Karlsruhe Institute of Technology
Department of Informatics

Chair of IT Security


Master's Thesis

# PolySphinx

## Extending the Sphinx Mix Network with Better Multicast Support


*Author*
Daniel Schadt

*Reviewers*
Prof. Dr. Thorsten Strufe

*Advisors*
M. Sc. Christoph Coijanovic
Dr. Christiane Kuhn


8 December 2021 – 8 June 2022

Ich versichere wahrheitsgemäß, die Arbeit mit dem Titel "PolySphinx — Extending the Sphinx Mix Network with Better Multicast Support" selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

**Karlsruhe, den 08.06.2022**

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(Daniel Schadt)

# Acknowledgements

I would like to thank my advisor Christoph for the informative discussions and comments he gave, as well as Christiane for the initial topic suggestion and the feedback she provided. Working with this subject has been fun and it fit my personal interests well.

Furthermore, I want to thank my family and especially my parents for their continued support throughout my years of studying. Without their help, my journey would've been much harder.

Finally, I would like to thank my partner for her emotional support, especially during the more stressful periods. She has provided some needed distractions and kept my spirit up.

**Abstract**

A lot of communication in the modern world happens digitally. However, when used carelessly, such communication channels can leak private information to third parties such as network operators or service providers. While end-to-end encryption can protect the content of messages, metadata such as the sender-recipient relationship is still leaked. One way to preserve privacy are mix networks, as introduced by Chaum [8], which relay messages over a series of nodes to make them untraceable.

A building block for a mix network is the mix format, which describes how the routing information needs to be packed and encrypted for the mix nodes to consume. Unfortunately, common mix formats are built for one-to-one messaging, even though a lot of communication is done in groups [25]. This makes them inefficient to use.

In this thesis, we will extend the Sphinx mix format [11] with capabilities that make it better suited for multicast messages, for example for group communication, by lowering the latency and the bandwidth requirements for multicast messages. Our *PolySphinx* approach allows a mix node to send the same payload to multiple recipients and relies on deterministic key trees that allow the recipient to reconstruct the original message.

We also devise and analyse alternative approaches based on key-homomorphic pseudorandom functions [5] and updatable encryption [20] and evaluate them against our PolySphinx suggestion.

Our evaluation shows that we can reduce the latency of multicast messages in a group with 128 members from 34.9 seconds for the naïve approach and 8.7 seconds with MultiSphinx to 2.1 seconds with PolySphinx while using less bandwidth in sending the data. This opens up new time-sensitive applications such as collaborative editing or instant messaging to be used with mix networks.

**Zusammenfassung**

Viel Kommunikation in der heutigen Welt findet digital statt. Falls solche digitalen Kommunikationskanäle jedoch unachtsam genutzt werden, können private Informationen an Drittparteien wie Netzbetreiber oder Dienstleister gelangen. Ende-zu-Ende Verschlüsselung hilft dabei, den Nachrichteninhalt zu schützen, jedoch werden Metadaten wie Sender und Empfänger nicht geschützt. Eine Möglichkeit, die Privatsphäre zu schützen, liegt in Mixnetzwerken wie sie von Chaum [8] vorgestellt wurden. Dabei werden Nachrichten über eine Folge von Mixknoten gesendet, damit sie nicht zurückverfolgt werden können.

Ein Baustein von Mixnetzwerken ist das Mixformat, das angibt, wie die Pfadinformationen in einer Nachricht kodiert werden müssen, sodass sie von den Mixknoten gelesen werden können. Leider sind gängige Mixformate für eine 1:1-Kommunikation entworfen, jedoch findet viel tatsächliche Kommunikation in Gruppen statt [25]. Daher sind sie ineffizient in der Nutzung.

In dieser Arbeit erweitern wir das Sphinx Mixformat [11] mit Möglichkeiten, die es besser für Gruppenkommunikation einsetzbar machen, indem wir die Latenz und die gesendete Datenmenge für Gruppennachrichten reduzieren. Unser *PolySphinx*-Ansatz ermöglicht es einem Mixknoten, dieselben Nutzdaten an mehrere Empfänger zu senden und basiert auf einem deterministisch aufgebauten Schlüsselbaum, der es den Empfängern ermöglicht, den Nachrichtentext zu rekonstruieren.

Wir stellen außerdem Alternativen basierend auf *key-homomorphic pseudorandom functions* [5] und *updatable encryption* [20] vor, und evaluieren diese gegenüber unserem PolySphinx-Ansatz.

Unsere Auswertung zeigt, dass wir die Latenz von Gruppennachrichten in einer Gruppe mit 128 Mitgliedern von 34,9 Sekunden mit dem naiven Ansatz und 8,7 Sekunden mit MultiSphinx auf 2,1 Sekunden mit PolySphinx reduzieren können, und dabei eine geringere Datenmenge senden. Dies ermöglicht neue Anwendungsfälle für Mixnetzwerke, wie das gemeinsame Echtzeit-Bearbeiten von Dokumenten oder Instant-Messaging.

# Contents


**1 Introduction** **8**

**2 Related Work** **11**

**3 Background** **12**
- 3.1 General Notation . . . 12
- 3.2 Cryptographic Primitives . . . 13
- 3.3 Sphinx . . . 14
  - 3.3.1 Threat Model . . . 15
  - 3.3.2 Required Primitives . . . 15
  - 3.3.3 Overview . . . 15
  - 3.3.4 Message Creation . . . 16
  - 3.3.5 The Sphinx Interface . . . 17

**4 Threat Model** **18**
- 4.1 Sequential Unicast . . . 19
- 4.2 Multicast at a Multiplication Node . . . 19
  - 4.2.1 Honest Node Before Multiplication . . . 21
  - 4.2.2 Honest Node After Multiplication . . . 21
  - 4.2.3 Multiplication Node is Honest . . . 22
  - 4.2.4 Summary . . . 22

**5 Our Approach** **24**
- 5.1 High-Level Overview . . . 24
- 5.2 The Sender Algorithm . . . 25

**6 Protocol Design** **27**
- 6.1 Key Tree . . . 27
- 6.2 Header Structure . . . 28
- 6.3 Creating a Header . . . 29
- 6.4 Creating a Message . . . 31
- 6.5 Message Processing at Mix Nodes . . . 32

**7 Security** **36**
- 7.1 Intra-Level Indistinguishability . . . 36
- 7.2 Inter-Level Indistinguishability . . . 39

**8 Implementation** 42
8.1 Chosen Primitives . . . 42
8.2 Rust Prototype . . . 42
8.2.1 External Libraries . . . 43
8.2.2 Prototype API . . . 43
8.2.3 Demonstration Program . . . 44

**9 Protocol Extensions** 45
9.1 Sender-Anonymous Direct Messages . . . 45
9.2 Improved Anonymity for Other Recipients . . . 45

**10 Alternative Approaches** 47
10.1 Key-Homomorphic Pseudorandom Functions . . . 47
10.1.1 The Learning-With-Errors Problem . . . 48
10.1.2 Constructions of Key-Homomorphic Pseudorandom Functions . . . 48
10.1.3 Evaluation in PolySphinx . . . 49
10.2 Updatable Encryption . . . 50
10.2.1 Security Notions . . . 50
10.2.2 Construction . . . 52
10.2.3 Evaluation in PolySphinx . . . 53
10.3 Re-Randomization of Ciphertexts . . . 53
10.3.1 Construction of a Re-randomisable Scheme . . . 53
10.3.2 Evaluation in PolySphinx . . . 54

**11 Evaluation** 56
11.1 Encryption Benchmark Results . . . 56
11.2 Latency Simulation Results . . . 58
11.2.1 Online Results . . . 59
11.2.2 Offline Results . . . 59
11.3 Overhead Analysis . . . 61

**12 Discussion** 66

**13 Conclusion** 67

# 1 Introduction

Digital communication is prevalent in the modern world. It appears in a lot of different applications and in many variations, such as emails, instant messaging, online conferences, internet telephony, and many more. While those applications benefit the user, they also pose a risk: A third party such as an eavesdropper or the service provider itself can easily listen in and analyse the data to use it for purposes other than those the user intended.

A popular way to combat this problem is to use protocols that are *end-to-end encrypted* (E2EE), such as Matrix[1] or Signal[2]. In this case, neither an eavesdropper nor the service provider can access the content of a message, as only the sender and the recipients have the proper keys to decrypt it. While this is a good step in the right direction, it does not prevent other information from being leaked. The service provider for example still knows who sent a message and for whom the message is destined, revealing the *sender-recipient-relationship*.

This relationship is enough to gather private and personal information: Imagine a user who frequently contacts an oncologist; in this case, the service provider might deduce that the user has cancer. Similarly, if a user frequently contacts a self-help group for addicts, the service provider could deduce that the user suffers from an addiction. Especially for sensitive information such as the aforementioned medical examples, this can be a big problem.

One way to protect this information and to hide the sender-recipient-relationship are *mix networks* (“mixnets” for short), as introduced by David Chaum [8]. In a mix network, a message is sent across multiple *mix nodes*, which collect incoming messages and relay them to their next destination in a way that an observer cannot relate an outbound message back to the corresponding incoming message. As such, as long as a message is sent through at least one mix node that is working as the protocol defines, and does not share its secrets with an adversary, nobody can trace it back to the sender. This removes the link between sender and recipient, both for outside observers and for the recipient.

Messages in such a mix network are packed according to a *mix format*, which defines how the metadata and the message content have to be prepared. In particular, the format describes how the routing information needs to be packed and how a mix node needs to process the message. The security of the mix format is fundamental to the security of a mix network: If the format does not properly encrypt all information, or if the format allows a message to be linked with its processed form, the privacy guarantees

[1] `https://matrix.org/`
[2] `https://signal.org/`

of the mix network are broken. Sphinx [11] describes such a format that promises both provable security and compactness.

One shortcoming of Sphinx is that it has no support for multicast messaging. Multicast messaging occurs every time a single message is sent to multiple recipients, such as in a mailing list or a group chat. How important such a feature is in practice shows a survey by Seufert *et al.*, according to which "it can be assumed that the group chat feature is used frequently by nearly every WhatsApp user, which makes it a key function of WhatsApp" [25, p. 229].

Of course, a naïve solution to the multicast problem would be to send multiple copies of the same message manually — an approach called *sequential unicast* [15]. However, clients are usually rate limited: To hide when messages are sent, every member sends messages at a fixed rate, using cover traffic when no messages are available. Clients generally use a lower sending rate than mix nodes, as they usually have fewer resources available. Therefore, the unicast approach has a huge cost in latency (due to the lower sending rate), as well as bandwidth consumption (as the message content has to be sent multiple times).

One optimisation to the unicast approach is *MultiSphinx* [15], a Sphinx extension in which multiple Sphinx messages are packed into a single MultiSphinx packet. This approach helps with the latency issue, as multiple messages can be sent at once. However, MultiSphinx still requires us to create a separate Sphinx message per recipient, meaning that we still end up duplicating the payload — and therefore not saving bandwidth compared to unicast.

In this thesis, we will propose an extension to the Sphinx mix packet format called *PolySphinx* that allows for efficient multicast messages to be sent. In particular, we want to improve on the bandwidth requirements and the latency problems that existing approaches have while keeping the security and anonymity aspects of Sphinx.

The main idea of PolySphinx is to enable a mix node along the path of a message to act as a multiplication node, sending the message to multiple recipients at once — in contrast to MultiSphinx however, we do not require the duplication of the payload. We adapt the Sphinx format in such a way that we can create multiple, indistinguishable copies of a given ciphertext while ensuring that each recipient can still obtain the plaintext.

**Our Contributions** We will introduce the PolySphinx format, which allows a message to be multicast without payload duplication. We will prove its security properties as well as give possible extensions and alternative approaches that we have considered. Furthermore, will evaluate our construction and the suggested alternatives in terms of computational overhead, message latency and size overhead, and compare it to the unicast strategy.

**Thesis Structure** In Chapter 3, we will introduce the required background and the existing Sphinx format that we want to improve upon. In Chapter 4, we will discuss the threat model and describe which information we want to protect and the type of adversary that we're considering. In Chapter 5, we will give a high-level description of

our PolySphinx approach, which we then define in more detail in Chapter 6. Chapter 7 contains the proof that our format fulfils our security requirements. In Chapter 8, we talk about the implementation of our scheme, both in terms of chosen primitives as well as the prototype implementation. In Chapter 9, we suggest possible extensions to our format, and in Chapter 10 we talk about alternative implementations. Chapter 11 contains the evaluation and comparison between PolySphinx, its variants and other multicast strategies. Finally, we end with the discussion in Chapter 12 and our conclusion in Chapter 13.

# 2 Related Work

Secure group communication is an important topic for practical applications. Cohn-Gordon *et al.* introduce the concept of an *Asynchronous Ratcheting Tree* (ART), which allows a group to derive a shared key efficiently, while also providing post-compromise and forward security [10]. However, their construction is focused on end-to-end encryption and does not try to hide the information about the sender in messages.

Chen and Chen improve on ARTs by introducing *Anonymous Asynchronous Rachet Trees* (AART), which provide the same functionality that ARTs provide while also providing anonymity [9]. Since ART originally relies on messages being delivered by a central server, they propose using one-time addresses and a blockchain in order to make the messages untraceable.

A shortcoming of Chen and Chen's AART is the disclosure of membership information and the reliance on a trusted third party, which Emura *et al.* note [13, p. 3]. Emura *et al.* instead propose a different extension to ARTs that can hide the information which user has updated a key, therefore enabling encrypted messaging without the need to disclose the sender of a message.

A mixnet that has multicast capabilities built-in is M2 [23]. The idea behind M2 is that there are "producers" that produce content that should be sent to many "consumers". The consumers register themselves with the producer and a path through the mix network, and the mixes along the way can identify shared prefixes of those paths and automatically multicast the message accordingly.

An extension to the Tor network allowing for multicast messages is MTor [21]. Here, the group members establish a multicast tree with a multicast relay at the root. Messages sent to the relay can then be multicast and sent to the remaining group members by the relay.

An approach that is completely different from mixnets are DC-nets, also introduced by Chaum [7]. They support the anonymous broadcast of messages by transmitting messages bit-by-bit, using random coin flips and parity broadcasts to hide information about the sender. They do however require a shared broadcast medium as well as a collision detection mechanism.

# 3 Background

In this section, we want to introduce notation and concepts that will be used throughout this thesis.

## 3.1 General Notation

We will use $\kappa$ to denote our security parameter. The bigger $\kappa$ is chosen to be, the more work an attacker has to do in order to break our security guarantees.

Given a finite set $A$ of elements, we write $x \xleftarrow{R} A$ to denote drawing a random element $x \in A$ from $A$, where each element has the same probability.

The term "probabilistic polynomial-time" (PPT) denotes that an algorithm can be simulated on a Turing machine that runs in polynomial time and may make probabilistic choices. We use the term "efficiently computable" to be synonymous with PPT.

We use the term *negligible* to be defined as follows, and we write *negl* to mean an arbitrary negligible function:

**Definition 1** (Negligible Functions [17])**.** *A function $f : \mathbb{N} \to \mathbb{R}$ is said to be* negligible*, if for every positive polynomial $p$ there is an integer $N$ such that for all integers $n > N$ it holds that $f(n) < p(n)^{-1}$.*

Informally, the Decisional Diffie-Hellman Assumption (DDH) is 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. In groups where the DDH assumption is said to "hold", such tuples are not distinguishable efficiently.

**Definition 2** (Decisional Diffie-Hellman Assumption [4])**.** *Let $\mathcal{G}$ be a group with generator $g$. We say that a group satisfies the DDH assumption iff for all PPT algorithms $\mathcal{A}$ and randomly drawn $a, b, c \xleftarrow{R} \{1, \ldots, |\mathcal{G}|\}$ the following holds:*

$$|\Pr[\mathcal{A}(|\mathcal{G}|, g, g^a, g^b, g^{ab}) = 1] - \Pr[\mathcal{A}(|A|(|\mathcal{G}|, g, g^a, g^b, g^c) = 1]| = \text{negl}$$

We will use $\mathcal{G}$ to be a prime-order cyclic group in which the Decisional Diffie-Hellman Assumption holds, $\mathcal{G}^*$ to be the subset of $\mathcal{G}$ without identity element, $g$ to be a generator of $\mathcal{G}$ and $q$ to be the order of $\mathcal{G}$.

We use the concept of *sequences*, which are similarly to sets a collection of elements. Unlike sets, however, a sequence keeps the order of elements, such that each element can be referred to by its index. Both sets and sequences are denoted by uppercase letters ($S$).

Given a bit-string $a \in \{0,1\}^*$, we write $a_{[x..y]}$ to denote the substring of $a$ spanning from bits $x$ to $y$, inclusive. Given two bit-strings $a, b \in \{0,1\}^*$, we use $a +\!\!+ b$ to denote the concatenation of $a$ followed by $b$. Given two bit-strings $a, b \in \{0,1\}^k$ of same length $k$, we use $a \oplus b$ to denote the bitwise exclusive-or ("xor") of the two bit-strings. We write $\mathtt{0}_k$ to denote a bit-string consisting of $k$ zero-bits.

## 3.2 Cryptographic Primitives

We will introduce a number of cryptographic primitives, which will be used in both Sphinx and our PolySphinx extension:

**Definition 3** (Hash Function)**.** *A hash function $h$ is an efficiently computable mapping $h : \mathcal{X} \to \mathcal{Y}$ where $|\mathcal{Y}|$ is finite.*

We model hash functions as random oracles, meaning that we expect the output of $f$ to look uniformly random:

**Definition 4** (Random Oracle)**.** *A function $h$ is said to be a* random oracle *if $h(x)$ is chosen uniformly random.*

**Message Authentication Code** A *message authentication code* (MAC) is an algorithm that can be used to verify the authenticity and integrity of a message [17, p. 109]. We model a MAC as a single function that takes as input a key and the message and outputs a "tag" which can be stored alongside the message. To verify a message, we can compute the MAC again using the same key and compare whether the computed tag matches the stored tag. The security of a MAC is defined by the inability to create a valid tag without knowledge of the right key [17, p. 110].

**Definition 5** (Message Authentication Code)**.** *A MAC is an efficiently computable function $\mu : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ where*

- *$|\mathcal{K}|$ is finite and $\mathcal{K}$ is called the key space*
- *$\mathcal{X}$ is called the message space*
- *$|\mathcal{Y}|$ is finite and $\mathcal{Y}$ is called the tag space*

Again, we view $\mu$ with a fixed key as a random oracle, essentially treating $\mu(k, \cdot)$ as a hash function.

**Pseudorandom Generator** A *pseudorandom generator* (PRNG) is an algorithm that takes a few bits of (truly) random input and produces a longer sequence of "pseudorandom" output. More formally, we can define a PRNG as follows:

**Definition 6** (Pseudorandom Generator [17, p. 62])**.** *Let $l(\cdot)$ be a polynomial and $\rho$ be a deterministic polynomial-time computable function $\rho : \{0,1\}^\kappa \to \{0,1\}^{l(\kappa)}$.*

*We say that $\rho$ is a pseudorandom generator if*

- $\forall \kappa : l(\kappa) > \kappa$
- *For all PPT algorithms $\mathcal{A}$, the advantage that $\mathcal{A}$ has when distinguishing between true randomness and the output of $\rho$ is negligible. More formally, given $r \leftarrow^R \{0,1\}^{l(\kappa)}$ and $s \leftarrow^R \{0,1\}^\kappa$, we require that*

$$|\Pr[\mathcal{A}(r) = 1] - \Pr[\mathcal{A}(\rho(s)) = 1]| \leq \mathit{negl}(\kappa)$$

In practice, we can generate more than $l(\kappa)$ bits from the PRNG by re-seeding it with a part of its output. However, that does not mean that we gain "more security": In the worst case, an attacker can always guess the initial seed — which only has $\kappa$ bits of security.

Since an "infinitely long" bit-string can never be constructed, we instead write $\rho(x)_{[a..b]}$ to denote the output bits $a$ to $b$ (inclusive) of the PRNG seeded by $x$.

**Pseudorandom Permutation** A *pseudorandom permutation* (PRP) is a pseudorandomly chosen bijective mapping $\{0,1\}^n \to \{0,1\}^n$. Generally, we want a whole family of pseudorandom permutations, indexed by a key. We call this a pseudorandom permutation family:

**Definition 7** (Pseudorandom Permutation Family [17, p. 94])**.** *A pseudorandom permutation family consists of functions $\pi : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ and $\pi^{-1} : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$, such that*

- *The functions $\pi$ and $\pi^{-1}$ are efficiently computable.*
- $\forall k \in \{0,1\}^\kappa \, \forall m \in \{0,1\}^n : \pi^{-1}(k, \pi(k,m)) = m$
- *For all PPT algorithms $\mathcal{A}$, the advantage that $\mathcal{A}$ has when distinguishing between a random mapping $F$ and the output of $\pi(k,\cdot)$ for a fixed key is negligible. More formally, for $k \leftarrow^R \{0,1\}^\kappa$ and $F \leftarrow^R (\{0,1\}^n \to \{0,1\}^n)$, we require that*

$$|\Pr[\mathcal{A}(F(\cdot)) = 1] - \Pr[\mathcal{A}(\pi(k,\cdot)) = 1]| \leq \mathit{negl}(\kappa)$$

## 3.3 Sphinx

Sphinx was introduced by Danezis and Goldberg in 2009 [11]. It provides a mix format that promises both compactness and provable security. Since PolySphinx is an extension of the Sphinx format, we want to first lay down the basics of Sphinx by introducing the relevant aspects.

### 3.3.1 Threat Model

Sphinx assumes the existence of an active adversary that can observe all traffic in the network as well as inject arbitrary messages. It is assumed that a subset of the mix nodes can be *corrupt*, meaning that those nodes cooperate with the adversary. In this case, the adversary knows the corrupt node's private key, and it may instruct the node to act in ways that are not conforming to the protocol. We call nodes that are not corrupt and that do not cooperate with the adversary *honest*.

The goal of the adversary is to infer information about the sender-recipient relationship of a message. Sphinx aims to hide this information as long as there is at least one honest node along the path of a message.

### 3.3.2 Required Primitives

Sphinx makes use of a number of cryptographic primitives:

**Message Authentication Code** We need a MAC, where our MAC key and tag are bit-strings of length $\kappa$. We denote our MAC as $\mu : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^\kappa$. We further use a hash function $h_\mu : \mathcal{G}^* \to \{0,1\}^\kappa$ to derive a key for our MAC from a group element.

**Pseudorandom Generator** We use $\rho : \{0,1\}^\kappa \to \{0,1\}^\infty$ to denote a pseudorandom generator which internally re-seeds itself and is able to output arbitrarily many bits. We will use this PRNG as a stream cipher.

We also define a corresponding hash function $h_\rho : \mathcal{G}^* \to \{0,1\}^\kappa$ that we can use to derive a seed for our PRNG from a group element.

**Pseudorandom Permutation** In the context of Sphinx, we use $\pi : \{0,1\}^\kappa \times \{0,1\}^{l_\pi} \to \{0,1\}^{l_\pi}$ to denote a family of pseudo-random functions that is used to encrypt the message payload between hops. We further use a hash function $h_\mu : \mathcal{G}^* \to \{0,1\}^\kappa$ to derive a key for $\pi$ from a group element.

**Blinding Factors** The last primitive that we need is a hash function $h_b : \mathcal{G}^* \times \mathcal{G}^* \to \mathbb{Z}_q^*$ to generate so-called *blinding factors* from two group elements.

### 3.3.3 Overview

A Sphinx network consists of mix nodes and users. If a user wants to send a message to another user, they first pick a random path of mix nodes through the network, generate the header for the message, and then send it to the first mix node along the path. The message is then relayed from one mix node to the next mix node along the path until it arrives at the final node. The final node acts as the "exit node", which is responsible for delivering the message to the actual recipient.

Sphinx identifies mix nodes $n$ by a bit-string $n \in \{0,1\}^\kappa$ and recipient addresses by a bit-string $d \in \{0,1\}^{2\kappa}$. The set of all mix nodes shall be denoted as $\mathcal{N} \subset \{0,1\}^\kappa$, and each mix node is assigned a private key $x_n \leftarrow^R \mathbb{Z}_q^*$ and a public key $y_n = g^{x_n} \in \mathcal{G}^*$. The set of all destination addresses is denoted as $\mathcal{D}$. We note that the union of mix node identifiers and destination addresses $\mathcal{N} \cup \mathcal{D}$ must be prefix-free: This construction allows a mix node to determine whether it should relay the message to another node or deliver it to its intended recipient without the need for extra information.

Throughout this thesis we will use $r$ to be the number of maximum mix node "hops" that can be encoded in the header. This limits the number of mix nodes a message can travel through.

A Sphinx message consists of two main parts: A header $h$, consisting itself of three elements $h = (\alpha, \beta, \gamma)$, as well as the payload $\xi$. A mix node is interested mainly in the header, as it contains the information necessary to forward the message to the next node, whereas the payload is simply "passed through" — after the mix node has unwrapped a layer of encryption from the payload.

Taking a closer look at the elements of a header, we first have $\alpha \in \mathcal{G}^*$. This is the so-called *Sphinx group element* which can be seen as a pre-computed Diffie-Hellman key exchange. The mix node can use $\alpha$ together with its private key $x_n$ to compute a shared secret between the node and the sender. This shared secret is the basis for most of the processing done by the mix node, which has two benefits: The secret is unknown to the other mix nodes, as they do not have access to the same private key, but the secret is known to the sender, who can use it as a shared key to prepare and encrypt the routing information.

The second part of the header is $\beta \in \{0,1\}^{(2r+1)\kappa}$, which we can call the *routing data*. It contains the information necessary for the mix node to find the next destination for the message. The routing data is encrypted via the stream cipher $\rho$ keyed by the shared secret, such that the mix node cannot access more data than it needs.

Finally, $\gamma$ is a MAC that ensures that the header is genuine and has not been tampered with. Again, the MAC is keyed by the shared secret, which ensures that other mix nodes along the path cannot tamper with the header either.

### 3.3.4 Message Creation

If a sender wants to create a Sphinx message, it does so in two parts: First, the Sphinx header has to be created, and then the message has to be prepared.

In order to create the header, the sender draws a random starting point $x \leftarrow^R \mathbb{Z}_q^*$. This is used as the basis for calculating the Sphinx group elements and therefore for the Diffie-Hellman key exchanges. Together with the generator $g$ and the public keys of the nodes along the path, the sender can compute the sequence of Sphinx group elements $\alpha_i$ and the shared secrets $s_i$. The blinding factors generated by $h_b$ are used to make the Sphinx group element unrecognizable between hops, by using $\alpha_{i+1} = \alpha_i^{h_b(\alpha_i, s_i)}$ as the value for the next hop. Otherwise, a message could be identified by its value of $\alpha$.

In order to prepare the payload, the sender wraps it in multiple layers of encryption,

in reverse order of the mix nodes along the path: If $s_{\upsilon-1}$ denotes the shared secret with the last mix node, the payload is prepared as

$$\pi\left(h_\pi\left(s_0\right), \pi\left(h_\pi\left(s_1\right), \ldots, \left(\pi\left(h_\pi\left(s_{\upsilon-1}\right), \xi\right)\right)\right)\right)$$

Such schemes are commonly called *onion encryption*: The idea here is that each mix node "unwraps" one layer of encryption by applying $\pi^{-1}$ with its shared secret, so that once the message arrives at the destination, all layers have been removed — similarly to an onion that has its layers peeled away piece by piece.

### 3.3.5 The Sphinx Interface

If we think of Sphinx as a library to use, we can define the following algorithms to interface with Sphinx:

CreateSphinxHeader($\Delta, I, N$) (Section 3.2 in [11]): This is the algorithm that creates a Sphinx header. It takes as input the recipient's address $\Delta$, an identifier $I$ and the sequence of mix nodes $N$ that the message should traverse. The output of the algorithm is a tuple $(M_0, S)$, where $M_0$ is the bit-string that represents the header, and $S$ is the sequence of shared secrets for the nodes along the path. We note that the header creation works independently from the payload, which is useful if the sender wants to prepare a header for future use (such as for reply blocks).

CreateSphinxMessage($m, \Delta, N$) (Section 3.3 in [11]): This is the algorithm that creates a message. It takes as input the payload $m$, the recipient's address $\Delta$ and a sequence of mix nodes $N$ that the message should traverse. Internally, it uses CreateSphinxHeader to create the appropriate header first and then encrypts the payload with the shared secrets. The output of the algorithm is the tuple $(M_0, \rho)$, where $M_0$ is the created header and $\rho$ is the encrypted payload.

UnwrapSphinxHeader($M$) (Section 3.6 in [11]): This is the inverse algorithm to CreateSphinxHeader. It uses a node's secret key to unwrap the bit-string and extract the node-specific routing information. Depending on the routing information, it then returns the address of the next mix node, or the recipient to forward the message to.

# 4 Threat Model

Our threat model is similar to the one that Sphinx is evaluated against (see Section 3.3.1), with extensions for multicast messaging. Again, we assume that our adversary can observe the complete traffic flow in the network, as well as modify messages during their transit or inject new messages into the network. It may also corrupt a subset of the mix nodes in the network, gaining access to their private keys and being able to control their functioning.

Before we continue to talk about and compare the different multicast strategies, we want to think about the information that an adversary could learn in the first place:

- *Sender-Recipient Relationship*: The sender-recipient relationship describes the fact that the sender and recipient are communicating with each other. The goal of the mixnet is to hide this information from the adversary.

  In the context of multicast messaging, we view each recipient as a separate sender-recipient relationship.

- *Message Sender*: Hiding the message sender means that the recipient (honest or not) cannot determine who sent the message. This gives us "sender anonymity".

- *Other Recipients*: Given a multicast message, a malicious recipient could learn the other recipients of the message.

In order to define the model more precisely, we will look at it from different points of view:

- *External Adversary*: This is the point of view of our global adversary that may corrupt mix nodes, but no recipients.

- *Internal Adversary*: This is the point of view of an honest (but curious) recipient that does not work together with the adversary or other recipients but still tries to extract as much information as possible from the message that it has received.

- *Internal + External Adversary*: This is the point of view of a corrupt recipient that may work together with the adversary.

We note that we do not concern ourselves with side-channel attacks (such as timing attacks) or attacks on the availability of the network or single mix nodes (such as denial of service attacks). Those attacks are considered out of scope for this thesis, and they have to be solved by different mechanisms.

In the following sections, we will look at the different multicast strategies to see which kind of information they leak to which adversary. This allows us to draw comparisons between the privacy goals that those strategies can achieve.

## 4.1 Sequential Unicast

Sequential unicast works by sending $p$ copies of the message to the recipients $r_1, \ldots, r_p$. For each of those copies, a new path is generated by the sender $s$. As the security of this method is only based on the underlying Sphinx guarantees for $p$ independent messages, this strategy is optimal in terms of privacy protection.

**Sender-Recipient Relationship** If a path $s \to r_i$ has at least one honest node, this sender-recipient relationship is protected — as per the standard Sphinx guarantees. This means that even if all other paths consist of only corrupt nodes, the communication between $s$ and $r_i$ is not traceable for the adversary.

Conversely, if a path $s \to r_i$ consists of only corrupt nodes, the adversary may learn the relationship between $s$ and $r_i$, but it does not gain an advantage in guessing the relationships between $s$ and $r_{j \neq i}$.

**Message Sender** Neither the adversary nor the recipient can learn the identity of the sender of the message, given that at least one honest node is on each message path.

If there is a path that has no honest node between the sender $s$ and a corrupt recipient $r_i$, other corrupt recipients may collaborate with $r_i$ to compare the message content and then assume that the same sender has sent the message. Such an attack however is hard to prevent: Once one recipient has identified the sender, the other recipients can always collaborate and assume the same sender has sent the message — no matter the underlying multicast scheme.

**Other Recipients** Since each message is sent individually, neither the recipient nor the adversary gets any information about the other recipients. This also means that a corrupt recipient cannot help the adversary in determining the remaining recipients.

Furthermore, as sequential unicast does not have the concept of "groups" built-in, the adversary cannot learn anything about group members. Since this strategy can be used to send messages to "ad-hoc groups" that have not been set up in advance, such a list of group members may or may not exist in the first place.

The information that is leaked is summarized in Table 4.1.

## 4.2 Multicast at a Multiplication Node

Ideally, we would like to view the "multicast at a multiplication node" approach that MultiSphinx and PolySphinx take as a special case of sequential unicast, in which the first part of the path is the same for every message. This would change the chance that

| Type of adversary | External | Internal + External | Internal |
|---|---|---|---|
| Sender-Recipient Relationship | ✗ | ✗ | ✗ |
| Message Sender | ✗ | ✗ | ✗ |
| Other Recipients | ✗ | ✗ | ✗ |

Table 4.1: Information leaks with the sequential unicast strategy, assuming an honest node on every path between the sender and a recipient. A ✓ indicates that the information may be leaked, a ✗ means that it is safe.

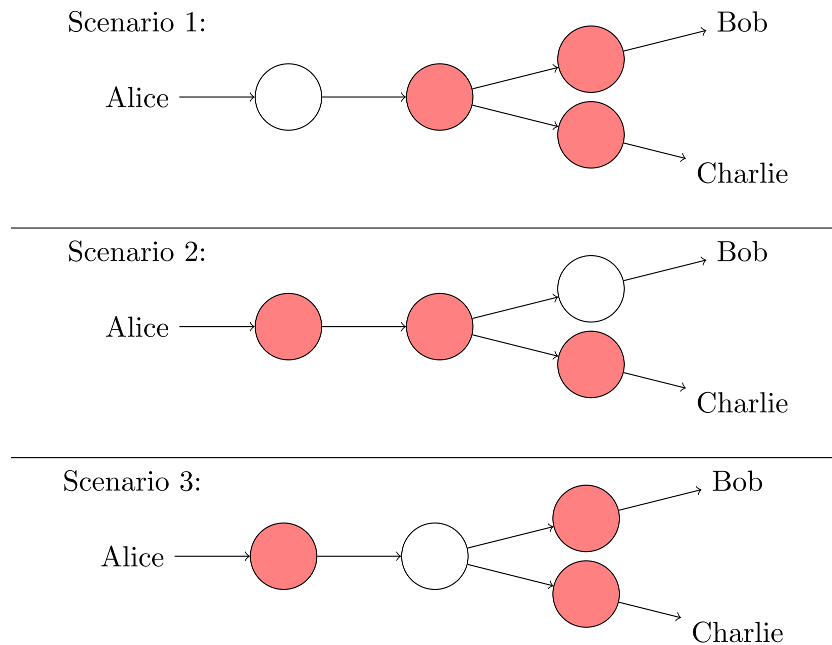


Figure 4.1: Different scenarios based on where the honest node sits. We assume that the red nodes are corrupt.

a corrupt node is included (as fewer random choices have to be made), but it would still give strong security guarantees.

However, due to the addition of the multiplication node, those two strategies are not equivalent: While the argument works in terms of protecting the anonymity of the sender, the adversary can gain an advantage in determining the recipients of the message depending on which nodes are corrupt.

In order to properly analyse the multicast approach, we, therefore, have to distinguish different cases based on where the honest node is in relation to the multiplication node, as shown in Figure 4.1.

### 4.2.1 Honest Node Before Multiplication

In this situation, the honest node sits between the sender and the multiplication node, depicted as Scenario 1 in Figure 4.1. We can say the following about possible information leaks to the adversary:

**Sender-Recipient Relationship** As the honest node doesn't allow the message to be traced back all the way to the sender, every sender-recipient relationship is protected and not leaked to the adversary.

**Message Sender** From the point of view of every recipient, the honest node is always part of the path to the sender. Unless the sender embeds information that allows the recipients to learn its identity, the sender is therefore protected.

**Other Recipients** Since the only honest node is between the sender and the multiplication node, we can assume that the multiplication node and all nodes behind it are corrupt. In this case, it is possible to trace a message back to the multiplication node and then to the other recipients.

### 4.2.2 Honest Node After Multiplication

In this situation, the honest node sits between the multiplication node and the recipient, depicted as Scenario 2 in Figure 4.1.

**Sender-Recipient Relationship** Similarly to the sequential unicast strategy, the honest node in this case protects the single sender-recipient relationship on which path it is. The adversary can still learn about the other sender-recipient relationships. For example, if the honest node is between Alice and Bob, then the adversary cannot learn that those two are communicating with each other. However, it can still learn that Alice is communicating with Charlie.

Assuming that there is an honest node on every path between the multiplication node and the recipients, the adversary will not be able to learn any relationship.

**Message Sender** Similarly to the previous scenario, the adversary may not gain information about the message sender for the one recipient that is behind the honest node.

**Other Recipients** The recipient that sits behind the honest node is safe, as an adversary cannot trace the message to other recipients, and vice versa. However, all recipients that do not have an honest node in their path can be linked together.

### 4.2.3 Multiplication Node is Honest

We can treat an honest multiplication node as the combination of advantages from both other scenarios: While messages can be traced back to the multiplication node, they cannot be traced to other recipients, nor to the sender.

The downside of this scenario is that the sender cannot rely on the multiplication node being honest — if the sender could trust the multiplication node, there would be no need to even include other mix nodes. We still need to include this case for the chance of it happening though, to ensure that the strategy won't have "edge cases" that lead to a lowered security.

### 4.2.4 Summary

The leaked information is summarised in Table 4.2. We can see an interesting trade-off: In every scenario, the sender is protected. Therefore, the best strategy would be to keep the path in front of the multiplication node as long as possible to increase the chance of including an honest node. The paths after the multiplication node can then be short. This would give the best protection to the sender, while also keeping the strain on the network to a minimum.

On the other hand, the recipients and group members have better protection when the honest node is after the multiplication node. In this case, the sender would like to keep the path before the multiplication node short, and then choose long paths after the multiplication. This ensures the best protection but strains the network more and requires more choices from the sender.

Scenario 1:

| Type of adversary | External | Internal + External | Internal |
|---|---|---|---|
| Sender-Recipient Relationship | ✗ | ✗ | ✗ |
| Message Sender | ✗ | ✗ | ✗ |
| Other Recipients | ✓ | ✓ | ✗ |

Scenario 2, assuming an honest node between the multiplication node and every recipient:

| Type of adversary | External | Internal + External | Internal |
|---|---|---|---|
| Sender-Recipient Relationship | ✗ | ✗ | ✗ |
| Message Sender | ✗ | ✗ | ✗ |
| Other Recipients | ✗ | ✗ | ✗ |

Scenario 3:

| Type of adversary | External | Internal + External | Internal |
|---|---|---|---|
| Sender-Recipient Relationship | ✗ | ✗ | ✗ |
| Message Sender | ✗ | ✗ | ✗ |
| Other Recipients | ✗ | ✗ | ✗ |

Table 4.2: Information leaks with a multiplication node strategy. A ✓ indicates that the information may be leaked, a ✗ means that it is safe.

# 5 Our Approach

This section provides the general ideas behind our approach and how it differs from Sphinx [11] and MultiSphinx [15].

## 5.1 High-Level Overview

The basic idea behind our approach is to extend Sphinx such that a node along the path of a message can act as a multiplication node, multiplying a single message and sending it to multiple recipients. We will use $p$ to refer to the "multiplication factor", which is the number of copies that the multiplication node splits a message into.

We keep the payload the same for every recipient, avoiding the need to duplicate this information — as contrasted to the MultiSphinx approach introduced by Hugenroth *et al.* [15]. This has a big benefit: Instead of having a huge overhead through payload duplication, we restrict ourselves to a small overhead that only contains the receiver-specific routing and encryption information. In addition, instead of requiring the sender to provide multiple encryptions of the payload, we enable the multiplication node to create multiple indistinguishable copies on its own.

There are two problems that need solving: The first one is that there is additional routing information that the nodes need, especially the multiplication node. Sphinx already provides a mechanism to get information to the right node, and we know that Sphinx headers are "secure" in the sense that a node can only decrypt the part of the header that belongs to it — and only after it has passed through the right path. Therefore, our approach involves increasing the header size and then putting the extra information in the header.

The other important problem that arises is how to recover the message at the recipient's end. Since the message may take different paths after the multiplication node, we cannot "prepare" the payload in the correct way when sending it, as the payload is shared. We also cannot trust the multiplication node to do this for us, as that would leak too much information to the multiplication node — which may be compromised.

Instead, we ensure that the multiplication node does not learn too much information by doing as much work as we can on the sender-side, and then providing the pre-built pieces to the multiplication node. Rather than sending the names or addresses of the recipients to the multiplication node, we embed pre-built headers (one for each recipient) into the header of the original message. The multiplication node can then extract the headers for the following paths, combine them with the payload and send the resulting messages on their way.

For this to work, however, we need to have a hierarchical system: Since message

headers are padded to the same length in the Sphinx format, embedding a header into a header would lead to an infinitely large structure. We solve this by defining different "layers" of the network, with different header sizes (and therefore a different number of recipients) per layer. A header of layer $n$ can then embed headers of layer $n-1$.

It is useful to think of the original function CreateSphinxHeader($\Delta, I, N$) (Section 3.3.5) as being extended to carry arbitrary information to a node: Instead of a simple sequence of nodes $\{n_1, n_2, \ldots, n_\nu\}$, it takes pairs $\{(n_1, d_1), (n_2, d_2), \ldots, (n_\nu, d_\nu)\}$ where $d_i$ is the extra-information to pass to $n_i$.

Additionally, we let the messages along the way be encrypted by pseudorandomly chosen keys. The sender embeds a pseudorandom secret key into the header for each mix node, which it will use to "add" an encryption layer — therefore making the incoming and outgoing message unlinkable for an outside observer. In order to allow the recipient to recover the message, the sender embeds the pseudorandom seed into the final header, allowing it to reconstruct the secret keys used along the path.

This approach is in contrast to the way that standard Sphinx works: In Sphinx, the layered encryption (the "onion") is built at the start, such that every mix node removes one layer of encryption, and the message arrives in plain at the recipient's end. This is possible because the sender can pre-calculate the shared secrets with the mix nodes, and therefore encrypt the payload accordingly. In our approach, however, the onion starts without encryption layers, and each mix node adds a layer of encryption with the key provided by the sender. The recipient can then unwrap all onion layers with the keys generated from the pseudorandom seed.

## 5.2 The Sender Algorithm

If a sender $s$ wants to send a message with payload $\xi$ to $j$ recipients, $r_1, r_2, \ldots, r_j$, the sender first constructs the "multiplication schedule". That is, the sender needs to determine at which point the message should be multiplied by a multiplication node.

Afterwards, the sender will pick a random seed $\mathcal{S}$ and generate a *key tree* $\mathcal{K}$ (Figure 5.1). The key tree allows the sender to derive the pseudorandom keys that will be used by the mix nodes: As long as the message has not been multiplied yet, we move along a single path in the key tree. Once the message has been multiplied, the path splits up and different keys are used in order to make the copies of the message unlinkable.

The sender now picks the mix nodes $n_i$ that should be used, each mix node corresponds to a node in the path through the key tree. We will denote a path for a message to $r_i$ by $P_i$, and the single path choices by $p_{i,x}$ with $1 \leq x \leq \nu$.

Now, $s$ can prepare the message header iteratively, by first computing the headers for the lowest level — that is, the level that brings the message to a single recipient — and then combining the lower levels into a higher-level message.

In order to compute the lowest level, for each recipient $r_i$, its corresponding "last mile" $n_\mu, \ldots, n_l$ and the corresponding choices in the key tree $p'_\mu, \ldots, p'_l$, the headers can be

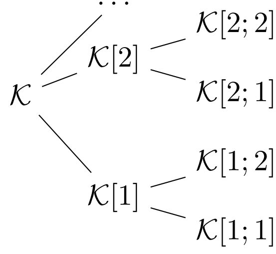


Figure 5.1: Schematic view of the key tree. Each node has $p$ children.

computed as follows:

$$\begin{aligned} h_i = \textsc{CreateSphinxHeader}(\Delta &= r_i, \\ I &= (\mathcal{S}, P_i), \\ N &= \{(n_{\mu+1}, H(\mathcal{K}[\ldots; p'_\mu])), \ldots, (n_l, H(\mathcal{K}[\ldots; p'_l]))\}) \end{aligned}$$

Intuitively speaking, the sender embeds the key tree seed and the path that the message took (in the key tree) for the recipient. For the mix nodes, the sender only embeds the derived key, so as to not leak the previous or next keys to the mix node.

Now, the headers $h_i$ of a lower level can be combined into a multicast header:

$$\begin{aligned} h' = \textsc{CreateSphinxHeader}(\Delta &= n_\mu, \\ I &= \{h_1, \ldots, h_j\}, \\ N &= \{(n_1, H(\mathcal{K}[p_1])), \ldots, (n_{\mu-1}, H(\mathcal{K}[\ldots; p_{\mu-1}]))\}) \end{aligned}$$

The sender can then send $(h', \rho)$ through the network, and each recipient has the information necessary to compute the encryption keys that were used along the path of the message. Therefore, the recipient can “unwrap” the encryption layers and restore the message plaintext.

# 6 Protocol Design

In this chapter, we will give a more detailed description of how the protocol works, and how a message can be constructed. We define the protocol bottom-up, by first introducing the lower-level designs, and then combining them at the end.

Similar to a standard Sphinx message, a PolySphinx message consists of two parts: There is the message header which contains the necessary information for routing and forwarding, as well as the message payload. We will denote a message as a tuple $(h, \xi)$, where the first element is the header and the second element is the payload.

## 6.1 Key Tree

Before we talk about the generation of the header, we first need to concern ourselves with the generation of the keys that will be used along the message path. We do this in the form of a *key tree*, where a child node is a key that is derived from its parent and the path, and the root is derived from an initial random seed.

The advantage of the key tree compared to drawing all keys randomly is that the receiver can generate the same keys that the sender has used from just the seed and a few bits of path information. On the other hand, using a full tree instead of a predefined partial tree allows for more flexibility when choosing the path through the mix network, as the sender can choose freely when and how often the message should be multiplied.

We denote the key tree as $\mathcal{K}$, and we define how to generate it recursively:

**Definition 8.** *Given a seed* $\mathcal{S} \in \{0,1\}^\kappa$ *and a hash function* $H : \{0,1\}^\kappa \to \{0,1\}^\kappa$, *we define our key tree as follows:*

- *The root* $\mathcal{K}[]$ *is defined as* $\mathcal{K}[] \coloneqq H(\mathcal{S})$.
- *Given a parent* $\mathcal{K}[\ldots]$, *its* $i$*th child* $\mathcal{K}[\ldots; i]$ *is defined as* $\mathcal{K}[\ldots; i] \coloneqq H(\mathcal{K}[\ldots] + i)$ *for* $1 \leq i \leq p$.

It is important that the sender will not embed the values from the key tree nodes directly: Since knowing the value of a node allows you to compute the descendent nodes easily, "leaking" this information to a mix node would allow it to compute the following secret keys as well.

In order to prevent the mix nodes from deriving other keys in the key tree, the sender will not embed the tree nodes directly. Instead, an additional hash function is applied to the generated nodes, hiding the actual values.

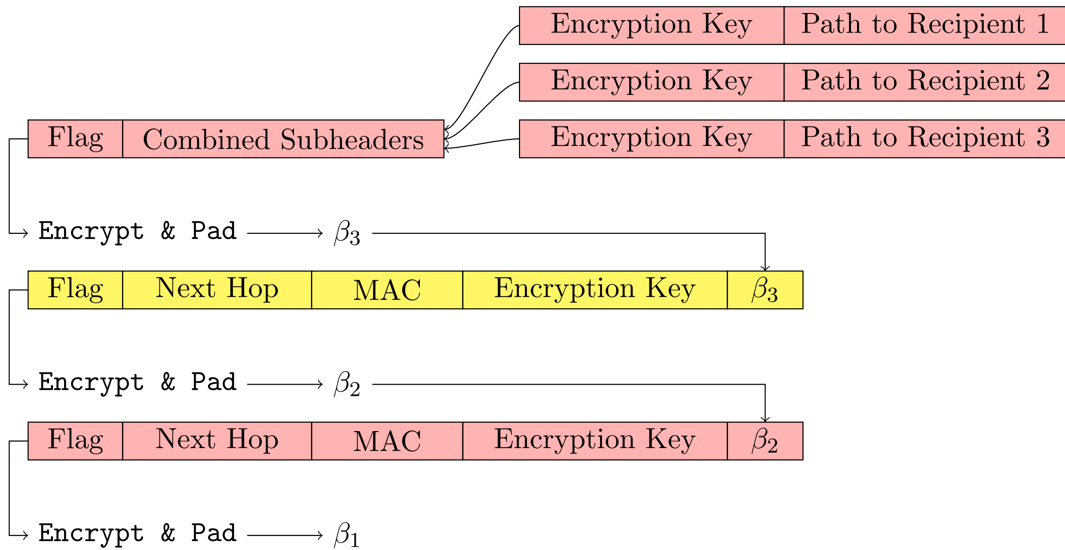


Figure 6.1: The high-level structure of the header. As an example, the information in yellow is visible to the second mix node, while the parts in red are encrypted and inaccessible.

## 6.2 Header Structure

Our header has a very similar structure to the Sphinx header [11, p. 5], consisting of the same three parts, $h = (\alpha, \beta, \gamma)$.

There are a few differences, however: First of all, we add $\kappa$ bits of extra information to be carried to each mix node, allowing us to embed the extra encryption key for each node.

Additionally, we allow extra information to be embedded for the final node. In the original Sphinx approach, this is used to carry a $\kappa$-bit long "identifier", but in our case, we want to expand this space to carry the subheaders: A header of level $\lambda \neq 0$ then contains $p$ headers of level $\lambda - 1$, while a header of level $\lambda = 0$ contains a single destination address and the information necessary for decryption.

Finally, we make use of *flag bytes* in our header: Even if the intersection of mix node addresses and recipient addresses $\mathcal{N} \cup \mathcal{D}$ is prefix-free — as per the original Sphinx construction — a mix node would not know whether it should parse a single address for the next hop, or if it should act as the multiplication node. We could fix this problem by having designated multicast nodes, however, to keep the protocol flexible, we instead introduce flag bytes. Those are bytes with a distinguished value that prefix the routing data and signal the mix node which action it should take.

A sketch of the header structure is shown in Figure 6.1. The elements on the left are the ones that are "visible" to the node, while the elements to the right are hidden behind more layers of encryption — as per the original Sphinx approach.

Before we can construct a header, however, we need to determine a header's size,

more specifically the size of the routing data $|\beta|$. This is important in order to properly encrypt and pad the header.

**Size per Hop** No matter the level of the header in the hierarchy, for every hop that our message should take, we need $\kappa$ bits for the next node identifier and the MAC each and $\kappa$ bits to store the encryption key. Additionally, we prefix the header with a designated flag byte $f_{\text{Relay}}$ to indicate that the message should be relayed, arriving at a cost of $8 + 3\kappa$) bits per hop.

Depending on the level, the header then needs to include a different amount of routing data.

**Base Level** For the level $\lambda = 0$, which is the level that does not allow for any subheaders to be embedded, we need $2\kappa$ bits to store the recipient's address, $\kappa$ bits to store the identifier, $\kappa$ bits to store the key tree seed and $r \cdot \lceil \log_2 p \rceil$ bits to store the path through the key tree. Additionally, we use 8 bits to prefix the header payload with a designated flag byte $f_{\text{Destination}}$.

We therefore get the following size in bits for the base level:

$$\begin{aligned} \text{SIZE}_0 &= (r-1) \cdot (8 + 3\kappa) + 8 + 4\kappa + r\lceil \log_2 p \rceil \\ &= r(8 + 3\kappa) + \kappa + r\lceil \log_2 p \rceil \end{aligned}$$

**Multicast Levels** A header of level $\lambda > 0$ must contain $p$ headers of level $\lambda - 1$. In addition, for each header, we also need to include $\kappa$ bits to store the identifier of the next hop, as well as $\kappa$ to store an encryption key. To signal the mix node that the multiple headers should be read, we use a designated flag byte $f_{\text{Multicast}}$.

We note that a header of level $\lambda - 1$ needs more than $\text{SIZE}_{\lambda-1}$ bits to store, as that is only the size of the routing data $\beta$. In addition to that, we need $2\kappa$ bits to store $\alpha$ and $\kappa$ bits to store $\gamma$.

For all $\lambda > 0$ we therefore get the following size for our header:

$$\text{SIZE}_\lambda = (r-1) \cdot (8 + 3\kappa) + 8 + p \cdot (5\kappa + \text{SIZE}_{\lambda-1})$$

## 6.3 Creating a Header

Similar to the header creation in Sphinx [11, p. 5], we want to define how a PolySphinx header can be created. The main difference in our extended version is that we will embed the encryption key for each node explicitly, rather than the node deriving the encryption key from the shared secret with the sender.

We denote the function to create a header by $\text{CreatePolyHeader}_\lambda(\Delta, N)$, where

- $\lambda$ is the level of the header that we want to create.

- $\Delta$ is the routing data to be transmitted to the final node
  - For $\lambda = 0$, this is the address of the recipient together with the identifier and the information needed for decryption.

    In this case, we limit the size of $\Delta$ to $|\Delta| = 4\kappa + \lceil \log_2 p \rceil$.
  - For $\lambda > 0$, this is the set of subheaders that the multiplication node needs.

    In this case, we limit the size of $\Delta$ to $|\Delta| = p(5\kappa + \text{SIZE}_{\lambda-1})$.
- $N$ is a sequence of pairs, each containing a mix node and the encryption key: $N = \{(n_0, \sigma_0), \ldots, (n_{\upsilon-1}, \sigma_{\upsilon-1})\}$

The actual header generation now works similarly to Sphinx [11, Section 3.2]: We first pick a random $x \leftarrow^R \mathbb{Z}_q^*$. Afterwards, we compute the Sphinx group elements $\alpha_i$, the shared secrets $s_i$ and the blinding factors $b_i$ $(0 \leq i < \upsilon)$ for each mix node along the path.

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

We then compute the filler strings $\phi_i$, which are used to ensure that the sender computes the correct MAC for each node. This is done via an iterative algorithm adapted from Sphinx:

- $\phi_0 = \varepsilon$
- For $0 < i < \upsilon$:

$$\phi_i = (\phi_{i-1} \mathbin{+\!\!+} 0_{8+3\kappa}) \oplus \Big(\rho\left(h_\rho\left(s_{i-1}\right)\right)_{[\text{SIZE}_\lambda - (i-1)\cdot(8+3\kappa)..\text{SIZE}_\lambda + 8 + 3\kappa - 1]}\Big)$$

We define the right flag to use for the header depending on the level that we're creating a header for:

$$
\begin{aligned}
f_0 &= f_{\text{Destination}} \\
f_x &= f_{\text{Multicast}} \text{ for } x > 0
\end{aligned}
$$

Lastly, we can build the remaining header elements $\beta_{\upsilon-1}, \ldots, \beta_0$ and $\gamma_{\upsilon-1}, \ldots, \gamma_0$:

$$\beta_{\upsilon-1} = \Big( (f_\lambda + \!\!\!+ \Delta) \oplus \rho \left( h_\rho \left( s_{\upsilon-1} \right) \right)_{[0..8+|\Delta|]} \Big) + \!\!\!+ \phi_{\upsilon-1}$$

For $0 \leq i < \upsilon - 1$:

$$\beta_i = \Big( f_{\text{Relay}} + \!\!\!+ n_{i+1} + \!\!\!+ \gamma_{i+1} + \!\!\!+ \sigma_i + \!\!\!+ \beta_{i+1[0..\text{SIZE}_\lambda - 3\kappa - 1]} \Big) \oplus \rho(h_\rho(s_i))_{[0..\text{SIZE}_\lambda - 1]}$$

For $0 \leq i \leq \upsilon - 1$:

$$\gamma_i = \mu \left( h_\mu \left( s_i \right), \beta_i \right)$$

We can then use the header $h_0 = (\alpha_0, \beta_0, \gamma_0)$ to send the message to the first hop.

## 6.4 Creating a Message

In order to create a message to a number of recipients $\{r_1, \ldots, r_n\}$, the sender first constructs a *multiplication schedule*. That is, the sender determines at which point in the path — that is, after how many mix nodes — the message should be multiplied, and how many nodes after the multiplication the message should pass.

The sender then picks a random seed $\mathcal{S}$ to generate the key tree $\mathcal{K}$. Additionally, the sender randomly picks mix nodes in a way that every node in the key tree that is used can be identified with a chosen mix node, therefore mapping the planned multiplication schedule to a choice of actual mix nodes. Now, the sender can generate the header for the message from the bottom up — meaning we first generate the lowest-level headers, then bundle them up into a higher-level header.

In order to generate the innermost headers, for each recipient $r_i$, the corresponding path from the last multiplication node to the recipient $n_\mu, \ldots, n_l$ and the corresponding choices in the key tree $K_i = \{k_{i,\mu}, \ldots, k_{i,l}\}$, the sender computes the header as $h_i = \textsc{CreatePolyHeader}_0(\Delta, N)$, where

- The information for the final node $\Delta = \mathcal{S} + \!\!\!+ K_i + \!\!\!+ r_i$ contains the key tree seed, the path that was taken and the recipient.
- The sequence of mix nodes $\mathbf{N} = \{(n_{\mu+1}, H(\mathcal{K}[\ldots; k_{i,\mu}])), \ldots, (n_l, H(\mathcal{K}[\ldots; k_{i,l}]))\})$ describes the path of the message from the multiplication node to the recipient.

**Remark.** *In our construction, the exit node learns the address of the recipient and the information necessary to unwrap the message. This is in line with the original Sphinx approach, where the exit node is responsible for forwarding the plaintext message to the recipient [11, p. 6].*

*It is easy however to modify the protocol to work differently, by encrypting the decryption information with the recipient's public key and letting the exit node simply forward this information. This places less trust in the exit node at the cost of a slightly more complex unwrapping.*

Once the sender has generated a level of headers, it can iteratively construct the higher-level headers by combining $p$ level $\lambda - 1$ headers into a single level $\lambda$ header. In order to do so, the sender groups the messages that share a common prefix into a group to then bundle them up. We are considering the path a message takes from one multiplication node $n_\mu$ to the next multiplication node $n_{\mu'}$, again with the respective path in the key tree $k_\mu, \dots, k_{\mu'}$. For every group, the sender can then compute the combined header as $h = \textsc{CreatePolyHeader}_\lambda(\Delta, N)$, where

- We denote $H_i$ as the headers of level $\lambda - 1$ in the current group, and $n'_i$ denote the addresses of the next hops for each header.
- We define $\Delta_i = n'_i \mathbin{+\!\!+} H(\mathcal{K}[\dots; k_i]) \mathbin{+\!\!+} H_i$ as the packed header, together with the next mix hop and the encryption key.
- We define the complete header payload $\Delta = \Delta_1 \mathbin{+\!\!+} \Delta_2 \mathbin{+\!\!+} \dots \mathbin{+\!\!+} \Delta_p$ as all inner headers combined.

An exemplary scenario for a multicast factor of two is given in Figure 6.2. In this example, the sender wants to send a message to two recipients. It decides to include a mix node before and after the multiplication node, giving us a total path length of three. We can see that the messages share a prefix in the key tree, up to $\mathcal{K}[1]$ — that is the key that is used by the mix node in front of the multiplication node. The multiplication node itself will be given keys derived from $\mathcal{K}[1;1]$ and $\mathcal{K}[1;2]$ in order to generate two unlinkable copies of the message. Finally, each message passes another mix node before being delivered to the recipient.

After arriving, there are two messages that can be decrypted to the same plaintext. One has encryption layers with the keys $\{\mathcal{K}[], \mathcal{K}[1], \mathcal{K}[1;1], \mathcal{K}[1;1;1]\}$, the other has encryption layers $\{\mathcal{K}[], \mathcal{K}[1], \mathcal{K}[1;2], \mathcal{K}[1;2;1]\}$.

## 6.5 Message Processing at Mix Nodes

When a mix node $n$ with private key $x_n$ receives a message $((\alpha, \beta, \gamma), \xi)$, it starts by computing the shared secret $s = \alpha^{x_n}$ and the MAC $\gamma' = \mu(h_\mu(s), \beta)$. If $\gamma \neq \gamma'$, the node can discard the message as the header has been tampered with.

Afterwards, the node can decrypt the content of the routing information and extract the flag byte $f$:

$$
\begin{aligned}
B &= (\beta \mathbin{+\!\!+} \mathtt{0}_{8+3\kappa}) \oplus \rho\left(h_\rho\left(s\right)\right)_{[0..\textsc{size}_\lambda+7+3\kappa]} \\
f &= B_{[0..7]}
\end{aligned}
$$

The further processing now depends on the value of the flag.

- If $f = f_{\text{Relay}}$, the node should simply relay the message. In this case, it needs to extract the information about the next hop as well as the encryption key from the routing information:

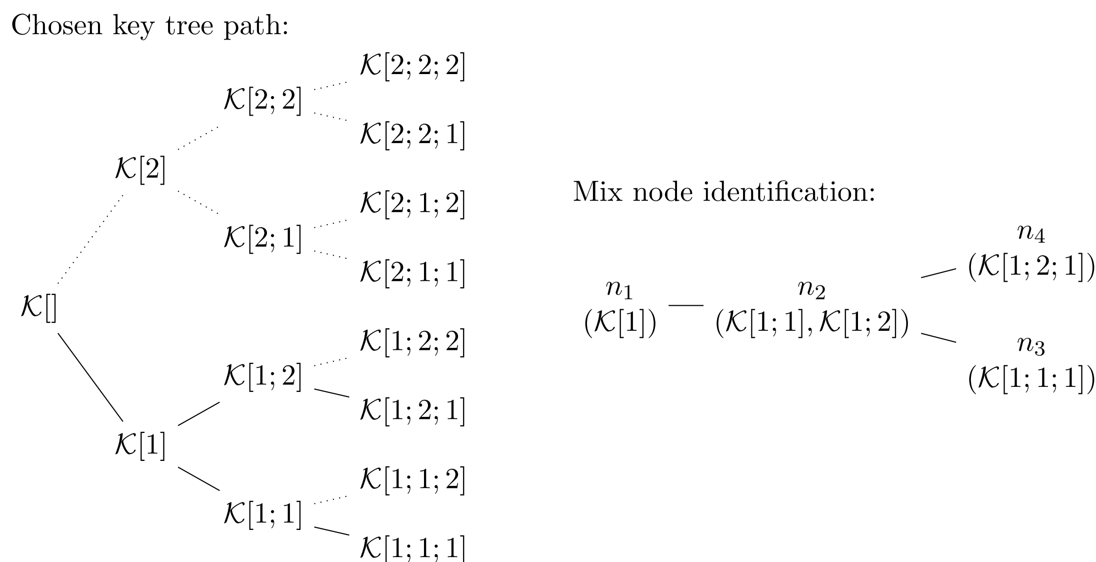


Figure 6.2: Exemplary path through a key tree for a multicast factor of $p = 2$.

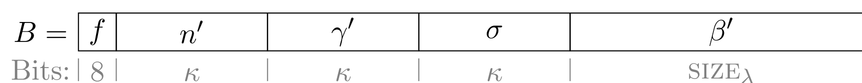


Figure 6.3: The layout of $B$ visualised.

$$
\begin{aligned}
n' &= B_{[8..7+\kappa]} \\
\gamma' &= B_{[8+\kappa..7+2\kappa]} \\
\sigma &= B_{[8+2\kappa..7+3\kappa]} \\
\beta' &= B_{[8+3\kappa..7+3\kappa+\text{SIZE}_\lambda]} \\
b &= h_b(\alpha, s) \\
\alpha' &= \alpha^b
\end{aligned}
$$

This structure is also shown in Figure 6.3.

Now, $n'$ is the address of the next hop, $(\alpha', \beta', \gamma')$ is the new header and $\sigma$ is the encryption key that $n$ should apply to the payload before sending it.

Therefore, the node sends $((\alpha', \beta', \gamma'), \textsc{Enc}(\sigma, \xi))$ to $n'$.

- If $f = f_{\text{Destination}}$, the message has arrived at the exit node. The node must now extract the recipients address and unwrap the onion encryption layers:

$$\mathcal{S} = B_{[8..7+\kappa]}$$
$$K = B_{[8+\kappa..7+\kappa+r\lceil\log_2 p\rceil]}$$
$$\Delta = B_{[8+\kappa+r\lceil\log 2p\rceil..7+4\kappa+r\lceil\log_2 p\rceil]}$$

Now the recipient's address is available in $\Delta$. Furthermore, given the key tree seed $\mathcal{S}$ as well as the path $K$ through the key tree, we can recover the plaintext by applying DEC in reverse order.

The node can therefore deliver the plaintext to the recipient.

- If $f = f_{\text{Multicast}}$, the node is the designated multicast node and should send $p$ copies of the message. In order to do so, the node first extracts the embedded subheaders:

$$\Sigma = 3\kappa + \text{SIZE}_\lambda$$

For $0 \leq i < p$:

$$n_i = B_{[i\Sigma..i\Sigma+\kappa-1]}$$
$$\sigma_i = B_{[i\Sigma+\kappa..i\Sigma+2\kappa-1]}$$
$$\alpha_i = B_{[i\Sigma+2\kappa..i\Sigma+4\kappa-1]}$$
$$\gamma_i = B_{[i\Sigma+4\kappa..i\Sigma+5\kappa-1]}$$
$$\beta_i = B_{[i\Sigma+5\kappa..i\Sigma+5\kappa+\text{SIZE}_{\lambda-1}-1]}$$

This structure is also shown in Figure 6.4.

The node can now send $p$ messages: For $i = 0$ up to $p-1$, it sends $((\alpha_i, \beta_i, \gamma_i), \text{ENC}(\sigma_i, \xi))$ to $n_i$.

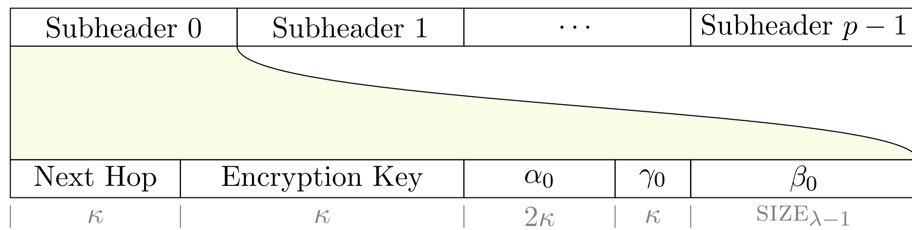


Figure 6.4: The embedded subheader structure. The grey numbers refer to the size of the element in bits.

# 7 Security

According to Camenisch and Lysyanskaya, an onion routing scheme is considered secure when it satisfies correctness, integrity and security [6, p. 179]. Intuitively, those terms can be described as follows:

- Correctness means that a well-formed message is processed correctly, in the right order of the nodes.
- Integrity means that the length of the path of a message cannot exceed a fixed upper bound.
- Security means that an adversary cannot distinguish between different messages, even if it knows the path and content of a message. It must also be impossible to wrap the message by adding another hop.

Since we do not modify the routing part of Sphinx, we refer to the proof of correctness, integrity and wrap-resistance of the original format [11, Section 4]. In order to prove that PolySphinx is also secure, we will prove that the indistinguishability of messages still holds in our construction.

We do so in two parts: First, we look at a message on a given level $\lambda$, meaning that an adversary cannot link incoming and outgoing messages at an honest mix node. Afterwards, we prove that the multiplication node is also secure, in the sense that an adversary cannot link outgoing to incoming messages, and it cannot link two messages that originate from the same incoming message.

Those two securities are enough to provide indistinguishability: Either a mix node acts as a relay, in which case the intra-level indistinguishability completes the security requirement. Alternatively, if the mix node acts as the multiplication node, the inter-level indistinguishability completes the security requirement.

## 7.1 Intra-Level Indistinguishability

We first define the game for the first step of our security proof, similarly to that of other classic indistinguishability notions:

**Definition 9** (PolySphinx game)**.** *The adversary selects two paths of mix nodes, $N_0$ and $N_1$, two final node information values $\Delta_0$ and $\Delta_1$ as well as two corresponding payloads $m_0$ and $m_1$. The challenger now chooses a random bit $b$ and creates a PolySphinx message with path $N_b$ and payload $m_b$ as described in Section 6.4. The adversary is given the created message and should determine the value of $b$.*

*The adversary may choose the secret keys of all but the first node in the path, and the adversary has access to an oracle that simulates the message processing at the mix node — however, the adversary may not submit the challenge message to the oracle.*

We can see why the game models what we would intuitively describe as "security": The adversary is given the point of view of a message before the processing of a mix node in the form of correctly created messages and headers. Now, even if the adversary can choose the path, the content and even the secret keys of the mix nodes that come after, it should be impossible to tell which of the message has been wrapped.

Before we go on to prove the security of PolySphinx, we will note a few assumptions that we will base our proof on:

**Assumption 1.** *The underlying encryption scheme $\pi$ is IND-CPA secure.*

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

In order to show that our PolySphinx scheme is secure, we will make a hybrid argument with five games. The first two transformations show that PolySphinx is not less secure than Sphinx, while the last two transformations are the adapted equivalent from the original Sphinx security proof [11, Section 4.4].

We now give a series of games, each with a slight modification compared to the previous one:

**Hybrid** (PolySphinx)**.** *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.*

**Hybrid** (Sphinx)**.** *In game $H_2$, 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.*

**Hybrid.** *In game $H_3$, the challenger changes the way the shared secret is computed and chooses $s_0 \leftarrow^R \mathcal{G}^*$ randomly instead of using $s_0 = y_{n_0}^x$. This is equivalent to $\mathbf{G}_1$ in the Sphinx proof [11, p. 10].*

**Hybrid** (Ideal World)**.** *In game $H_4$, the challenger replaces the header values $\beta$ and $\gamma$ with random values. This is equivalent to $\mathbf{G}_2$ in the Sphinx proof [11, p. 10].*

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

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

*Proof.* The key that the sender embeds for the mix node is the hash of a key tree node, $H(\mathcal{K}[\ldots;x])$. As per our definition of a hash function as a random oracle, this value is randomly chosen. The only way for the node to distinguish between $H_1$, in which a truly random key is embedded, and $H_0$, in which the output of $H$ is embedded, would be to know the input value $\mathcal{K}[\ldots;x]$.

The value $\mathcal{K}[\ldots;x]$ itself however is either the output of a random oracle $H(\mathcal{K}[\ldots]+x)$, or the randomly chosen initial seed — neither of which values is embedded into the message. Since 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 only difference between $H_2$ and $H_1$ is the fact that in $H_2$, 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 attacker $\mathcal{D}_{\text{IND}}$:

- $\mathcal{D}_{\text{IND}}$ calls $\mathcal{D}$ to get the message $m_0$.
- $\mathcal{D}_{\text{IND}}$ draws a message $m_1$ randomly.
- $\mathcal{D}_{\text{IND}}$ forwards $m_0$ and $m_1$ to the IND-challenger and receives the challenge $c$.
- $\mathcal{D}_{\text{IND}}$ forwards $c$ to $\mathcal{D}$ and receives the guess $b$.
- $\mathcal{D}_{\text{IND}}$ forwards the guess $b$ to the IND-challenger.

As per Assumption 1 however, no such distinguisher $\mathcal{D}_{\text{IND}}$ can exist, and therefore the two games are indistinguishable. □

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

*Proof.* The difference between $H_3$ and $H_2$ is the way that the shared secret is calculated. In $H_2$, we have $s_0 = y_{n_0}^x$, whereas in $H_3$ we have $s_0 \leftarrow^R \mathcal{G}^*$.

If a distinguisher $\mathcal{D}$ exists that could differentiate between those two games, we could use it to construct a DDH distinguisher $\mathcal{D}'$:

- $\mathcal{D}'$ receives the challenge tuple $(a, b, c)$.
- $\mathcal{D}'$ constructs a PolySphinx message using $a$ as the public key of the first mix node ($y_{n_0} = a$), $b$ as the first Sphinx group element ($\alpha_0 = b$) and $c$ as the first shared secret ($s_0 = c$).
- $\mathcal{D}'$ now passes the constructed message to $\mathcal{D}$ and receives the response bit $b$.
- $\mathcal{D}'$ passes $b$ to its challenger.

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

We note that the adversary cannot use the mix node oracle to distinguish between the games: It cannot send the challenge message to the mix node, as that is forbidden by the game definition. It can also not construct a different message $(\alpha, \beta', \gamma')$, as that would require it to forge a valid mac with the mac key $h_\mu(\alpha^{x_{n_0}})$. Since $h_\mu$ is a random oracle, the adversary must know the value of $\alpha^{x_{n_0}}$. In this case, however, the adversary does not gain any advantage from the oracle, as it would possess the shared secret and could simply decrypt the header [11, p. 10]. □

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

*Proof.* Per definition, $\beta$ is combined with the output of a pseudorandom generator $\rho$ and as such indistinguishable from randomness. Likewise, $\mu$ is the output of a random oracle with a random input key and again indistinguishable from randomness. □

We now show that the adversary does not have an advantage in the "ideal world":

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

*Proof.* No value in the header or payload of the message in $H_4$ is dependent on the challenge bit $b$. Therefore, no adversary can have an advantage over random guessing [11, p. 10]. □

We can therefore conclude this section with our final theorem:

**Theorem 1** (PolySphinx is secure)**.** *PolySphinx fulfils the definition of security from Definition 9 under Assumption 1 and Assumption 2.*

*Proof.* The proof follows from Lemma 1, Lemma 2, Lemma 3, Lemma 4 and Lemma 5: As the adversary has no advantage in $H_4$, and each game is indistinguishable from the one before except for a negligible probability, the adversary overall has at most a negligible probability to win the original game $H_0$. □

**Corollary 1.** *The intra-level security of PolySphinx messages holds for any two messages of the same level $\lambda$.*

*Proof.* The structure of PolySphinx headers of different levels only differs in the value and length of $\Delta$. Since Theorem 1 holds for arbitrary values of $\Delta_0$ and $\Delta_1$, it also holds for $\Delta_0$ and $\Delta_1$ that represent a lower-level PolySphinx header, or a recipient's address. □

## 7.2 Inter-Level Indistinguishability

We can use Theorem 1 to argue about the security of PolySphinx messages on one level. In the context of multiplication nodes, this means we can use it to argue about the path before a multiplication node, or for the path after the multiplication node. However, the

multiplication node itself needs special treatment: It is the piece that brings a level $\lambda$ message to a level $\lambda - 1$ message by unwrapping the inner headers and multicasting it.

In terms of security, we want to ensure two things: First, it should not be possible to link an outgoing message back to the incoming message that contained it. Secondly, it should not be possible to link two outgoing messages that originate from the same incoming message.

In order to capture the first goal, we define the following game:

**Definition 10** (Multicast-Indistinguishability game)**.** *The adversary selects two paths of mix nodes, $N_0$ and $N_1$, two final node information values $\Delta_0$ and $\Delta_1$ as well as two corresponding payloads $m_0$ and $m_1$. The challenger now chooses a random bit $b$ and creates a* higher-level *PolySphinx message with inner path $N_b$ and payload $m_b$ as described in Section 6.4. The adversary is given the created message and should determine the value of $b$.*

*The adversary may choose the secret keys of all selected mix nodes in $N_0$ and $N_1$, but it may not know the secret key of the multiplication node. The adversary has access to an oracle that simulates the message processing at the mix node — however, the adversary may not submit the challenge message to the oracle.*

This game captures the multiplication node explicitly: We're giving the adversary access to the message just before multiplication, and (by virtue of the adversary choosing the paths) access to the resulting multicast messages. We treat the multiplication node as honest, as the adversary does not have access to its secret key. If the adversary cannot determine which inner path is wrapped, it cannot trace the outgoing message back to the incoming message.

We will prove that our construction is secure in the sense that no adversary has a non-negligible advantage at winning the Multicast-Indistinguishability game:

**Theorem 2.** *No PPT adversary has a non-negligible advantage in the Multicast-Indistinguishability game from Definition 10.*

*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 PolySphinx game:

- $\mathcal{A}'$ calls $\mathcal{A}$ to get the choices of $N_0, N_1, \Delta_0, \Delta_1, m_0$ and $m_1$.
- $\mathcal{A}'$ creates $\Delta_0'$ and $\Delta_1'$ as inner headers, as described in Section 6.4.
- $\mathcal{A}'$ chooses a random mix node $n$ and passes $n_0 = n_1 = n$, $\Delta_0'$, $\Delta_1'$, $m_0$ and $m_1$ to its challenger to receive the challenge message $c$.
- Now, $c$ has the same structure as a multicast message in the Multicast-Indistinguishability game, so $\mathcal{A}'$ forwards $c$ to $\mathcal{A}$ to get the guess $b$.
- $\mathcal{A}'$ forwards $b$ to the challenger.

□

In addition to the attacker not being able to trace the message through the honest multiplication node, we also want to ensure that the attacker cannot tell if two messages originate from the same incoming message. To capture this requirement, we define a second game:

**Definition 11** (Multicast-Unlinkability game)**.** *The adversary selects two paths of mix nodes, $N_0$ and $N_1$, two final node information values $\Delta_0$ and $\Delta_1$ 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 a higher-level PolySphinx message with inner messages $(N_0, \Delta_0, m)$ and $(N_1, \Delta_1, m)$. The challenger then processes this message and returns the two resulting messages $(m'_0, m'_1)$ to the adversary.*
- *If $b = 1$, the challenger creates two separate messages with $(N_0, \Delta_0, m)$ and $(N_1, \Delta_1, m)$ 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.** *No PPT adversary has a non-negligible advantage to win the Multicast-Unlinkability game from Definition 11.*

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

# 8 Implementation

So far, we have talked about PolySphinx in an abstract sense. In this chapter, we want to give concrete choices for our primitives and explain why we have chosen them, as well as introduce our implementation of PolySphinx in Rust.

## 8.1 Chosen Primitives

As our Diffie-Hellman group $\mathcal{G}$ we have chosen the Curve25519 elliptic curve, which was proposed by D. J. Bernstein [3]. This decision matches what the original Sphinx reference implementation `sphinxmix`[1] uses. Curve25519 has the benefit of having compact keys while being fast to evaluate.

For our pseudorandom function $\pi$, we chose to use AES in CTR mode. AES is fast to evaluate, especially on modern machines with hardware support for AES instructions. Furthermore, it is widely used and has implementations in many programming languages [26]. The counter mode was chosen because it allows for a constant-sized ciphertext and good parallelisation.

For our hash functions and random oracles, we have chosen to use truncated SHA256 hashes. This matches the suggestion in the original Sphinx construction [11, p. 4]. Similarly to AES, the SHA256 hash function is widely implemented, supported and used in practice. In order to generate different instances of the hash functions for the different places in which they are used, we have *salted* the SHA256 input with a different byte for each different use case. This ensures that the instances have different output values, even if their input is the same.

The mac $\mu$ is implemented as an HMAC with SHA256 as the underlying hash function. HMAC provides a standard way to define a hash-based message authentication code whose security is based on the underlying hash function [19]. This again matches the design choice in the reference Sphinx implementation and uses a tool that is widely available and used in practice.

For our pseudorandom generator $\rho$, we have chosen ChaCha [2], as it is performant, widely used and available in many programming languages [16].

## 8.2 Rust Prototype

We have implemented a prototype of PolySphinx in Rust[2]. The strong type system and memory safety of Rust are advantageous for security-related programs, as they prevent

[1] `https://github.com/UCL-InfoSec/sphinx`
[2] `https://www.rust-lang.org/`

common pitfalls such as buffer overflows or use-after-free errors that may arise in other languages.

In the following sections, we will introduce our prototype in more detail and talk about the API and benchmark programs that we have written (see also Chapter 11).

### 8.2.1 External Libraries

In order to use the primitives that we have chosen in the previous section, we rely on a number of third-party libraries (so-called "crates") to provide an implementation:

- Curve25519: `curve25519-dalek`
- AES-CTR: `aes` , `ctr`
- SHA256: `sha2`
- HMAC: `hmac`
- ChaCha: `rand_chacha`

### 8.2.2 Prototype API

We have chosen to implement our prototype in two layers: The first layer implements the PolySphinx mechanism of creating the header and embedding the re-encryption keys. The second layer implements the actual generation of the re-encryption keys and the payload re-encryption functionality. Communication between the layers is done via a generic interface ("trait"), such that the first layer can use the second layer's functionality without being hardwired to a specific implementation.

This separation allows us to implement different PolySphinx variants (see Chapter 10) while keeping the overall mechanism and interface the same. In addition, we can benchmark and compare the various implementations (see Section 11.1), without the need to re-implement the common parts of the protocol.

The first layer that implements the PolySphinx API consists of three main functions:

```
fn create_header(header_payload: Bits, nodes: &[&MixNode], re_keys:
↪  &[ReencryptionKey])
fn create_polyheader(path: &Path)
fn unwrap_header(node: &MixNode, header: &Header)
```

The first function, `create_header`, is the equivalent of CreateSphinxHeader. It is a low-level function that takes in the final routing information ($\Delta$ in Sphinx), the path of mix nodes, and the re-encryption keys, and it outputs the created header together with the randomly selected Sphinx group element.

The second function, `create_polyheader`, is the high-level function to create a (possibly nested) header. It takes as input the path — which may contain multiplication

points — and draws the re-encryption keys using the interface to the second layer. Internally, it uses `create_header` to create the necessary (sub-)headers and returns the final header together with the first encryption key.

The final function, `unwrap_header`, is the counterpart to UnwrapSphinxHeader. It uses the node's private key to unwrap the received header and returns the information necessary to further process the message.

The interface to the second layer contains four main functions:

```
trait Polyfication {
  fn prepare(path: &Path)
  fn encrypt(key: &EncryptionKey, data: &[u8])
  fn decrypt(key: &DecryptionKey, data: &[u8])
  fn reencrypt(key: &ReencryptionKey, data: &[u8])
}
```

The first two functions are used by the sender: The `prepare` function takes is used by `create_polyheader` to prepare the re-encryption keys, while `encrypt` takes the initial encryption key and payload to produce the first ciphertext.

The `decrypt` function is used by the exit node to recover the plaintext from the final ciphertext. The decryption key is returned by `unwrap_header` when the final destination is reached.

Finally, `reencrypt` is used by the mix nodes to perform the per-node re-encryption of the ciphertext. It is given the re-encryption key that is embedded in the header and the current ciphertext, and it returns the new ciphertext.

### 8.2.3 Demonstration Program

Our prototype contains a small demonstration program that internally creates several mix nodes and allows the user to "send" a multicast message over a randomly selected path.

This demonstration program not only shows how the PolySphinx API can be used in practice, but it also allows the user to explore different parameter choices and implementations locally, without the need to set up a proper mix network.

The demonstration program also contains implementations of the PolySphinx alternatives from Chapter 10.

# 9 Protocol Extensions

Due to the modifications that we make to the Sphinx format, we lose some of its anonymity guarantees. In particular, since the recipient can now reconstruct all intermediate keys that the mix nodes have used, it can trace back the message to the sender and to the other recipients of the message.

In the following sections, we want to give some possible extensions to the PolySphinx format that restore some of the lost anonymity. We will explain the possible extensions and their reasoning on a high level, without going into the bit-level details.

## 9.1 Sender-Anonymous Direct Messages

The original Sphinx format guarantees sender anonymity, meaning that the recipient cannot determine the sender of a message based on the message format alone. PolySphinx does not guarantee that, as the recipients can reconstruct the keys that have been used by the mix node, and therefore trace the messages through the mix node path.

While this is a necessary trade-off in our multicast design, it is not necessary to impose this restriction on single-recipient messages. We, therefore, propose an extension to PolySphinx that retains the ability to send sender-anonymous messages to single recipients:

Instead of sending the message in plain and letting it be encrypted along the way, the sender simply "wraps" the message in the correct encryption layers before sending, such that it gets decrypted along the way. This is similar to how Sphinx works, with the difference that the sender supplies the keys in the header explicitly in PolySphinx. For the recipient, a simple sentinel value (such as a null seed for the key tree) can be used to indicate that the message needs no further decryption.

With this method, the recipient can no longer retrace the message, as it lacks the keys used by the mix nodes, whereas the processing at the mix nodes doesn't need to change.

## 9.2 Improved Anonymity for Other Recipients

In the basic PolySphinx approach, a recipient can derive the keys for any path on the key tree — not just its own. Therefore, it is possible for a recipient to trace messages to other recipients as well. While this is necessary for at least parts of the path (the part that the messages share), it gives unnecessary power to a single recipient.

We can limit the information that a single recipient gets by having two different seeds embedded into the final message: The first one is for the key tree, shared amongst

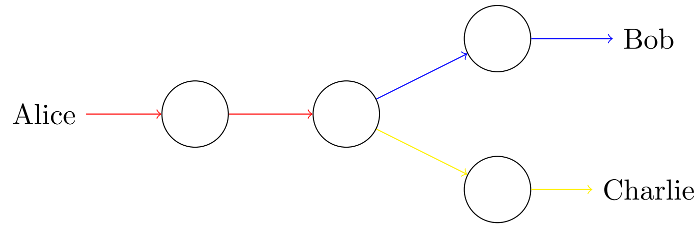


Figure 9.1: Separate seeds visualised. Only the red part is shared, while the blue and yellow parts are separate.

all recipients. However, once the message is split up, we make use of a separate per-recipient value which serves as the seed for the “last mile”, that is the path between the last multiplication node and the recipient.

With this extension, a recipient can no longer trace messages to every other recipient but instead is limited to tracing the message to the multiplication node. The overhead is small, as we only need $\kappa$ bits for a second seed value.

An example of this extension is shown in Figure 9.1. Here, only the red part before the multiplication node is shared, while the blue and yellow parts are separate for each recipient. Bob would get the seed values for the red and blue parts, while Charlie would get the seed values for the red and yellow parts. In this case, it would not be possible for Bob to trace the connection to Charlie or vice versa.

# 10 Alternative Approaches

One of the main questions in the design of PolySphinx was how multiple, unlinkable copies of a ciphertext should be generated. Ideally, we wanted to find an encryption scheme that allows keys to be “combined” in a homomorphic way. That is, given keys $k_1$, $k_2$, the encryption function $E$ and the decryption function $D$, we wanted a scheme in which there is a function $f$ that can generate a combined key:

$$D(f(k_1, k_2), E(k_2, E(k_1, m))) = m$$

Such a scheme would allow us to send the keys $k_1$, $k_2$, ... to the mix nodes, and then send the combined key to the recipient. If the message takes different paths after the multiplication, each recipient will simply get a different combined key.

A basic scheme fitting our requirement is trivial to construct, for example, $f(k_1, k_2) = k_1 \mathbin{+\!\!+} k_2$ could be the concatenation of two keys, and $D$ could then read multiple keys.

However, this scheme would only meet the basic requirement described above. We can think of a few more desirable properties:

**Compactness** The keys, especially the combined key, should be compact.

**Length Idempotency** The combined key should not reveal how many keys were combined. Ideally, the output of $f$ and a fresh key should be indistinguishable.

**Key Unlinkability** Given three keys $k_1$, $k_2$, $k_3$ and $f(k_1, k_2)$, it should not be possible to distinguish $k_2$ and $k_3$ given just $k_1$ and $f(k_1, k_2)$.

Especially the length idempotency and the key unlinkability aspects are important if we want to support sender-anonymous multicast messages, as those are the aspects that prevent the recipient (who has the combined key) to learn information about the previous mixes.

In the following sections, we will discuss some primitives that we had found as alternatives to the key-tree based re-encryption approach.

## 10.1 Key-Homomorphic Pseudorandom Functions

A key-homomorphic pseudorandom function (KHPRF) is a pseudorandom function $F$ in which $F(k_1, m) + F(k_2, m) = F(k_1 + k_2, m)$ [5]. We can see that such a KHPRF cannot be used directly to encrypt our payload, as each encryption is done on the plaintext instead of the previous ciphertext. However, Boneh *et al.* give a way to use KHPRFs that fits our use-case [5]:

By encrypting a nonce instead of the message, and then using the encrypted nonce as a "one-time pad", we can combine keys without the need to know the plaintext:

$$m + F(k_1, 0) + F(k_2, 0) = m + F(k_1 + k_2, 0)$$

The decryption can then be done by subtracting $F(k_1 + k_2, 0)$ again.

### 10.1.1 The Learning-With-Errors Problem

Boneh *et al.* also give a way of constructing KHPRFs based on the learning-with-errors (LWE) problem. The core idea behind LWE as introduced by Regev is that it is "hard" to solve a system of linear equations when those equations have a random error added to them [24]. More formally, we can define the *search LWE* problem as follows:

**Definition 12** (Search-LWE). *Given a prime $t$, a matrix of coefficients $\mathbf{A} \in \mathbb{Z}_t^{m \times n}$, a secret vector $\mathbf{s} \in \mathbb{Z}_t^n$, and an error vector $\mathbf{e} \in \mathbb{Z}_t^m$, the LWE problem asks to recover $\mathbf{s}$ given just $\mathbf{b} := \mathbf{A}\mathbf{s} + \mathbf{e}$.*

We can see that this is equivalent to solving a list of "equations with error", as Regev calls it:

$$\begin{aligned} \mathbf{a}_1\mathbf{s} + e_1 &= b_1 \mod t \\ \mathbf{a}_2\mathbf{s} + e_2 &= b_2 \mod t \\ \mathbf{a}_3\mathbf{s} + e_3 &= b_3 \mod t \\ &\vdots \end{aligned}$$

Regev goes on to show that for certain parameter choices, there exists a reduction from LWE to GapSVP, implying that LWE cannot be solved efficiently for those parameters [24].

Intuitively speaking, what makes LWE interesting as the base for cryptographic applications is the algebraic structure it shows: If we have two LWE instances $\mathbf{b}_1$ and $\mathbf{b}_2$, we can see that $\mathbf{b}_1 + \mathbf{b}_2 = \mathbf{A}(\mathbf{s}_1 + \mathbf{s}_2) + \mathbf{e}'$, where $\mathbf{e}'$ is the combined error term. As long as the accumulated error doesn't get too big, we can keep combining ciphertexts homomorphically.

### 10.1.2 Constructions of Key-Homomorphic Pseudorandom Functions

The construction of the KHPRF is parametrised by the integers $q$, $p$, $n$ and $m$ with $m = n\lceil \log q \rceil$ and $p$ dividing $q$. It then uses two random matrices $\mathbf{A}_0, \mathbf{A}_1 \in \mathbb{Z}_q^{m \times m}$ where each row is sampled from a binary distribution, those matrices are known to all parties as public parameters. The keys are vectors in $\mathbb{Z}_q^m$. The actual function is then defined as:

**Definition 13** (Boneh *et al.*'s KHPRF [5, p. 14])**.**

$$F(\mathbf{k}, x) = \left\lfloor \prod_{i=1}^{l} \mathbf{A}_{x_i} \cdot \mathbf{k} \right\rfloor_p$$

The "modular rounding" $\lfloor x \rceil_p$ is defined to be the integer $i$ such that $i \cdot \lfloor \frac{q}{p} \rfloor$ does not exceed $x$.

Boneh *et al.* show that this function is pseudorandom for certain parameter choices [5, Theorem 5.1], based on the LWE problem. It is also *almost* key-homomorphic [5, p. 18], meaning that it is key-homomorphic with a growing error term, and the homomorphism only holds as long as the error is below the threshold.

On the downside of this construction are the large parameters that are required for the function to be pseudorandom. To improve on that, Banerjee and Peikert generalize the construction and provide a KHPRF that works with smaller parameters [1]. Their definition, in addition to the parameters from above, uses matrices $\mathbf{A}_0, \mathbf{A}_1 \in \mathbb{Z}_q^{n \times nl}$ and a binary tree $T$, which are also considered public parameters. The construction then works as follows:

**Definition 14** (Banerjee and Peikert's KHPRF [1, p. 4])**.** *First, define $A_T : \{0,1\}^{|T|} \to \mathbb{Z}_q^{n \times nl}$ recursively:*

$$A_T(x) = \begin{cases} \mathbf{A}_x & \textit{if } |x| = 1 \\ A_{T.l}(x_l) \cdot G^{-1}(A_{T.r}(x_r)) & \textit{otherwise} \end{cases}$$

*In the second case, parse $x = x_l \mathbin{+\!\!+} c_r$ with $x_l \in \{0,1\}^{|T.l|}$ and $x_r \in \{0,1\}^{|T.r|}$, and $G^{-1}$ is a binary decomposition of the input matrix.*

*Then, define the KHPRF $F$ as follows:*

$$F(\mathbf{k}, x) = \lfloor \mathbf{k}^{\intercal} \cdot A_T(x) \rceil_p$$

While Banerjee and Peikert's function does use smaller parameters, it also does require more computational power to evaluate, as the recursive evaluation of $A_T$ requires a lot of matrix multiplications.

### 10.1.3 Evaluation in PolySphinx

We have implemented the KHPRF constructed by Banerjee and Peikert, as well as a PolySphinx variant that uses the KHPRF to multicast messages. The benchmarks done with this PolySphinx variant are labelled `PolySphinx/LWE`, and the results are shown in Chapter 11. We can see that a major downside of this PolySphinx variant is the huge performance impact, as the KHPRF relies heavily on slow matrix multiplications.

Another downside in practice is the choice of the parameters, as those dictate how many homomorphic transformations are possible before the accumulated error is too big. This increases the size of the message space, and therefore requires more bits to store the message — which increases the space overhead.

Lastly, it was unclear whether the construction with KHPRFs would be "secure" enough in the sense that neither the ciphertexts nor the re-encryption keys could be linked to each other. The schemes proposed by Boneh *et al.* in the original paper were questioned in their ability to fulfil the security notions for "unlinkability" [20, 18].

## 10.2 Updatable Encryption

An *updatable encryption* (UE) scheme is a scheme in which ciphertexts that were encrypted with one key $s_1$ can be transformed to be decrypted by a different key $s_2$, without decrypting and re-encrypting the ciphertext in the process. This is achieved by supplying a re-encryption key (usually called an "update token") that is generated from $s_1$ and $s_2$.

Formally, we can define an updatable encryption scheme with five algorithms:

**Definition 15** (Updatable Encryption Scheme [20, p. 692])**.** *An updatable encryption scheme consists of the following algorithms:*

- UE.SETUP *is a probabilistic algorithm that takes as input the security parameter and returns a secret key $k_0$.*
- UE.NEXT *is a probabilistic algorithm that takes a key $k_e$ and returns a new key $k_{e+1}$ together with the re-encryption key $\Delta_{e+1}$.*
- UE.ENC *is the encryption algorithm that takes as input a message $m$ and a key $k_e$ and returns the ciphertext of this message encrypted under the given key.*
- UE.DEC *is the decryption algorithm that takes a ciphertext $C$ and a key $k_e$ and recovers the message $m$.*
- UE.UPD *is the update algorithm that takes a ciphertext $C$ and the re-encryption key $\Delta_{e+1}$ and returns the updated ciphertext.*

In the context of PolySphinx, an updatable encryption scheme can be used to re-encrypt the ciphertext at the mix nodes. The sender would simply generate a random key for every hop between mix nodes and supply the re-encryption keys to the mix nodes. At the multiplication node, different re-encryption keys could be used to generate different versions of the ciphertext, each encrypted with a different key. Finally, the recipient could use the last key to decrypt the ciphertext.

### 10.2.1 Security Notions

The security of updatable encryption schemes can be seen in two parts: First, we don't want the ciphertext to reveal information about the contained plaintext, similarly to the notion of "semantic security". Secondly, we also want to ensure that updated ciphertexts cannot be linked to their "previous" version, a property we call *unlinkability.*

Anja Lehmann and Björn Tackmann define formal notions for both of these concepts which they call IND-ENC and IND-UPD [20]. Their definitions are reiterated here, starting with the oracles that are provided to the adversary:

- The encryption oracle $\mathcal{O}_{\text{enc}}(m)$ returns the ciphertext of the given message encrypted under the current key. The oracle keeps track of which ciphertexts it returned in a set $L$.
- The advancing oracle $\mathcal{O}_{\text{next}}$ generates a new key when triggered and updates the global state to reflect that. If the challenge query has already been made, the challenge ciphertext is updated as well and the returned ciphertext is remembered in a set $\tilde{L}$.
- The updating oracle $\mathcal{O}_{\text{upd}}(C_{e-1})$ updates a ciphertext from the previous key to the current key using the re-encryption algorithm. The oracle ensures that the ciphertext has been honestly generated (meaning that it is part of $L$) and adds the new ciphertext to $L$ as well.
- The corruption oracle $\mathcal{O}_{\text{corrupt}}(\{\text{key}, \text{token}\}, e^*)$ allows the adversary to request a key or update token from an arbitrary epoch. The oracle remembers the corrupted keys in a set $K$ and the corrupted tokens in a set $T$.
- The challenge oracle $\mathcal{O}_{\text{updC}}$ returns the current challenge ciphertext from $\tilde{L}$. The oracle keeps track of the challenge ciphertexts that the adversary requested in $C$.

The security notions are then defined in two games:

**Definition 16** (IND-ENC [20, p. 700])**.** *An updatable encryption scheme is said to be* IND-ENC*-secure if all PPT adversaries have a negligible success chance for Experiment 1.*

**Experiment 1:** IND-ENC

**begin**
- $k_0 \leftarrow \text{UE.SETUP}$
- $e \leftarrow 0$ `/* Epoch counter */`
- $(m_0, m_1, \text{state}) \leftarrow \mathcal{A}^{\mathcal{O}_{enc}, \mathcal{O}_{next}, \mathcal{O}_{upd}, \mathcal{O}_{corrupt}}$
- $\tilde{e} \leftarrow e$ `/* Remember the challenge epoch */`
- $d \leftarrow^R \{0, 1\}$
- $\tilde{C} \leftarrow \text{UE.ENC}(k_{\tilde{e}}, m_d)$
- $d' \leftarrow \mathcal{A}^{\mathcal{O}_{enc}, \mathcal{O}_{next}, \mathcal{O}_{upd}, \mathcal{O}_{corrupt}, \mathcal{O}_{updC}}(\tilde{C}, \text{state})$
- **return** *success* **iff**
  - *$d' = d$, and*
  - *$\mathcal{A}$ did not learn the key in an epoch where it had the challenge ciphertext or a corrupted token allowing it to get the challenge ciphertext*

We can see that IND-ENC resembles the idea of similar indistinguishability notions, adapted to the features that updatable encryption provides. The idea behind the extra conditions for the winning case is to exclude "trivial wins", meaning wins in which the challenger had access to a version of the challenge ciphertext to which it had a fitting corrupted key.

**Definition 17** (IND-UPD [20, p. 701])**.** *An updatable encryption scheme is said to be* IND-UPD*-secure if all PPT adversaries have a negligible success change for Experiment 2.*

---

**Experiment 2:** IND-UPD

---

**begin**

$k_0 \leftarrow \text{UE.SETUP}$

$e \leftarrow 0$ `/* Epoch counter */`

$(C_0, C_1, \text{state}) \leftarrow \mathcal{A}^{\mathcal{O}_{enc}, \mathcal{O}_{next}, \mathcal{O}_{upd}, \mathcal{O}_{corrupt}}$

$\tilde{e} \leftarrow e$ `/* Remember the challenge epoch */`

$d \leftarrow^R \{0, 1\}$

$\tilde{C} \leftarrow \text{UE.UPD}(\Delta_{\tilde{e}}, C_d)$

$d' \leftarrow \mathcal{A}^{\mathcal{O}_{enc}, \mathcal{O}_{next}, \mathcal{O}_{upd}, \mathcal{O}_{corrupt}, \mathcal{O}_{updC}}(\tilde{C}, \text{state})$

**return** *success* **iff**

- $d' = d$*, and*
- $\mathcal{A}$ *has not learned* $\Delta_{\tilde{e}}$*, and*
- $\mathcal{A}$ *did not learn the key in an epoch where it had the challenge ciphertext or a corrupted token allowing it to get the challenge ciphertext, and*
- *if* UE.UPD *is deterministic, then* $\mathcal{A}$ *has neither queried* $\mathcal{O}_{upd}(C_0)$ *nor* $\mathcal{O}_{upd}(C_1)$ *in epoch* $\tilde{e}$

---

The IND-UPD game is similar to IND-ENC, however, now the challenger picks one of the two ciphertexts to update. This is important for PolySphinx, as we must ensure that the ciphertexts from before the mix node and after the mix node are unlinkable.

### 10.2.2 Construction

Lehmann and Tackmann construct an IND-ENC and IND-UPD secure updatable encryption scheme based on ElGamal, called RISE:

**Definition 18** (RISE [20, p. 712])**.** *Let the following algorithms define the RISE updatable encryption scheme:*

- $\text{RISE.SETUP}(\lambda)$*:* $x \leftarrow^R \mathbb{Z}_q^*, k_o := (x, g^x)$*, return* $k_0$
- $\text{RISE.NEXT}(k_e)$*:* $k_e = (x, y)$*,* $x' \leftarrow^R \mathbb{Z}_q^*$*,* $k_{e+1} := (x', g^{x'})$*,* $\Delta_{e+1} := (\frac{x'}{x}, g^{x'})$*, return* $(k_{e+1}, \Delta_{e+1})$

- RISE.$\textsc{enc}(k_e, m)$: $k_e = (x, y)$, $r \leftarrow^R \mathbb{Z}_q$, *return* $C_e := (y^r, g^r m)$
- RISE.$\textsc{dec}(k_e, C_e)$: $k_e = (x, y)$, $C_e = (C_1, C_2)$, *return* $m' := C_2 \cdot C_1^{-x^{-1}}$
- RISE.$\textsc{upd}(\Delta_{e+1}, C_e)$: $\Delta_{e+1} = (\Delta, y')$, $C_e = (C_1, C_2)$, $r' \leftarrow^R \mathbb{Z}_q$, $C_1' := C_1^{\Delta} \cdot y'^{r'}$, $C_2' := C_2 \cdot g^{r'}$, *return* $C_{e+1} := (C_1', C_2')$

### 10.2.3 Evaluation in PolySphinx

We have implemented the RISE scheme, as well as a PolySphinx variant that uses RISE to multicast messages. The benchmarks done with this PolySphinx variant are labelled `PolySphinx/RISE`, and the results are shown in Chapter 11.

A downside of this approach is that the existing security goals do not cover the use of updatable encryption in PolySphinx: While IND-UPD security means that ciphertexts cannot be linked, it assumes a linear progression of updated ciphertexts — which is only true before the multiplication node, as afterwards multiple updates of one ciphertext can exist.

Another problem is the linkability of the re-encryption tokens to keys or other re-encryption tokens, something that is not covered by the IND-ENC or IND-UPD games. While there are recent approaches that limit the "leakage" of information from tokens, those constructions rely on key switching techniques and indistinguishability obfuscators and are therefore not practical [22].

## 10.3 Re-Randomization of Ciphertexts

The idea of "re-randomization" applies to encryption schemes that have a probabilistic encryption function: Given a ciphertext $c_1$ that contains some random value, we want to transform it into $c_2$ containing a different random value, while still being able to be decrypted by the original key. This process is also called "re-encryption", as the transformed ciphertext looks like it has been decrypted and freshly encrypted. If the re-encryption is possible without the knowledge of the key, it is called "universal re-encryption" [14].

In PolySphinx, we can use universal re-encryption similarly to how we would use updatable encryption (see Section 10.2): By re-encrypting the ciphertext at each mix node, we can make it unlinkable for an outside adversary. In the last step, the recipient can simply use the key to decrypt the payload.

### 10.3.1 Construction of a Re-randomisable Scheme

An encryption scheme that is re-randomisable is ElGamal's scheme:

**Definition 19** (ElGamal's Encryption [12]). *Given a message $m \in \mathcal{G}^*$, a public key $k \in \mathcal{G}^*$ and a random number $r \leftarrow^R \mathbb{Z}_q$. We define the ElGamal encryption of $m$ as*

$$E(k, m) = (g^r, mk^r)$$

Such a ciphertext can be re-randomised by using a different random value $r' \leftarrow^R \mathbb{Z}_q$ and then multiplying the encryption of the unit value onto both elements:

$$(g^r \cdot g^{r'}, mk^r \cdot 1k^{r'}) = (g^{r+r'}, mk^{r+r'})$$

The ciphertext now looks like it was encrypted using the random value $r + r'$.

The challenge is to construct a scheme that can do the re-randomisation without the knowledge of the key $k$, as providing the key to the mix nodes would allow them to recognize the message and therefore remove the indistinguishability. Golle *et al.* solve this problem by including an encrypted version of the public key that can be used for the re-randomisation:

**Definition 20** (Golle's Universal Re-encryption [14]). *The re-randomisable scheme is defined by the following algorithms:*

- RR.KeyGen() *draws* $x \leftarrow^R \mathbb{Z}_q$ *randomly and outputs* $(y = g^x, x)$ *as the public key and secret key.*
- RR.Enc$(y, m)$ *draws* $r_1 \leftarrow^R \mathbb{Z}_q$, $r_2 \leftarrow^R \mathbb{Z}_q$ *randomly and outputs*

  $$C = (C_1, C_2) = ((my^{r_1}, g^{r_1}), (y^{r_2}, g^{r_2}))$$

  .
- RR.Dec$(x, C)$ *pareses* $C$ *as* $C = ((\alpha_1, \beta_1), (\alpha_2, \beta_2))$, *computes* $m_1 = \frac{\alpha_1}{\beta_1^x}$ *and* $m_2 = \frac{\alpha_2}{\beta_2^x}$. *If* $m_2 = 1$, *then* $m_1$ *is returned as the decrypted value. Otherwise, the decryption fails.*
- RR.ReEnc$(C)$ *parses* $C$ *as* $C = ((\alpha_1, \beta_1), (\alpha_2, \beta_2))$ *and draws* $r_1, r_2 \leftarrow^R \mathbb{Z}_q$. *It then computes and outputs the re-randomised ciphertext*

  $$C' = ((\alpha_1 \alpha_2^{r_1}, \beta_1 \beta_2^{r_1}), (\alpha_2^{r_2}, \beta_2^{r_2}))$$

  .

We can see that one ciphertext consists of two parts: The first part is the encryption of the actual message, while the second part is the encryption of the key. On re-randomisation, both parts of the ciphertext are re-randomised with different random values.

### 10.3.2 Evaluation in PolySphinx

We have implemented Golle *et al.*'s universal re-encryption scheme, as well as a PolySphinx variant that uses it, labelled as `PolySphinx/Golle`. We have also implemented a hybrid scheme proposed by Golle *et al.* that uses symmetric encryption for the payload, and only re-encrypts the symmetric keys [14, p. 14]. This hybrid scheme is labelled `PolySphinx/GolleAes`, and the benchmarks are shown in Chapter 11.

We can see that a big disadvantage of the re-encryption scheme is the overhead, both computational and in the size of the message. Those issues are mitigated by the use of the hybrid approach, however, there is no benefit of the hybrid approach compared to the original PolySphinx approach: The overhead due to the keys is bigger, the computational overhead is slightly bigger, and there is no benefit to the privacy goals.

# 11 Evaluation

In order to evaluate our PolySphinx approach in a practical implementation, we will provide three sets of data points to judge three aspects of the protocol. For each aspect, we will provide data of our suggested PolySphinx implementation, the alternatives discussed in Chapter 10 and Rollercoaster [15].

The three aspects that we will evaluate are the following:

**Encryption Benchmarks** In order to judge the performance overhead that the protocol has, we will benchmark the computational overhead that the encryption brings. This is especially important for mix nodes, as they have to process a large number of messages in a short time, as well as for mobile devices, as the amount of computation power needed can have an impact on the battery life of the device.

**Latency Simulation** One of the motivations for PolySphinx is the high latency for existing multicast strategies. In order to judge whether PolySphinx can improve the situation, we will adapt and use the Rollercoaster simulation [15, Section 6] to gather data about the latency with PolySphinx messages.

**Overhead Analysis** One aspect of a mix format is the overhead that it adds to a message. As PolySphinx adds more information to the header, we want to analyse the space overhead of the proposed design and suggested alternatives.

## 11.1 Encryption Benchmark Results

The benchmarks were done with our Rust implementation of PolySphinx, and the `sphinxcrypto`[1] implementation of Sphinx. The benchmarks were executed on a machine with an AMD Ryzen 5 1600 Six-Core Processor (3.2 GHz) and 16 GiB of RAM, and the results were collected with the help of `criterion`[2]. We note that this does not benchmark all of Rollercoaster, as `sphinxcrypto` only implements the base Sphinx design. However, we believe that this is still reflective of Rollercoaster's performance, for two reasons:

First, to the mix node, the Rollercoaster messages are just "normal" Sphinx messages, and therefore they will be processed as fast as normal Sphinx messages. Second, the work to create the Rollercoaster schedule in the sender is small compared to the encryption work done at every mix step.

[1] `https://crates.io/crates/sphinxcrypto`
[2] `https://crates.io/crates/criterion`

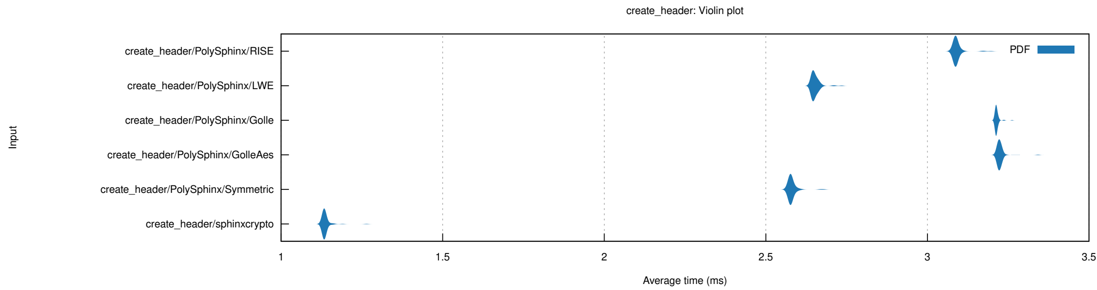


Figure 11.1: Benchmark results of the header creation function. In all cases, a path with length 5 has been created.

The first benchmark result that we provide is that of the header creation. This includes the packing and encryption of the initial header as done by a client that prepares a message to send, but it does not include the initial encryption of the payload. We include this benchmark because a header needs to be created for every message that a client sends, therefore adding overhead to every message.

The results of this benchmark, depicted in Figure 11.1, show that the PolySphinx variants take about twice the time as the Sphinx approach. This is to be expected, as they do the same work as the Sphinx implementation, but in addition generate and embed the re-encryption tokens for the mix node. We also note that the variants that depend on elliptic curve cryptography (RISE, Golle and GolleAes) take more time, as the group operations done on the curves are expensive.

In practice, we can reduce this slowdown by keeping a set of keys and re-encryption tokens ready for use: Since those keys are not bound to the payload, the recipient or the path that a message should take, it is possible to pre-calculate a set of them and then use them when a message should be sent quickly. However, the time saving is small.

A bigger question arises when we look at the payload encryption, as more time is spent there. The results of this benchmark are shown in Figure 11.2.

We can see that the variants that use a symmetric block cipher for the payload re-encryption are the fastest ones, their impact on the message processing is negligible. This is due to the hardware support for the chosen cipher, as modern processors have dedicated instructions for AES encryption. As such, they scale well even for bigger messages.

The hybrid scheme GolleAes uses a block cipher for the majority of the message, and only uses public-key cryptography for a fixed number of keys. Therefore, it has more overhead than a purely symmetrical scheme, but it still scales well and beats the schemes that only rely on non-symmetric cryptography.

The worst scheme is the LWE-based re-encryption scheme, as that involves a lot of matrix multiplications, as well as the elliptic-curve-based RISE and Golle schemes.

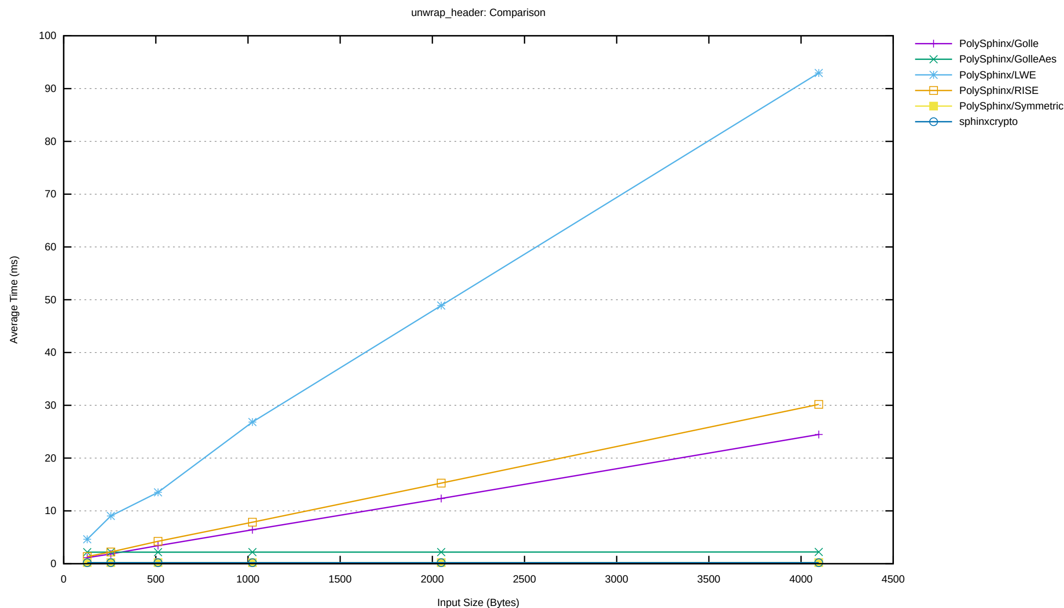


Figure 11.2: Benchmark results of the header unwrapping and payload decryption function, subject to the input size.

## 11.2 Latency Simulation Results

In order to evaluate the latency of multicast messages sent with PolySphinx, we use the `rollercoaster`[3] tool to simulate a mix network. The tool simulates the users' messaging behaviour by sending group messages at random intervals and then tracks the time that the message needed until it was delivered. The simulation does not include network-related latency nor the overhead due to the message processing at mix nodes, but since those delays are negligible compared to the delay that the mixing adds, we can assume that the numbers in practice would look similar.

We then compare the message latencies for the following strategies:

| | |
|---|---|
| Unicast | Sequential unicast |
| Unicast ($p = \ldots$) | Unicast over MultiSphinx with the given multiplication factor |
| RC ($k = p = \ldots$) | Rollercoaster with the given parameters |
| PolySphinx ($p = \ldots, \lambda \leq \ldots$) | PolySphinx with the given multiplication factor and maximum level |

The results for the Unicast and Rollercoaster simulation are also provided in the original Rollercoaster paper [15]. We use this opportunity to verify the numbers published there.

The simulations were run on a server with an 8 core 2.5 GHz CPU and 20 GiB of

[3] `https://github.com/lambdapioneer/rollercoaster`

| | 128 members | | | 256 members | | |
|---|---|---|---|---|---|---|
| | Avg | 99% | 90% | Avg | 99% | 90% |
| Unicast | 34.9 | 75.6 | 60.8 | 68.4 | 151.5 | 119.9 |
| Unicast + MultiSphinx | 8.7 | 16.8 | 14.1 | 15.5 | 30.2 | 26.1 |
| Rollercoaster | 7.0 | 12.3 | 9.9 | 8.3 | 14.5 | 11.7 |
| PolySphinx | 2.1 | 5.6 | 3.6 | 3.4 | 8.7 | 6.0 |

Table 11.1: Average latency and percentiles in seconds for the various strategies.

RAM over the span of a week. Each simulation represents a simulated time span of 24 hours.

### 11.2.1 Online Results

The "online" scenario simulates a network in which all participants are always online. The results (Table 11.1) show that for a group with 128 members, the average latency of 7.0 seconds using Rollercoaster is reduced to 2.1 seconds with PolySphinx. For a bigger group of 256 members, the average time of 8.3 seconds is reduced to 3.4 seconds.

We can see that the MultiSphinx and PolySphinx approaches behave very similar with a single multiplication level (Figure 11.3). This is to be expected, as both variants rely on a single multiplication along the path, cutting down the number of messages that a sender needs to send by a constant factor. Once multiple (nested) levels are allowed, the PolySphinx approach performs better.

Furthermore, especially for small groups, the approaches with a single multiplication (Unicast with MultiSphinx and PolySphinx) achieve a better latency than Rollercoaster. For larger groups, PolySphinx requires a bigger multiplication factor and a higher-level message in order to stay competitive. This can be seen in Figure 11.4. This is also in line with our expectations, as the Rollercoaster latency scales logarithmically with the group size, whereas PolySphinx scales linearly.

### 11.2.2 Offline Results

The "offline" scenario simulates a network in which participants are not always online. This is important for Rollercoaster, as the strategy uses the other group members to help in distributing a message. Since PolySphinx does not rely on group members for the message distribution, we expect that offline members do not hinder the message delivery.

We can see in the resulting latencies (Figure 11.5) that our expectation is met: PolySphinx compares favourably to the fault-tolerant version of Rollercoaster. The bar plot (Figure 11.6) confirms this and shows that even for big groups, PolySphinx performs well.

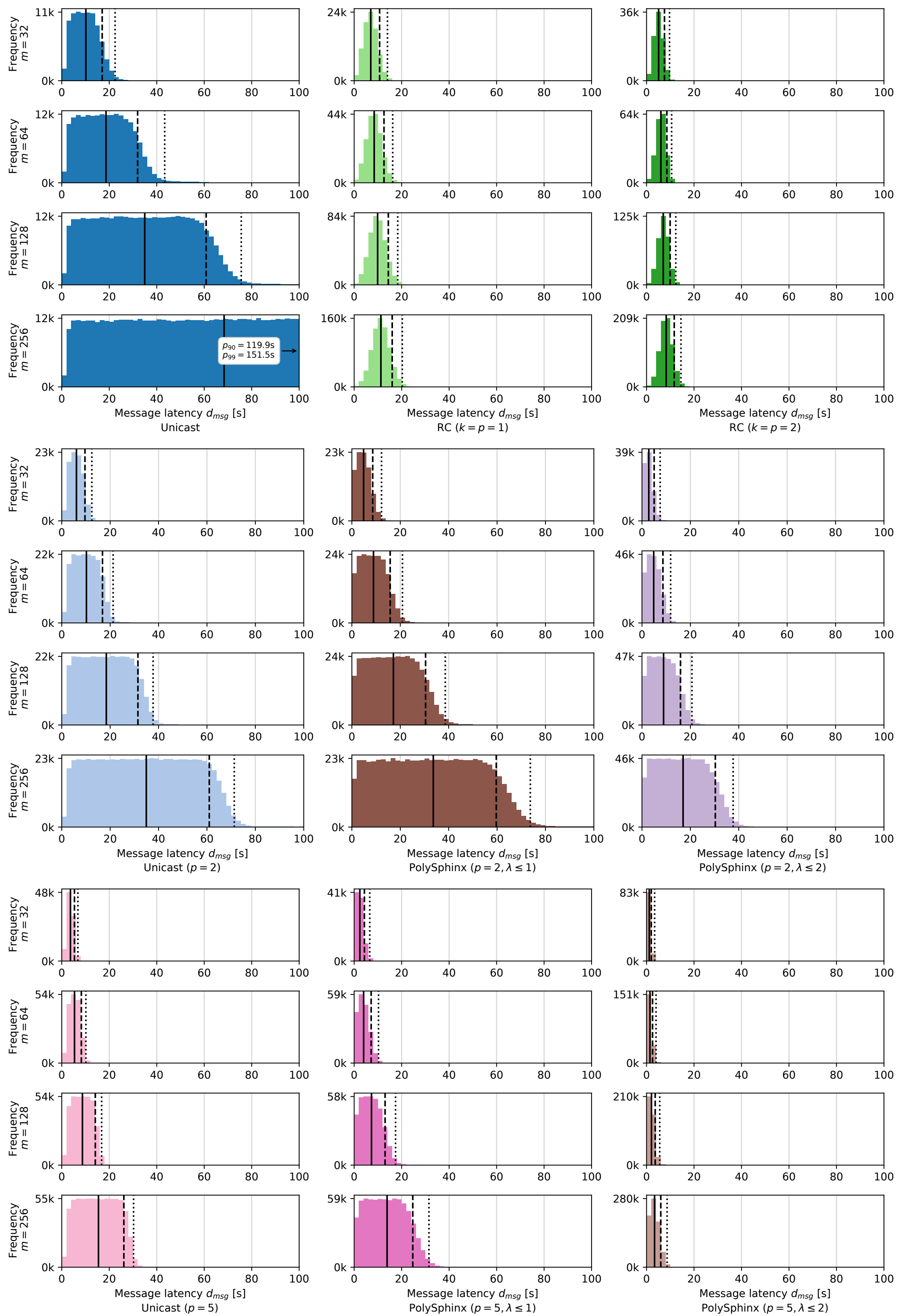


Figure 11.3: Histograms for message latency in the online scenario.

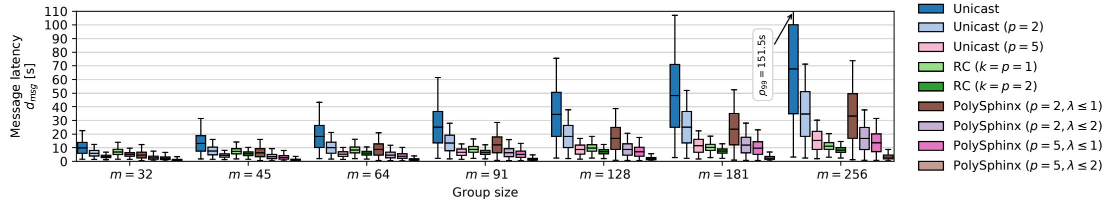


Figure 11.4: Bar plots for the average message latency depending on the group size, with every member online.

## 11.3 Overhead Analysis

An important aspect of a mix format is the space overhead that the header takes. Sphinx itself is very compact [11, Table 1], while Rollercoaster adds a constant overhead to the payload of every multicast message [15, Figure 3]. In this section, we want to analyse the overhead that a PolySphinx message has generally while giving concrete numbers for the alternatives discussed in Chapter 10.

We note that the design in Chapter 6 has a fixed view of the actual space requirements: The design talks about space in terms of $\kappa$ bits whenever keys are needed. However, for the alternative approaches, this is usually not true, as they use different cryptographic primitives. For example, the Curve25519 elliptic curve provides 128 bits of security, but requires 256 bits to store the actual keys [3, p. 208] — therefore, a key would need $2\kappa$ of space.

We will therefore introduce a few new variables that we will use in this section:

- We will denote the size of an asymmetric key as $s_a$. For our implementation from Chapter 8, this expands to $s_a = 256$.
- We will denote the size of a symmetric key as $s_s$. For our implementation from Chapter 8, this expands to $s_s = \kappa = 128$.
- We will denote the size of a re-encryption key as $s_r$. Our main design has $s_r = \kappa = 128$, however this value can change if an alternative approach (Chapter 10) is used.
- We will denote the size of the final key as $s_f$. Our main design has $s_f = \kappa + r\lceil \log_2 p \rceil$, however this value can change if an alternative approach (Chapter 10) is used.
- We will denote the payload blow-up $s_m : \mathbb{N} \to \mathbb{N}$ as a function that determines by how much the payload size is increased after encryption. Given the payload size $m$, $s_m(m)$ is the payload size after creating the multicast message.

Note that our values for $s_a$ and $s_s$ are in accordance with the values of $p$ and $s$, respectively, for the ECC variant from the Sphinx comparison table [11, Table 1]. For

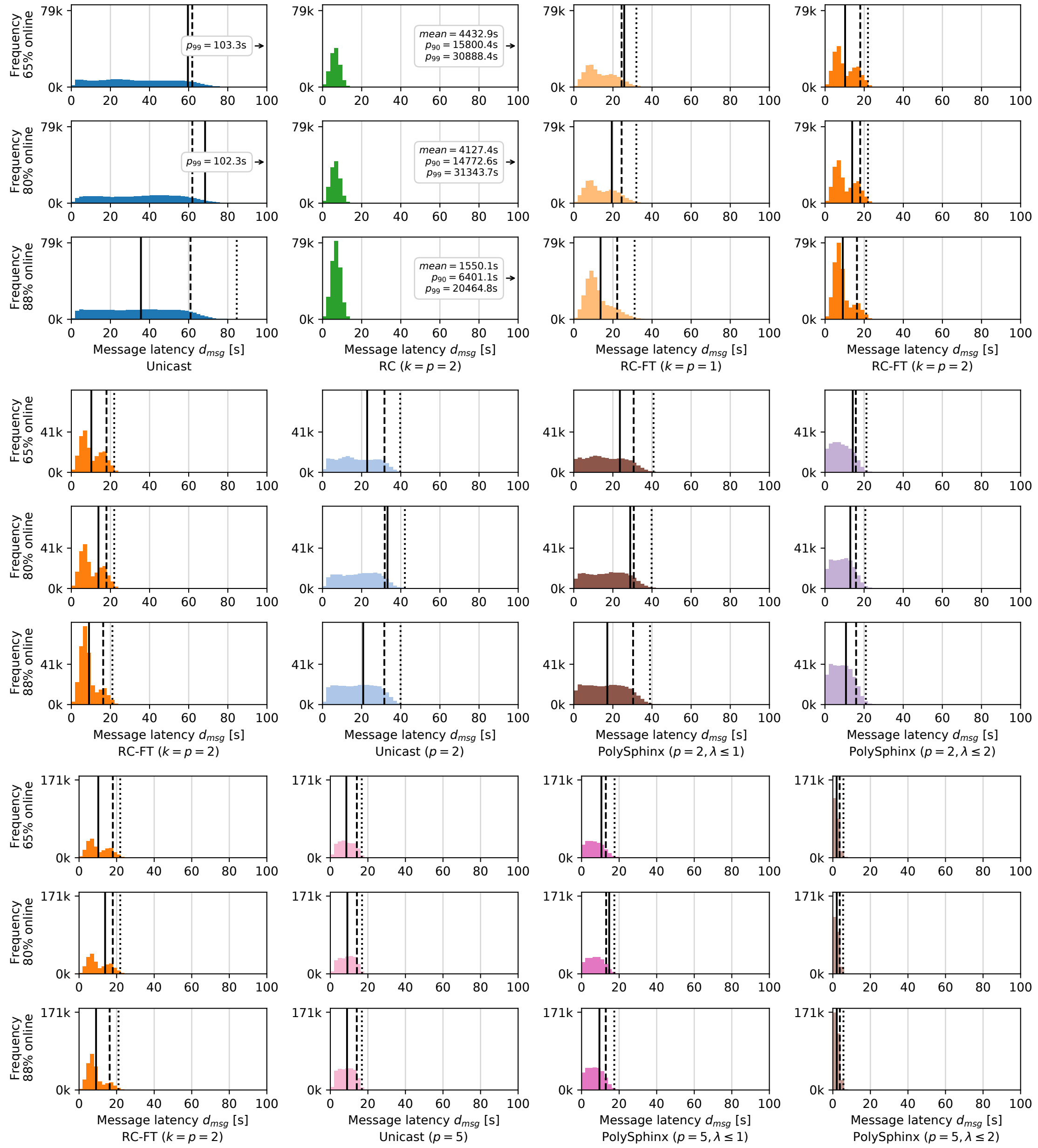

Figure 11.5: Histograms for message latency in the offline scenario.

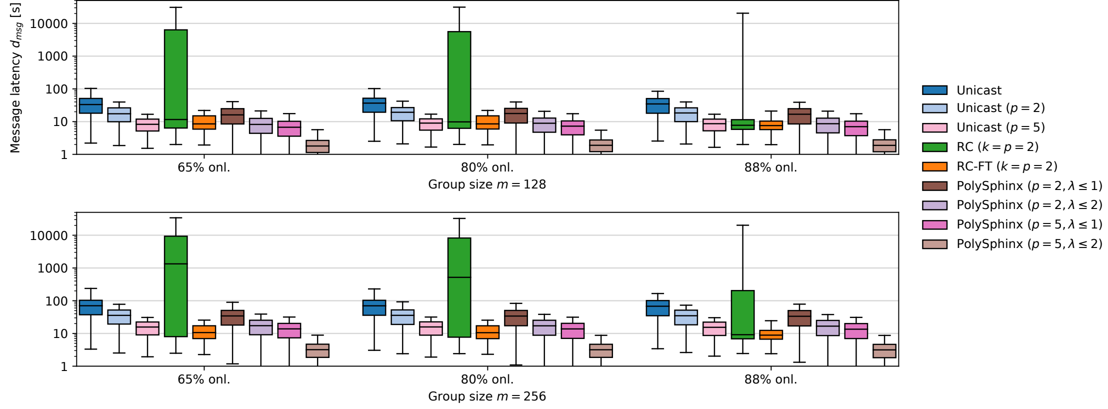


Figure 11.6: Bar plots for the average message latency depending on the group size, with members being offline.

$r$, the maximum path length, we will choose $r = 5$ as an example value. We will choose $p = 4$ as an example of a multicast factor.

The formulas to calculate the size of a header are as follows:

- For Sphinx, we use $s_a + (2r + 2)s_s$ as the size of a header [11, p. 12]. Note that this includes the size for the initial shared secret and the initial MAC. For Unicast, we therefore use $p \cdot (s_a + (2r + 2)s_s)$ as an approximation, as Unicast requires $p$ separate Sphinx headers to be sent.

- For a level 0 PolySphinx header, also including the initial shared secret and MAC, we have a size of

$$S_0 = s_a + s_s + (r - 1)(8 + 2s_s + s_r) + 8 + 3s_s + s_f$$

- For a level $\lambda > 0$ PolySphinx header, we have a size of

$$S_\lambda = (r - 1) \cdot (8 + 2s_s + s_r) + 8 + p(s_s + s_r + S_{\lambda-1})$$

The parameters for the various PolySphinx implementations are shown in Table 11.2. We can see that the schemes based on asymmetric cryptography have a big overhead in the size of the extra data that needs to be embedded into the headers. In addition to the high overhead in the header, they also lead to a size expansion of the payload, leading to twice the amount of space required to store the same information.

We can add the size of the header to the size of the payload to determine the size of the complete message, giving us the values in Table 11.3 for direct messages (one recipient) and the values in Table 11.4 for multicast messages ($p$ recipients).

We can see that all PolySphinx approaches have an overhead compared to Sphinx in direct messages: For PolySphinx, this is explained by the extra keys, for example,

| | $s_s$ | $s_a$ | $s_r$ | $s_f$ | $s_m(m)$ |
|---|---|---|---|---|---|
| Unicast | 128 | 256 | 0 | 0 | $pm$ |
| PolySphinx | 128 | 256 | 128 | 138 | $m$ |
| PolySphinx/RISE | 128 | 256 | 512 | 512 | $2m$ |
| PolySphinx/LWE | 128 | 256 | 288 | 288 | $2m$ |
| PolySphinx/Golle | 128 | 256 | 512 | 256 | $2m$ |
| PolySphinx/GolleAes | 128 | 256 | 512 | 256 | $m + 5 \cdot 128$ |

Table 11.2: Table with the system parameters for the various approaches.

| | $m = 8192$ | $m = 16384$ | $m = 32768$ |
|---|---|---|---|
| Unicast | 9984 | 18176 | 34560 |
| PolySphinx | 10674 | 18866 | 35250 |
| PolySphinx/RISE | 20776 | 37160 | 69928 |
| PolySphinx/LWE | 19656 | 36040 | 68808 |
| PolySphinx/Golle | 20520 | 36904 | 69672 |
| PolySphinx/GolleAes | 12968 | 21160 | 37544 |

Table 11.3: Sizes (in bits) for a direct message.

$9984 + 5 \cdot 8 + 4 \cdot 128 + 138 = 10674$, accounting for 5 flag bytes, 4 re-encryption keys and the final decryption information. We can also see that the overhead PolySphinx induces over Sphinx is rather small, only about 3.1% for a 1 KiB (8192 bits) message, and about 1.7% for a 2 KiB (16384 bits) message.

For the alternative approaches, however, except for the hybrid scheme, the situation is made worse by the fact that the overhead is not limited to the header, but also increases the payload size. Therefore, those approaches have a massive overhead compared to Sphinx, and this overhead scales with the message size.

For multicast messages, we see that the sequential unicast approach can be advantageous for very short messages — in this example, 140 bytes (1120 bits) would require the least amount of data if the payload is just sent multiple times. This is because the overhead that PolySphinx induces does not offset the data saved by the payload

| | $m = 1120$ | $m = 1787$ | $m = 8192$ | $m = 16384$ | $m = 32768$ |
|---|---|---|---|---|---|
| Unicast | 11648 | 14316 | 39936 | 72704 | 138240 |
| PolySphinx | 13648 | 14315 | 20720 | 28912 | 45296 |
| PolySphinx/RISE | 25480 | 26814 | 39624 | 56008 | 88776 |
| PolySphinx/LWE | 19208 | 20542 | 33352 | 49736 | 82504 |
| PolySphinx/Golle | 24456 | 25790 | 38600 | 54984 | 87752 |
| PolySphinx/GolleAes | 23976 | 24643 | 31048 | 39240 | 55624 |

Table 11.4: Sizes (in bits) for a multicast message.

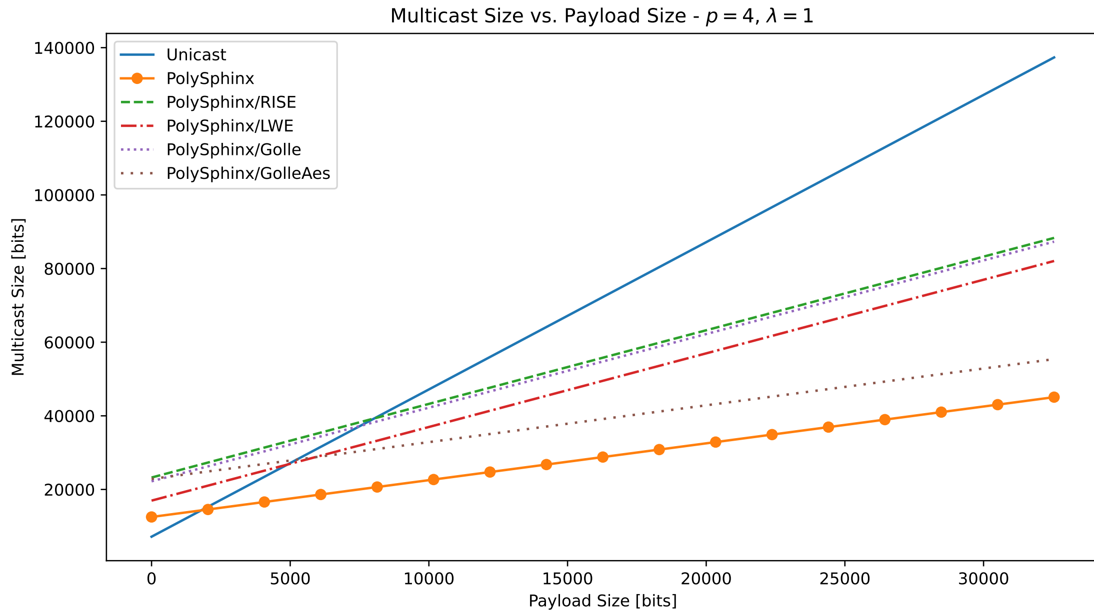


Figure 11.7: Graph of the size increase for multicast messages.

de-duplication.

However, the break-even point is at 1787 bits, meaning that messages bigger than 223 bytes would already benefit from the PolySphinx scheme, and the bigger the message is, the more bandwidth we can save. The alternative PolySphinx approaches do take longer to break even with the sequential unicast scheme, but all of them scale better than unicast and break even before a payload size of 1 KiB (8192 bits). A visual representation of this behaviour is also shown in Figure 11.7.

# 12 Discussion

We can see that MultiSphinx and PolySphinx share a few similarities: They both rely on the concept of a multiplication node that takes a single message and produces multiple outgoing messages. This also leads to a similar performance when put under similar circumstances.

The big difference however lies in the fact that MultiSphinx requires the payload to be duplicated, while PolySphinx can use the same payload for all recipients. The advantage of the MultiSphinx approach is that it is more flexible, as different messages can be combined into a single MultiSphinx message. This can already provide a benefit, for example in the power consumption of the client device, as sending one large message is favourable compared to sending multiple small messages [15, p. 5]. The disadvantage however lies in the fact that payloads must still be duplicated even if they contain the same plaintext, which means that bandwidth is wasted in those cases.

PolySphinx is the opposite: While bandwidth can be saved when dealing with identical message payloads, it cannot combine different payloads into one message. Especially for small, non-multicast messages, the added overhead can offset the savings from efficient multicast messages.

Therefore, a good use-case for a PolySphinx network would be one where most communication is done in small to medium-sized groups. Especially services that make use of multimedia such as audio, images and video data can benefit from the bandwidth saving, as those take more bandwidth to send in the first place. A messenger based on PolySphinx could support multimedia group communication while still having a low bandwidth overhead.

We note that another difference between MultiSphinx and PolySphinx is the hierarchical architecture of PolySphinx that allows multiple nested levels. This gives PolySphinx more flexibility in choosing the multiplication factor, as a higher level leads to more multiplication. However, we can imagine a MultiSphinx variant that offers a similar level of flexibility by also allowing nested MultiSphinx messages, or by having different multiplication nodes with different multiplication factors. Because of the payload duplication, those messages would get rather big though — or they would only support a small payload.

In the future, it might be possible to construct a hybrid scheme that supports both modes of operation, allowing for the flexibility of MultiSphinx if independent messages are sent and the bandwidth savings of PolySphinx when a group message is sent.

# 13 Conclusion

In this thesis, we have designed and constructed an extension to the Sphinx mix format that allows for an efficient multicast of messages, without the need to duplicate the payload by the sender. We have shown that our scheme is secure using a hybrid argument, and the security is based on well-understood concepts such as the Diffie-Hellman-Assumption and the indistinguishability of ciphertexts.

Furthermore, we have used simulations to show that PolySphinx performs well in a group setting, bringing down the average latency from 34.9 seconds with sequential unicast to 2.1 seconds with level-2 PolySphinx in a group of 128 members. For our simulations, we have used and extended the tool developed by Hugenroth *et al.* for the Rollercoaster evaluation [15]. As such, we have a common base for our results and we can give comparable data.

While our scheme allows the recipient to recover information about the sender, we have suggested possible extensions that limit this leakage. As such it is possible to hide the information about other recipients in a group message or to hide information about the sender in a direct one-to-one message.

For the design of PolySphinx, we have evaluated multiple alternative approaches based on recent cryptographic primitives such as key-homomorphic pseudorandom functions and updatable encryption. We have benchmarked all of those alternatives and we have found our main PolySphinx design to be the most efficient in terms of computational and space overhead. While those alternatives do not provide an advantage in the current design, it is possible that future developments in those areas could provide a better way to handle the multiplication of the payload.

The improved efficiency of PolySphinx opens up new areas in which mix networks could be used to improve users' privacy, for example, we could imagine new multimedia messengers built on top of mixnets.

# Bibliography


[1] Abhishek Banerjee and Chris Peikert. *New and Improved Key-Homomorphic Pseudorandom Functions.* Cryptology ePrint Archive, Report 2014/074. https://ia.cr/2014/074. 2014.

[2] Daniel J. Bernstein. *ChaCha, a variant of Salsa20.* Jan. 28, 2008.

[3] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Public Key Cryptography - PKC 2006.* Ed. by Moti Yung et al. Vol. 3958. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33851-2. DOI: 10.1007/11745853_14. URL: http://link.springer.com/10.1007/11745853_14 (visited on 01/24/2022).

[4] Dan Boneh. "The Decision Diffie-Hellman problem". In: *Algorithmic Number Theory.* Ed. by Joe P. Buhler. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 48–63. ISBN: 978-3-540-69113-6.

[5] Dan Boneh et al. *Key Homomorphic PRFs and Their Applications.* Cryptology ePrint Archive, Report 2015/220. https://ia.cr/2015/220. 2015.

[6] Jan Camenisch and Anna Lysyanskaya. "A Formal Treatment of Onion Routing". In: *Advances in Cryptology – CRYPTO 2005.* Ed. by Victor Shoup. Red. by David Hutchison et al. Vol. 3621. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 169–187. ISBN: 978-3-540-28114-6. DOI: 10.1007/11535218_11. URL: http://link.springer.com/10.1007/11535218_11 (visited on 04/25/2022).

[7] David Chaum. "The dining cryptographers problem: Unconditional sender and recipient untraceability". In: *Journal of Cryptology* 1.1 (Jan. 1988), pp. 65–75. ISSN: 0933-2790, 1432-1378. DOI: 10.1007/BF00206326. URL: http://link.springer.com/10.1007/BF00206326 (visited on 05/23/2022).

[8] David L. Chaum. "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms". In: *Commun. ACM* 24.2 (Feb. 1981), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/358549.358563. URL: https://www.chaum.com/publications/chaum-mix.pdf.

[9] Kaiming Chen and Jiageng Chen. "Anonymous End to End Encryption Group Messaging Protocol Based on Asynchronous Ratchet Tree". In: *Information and Communications Security.* Ed. by Weizhi Meng et al. Vol. 12282. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 588–605. ISBN: 978-3-030-61077-7. DOI: 10.1007/978-3-030-61078-4_33.

URL: https://link.springer.com/10.1007/978-3-030-61078-4_33 (visited on 05/17/2022).

[10] Katriel Cohn-Gordon et al. “On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18: 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto Canada: ACM, Oct. 15, 2018, pp. 1802–1819. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243747. URL: https://dl.acm.org/doi/10.1145/3243734.3243747 (visited on 05/17/2022).

[11] George Danezis and Ian Goldberg. “Sphinx: A Compact and Provably Secure Mix Format”. In: *2009 30th IEEE Symposium on Security and Privacy*. 2009, pp. 269–282. DOI: 10.1109/SP.2009.15. URL: https://www.cypherpunks.ca/~iang/pubs/Sphinx_Oakland09.pdf.

[12] T. Elgamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE Transactions on Information Theory* 31.4 (July 1985), pp. 469–472. ISSN: 0018-9448. DOI: 10.1109/TIT.1985.1057074. URL: http://ieeexplore.ieee.org/document/1057074/ (visited on 05/27/2022).

[13] Keita Emura et al. *Membership Privacy for Asynchronous Group Messaging*. Published: Cryptology ePrint Archive, Report 2022/046. 2022.

[14] Philippe Golle et al. “Universal re-encryption for mixnets”. In: *Cryptographers’ Track at the RSA Conference*. Springer. 2004, pp. 163–178.

[15] Daniel Hugenroth, Martin Kleppmann, and Alastair R. Beresford. “Rollercoaster: An Efficient Group-Multicast Scheme for Mix Networks”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3433–3450. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/hugenroth.

[16] IANIX. *ChaCha Usage & Deployment*. May 5, 2022. URL: https://ianix.com/pub/chacha-deployment.html (visited on 05/25/2022).

[17] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Second Edition. Chapman and Hall/CRC, 2015. ISBN: 978-1-4665-7027-6.

[18] Michael Klooß, Anja Lehmann, and Andy Rupp. “(R)CCA Secure Updatable Encryption with Integrity Protection”. In: *Advances in Cryptology – EUROCRYPT 2019*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11476. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 68–99. ISBN: 978-3-030-17652-5. DOI: 10.1007/978-3-030-17653-2_3. URL: http://link.springer.com/10.1007/978-3-030-17653-2_3 (visited on 02/16/2022).

[19] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Feb. 1997. DOI: 10.17487/RFC2104. URL: https://www.rfc-editor.org/info/rfc2104.

[20] Anja Lehmann and Björn Tackmann. "Updatable Encryption with Post-Compromise Security". In: *Advances in Cryptology – EUROCRYPT 2018*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 685–716. ISBN: 978-3-319-78371-0. DOI: 10.1007/978-3-319-78372-7_22. URL: https://link.springer.com/10.1007/978-3-319-78372-7_22 (visited on 02/18/2022).

[21] Dong Lin, Micah Sherr, and Boon Thau Loo. "Scalable and Anonymous Group Communication with MTor." In: *Proc. Priv. Enhancing Technol.* 2016.2 (2016), pp. 22–39.

[22] Ryo Nishimaki. *The Direction of Updatable Encryption Does Matter*. Published: Cryptology ePrint Archive, Report 2021/221. 2021.

[23] G. Perng, M.K. Reiter, and Chenxi Wang. "M2: Multicasting Mixes for Efficient and Anonymous Communication". In: *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06). Lisboa, Portugal: IEEE, 2006, pp. 59–59. ISBN: 978-0-7695-2540-2. DOI: 10.1109/ICDCS.2006.53. URL: http://ieeexplore.ieee.org/document/1648846/ (visited on 05/23/2022).

[24] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing - STOC '05*. the thirty-seventh annual ACM symposium. Baltimore, MD, USA: ACM Press, 2005, p. 84. ISBN: 978-1-58113-960-0. DOI: 10.1145/1060590.1060603. URL: http://portal.acm.org/citation.cfm?doid=1060590.1060603 (visited on 01/28/2022).

[25] Michael Seufert et al. "Analysis of Group-Based Communication in WhatsApp". In: *Mobile Networks and Management*. Ed. by Ramón Agüero et al. Cham: Springer International Publishing, 2015, pp. 225–238. ISBN: 978-3-319-26925-2.

[26] Wikipedia contributors. *AES implementations — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=AES_implementations&oldid=1084722376. [Online; accessed 25-May-2022]. 2022.