Technical Report

UCAM-CL-TR-957
ISSN 1476-2986

Number 957

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Rollercoaster: an efficient group-multicast scheme for mix networks

Daniel Hugenroth, Martin Kleppmann, Alastair R. Beresford

Originally published in Proceedings of the 30th USENIX Security Symposium

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Rollercoaster: An Efficient Group-Multicast Scheme for Mix Networks

Daniel Hugenroth
University of Cambridge

Martin Kleppmann
University of Cambridge

Alastair R. Beresford
University of Cambridge

Abstract

Mix network designs such as Loopix provide strong metadata anonymity guarantees that are crucial across many applications. However, because they limit the rate at which messages can be sent by each user, they incur high delays when sending many messages to multiple recipients – for instance, in decentralised collaborative apps.

In this paper we present an efficient multicast scheme named Rollercoaster that reduces the time for delivering a message to all members of a group of size m from $\mathcal{O}(m)$ to $\mathcal{O}(\log m)$. Rollercoaster can be deployed without modifications to the underlying mix network, allowing it to benefit from the anonymity set provided by existing users. We further develop an extension that achieves the same asymptotic guarantees in the presence of unreliable group members.

While the scheme is applicable to many mix network designs, we evaluate it for the Loopix network, which is the most advanced and practical design to date. For this evaluation we developed a network simulator that allows fast, reproducible, and inspectable runs while eliminating external influences.

1 Introduction

Information security often focuses on the confidentiality and integrity of electronic messages. However, metadata privacy is frequently also important since merely knowing the parties involved in a communication can reveal sensitive information and stigmatise groups and individuals. For example, revealing the names of people contacting a sexual health clinic may discourage individuals from seeking treatment; and potential whistle-blowers may be dissuaded from disclosing illegal or unethical behaviour to a journalist. Strong metadata privacy is critical across many domains, not just healthcare and journalism, but also in diplomatic services and military operations.

Protecting metadata privacy is not merely a theoretical requirement: we find ourselves in an era of mass-surveillance by well-funded state actors as well as pervasive data collection by private companies and service providers. In this reality there are many online applications where protecting metadata privacy is of practical importance.

The Tor [1] network is perhaps the best known example of a system that provides metadata privacy. Tor brought so-called anonymous communication networks to a large audience by providing low-latency communication and anonymous access to the Internet. However, while the Tor network provides high throughput and low latency, it does not provide metadata privacy in the presence of a global adversary who can observe all communication [2]. Mix network designs and broadcast schemes provide metadata privacy in the face of global adversaries, however they do so at the cost of significantly higher latency and lower overall throughput. A prominent recent medium-latency mix network design, which protects metadata privacy in the presence of a global adversary, is Loopix [3].

Many *collaborative apps* are in use today, including group messaging services such as WhatsApp, Signal, and iMessage; productivity tools such as Google Docs and Office365; and file sharing applications such as Dropbox and Box. At present, no mainstream collaborative apps provide metadata privacy. Hence, in this paper, we present a new architecture that enables strong metadata privacy for such applications.

We consider forms of collaboration in which a file or conversation thread is shared by a group of collaborators, and any update to it needs to be shared with all group members. Group messaging and collaboration can share the same underlying infrastructure [4]. In collaborative editing applications individual update messages are usually small and frequent [5]. Such apps therefore require an efficient, reliable, and timely method of sending messages to all members of a group. However, the original design of Loopix provides only one-to-one (unicast) messages, and no built-in mechanism for group communication (multicast). In this paper we show that naïvely implementing multicast in an anonymity network like Loopix results in significant overhead in terms of latency and throughput, typically exceeding the latency required to provide good user experience. We therefore extend Loopix to support low-latency group communication while preserving metadata privacy.

This paper makes the following contributions:

- An anonymous group communication scheme called *Rollercoaster*, which achieves a group multicast latency of $\mathcal{O}(\log m)$ for groups of size m, while ensuring strong metadata privacy against an active global adversary (§5). In our evaluation Rollercoaster achieves a 99th percentile (p_{99}) latency of 12.3 seconds for groups larger than 100 users, whereas the default implementation of Loopix incurs a latency of 75.6 s (§6.2). Rollercoaster works by involving many group members, not just the sender of a message, in the task of disseminating a message.

- An extension to the Rollercoaster scheme that adds fault-tolerance to gracefully handle the fact that some group members may be offline, while preserving scalability (§5.2). Even in the presence of faulty nodes, Rollercoaster performs better than default Loopix for mean, p_{90}, and p_{99} latency. Our solution reduces p_{99} latency to 21.9 s compared to 103.3 s for default Loopix (§6.3) when evaluated against realistic connectivity patterns.

- The design of the *MultiSphinx* packet format that allows limited multicast by designated mix nodes while preserving strong metadata privacy guarantees (§5.4).

- A deterministic, open-source simulator for Loopix and Rollercoaster that allows efficient, inspectable, and reproducible performance evaluations. We use it to empirically compare the latency properties of both systems. Compared to evaluations using a real network, it reduces the required CPU hours by a factor of 4500×, allowing us to explore significantly more scenarios and parameter choices (§6.1).

2 Threat Model and Goals

Our work guarantees strong anonymity against sophisticated adversaries while providing an efficient, low-latency, and fault-tolerant group-multicast anonymity network.

Assumptions We assume three types of participants in a mix network based on the Loopix model [3]: *Users* are members of one or more groups; group members can broadcast and receive messages to and from all members of the group. *Provider nodes* act as the users' entry points to the anonymity network; all communication to or from a specific user flows through their provider. *Mix operators* manage a mix node in the core of the network; mix nodes receive messages from other mix nodes or providers and send messages to other mix nodes or providers. Mix nodes do not communicate directly with users. For further details on the Loopix model and how these participants communicate, see Section 3.

Security and Anonymity We assume a global active adversary who can observe all traffic, manipulate traffic to remove messages and insert new ones, as well as corrupt a subset of mix nodes and providers. As in Loopix, sending a message to a Rollercoaster user requires that the sender knows both the addresses and public keys for their provider, the recipient, the recipient's provider, and the mix nodes.

Our scheme provides message confidentiality and integrity as well as the same strong metadata privacy guarantees as Loopix, including *sender-recipient unlinkability* (preventing an adversary from deducing which users are communicating with each other) and *sender/recipient online unobservability* (preventing an adversary from deducing which users are currently participating in any communication). More details on these and further definitions of metadata privacy are given by Pfitzmann and Hansen [6]. In addition we provide *membership unobservability* (preventing anyone outside the group from determining group membership or group size). We assume a group is composed of trusted members and therefore we do not provide unlinkability or unobservability guarantees against an attacker who compromises or colludes with group members. The goal of the attacker is to break the confidentiality, integrity, or metadata privacy guarantees.

Our scheme supports efficient communication for group sizes of two or more and therefore we handle pairwise and group communication in the same way. An attacker cannot distinguish between two-party communication and communication in a larger group.

Application Requirements Low latency is often a requirement in group communication. For example, user studies have highlighted the negative implications of high network delays in collaborative editing. One previous study [7] asked a group of participants to transcribe audio lectures using collaborative text editing software. The researchers investigated the effect of communication latency by repeating the experiment multiple times and varying artificial delay on all communication between participants. A delay of 10 seconds or more had a significant impact in their study, with an increase of error rates and content redundancy by more than 50%. We therefore set our target for group multicast latency at 10 seconds for group sizes of up to 100 people. The group size is motivated by the active editor limit of Google Docs (100 users) and Microsoft Sharepoint (99 users). We further require the latency to grow sub-linearly with the size of the group, allowing effective collaboration in large groups. In many multi-user applications, a large fraction of the data is generated by a small fraction of the users (a trend that is known as *participation inequality* [8]), and our scheme fares well in a system with such a distribution of activity.

Offline support is required since mobile devices do not always have connectivity. As in the Loopix design, provider nodes in Rollercoaster store messages on behalf of the user until the user is next online and able to download them.

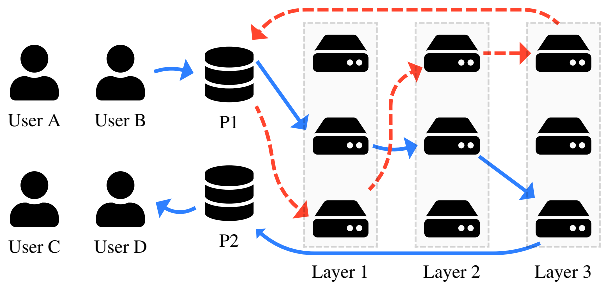


Figure 1: Schematic for a Loopix network with four users (A, B, C, D), two providers (P1, P2), and a three-layer mix network. Each node of mix layer L is connected to each node of layer $L+1$. The solid blue arrows depict one possible path for a payload or drop message from user B to user D. The dashed red line represents loop traffic induced by a mix node.

On mobile devices, the frequency of sending network packets has a large impact on energy efficiency. Every transmission promotes the mobile network connection from idle to an active sending state after which it remains in a tail state for a few seconds [9, §5.1]. During the active sending/receiving state (1680 mW, data for LTE) and the tail state (1060 mW) the power consumption is higher than during idle (594 mW) [9, Table 3]. Every promotion from idle to active comes with additional energy costs. Therefore, sending few but large messages with long intra-packet pauses is advantageous for battery life on mobile devices, even if the total volume of data transmitted is the same. On the other hand, smaller and more frequent messages lead to lower latency.

We assume that the group membership is fixed and known to all members; we leave the problems of group formation and adding or removing group members for future work.

3 Background

Our work builds on Loopix, which we introduce in this section. Section 3.2 introduces multicast as it is used in this paper.

3.1 Loopix

Loopix is a mix network [10]: messages are sent via several mix nodes to conceal their sender and destination. The route is chosen by the sender and encoded in message headers.

Several mix network designs have been proposed: for example, in the *threshold* approach, a mix node waits until a fixed number of messages have arrived, and then forwards them to their next hops in a random order. A mix node must wait for a sufficient number of messages to arrive before forwarding them to ensure there is significant uncertainty in the mapping between incoming and outgoing messages. Unfortunately, this batching process can lead to high latency.

Loopix takes a different approach to mixing: whenever a message passes through a node, it delays that message by a duration d_μ. For each hop the sender independently chooses d_μ randomly from the exponential distribution with rate parameter λ_μ, and includes that value in the message header.

Moreover, Loopix ensures that the timings of messages sent by any node can be modelled as a *Poisson process* (i.e. the interval between messages is exponentially distributed). Applying exponentially distributed random delay to a Poisson process yields another Poisson process; moreover, aggregating the events from several Poisson processes yields another Poisson process [3]. Message senders can adjust λ_μ to balance the trade-off between reducing latency (increase λ_μ) and strengthening anonymity (decrease λ_μ).

An individual mix node may be compromised by the adversary, allowing it to learn the mapping between input and output messages. However, a mix network provides strong anonymity guarantees when at least one of the mix nodes on the message's path is trustworthy. Cover traffic is added to hide communication patterns and to prevent an attacker from inferring message senders and recipients merely by looking at the set of messages sent and received over time.

Loopix arranges mix nodes in l layers (where $l = 3$ is a typical choice), forming a *stratified topology*. In this arrangement, each node is connected to all nodes of the next layer, and a message flows through one mix node in each layer. The system's message throughput capacity can be increased by adding more nodes to each layer.

Access to the mix network is mediated by provider nodes (see Figure 1). Providers receive and store incoming messages for each user in an inbox, allowing the end-user device to be offline and download messages from the provider later. These messages are still end-to-end encrypted and providers cannot distinguish them from cover traffic (see below). The provider nodes are a required component if the end-user device (e.g. smartphone) is not always connected to the Internet. Moreover, the provider nodes support revenue generation since the provider can charge users to cover operating costs without knowing who their customers are communicating with.

3.1.1 Messages and Traffic

All Loopix messages are encrypted and padded to a fixed size[1] using the Sphinx [11] mix message format. The Sphinx message format uses layered encryption and ensures the contents of messages change at every hop in the mix network. Fixed-size padding renders messages containing payload traffic indistinguishable from cover traffic messages. This approach means the attacker cannot correlate incoming and outgoing messages based on payload contents or length. Loopix uses three types of messages:

Drop messages are the primary form of cover traffic. They are sent by users as a Poisson process with rate parameter λ_d

[1]The implementation accompanying the original Loopix paper uses a message size of 1024 bytes, including headers and overheads.

$d_p \sim \exp(\lambda_p)$	Delay between successive payload messages
$d_d \sim \exp(\lambda_d)$	Delay between successive drop messages
$d_l \sim \exp(\lambda_l)$	Delay between successive loop messages
$d_\mu \sim \exp(\lambda_\mu)$	Delay applied on message forwarding
Δ_{pull} (constant)	Polling interval for checking inboxes
$d_M \sim \exp(\lambda_M)$	Delay between successive loop messages sent by mix nodes

Table 1: Delays are either constant or chosen from an exponential (exp) distribution with the given parameter. Our notation slightly differs from the original paper.

and addressed to a randomly chosen user's inbox. They follow the full transport route from the sender's provider through all layers of the mix network to the recipient's device. Recipients download the message from their inboxes, decrypt it, and only then identify it as drop traffic and discard it.

Payload messages contain application data and are sent as a Poisson process with rate parameter λ_p. When an user sends multiple messages in quick succession, they are added to a send queue at the client and forwarded to the user's provider at an average rate of λ_p. While they are in the payload send queue, messages experience delay d_Q. When there are no payload messages waiting to be sent, a drop message is sent instead. Keeping the send rate constant prevents irregular traffic patterns that may reveal whether a user is currently actively communicating.

Loop messages defend against active attacks such as $(n-1)$ *attacks* [12]. In such an attack an adversary tries to follow the path of a message by blocking all other incoming traffic for the mix node or replacing it with its own. Loop messages are injected by both users (at rate λ_l) and mix nodes (at rate λ_M); these messages travel in a loop though all mix layers, via a provider node, back to the sender. If the loop messages sent by a node fail to be delivered back to that node, it can suspect that an active attack is taking place and employ countermeasures as described in the Loopix paper [3, §4.2.1].

Choosing suitable rate parameters depends heavily on the application behaviour, the message size, and the capacity of the underlying network. In the original Loopix paper the values of the parameters λ_p, λ_d, and λ_l range from one message per second to one message per minute. With a total message size of $|msg|$ bytes and the rates given in messages/s, the required bandwidth of a client can be estimated as $(\lambda_p + \lambda_d + \lambda_l) \cdot |msg|$ bytes/s.

3.2 Multicast and Group Messaging

A *multicast* protocol allows a single message to be delivered to all members of a group. Broadly speaking, there are two approaches for implementing multicast: by sending each message individually to each recipient over unicast, or by relying on the underlying network to make copies of a message that are delivered to multiple recipients. IP multicast [13] is an example of the latter approach, which avoids having to send the same message multiple times over the same link.

In this paper we are interested in *group multicast*, a type of multicast protocol in which there is a pre-defined, non-hierarchical group of users U. At any time any member of the group might send a message to all other group members. We call the initial sender *source* s and all others the intended recipients $U_{recv} = U \setminus s$.

4 Naïve Approaches to Multicast

In this section we discuss the reasons why message delays occur in Loopix. We then explore two simple approaches to implementing multicast on Loopix, and explain why they are not suitable, before introducing Rollercoaster in Section 5.

We define the message latency d_{msg} of a single unicast message msg from user to A to user B:

$$d_{msg} = T_{recv,B} - T_{send,A} \tag{1}$$

where $T_{send,A}$ is the time at which user A's application sends msg, and $T_{recv,B}$ is the time at which user B's application receives the message.

In Loopix, message delays are the sum of delays at various points in the network. First, any outbound message sent by the user to the provider experiences a queuing delay d_Q based on the number of messages in the send queue. The delay between two successive messages in the queue being sent, d_p, is exponentially distributed with a rate parameter λ_p (see Table 1). Hence, a message's time spent in the send queue, d_Q, is a random variable with a Gamma distribution $\Gamma(n, \frac{1}{\lambda_p})$, where the shape parameter n denotes the number of messages in the queue ahead of our message msg.

Secondly, the payload message is held up at the ingress provider and each of the l mix nodes by an exponentially-distributed delay d_μ. Finally, the receiving user checks their inbox in fixed time intervals of Δ_{pull}, leading to a delay d_{pull} that is uniformly distributed between 0 and Δ_{pull}. Therefore the message delay in a Loopix network with l layers can be expressed as a sum of these components:

$$d_{msg} = d_Q + d_p + (l+1) \cdot d_\mu + d_{pull} \tag{2}$$

The above equation ignores processing and network delays. The Loopix paper demonstrates that these are negligible compared to the delays imposed by sensible rate parameters.

For a Poisson distribution with parameter λ, the expected mean is $1/\lambda$. The Gamma distribution $\Gamma(n, \frac{1}{\lambda_p})$ has the mean $\frac{n}{\lambda_p}$. For the pull interval, the expected mean delay is $\Delta_{pull}/2$.

Hence, the mean latency for Equation 2 is:

$$\text{mean}(d_{msg}) = \frac{n+1}{\lambda_p} + \frac{l+1}{\lambda_\mu} + \frac{\Delta_{pull}}{2} \tag{3}$$

When a source s wants to send a payload to a group by multicast, we define the *multicast latency* D to be the time from the initial message sending until all of the recipients U_{recv} have received the message:

$$D = \max_{u \in U_{recv}} (T_{recv,u}) - T_{send,s} \tag{4}$$

4.1 Naïve Sequential Unicast

In the simplest implementation of multicast, the source user s sends an individual unicast message to each of the recipients $u \in U_{recv}$ in turn. While the messages can travel through the mix network in parallel, their emission rate is bounded by the payload rate λ_p of the sender.

For a recipient group of size $|U_{recv}| = m-1$, the last message in the send queue will be behind $n = m-2$ other messages. Further, the last message will incur the same network delay and pull delay as all other unicast messages. The average delay for the last message therefore describes the multicast latency for when performing sequential unicast:

$$D_{unicast} = \frac{m-1}{\lambda_p} + \frac{l+1}{\lambda_\mu} + \frac{\Delta_{pull}}{2} = \mathcal{O}(m) \tag{5}$$

The mean delay $D_{unicast}$ therefore grows linearly with m. As we show in Section 6, sequential unicast is too slow for large groups with realistic choices of parameters (λ_p is typically set to less than one message per second).

Another problem with the sequential unicast approach is that the effective rate at which a user can send messages to the group is $\frac{\lambda_p}{m-1}$, as all copies of the first message need to be sent before the second multicast message can begin transmission.

One might argue that this problem can be addressed by increasing the payload bandwidth by increasing the value for λ_p. However, this would require similar adjustments to the rates for drop and loop messages to preserve the network's anonymity properties. As these parameters are fixed across all users, this would lead to a proportional increase in overall bandwidth used by the network. Moreover, the factor by which we increase λ_p would be determined by the largest group size we want to support. As a result, users participating in smaller groups would face an unreasonable overhead. This inefficiency particularly applies to users who mostly receive and only rarely send messages.

4.2 Naïve Mix-Multicast

An alternative approach shifts the multicast distribution of a message to mix nodes. In this scheme, the source chooses one mix node as the *multiplication node*. This node receives a single message from the source and creates $|U_{recv}| = m-1$ mix messages sent on to the other group members. A provider node would not be suitable as a multiplication node as it would learn about the group memberships of its users and their group sizes.

When the multiplication node receives such a multicast message, it inserts $m-1$ messages into its input buffer, one for each of the recipients, and processes them as usual. This provides optimal group message latency of $D = d_{msg}$ as there is no rate limit on messages sent by a mix node, and hence no queuing delay. However, this design has significant flaws.

First, a corrupt multiplication mix node can learn the exact group size $|U| = m$, in contravention of our threat model. This is undesirable as it may allow an attacker to make plausible claims regarding the presence or absence of communication within certain groups. Even without corrupting a node, an adversary can observe the imbalance between incoming and outgoing messages of a multiplication node.

The weakened anonymity properties could perhaps be mitigated with additional cover traffic that incorporates the same behaviour as the payload traffic. In particular, the cover traffic must model all possible group sizes. Allowing a group size of 200 requires cover traffic to multicast by factor 200 as well. However, this would significantly increase the network bandwidth requirements in the following mix layers, increasing the cost of operating the network.

Permitting message multiplication also opens up the risk of denial of service attacks: a malicious user could use the multicast feature to send large volumes of messages to an individual provider, mix node, or user, while requiring comparatively little bandwidth themselves.

Finally, supporting group multicast in a mix node requires the input message to contain $m-1$ payloads and headers, one for each outgoing message. As all outgoing messages must travel independently of each others they must be encrypted with different keys for their respective next hops. Otherwise, all outgoing messages share the same encrypted payload. This makes it trivial for an observer to identify the recipients of this group message. The only solution is to either increase the size of *all* messages in the system or enforce a very low limit on maximum group size.

In summary, naïvely performing message multiplication on mix nodes is not a viable option. However, a viable variant of this approach is possible by fixing the multiplication factor of messages to be a small constant (e.g. $p = 2$). We discuss this design in Section 5.4 where we present MultiSphinx.

5 Rollercoaster

We propose *Rollercoaster* as an efficient scheme for group multicast in Loopix. Rollercoaster distributes the multicast traffic over multiple nodes, arranged in a distribution graph. This not only spreads the message transmission load more uniformly across the network, but it also improves the balance

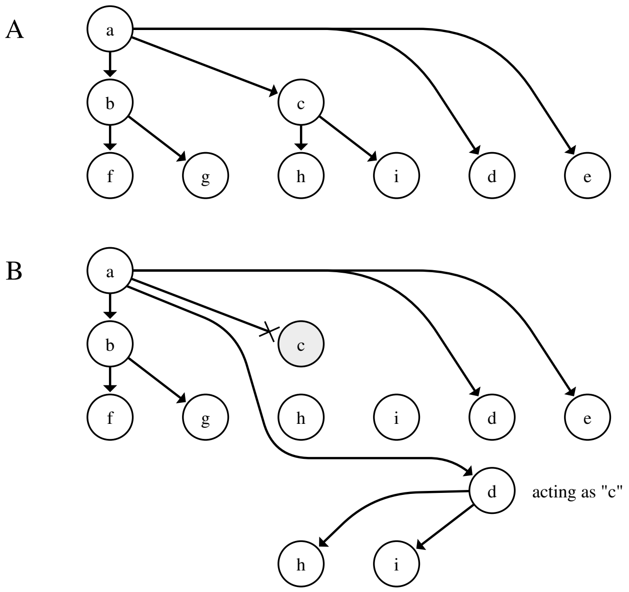


Figure 2: Message distribution graph for a group of size $m = 9$ and branching factor $k = 2$. *Graph A:* Expected delivery from source $s = a$. *Graph B:* The node c is offline and breaks delivery to h and i. Using the fault-tolerant variant the node d is assigned the role of c and delivers the payload to h and i.

of payload and cover traffic. Rollercoaster is implemented as a layer on top of Loopix, and it does not require any modifications to the underlying Loopix protocol (we discuss an optional protocol modification in Section 5.4).

As we have seen with naïve sequential unicast, messages slowly trickle from the source into the network as the source's message sending is limited by the payload rate λ_p. However, users who have already received the message can help distribute it: after the source has sent the message to the first recipient, both of them can send it to the third and fourth recipient concurrently. Subsequently, these four nodes then can send the message to the next four recipients, and so on, forming a distribution tree with the initial source at the root.

The distribution tree for a set of users U is structured in *levels* such that each parent node has k children at each level, until all recipients have been included in the tree. An example with eight recipients is shown in Figure 2. With each level the total number of users who have the message increases by a factor of $k+1$, which implies that the total number of levels is logarithmic in the group size $|U|$.

In this section we first detail the construction of Rollercoaster in Section 5.1. As a second step, Section 5.2 adds fault tolerance to ensure that the scheme also works when nodes are offline. Asymptotic delay and traffic properties are analysed in Section 5.3. Section 5.4 develops the *MultiSphinx* message format, which allows restricted multicast through designated mix nodes. Further optimisations to the scheme are briefly discussed in Section 5.5.

5.1 Detailed Construction

The Rollercoaster scheme is built upon the concept of a *schedule*. This schedule is derived deterministically from the source s, the total set of recipients U_{recv}, and the maximum branching factor k following Algorithm 1. First, a list U of all group members is constructed with the initial source at the 0-th index. The group size $|U|$ and branching factor k lead to a total of $\lceil \log_{k+1} |U| \rceil$ levels. In the t-th level the first $(k+1)^t$ members have already received the message. All of them send the message to the next w recipients, increasing the next group of senders to $(k+1)^{t+1}$. In the 0-th level only $U[0]$ (the initial sender) sends k messages to $U[1] \dots U[k]$.

Algorithm 1 The basic Rollercoaster schedule algorithm for a given initial source s, list of recipients U_{recv}, and branching factor k. The *schedule* contains a list for every level with a tuple $(sender, recipient)$ for each message to be sent.

procedure GENSCHEDULE(s, U_{recv}, k)
  $U \leftarrow [s] + U_{recv}$
  $L \leftarrow \lceil \log_{k+1} |U| \rceil$ ▷ number of levels
  $schedule \leftarrow []$
  for $t = 0$ until $L-1$ **do**
    $p \leftarrow (k+1)^t$ ▷ first new recipient
    $w \leftarrow \min(k \cdot p, |U| - p)$
    $R \leftarrow []$
    for $i = 0$ until $w-1$ **do**
      $idx_{sender} \leftarrow \lfloor \frac{i}{k} \rfloor$
      $idx_{recipient} \leftarrow p + i$
      $R[i] \leftarrow (U[idx_{sender}], U[idx_{recipient}])$
    $schedule[t] \leftarrow R$
  return $schedule$

In order to associate an incoming message with the correct source node and group of all recipients, all Rollercoaster payloads contain a 16 byte header as illustrated in Figure 3, in addition to the *Sphinx* packet header used by Loopix. Each group is identified by a 32-bit *groupid* shared by all group members. The 32-bit *nonce* identifies previously received messages, which becomes relevant with fault-tolerance (Section 5.2). The fields *source*, *sender*, and *role* refer to individual group members and have a 10-bit size, allowing groups with up to 1024 members. The *source* field indicates the original sender and is necessary to construct the distribution graph at the recipient. The fields *sender* and *role* are used by the fault tolerant variant in Section 5.2 for acknowledgement messages and to route around nodes that are offline. Field lengths can easily be increased or decreased as they do not have to be globally the same across all Loopix clients. Finally, the header contains a signature that is generated by the original source and covers the payload as well as all static header fields. It assures recipients that the message indeed originated from a legitimate group member and that they are not tricked by an adversary to start distributing a fake message to group

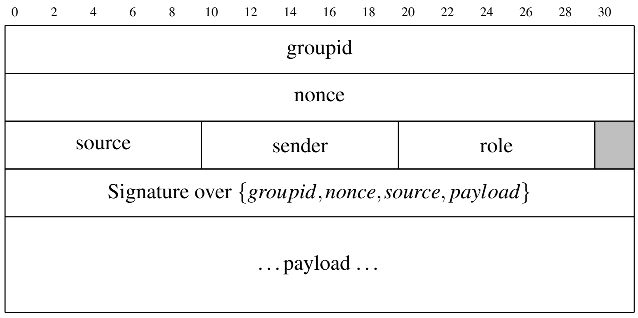


Figure 3: Payload header for the Rollercoaster scheme containing both the fields for the minimal scheme and the fields necessary for the fault-tolerance variant and further optimisations.

members. The *sender* and *role* fields are not covered by the signature, allowing nodes that are not the original source to modify these fields without invalidating the signature.

5.2 Adding Fault Tolerance

The basic Rollercoaster scheme of Section 5.1 fails when users are offline and cannot perform their role of forwarding messages. In this case, one or more recipients in later levels would not receive the message until their parent node returns online. The risk of this approach becomes apparent when looking at the graph in Figure 2B, where a single unavailable node causes message loss for its entire subtree. In principle, the responsibility for forwarding messages could be delegated to the provider nodes, which are assumed to always be online. However, we consider this approach not to be desirable as the adversary could learn about the group membership by compromising a provider.

Rollercoaster with fault-tolerance achieves reliable delivery through *acknowledgement* (ACK) replies to the source and reassignment of roles. When the source sends a message it sets timeouts by which time it expects an acknowledgement from the recipient and each of its children. The individual timeouts account for the number of hops and the expected delays at each hop due to mix node delays and messages waiting in send queues. ACKs are sent through the mix network like any other unicast message. When receiving an ACK from a node, the source marks the sending node as delivered. Choosing the source as the main coordinator is reasonable as it has the strongest incentive for ensuring delivery of all messages. Loopix allows a high rate of messages received by users, so it is not a problem if one user receives a large number of ACKs.

The source responds to a timeout by sending the message to a different node. For this, each node maintains a list of most-recently-seen nodes based on received messages and chooses one from it heuristically. The source itself is part of that list as the ultimate replacement node. A replacement node is only necessary when the failing node would have forwarded the message to others, i.e. when it is not a leaf node of the distribution tree (see Algorithm 10 in Appendix B). Independently of this and in case that the message did not reach the intended recipient due to message loss, a *retry message* is sent (with exponential back-off) to the failed node again with its own timeout.

We start the timeouts associated with a message when the underlying Loopix implementation sends the message to the provider, so that the timeouts do not need to include the sender's queuing delay. Since the sender knows the global rate parameters λ_p and λ_μ, it takes these into account when determining timeouts. The timeout may further be adjusted based on the network configuration and application requirements.

The fault-tolerance mechanism makes use of the message fields *source*, *sender*, and *role* shown in Figure 3. The *source* field remains unchanged as the message is forwarded because it is required for constructing the schedule at each node. It also indicates the node to which the ACK should be sent. The *sender* field is updated when forwarding a message or sending an ACK and used by the recipient to update their list of most-recently-seen nodes. The *role* field indicates the role that the receiving node should perform, usually their natural identity. However, when a node is offline, another node might be assigned its role, i.e. its position in the distribution tree. In this case, the role field indicates the node as which the recipient should act. Retry messages to failed nodes have an empty role field, because the role has already been reassigned.

On receiving any payload message *msg*, the recipient node hands over the payload to the application and reconstructs the schedule using *msg.source*, *msg.groupid*, and *msg.nonce*. For every child node of *msg.role* in the schedule, the node enqueues a message for the respective recipient, making sure to update *msg.role*. The ACK reply is enqueued after the payload messages so that no ACK is sent if a node goes offline before forwarding a message to all of its children in the distribution tree.

ACK messages contain the *groupid*, *nonce*, *source*, and *role* fields of the original message and an updated *sender* field, which allow the recipient of the ACK (i.e., the source) to identify and cancel the corresponding timeout. The sender adds a signature covering *all* header fields to ensure that the ACK message cannot be forged. When an ACK is not received on time, the message is sent to a different node as described above.

If the connection between a user and their provider is interrupted, we rely on the fact that Loopix allows users to retrieve received messages from their inbox later. The user's software notices a loss of connection and pauses timeouts until it has had a chance to check the inbox on the provider again.

After a long offline period, a node's inbox may contain a large backlog of messages that were received by the provider while the user was offline. When a node comes back online, it treats this backlog differently from messages received while online: for any messages received while offline, a node only delivers the payloads to the application, but it does not send ACK messages or forward messages to other nodes. Here the node avoids doing unnecessary work for messages where the timeout is likely to have already expired.

Algorithm 8 in Appendix B describes the behaviour of the fault-tolerant variant in detail.

5.2.1 Eventual Delivery and Byzantine Fault Tolerance

The fault-tolerant variant of Rollercoaster assumes that the source node acts honestly and does not disconnect permanently (but can do so intermittently). This is reasonable as the sending user has high incentive to see through the delivery of their message. We prove eventual delivery under this assumption in Appendix C . An application might provide the user with a suitable user interface that shows the delivery process.

Proof sketch: Everyone who does not ACK the payload will eventually receive it directly from the source, and will read it from their inbox when they return online. This works even in the presence of malicious nodes that acknowledge a message without forwarding it, since the source has individual timeouts for each group member. Therefore, the source will detect when a node's children do not send ACKs.

However, the source node might be disconnected permanently. To nevertheless guarantee eventual delivery, every group member can periodically pick another group member at random and send it a hash of the message history it has seen so far (ordered in a deterministic way so that two users with the same set of messages obtain the same hash). If the recipient does not recognise the hash, the users run a reconciliation protocol [14] to exchange any messages that are known to only one of the users. Such a protocol provably guarantees that every user eventually receives every message, even if some of the users are Byzantine-faulty, provided that every user eventually exchanges hashes with every other user [14].

5.3 Exploring Delay and Traffic

We first analyse the expected multicast latency of Rollercoaster without fault tolerance by considering the levels of the distribution tree, as illustrated in Figure 2. The expected multicast latency $D_{rollercoaster}$ is determined by the longest message forwarding paths $C_1, C_2, \ldots$. Each such path is defined as $C = e_0, \ldots, e_{|C|-1}$ where e_i is a edge from a node on level i to a node on level $i+1$. We call these edges *one-level edges*. The number of levels of the schedule generated by Algorithm 1 is $L = \lceil \log_{k+1} |U| \rceil$ as discussed in §5.1. Hence, no path is longer than L. An example of a longest path is $C = (a,b)(b,g)$ in Figure 2. The mean message delay when traversing each edge of the graph is $\bar{d}_{msg} = \bar{d}_Q + \bar{d}_t$, where $\bar{d}_Q$ is the mean queuing delay and $\bar{d}_t = \bar{d}_p + (l+1) \cdot \bar{d}_\mu + \bar{d}_{pull}$ is the message's mean travel time through the network, as in (2). Since each node sends no more than a total of k messages to the directly subsequent level, the expected queuing delay for the last message is $\bar{d}_Q = \frac{k-1}{\lambda_p}$.

Scheme	Latency D	Packet size overhead
Naïve Unicast	$\mathcal{O}(m)$	–
Naïve Multicast	$\mathcal{O}(1)$	$\mathcal{O}(m)$
Rollercoaster	$\mathcal{O}(\log m)$	$\mathcal{O}(1)$

Table 2: Overhead of the presented multicast schemes in terms of group multicast delay and packet size overhead.

However, there are also edges from a node on level i to a node on level $i+j$ where $j > 1$. One example is (a,d) in Figure 2. Messages from level i to level $i+1$ are sent before any messages that skip levels, and therefore any level-skipping messages may experience higher queuing delay before they are sent. Concretely, the edges from level i to level $i+j$ will incur an additional expected queuing delay of at most $(j-1) \cdot \bar{d}_Q$ compared to one-level edges. At the same time, these edges save $j-1$ hops, which would have incurred both a queuing delay $\bar{d}_Q$ and a travel time $\bar{d}_t$ each. Hence, the time saved by the reduced hop count outweighs the extra queuing delay.

Thus, the expected time for a message to be received by all nodes is determined by the longest path consisting of only one-level edges, with a queuing delay of $\bar{d}_Q = \frac{k-1}{\lambda_p}$ at each hop:

$$D_{rollercoaster} = L \cdot (\bar{d}_Q + \bar{d}_t) = \lceil \log_{k+1} m \rceil \cdot \bar{d}_{msg} \tag{6}$$

Hence, the group multicast latency is logarithmically dependent on the group size m and contains a multiplicative factor that equals the time to send a single message after being queued behind at most k messages.

When a node is offline, it will only be able to receive messages when it comes online and queries its inbox. In case the offline node is a forwarding node, the source will detect the lack of an ACK after the timeout expired. In this case the latency penalty for the children of the failed node is the timeout of the parent node, which is typically proportional to the expected delivery time.

5.4 p-Restricted Multicast with MultiSphinx

As specified so far, Rollercoaster uses the unmodified Loopix protocol. However, even though Rollercoaster spreads the work of sending a multicast message more evenly across the network than sequential unicast, payload messages and ACKs are still demanding for nodes' send queues.

In this section, we consider a modification to the Loopix protocol that further improves multicast performance: namely, we allow some mix nodes to *multiply* one input message into multiple output messages, which may be sent to different

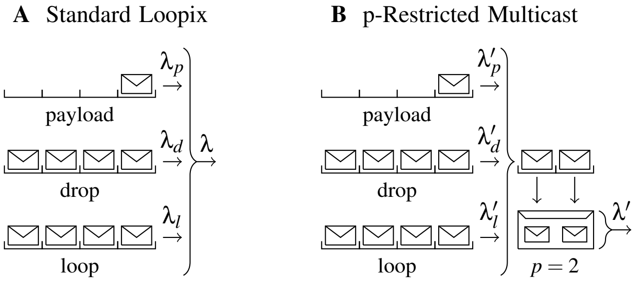


Figure 4: Standard Loopix (A) sends out a message if any of its Poisson processes triggers, so the rate of messages sent is $\lambda = \lambda_p + \lambda_d + \lambda_l$. In p-restricted multicast (B) these Poisson processes are still independent, but the node has an extra layer that awaits p messages, which are then wrapped into a MultiSphinx message. The sender can increase λ'_p to $p\lambda_p$ (same for λ'_d, λ'_l) while keeping $\lambda' = \lambda$.

recipients. The naïve mix-multicast we considered in Section 4.2 allows arbitrary multiplication factors. Here we show how to make mix-node-supported multicast safe by restricting the multiplication factor to a fixed constant p. We call this approach *p-restricted multicast* where clients can send p messages inside one MultiSphinx package; with $p = 1$ this scheme is identical to the regular Rollercoaster.

In p-restricted multicast, only mix nodes in one designated layer may multiply messages. In our design, we perform multiplication in the middle layer (layer 2 of 3) and we refer to these mix nodes as *multiplication nodes*. To ensure unlinkability of mix nodes' inputs and outputs, *every* message processed by a multiplication node must result in p output messages, regardless of the message type or destination. Mix nodes in other layers retain the standard one-in-one-out behaviour of Loopix. Since layer 3 of the mix network needs to process p times as many messages as the earlier layers, layer 3 should contain p times as many mix nodes as layers 2.

This paper uses the parameter p for p-restricted multicast and k for the schedule algorithm. These can be chosen independently of each other. However, for simplicity and practical interdependence we often set both to the same value $k = p$.

Effectively, p-restricted multicast allows p messages to different recipients to be packaged as a single message up to p times the size. Sending fewer but larger messages allows for lower power consumption on mobile devices, as discussed in our application requirements (§2). We show in our evaluation in §6.5 that p-restricted multicast allows choosing much larger λ values while maintaining low latency.

5.4.1 The MultiSphinx message format

Loopix encodes all messages using the *Sphinx* message format [11], which consists of a header M containing all metadata and an encrypted payload δ. Using the header, each mix node n_i derives a secret shared key s_i. Due to the layered encryption of the header and payload, an adversary cannot correlate incoming and outgoing packets when observing mix nodes. Our construction is based on the improved Sphinx packet format [15] which uses authenticated encryption (AE). In particular, we use a stream cipher C in an encrypt-then-MAC regime and require that without the knowledge of the key, the generated ciphertext is indistinguishable from random noise (which is believed to be the case for modern ciphers such as AES-CTR). Every hop verifies integrity of the entire message to prevent active tagging attacks. The improved Sphinx packet format satisfies the ideal functionality of Sphinx [16]. The per-hop integrity checks of the entire message come at the cost of lacking support for anonymous reply messages, but these are not used by Loopix.

Sphinx assumes that each input message to a mix node results in exactly one output message. In order to support p-restricted multicast we introduce the *MultiSphinx* message format, which can wrap p messages. A MultiSphinx message is unwrapped at a designated mix node, and split into p independent messages. For anyone other than the designated multiplication node, MultiSphinx messages are indistinguishable from regular Sphinx packets. We now describe the *MultiSphinx* design for $p = 2$ by describing the creation and processing of these messages. The detailed construction and processing is formalised in Appendix A.2.

For $p = 2$, the sender waits until its message queues (payload, drop, loop) have released two messages. The sender then combines their payloads δ_A, δ_B and recipients U_A, U_B into a single message that is inserted into the mix network, as shown in Figure 4. As we want to fit both payloads and two headers into our message to the multiplication node, $|\delta_A|$ and $|\delta_B|$ must be smaller than the global Sphinx payload size.

The combined message is sent via a mix node n_0 in the first layer to the designated multiplication node n_1, where its inner messages are extracted and added to its input buffer. The inner message containing δ_A will be processed by n_1 and routed via $n_{2,A}$ to the recipient n_A (and similarly for B). The multiplication node derives the secret key s_1 from the incoming message's header and additional secret keys $s_{1,A}, s_{1,B}$ from the headers of the inner messages. We omit provider nodes.

The sender first computes all secret keys. Using these secret keys it encrypts the payloads δ_A, δ_B between the recipients and the multiplication node. However, the resulting encrypted payloads are smaller than the regular Sphinx payload lengths.

To ensure all messages have the same size, we use a pseudorandom function (PRF, e.g. HMAC) ρ to add padding to the encrypted payloads $\delta_{1,A}$ and $\delta_{1,B}$. ρ is keyed with the shared secret s_1 and the payload index (`A` or `B`) so that the padding is unique. The resulting payloads have the format $\delta'_{1,A} = \delta_{1,A} \parallel \rho(s_1 \parallel \texttt{A})$ (and similarly for B). Now the sender computes the headers and MACs along the path from the multiplication node to the recipients by simulating the decryption of the payload at each step. This results in two Sphinx headers $M_{1,A}$ and $M_{1,B}$. Finally, we create the message for the path from the sender to the multiplication node using the regular Sphinx construction. We set the payload of that message to the concatenation $\delta_{combined} = M_{1,A} \parallel \delta_{1,A} \parallel M_{1,B} \parallel \delta_{1,B}$. Appendix A.2 contains pseudocode for this construction.

The processing of incoming messages at the multiplication node differs from other nodes. First, the payload is decrypted and split into the message headers and payloads. Then, the payloads are deterministically padded using the PRF ρ as described above. To ensure that the messages are hard to correlate, they are added to the node's input buffer, decrypted again (now deriving secrets $s_{1,\{A,B\}}$), and delayed independently as defined by their individual delay parameter.

5.4.2 Anonymity of MultiSphinx

All MultiSphinx messages (before and after the multiplication node) have the same header length and payload length as regular Sphinx messages. Sphinx headers do not leak the number of remaining hops and the ciphertext is indistinguishable from random noise. Therefore, MultiSphinx messages are indistinguishable from regular Loopix messages(Lemma 9, Appendix D). At the same time, the multiplication node maintains the unlinkability between the incoming messages and outgoing messages as these are delayed independently.

An adversary might also corrupt mix nodes. Even in this case they do not gain advantage over regular Sphinx message with regards to sender and recipient anonymity and unlinkability(Theorem 13, Appendix D). These results also hold for active adversaries with the capabilities from the original Loopix paper(Theorem 19, Appendix D).

If an adversary controls a p-restricted multiplication node and c_3 of the n_3 mix nodes of the third layer, they can trace some messages from their multiplication to their delivery at providers. On the basis that the p recipients of a MultiSphinx message are likely to be members of the same group, the adversary then has a chance to guess that any two of the users from these providers share a group membership. In Theorem 17 (Appendix D) we show that the probability of correctly guessing two group members given a group message is less than $(1-(\frac{n_3-c_3}{n_3})^{p-1}) \cdot \frac{|\mathcal{P}|^2}{|\mathcal{U}|^2}$ if all $|\mathcal{U}|$ users are evenly distributed among $|\mathcal{P}|$ providers. This attack is prevented if the multiplication node *or* all but one of the chosen nodes in the third layer are trustworthy. (In contrast, standard Loopix requires only that any mix node on the message path is trustworthy.) MultiSphinx does not leak any information regarding group sizes(Theorem 15). Appendix Dcontains theorems and proofs for our claims.

In addition to these properties, it is possible to achieve sender anonymity by first forwarding the message to a trusted group member. The sender can prove its membership through a shared group secret. We leave receiver anonymity and unlinkable group membership for future work.

5.5 Further Optimisations

The schedule computed by GENSCHEDULE in Algorithm 1 delivers the first messages to the nodes at the beginning of the provided recipient list U_{recv}. These nodes will always act as the forwarding nodes. To better balance these among all group members, one can shuffle the list based on a *nonce* value that is part of the message. This variant is described in Algorithm 2. As the GENSCHEDULERAND algorithm is still deterministic and the nonce is part of the Rollercoaster header, each node reconstructs the same schedule.

Algorithm 2 Creating a pseudorandomized schedule for a given nonce

```
procedure GENSCHEDULERAND(s, U_recv, k, nonce)
    R ← NEWPRNG(nonce)
    U'_recv ← R.shuffle(U_recv)
    return GENSCHEDULE(s, U'_recv, k)
```

Further optimisation is possible if different sub-groups display different levels of activity and connectivity. For example, if there is a small, active sub-group communicating while the rest of the group remains passive, it is more important for messages to travel faster between active nodes to support swift, effective collaboration. Active nodes can often be assumed to be more likely to be online. Agreeing on the full order is no longer possible through a single nonce value. However, the source can randomly compute a subset of all schedules, evaluate the generated schedule against its information about the group members, and choose one that creates a schedule with the most desirable properties.

6 Evaluation

For the empirical evaluation we developed a mix network simulation tool that provides fully reproducible results. First, we discuss the behaviour and results of the Rollercoaster scheme in an ideal scenario where all participants are online throughout. Second, we discuss the impact of offline nodes and how this is addressed by the fault-tolerant variant of Rollercoaster. Finally, we discuss the impact of multi-group membership, sending multiple messages at once, and p-restricted multicast.

6.1 Methodology

Since the real-world performance of Loopix has been practically demonstrated [3] we run a simulation instead of an experiment on a real network. This provides clear practical advantages: First, it allows us to eliminate external influences such as network congestion due to unrelated traffic or CPU usage by other processes. Second, the simulated time can run faster than real-time, allowing us to gather significantly more results using less computational resources. Third, it makes monitoring and categorising traffic easier as packets and node state can be inspected. Finally, by initialising the PRNG with a fixed seed, the results of this paper are fully reproducible.

The simulator runs the entire mix network on a single machine, with nodes communicating through shared memory simulating a network. It instantiates objects for each participating user, provider, and mix node. All objects implement a `tick()` method in which they process incoming messages and mimic the designed node behaviour such as delaying and forwarding packets. As we are primarily interested in the traffic behaviour, no actual encryption is performed. The original Loopix paper has shown that the queuing time and per-hop delays dominate the message delay, and that CPU time for cryptographic operations is insignificant in comparison. Similarly, the network delay is negligible.

For the final evaluation we ran 276 independent simulations, covering more than 992,160 hours of simulated node time in less than 209 hours of CPU core time. This is a relative speed up by factor 4500× compared to a real network experiment of the same scope. In every simulation step the application (see below) measures the message latency d_{msg} of each delivered message between the original source and each recipient. We verified that our simulator behaves faithfully to the Loopix implementation by reproducing a latency distribution graph from the original paper [3, Figure 11], as shown in Appendix E . Our simulator is implemented in less than 2,000 lines of Python code including tests and is available as an open-source project.[2]

The network simulator assigns 16 users to each provider. We set the Loopix rates $\lambda_p = \lambda_d = \lambda_l = 2/\text{s}$ for the client nodes and the delay rate $\lambda_\mu = 3/\text{s}$. Hence, the overall sending rate of the clients is $\lambda = 6/\text{s}$. This meets the requirement $\lambda/\lambda_\mu \geq 2$ that is suggested by the Loopix paper [3, p. 1209, $\lambda_\mu = \mu$]. The mix network consists of 3 layers containing 3 mix nodes each (mix loop injection rate $\lambda_M = 2/\text{s}$). All simulations are run with a granularity of 10 ms per tick. The simulated time span for all configurations is 24 h.

The application behaviour is modelled by a Poisson process. On average every 30 s one of the online nodes sends a single message to all other group members. We account for participation inequality [8] by dividing the group using an 80/20 rule: 20% of users in the group send 80% of all messages, and vice versa.

6.2 Results with All Users Online

For a group of size 128, the average latency is reduced from 34.9 s in sequential unicast to 7.0 s (8.3 s for group size 256) in Rollercoaster with $k = p = 2$. This fulfils our application requirements that were derived from the user study concerning delay in collaborative applications [7]. The results are compatible with our analytical results as discussed in Section 5.3. For Rollercoaster not only is the average latency low, but most of the latency distribution falls within fairly tight

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

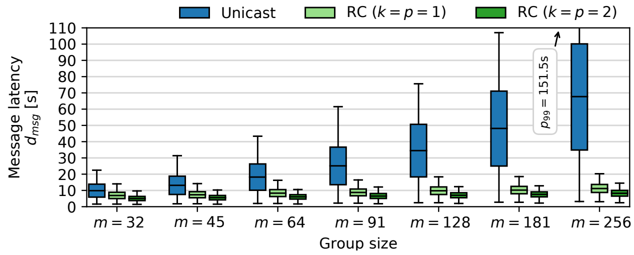


Figure 5: This box plot shows the distributions of message latency d_{msg} for increasing group sizes for the strategies *naïve sequential unicast* and *Rollercoaster (RC)*. The Rollercoaster strategies show different k and p parameters. The boxes span from the first quartile to the third quartile (middle line is the median) and the whiskers indicate the 1st and 99th percentile.

bounds – that is, very large latencies are rare. Figure 5 shows the latency achieved by the Rollercoaster scheme with and without p-restricted multicast for different percentiles and compares them to unicast. For a group with 128 members the 99th percentile p_{99} for Rollercoaster is 12.3 s (p_{90}: 9.9 s) whereas in unicast it is 75.6 s (p_{90}: 60.8 s). We provide detailed histograms in Appendix H .

6.3 Results for Fault-Tolerance Scenarios

The evaluation of the fault tolerance properties requires a realistic model of connectivity of mobile devices. For this we processed data from the *Device Analyzer* project [17] that contains usage data of volunteers who installed the Android app. The online/offline state of a device is derived from its trace information regarding network state, signal strength, and airplane mode. We limit the dataset ($n = 27790$) to traces that contain connectivity information ($n = 25618$), cover at least 48 hours ($n = 20117$), and have no interval larger than 12 hours without any data ($n = 2772$).

Inspecting the traces we identify three archetypes of online behaviour. The first group is online most of the time and is only interrupted by shorter offline segments of less than 60 minutes. Members of the second group have at least one large online segment of > 8 hours and are on average online 50% or more of the time. Finally, the third group is online less than 50% of the time with many frequent changes between online and offline states. As the dataset is more than five years old we decided to use the characteristics of these groups to build a model. Using a model allows us to extrapolate offline behaviour into scenarios with increased connectivity. In the model following the parameters of the original dataset, the fraction of all users' time spent online is 65%. In a second and third model with increased connectivity, we increase this percentage to 80% and 88%, respectively, while preserving the behaviour of the archetype groups. The generated models are visualised in Appendix F .

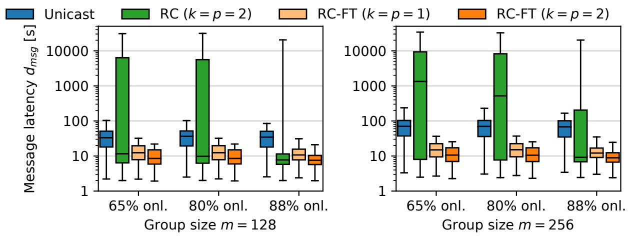


Figure 6: The distribution of message latency d_{msg} for different offline scenarios. From left to right the strategies are Unicast, Rollercoaster without fault-tolerance (RC), and Rollercoaster with fault-tolerance (RC-FT). Boxes and whiskers as in Figure 5.

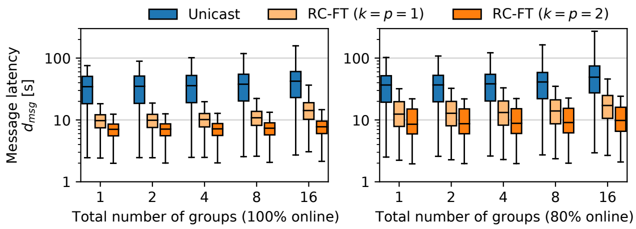


Figure 7: Message latency d_{msg} for an increasing number of groups for 128 users (every user is member of every group). Boxes and whiskers as in Figure 5.

For our discussion of offline behaviour we refine our previous definition of message latency d_{msg}: we ignore all latencies where the intended recipient was offline when the message was placed into their inbox by the provider node. This change has the practical benefit of excluding outliers. More importantly, fast delivery to an offline user has no real-world benefit. Instead, a good multicast algorithm should optimise the delivery to all nodes that are active and can actually process an incoming message. The source might go offline at any time regardless of outstanding messages.

Without fault tolerance, the presence of offline nodes greatly increases the 99th percentile (p_{99}) for Rollercoaster (RC) to more than 10,000 s for a group of 128 members. The fault-tolerant variant (RC-FT) reduces the 99th percentile to less than 21.9 s (p_{90}: 18.0 s). In unicast p_{99} latency is 103.3 s (p_{90}: 61.9 s). Figure 6 shows that the fault-tolerant variant generally outperforms unicast at various percentiles. We provide detailed histograms in Appendix H .

6.4 Multiple Groups and Message Bursts

Users might be part of multiple groups, which increases their burden of distributing messages. In this evaluation we assign 128 users to a growing number of groups. Figure 7 shows that the number of group memberships has little impact on Rollercoaster's performance both for online and offline scenarios.

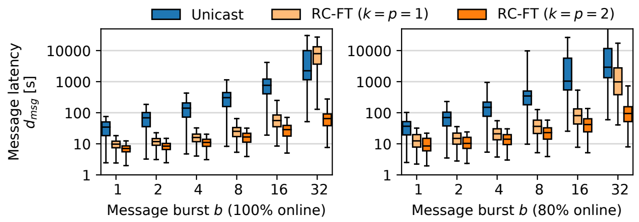


Figure 8: Message latency d_{msg} for applications that send b messages at once. The group size is $m = 128$. Boxes and whiskers as in Figure 5.

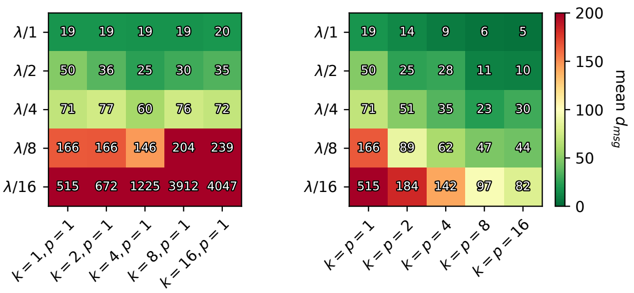


Figure 9: Heatmaps showing the mean message latency for reduced sending rates (y-axis) and different Rollercoaster parameters (x-axis). In the left graph only the logical branch factor k is increased. In the right graph the multicast factor p is increased at the same time. Group size is $m = 128$ and 80% online. More simulations in Appendix G.

Similarly, users might be sharing large payloads (e.g. images) or sending multiple updates at once. Both translate into many messages being scheduled for distribution at the same time, which risks overwhelming the payload queue. Figure 8 shows that Rollercoaster can handle many more messages sent in bursts than unicast. We observed that with unicast and some Rollercoaster configurations some nodes had indefinitely growing send buffers as the simulation progressed. The effect of this can be seen by the higher message latencies for $b = 32$. This threshold is higher for p-restricted multicast.

6.5 Results for p-Restricted Multicast

In this evaluation we show that p-restricted multicast allows us to drastically lower the sending rates $\lambda_{\{p,d,l\}}$ of the clients while achieving similar performance. A low sending rate is desirable as it allows the radio network module to return to standby and thereby saving significant battery energy on mobile devices (see §2). Figure 9 shows that just increasing k (left) has negligible or even negative impact, while increasing k and p together (right) allows for lower sending rate λ while maintaining good enough performance. We decrease λ_μ accordingly to maintain the $\lambda/\lambda_\mu \geq 2$ balance (see §6.1) which increases the per-hop delays.

7 Related Work

Previous research on efficient anonymity networks achieves strong security goals, high efficiency, scalability, and offline support. However, decentralised low-latency group multicast while guaranteeing the strongest privacy guarantees against a global adversary has not yet received due attention.

Work based on *Dining Cryptographer networks (DC-nets)* [18] is inherently broadcast-based as the round results are shared with all nodes. These designs generally provide sender anonymity and impressive functionality. However, the required synchronisation and communication overhead render them unsuitable for low latency applications. As the rounds depend on the calculations of all clients, they can be susceptible to interference by malicious participants. The Xor-Trees by Dolev et al. [19] achieve efficient multicast, but only in the absence of an active attacker. Dissent [20] can provide protection against such active attacks. However, its design does not scale as well as Loopix due to its need to broadcast messages to all clients, and not just the intended group of recipients.

Circuit-based onion routing networks such as Tor [1] establish long-living paths through multiple hops. All messages from and to the client are transmitted via the same path with every node peeling off the outer-most encryption layer. They are arguably the most widely deployed and accessible class of anonymity network designs. While the onion path approach allows for low latency communication, it is known to be vulnerable against global adversaries performing *traffic analysis* attacks [2, 21]. Most mainstream designs consider one-to-one communication, but there is interesting work on building multicast trees using onion-routing techniques. Examples are AP3 [22], M2 [23], and MTor [24]. When facing a global adversary, they share similar vulnerabilities to Tor.

Multicast in friend-to-friend overlays as in VOUTE [25,26] share a similarity with our work as trusted peers help with message distribution. However, to our knowledge, there are no practical implementations with performance similar to Loopix. Using real-world trust relationships together with Rollercoaster for inter-group communication is an interesting direction for future work.

The recent Vuvuzela design [27] cleverly leverages dead drops and cover traffic to achieve strong metadata privacy while maintaining a high throughput of messages. Pursuing the goal of limiting network bandwidth use results in delays of up to 10 minutes to initiate a call and more than 30 seconds latency for messages, which we consider too large for many collaborative applications. Its privacy guarantees can be limited in the case of an active attacker with a priori suspicion of a certain group of users communicating.

Work based on *private information retrieval* (PIR) such as Pung [28] and Talek [29] allows for low-latency group communication with strong security guarantees. However, these systems are not decentralised and rely on the availability of high-spec servers. Moreover, their latency scales with the total number of users n rather than the group sizes.

We note that our evaluation differs from the standard methods in similar papers [3, 20, 27] using real servers and networks. Since it is already established that the performance of Loopix is viable in practise, we can build on top of this and focus on more inspectable and reproducible evaluations through deterministic simulation.

The Shadow project [30] can simulate actual anonymity network implementations in a network topology on a single machine. With extensive modelling options the network and user behaviour can be modelled deterministically. However, since the application binaries remain black-boxes it cannot guarantee complete deterministic behaviour. White-box simulators such as Mixim [31] calculate the entropy as messages pass through the system.

Many multicast systems use distribution trees [32–36]. However, to our knowledge, these protocols have not yet been applied in the context of mix networks, where the limited send rate and artificial message delays introduce particular challenges not considered by existing multicast protocols.

8 Conclusion

In this paper we have presented an efficient scheme for multicast in mix networks named *Rollercoaster*. Compared to the sender of a message naïvely sending it to all other group members by unicast, our scheme significantly lowers the time until all group members receive the message. For a group of size $m = 128$, Rollercoaster is faster by a factor of 5, reducing the average delay from 34.9 s to 7.0 s and reducing the 99th percentile from 75.6 s to 12.3 s. We do this by involving more users than just the original sender in the process of disseminating a message to group members. This also reduces the asymptotic growth of the expected delay to $\mathcal{O}(\log m)$. A key ingredient for this is the deterministic GENSCHEDULE algorithm that allows users to share plans for message distribution using a single nonce.

Faced with the challenge of unreliable and offline nodes, we have introduced a variant of our algorithm that allows acknowledgement and retry of message delivery as well as reassignment of tasks from offline to online users. In the failure-free case, it adds a constant message overhead that does not worsen the results measured. When nodes are offline it significantly improves reliability and delays.

Our simulation tool enabled us to obtain reproducible and inspectable performance measurements. The low cost of simulation enabled us to efficiently explore the behaviour of many system configurations with a large number of users.

In future work we plan to implement and run collaborative applications and group messaging protocols on a network using Rollercoaster. We also hope to extend Rollercoaster with facilities to add or remove members of a group.

Acknowledgements

We thank Steven J. Murdoch, Killian Davitt, and our anonymous reviewers for the helpful discussions and their valuable input. Daniel Hugenroth is supported by a Nokia Bell Labs Scholarship and the Cambridge European Trust. Martin Kleppmann is supported by a Leverhulme Trust Early Career Fellowship, the Isaac Newton Trust, Nokia Bell Labs, and crowdfunding supporters including Ably, Adrià Arcarons, Chet Corcos, Macrometa, Mintter, David Pollak, RelationalAI, SoftwareMill, Talent Formation Network, and Adam Wiggins. Alastair R. Beresford is partially supported by EPSRC [grant number EP/M020320/1].

References

[1] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," tech. rep., Naval Research Lab Washington DC, 2004.

[2] S. J. Murdoch and G. Danezis, "Low-cost traffic analysis of Tor," in *2005 IEEE Symposium on Security and Privacy*, pp. 183–195, IEEE, 2005.

[3] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The Loopix anonymity system," in *26th USENIX Security Symposium*, pp. 1199–1216, 2017.

[4] M. Kleppmann, S. A. Kollmann, D. A. Vasile, and A. R. Beresford, "From secure messaging to secure collaboration," in *26th International Workshop on Security Protocols*, pp. 179–185, Springer, 2018.

[5] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, "Evaluating CRDTs for real-time document editing," in *11th ACM Symposium on Document Engineering*, pp. 103–112, ACM, Sept. 2011.

[6] A. Pfitzmann and M. Hansen, "A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management," Aug. 2010. v0.34, `http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf`.

[7] C.-L. Ignat, G. Oster, O. Fox, V. L. Shalin, and F. Charoy, "How do user groups cope with delay in real-time collaborative note taking," in *14th European Conference on Computer Supported Cooperative Work*, pp. 223–242, Springer, Sept. 2015.

[8] J. Nielsen, "The 90-9-1 rule for participation inequality in social media and online communities," 2006. `https://www.nngroup.com/articles/participation-inequality/`.

[9] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4G LTE networks," in *10th International Conference on Mobile Systems, Applications, and Services*, pp. 225–238, 2012.

[10] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, vol. 24, no. 2, 1981.

[11] G. Danezis and I. Goldberg, "Sphinx: A compact and provably secure mix format," in *30th IEEE Symposium on Security and Privacy*, pp. 269–282, IEEE, 2009.

[12] A. Serjantov, R. Dingledine, and P. Syverson, "From a trickle to a flood: Active attacks on several mix types," in *International Workshop on Information Hiding*, pp. 36–52, Springer, 2002.

[13] S. Deering, "Host extensions for IP multicasting," STD 5, RFC Editor, August 1989. `http://www.rfc-editor.org/rfc/rfc1112.txt`.

[14] M. Kleppmann and H. Howard, "Byzantine eventual consistency and the fundamental limits of peer-to-peer databases," *arXiv preprint arXiv:2012.00472*, 2020.

[15] F. Beato, K. Halunen, and B. Mennink, "Improving the Sphinx mix network," in *International Conference on Cryptology and Network Security*, pp. 681–691, Springer, 2016.

[16] C. Kuhn, M. Beck, and T. Strufe, "Breaking and (partially) fixing provably secure onion routing," *arXiv preprint arXiv:1910.13772*, 2019.

[17] D. T. Wagner, A. Rice, and A. R. Beresford, "Device analyzer: Understanding smartphone usage," in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pp. 195–208, Springer, 2013.

[18] D. Chaum, "Security without identification: Transaction systems to make Big Brother obsolete," *Communications of the ACM*, vol. 28, no. 10, pp. 1030–1044, 1985.

[19] S. Dolev and R. Ostrobsky, "Xor-trees for efficient anonymous multicast and reception," *ACM Transactions on Information and System Security*, vol. 3, no. 2, pp. 63–84, 2000.

[20] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson, "Dissent in numbers: Making strong anonymity scale," in *10th USENIX Symposium on Operating Systems Design and Implementation*, pp. 179–182, 2012.

[21] G. Danezis and A. Serjantov, "Statistical disclosure or intersection attacks on anonymity systems," in *International Workshop on Information Hiding*, pp. 293–308, Springer, 2004.

[22] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach, “AP3: Cooperative, decentralized anonymous communication,” in *11th ACM SIGOPS European workshop*, p. 30, ACM, 2004.

[23] G. Perng, M. K. Reiter, and C. Wang, “M2: Multicasting mixes for efficient and anonymous communication,” in *26th IEEE International Conference on Distributed Computing Systems*, pp. 59–59, IEEE, 2006.

[24] D. Lin, M. Sherr, and B. T. Loo, “Scalable and anonymous group communication with MTor,” *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 2, pp. 22–39, 2016.

[25] S. Roos, M. Beck, and T. Strufe, “Anonymous addresses for efficient and resilient routing in F2F overlays,” in *35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, IEEE, 2016.

[26] S. Roos, M. Beck, and T. Strufe, “Voute-virtual overlays using tree embeddings,” *arXiv preprint arXiv:1601.06119*, 2016.

[27] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich, “Vuvuzela: Scalable private messaging resistant to traffic analysis,” in *25th Symposium on Operating Systems Principles*, pp. 137–152, 2015.

[28] S. Angel and S. Setty, “Unobservable communication over fully untrusted infrastructure,” in *12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 551–569, 2016.

[29] R. Cheng, W. Scott, E. Masserova, I. Zhang, V. Goyal, T. Anderson, A. Krishnamurthy, and B. Parno, “Talek: Private group messaging with hidden access patterns,” *arXiv preprint arXiv:2001.08250*, 2020.

[30] R. Jansen and N. Hopper, “Shadow: Running Tor in a box for accurate and efficient experimentation,” in *19th Symposium on Network and Distributed System Security*, Internet Society, February 2012.

[31] I. B. Guirat, D. Gosain, and C. Diaz, “Mixim: A general purpose simulator for mixnet,” *Privacy Enhancing Technologies Symposium – HotPETs Workshop*, 2020.

[32] M. Hosseini, D. T. Ahmed, S. Shirmohammadi, and N. D. Georganas, “A survey of application-layer multicast protocols,” *IEEE Communications Surveys & Tutorials*, vol. 9, no. 3, pp. 58–74, 2007.

[33] A. Popescu, D. Constantinescu, D. Erman, and D. Ilie, “A survey of reliable multicast communication,” in *Conference on Next Generation Internet Networks*, NGI, pp. 111–118, IEEE, 2007.

[34] C. K. Yeo, B.-S. Lee, and M. H. Er, “A survey of application level multicast techniques,” *Computer Communications*, vol. 27, no. 15, pp. 1547–1568, 2004.

[35] J. Leitão, J. Pereira, and L. Rodrigues, “Epidemic broadcast trees,” in *26th IEEE International Symposium on Reliable Distributed Systems*, SRDS 2007, pp. 301–310, IEEE, Oct. 2007.

[36] J. Leitão, J. Pereira, and L. Rodrigues, “Gossip-based broadcast,” in *Handbook of Peer-to-Peer Networking*, pp. 831–860, Springer, Oct. 2009.

A MultiSphinx Construction

In this Appendix we provide detailed algorithms for constructing and processing both the regular Sphinx messages (A.1) and our MultiSphinx messages (A.2). The regular construction is based on the original Sphinx paper [11] and the proposed improvement using authenticated encryption [15]. For both schemes we will use three hops n_0, n_1, n_2 for the mix nodes and a final hop n for the recipient[3] that extracts the payload from the inner-most encryption (see Figure 10).

A Sphinx header M consists of a group element α for deriving shared secrets, authenticated data β, and an authentication tag γ. In the original Sphinx paper β is used to store the address of the next hop. For the final hop the distinguished element $*$ is used to signal that the payload reached its intended destination. Loopix adds per-hop delays to this routing information.

We assume that all nodes n_i have access to the public keys of all other nodes without us passing these explicitly. We assume the existence of a method ProcessHeader that takes a header of a Sphinx packet and returns all metadata contained in β (next hop identifier, delay) and the header for the next hop. We assume the existence of a method ComputeSecrets that takes a list of hops $n_0, n_1, \ldots$ and outputs

[3] We omit the provider nodes here to improve readability.

Regular Sphinx

$$s \xrightarrow{(M_0,\delta_0)} n_0 \xrightarrow{(M_1,\delta_1)} n_1 \xrightarrow{(M_2,\delta_2)} n_2 \xrightarrow{(M_3,\delta_3)} n \xrightarrow{\delta} *$$

p-restricted MultiSphinx ($p = 2$)

$$\begin{tikzcd}
& & & n_{2,A} \arrow[r, "{(M_{3,A},\delta_{3,A})}"] & n_A \arrow[r, "\delta_A"] & * \\
s \arrow[r, "{(M_0,\delta_0)}"] & n_0 \arrow[r, "{(M_1,\delta_1)}"] & n_1 \arrow[ur, "{(M_{2,A},\delta_{2,A})}"] \arrow[dr, "{(M_{2,B},\delta_{2,B})}"] \arrow[loop below, "{\begin{matrix}(M_{1,A},\delta_{1,A})\\(M_{1,B},\delta_{1,B})\end{matrix}}"] & & & \\
& & & n_{2,B} \arrow[r, "{(M_{3,B},\delta_{3,B})}"] & n_B \arrow[r, "\delta_B"] & *
\end{tikzcd}$$

Figure 10: Schematic of messages (header, payload) for Sphinx and MultiSphinx.

a list of shared secrets $s_0, s_1, \ldots$. We assume the existence of a method CREATEHEADER that takes a shared secret s_i, the next hop identifier n_{i+1}, and (optionally) a header M_{i+1} to wrap. The details of these operations can be found in the Sphinx paper [11, §3.2 and §3.6]. In line with Loopix the sender chooses a random per-hop delay for each hop and includes it in the authenticated metadata in the header. This happens transparently in the CREATEHEADER method.

We assume the existence of an authenticated encryption (AE) scheme as required by the improved Sphinx format [15]. An AE scheme provides an encryption function AE_{enc} that takes a secret key s, a message *msg*, and optional metadata *meta* and outputs a ciphertext *ctext* and an authentication tag *auth*. It also provides a decryption function AE_{dec} that takes a secret key s, a ciphertext *ctext*, an authentication tag *auth*, and metadata *meta*. It returns the decrypted message if the authentication tag verifies the integrity of ciphertext and metadata or $\perp$ otherwise.

We assume that the AE scheme is based on an encrypt-then-mac regime using a stream cipher C (e.g. AES-CTR), a message authentication code MAC (e.g. HMAC), and a keyed key derivation function KDF (e.g. HKDF). Stream ciphers have the property that changing a given bit of the ciphertext/plaintext only changes the bit at the same position in the plaintext/ciphertext after decryption/encryption. Arbitrary changes will lead to an invalid auth tag – but we might intentionally ignore this during our constructions and recalculate the auth tags later. Since Sphinx uses fresh secret keys for every message and hop, we can leave the nonce for the stream cipher constant. We show our construction of AE_{enc} and AE_{dec} in Algorithm 3.

Algorithm 3 The authenticated encryption scheme AE based on stream cipher C, a MAC, and a keyed KDF.

procedure $AE_{enc}(s, msg, meta)$
$\quad s_{cipher}, s_{mac} \leftarrow \text{KDF}(s, \texttt{cipher}), \text{KDF}(s, \texttt{mac})$
$\quad ctext \leftarrow \text{C}(s_{cipher}) \oplus msg$
$\quad auth \leftarrow \text{MAC}(s_{mac}, ctext \,\|\, meta)$
$\quad$ **return** $(ctext, auth)$

procedure $AE_{dec}(s, ctext, auth, meta)$
$\quad s_{cipher}, s_{mac} \leftarrow \text{KDF}(s, \texttt{cipher}), \text{KDF}(s, \texttt{mac})$
$\quad$ **if** $\text{MAC}(s_{mac}, ctext \,\|\, meta) \neq auth$ **then**
$\quad\quad$ **return** $\perp$
$\quad msg \leftarrow \text{C}(s_{cipher}) \oplus ctext$
$\quad$ **return** msg

A.1 Normal Sphinx (existing solution)

The algorithms in this section summarise the existing literature [11, 15], but we have adapted the notation to be more concise. Algorithm 4 shows the creation of the a regular Sphinx message by the sender. While the original Sphinx papers can create all headers before encrypting the payload, the improved variant with AE requires us to do these operations simultaneously as the encryption affects the authentication tag γ of this and the following message headers.

Algorithm 4 Creating a packet to be routed through hops n_0, n_1, n_2 to node n.

procedure $\text{CREATE}(\delta, n_0, n_1, n_2, n)$
$\quad$ **assert** $|\delta| = \text{MAXMSGLEN}$
$\quad s_0, s_1, s_2, s_3 \leftarrow \text{COMPUTESECRETS}(n_0, n_1, n_2, n)$
$\quad M_3 \leftarrow \text{CREATEHEADER}(s_3, *)$
$\quad \delta_3,\ M_3.\gamma \leftarrow AE_{enc}(s_3, \delta, M_3.\beta)$
$\quad M_2 \leftarrow \text{CREATEHEADER}(s_2, n, M_3)$
$\quad \delta_2,\ M_2.\gamma \leftarrow AE_{enc}(s_2, \delta_3, M_2.\beta)$
$\quad M_1 \leftarrow \text{CREATEHEADER}(s_1, n_2, M_2)$
$\quad \delta_1,\ M_1.\gamma \leftarrow AE_{enc}(s_1, \delta_2, M_1.\beta)$
$\quad M_0 \leftarrow \text{CREATEHEADER}(s_0, n_1, M_1)$
$\quad \delta_0,\ M_0.\gamma \leftarrow AE_{enc}(s_0, \delta_1, M_0.\beta)$
$\quad$ **return** (M_0, δ_0)

Algorithm 5 shows how a mix node processes a message it has received. First the message is unpacked into the header and the payload. Then the tag is derived and compared against previously seen *tags* to protect against replay attacks. Afterwards, the decryption verifies that the authentication tag matches the message and header metadata. Finally the header is unwrapped and a send operation is scheduled according to the next hop identifier and delay from the metadata.

Algorithm 5 Processing of an incoming packet at mix node n with secret key x_n.

procedure $\text{PROCESS}(packet)$
$\quad (M, \delta) \leftarrow packet$
$\quad s \leftarrow (M.\alpha)^{x_n}$
$\quad$ **if** $h_\tau(s) \in tags$ **then abort**
$\quad tags \leftarrow tags \cup \{h_\tau(s)\}$
$\quad \delta' \leftarrow \text{AE}_{dec}(s, \delta, M.\gamma, M.\beta)$
$\quad$ **if** $\delta' = \perp$ **then abort**
$\quad (n', delay), M' = \text{PROCESSHEADER}(M)$
$\quad \text{QUEUEFORSEND}(n', (M', \delta'), delay)$

A.2 MultiSphinx (our solution)

We now describe our MultiSphinx construction and highlight the changes relative to the normal Sphinx construction in blue. To allow for a readable description we describe everything for $p = 2$ however the general case follows easily.

We use the pseudo-random function (PRF) ρ together with its key-generating function h_ρ from the original Sphinx paper to create a deterministic pseudo-random padding. Since we need two derive to independent keys from the same secret, we extend h_ρ with another parameter that can be an arbitrary string. This extension can be implemented using any suitable HKDF function.

Algorithm 6 explains the creation of MultiSphinx messages by the sender. The part concerning the "two legs" of the message graph is only shown once for A to allow for a more readable presentation. Line 21 instructs which lines are meant to be repeated for the other $p-1$ recipients. In line 4 the secret s_1 is computed which is required for the padding construction in line 11. Lines 6-9 encrypt the actual payload from the recipient n_A to the multiplication node $n_{1,A}$ (going backwards). The encrypted payloads $\delta_{3,A}, \delta_{2,A}, \delta_{1,A}$ are all smaller than the normal payload length of messages. This would allow an attacker to distinguish such messages from other Loopix messages (e.g. when the middle mix layer sends loop messages). Therefore, the ciphertext is padded in line 11 with our PRF ρ. To correctly compute the MACs and headers in lines 15-20, we first simulate (going forwards) how the payloads will be affected by the decryption (line 12f).

Algorithm 7 explains the processing step at a mix node. Regular mix nodes operate as before (line 10). However, at multiplication nodes incoming message payloads are split into p headers and p payloads (line 12). In lines 13-16 the pseudo-random paddings are added. This process is also visualised in Figure 11. The newly created packets are processed recursively and then scheduled for sending based on their individual delay (line 15f). This "self-delivery" corresponds to the loop edge of n_1 in Figure 10. The extra hop allows for delaying both messages independently at the multiplication node (two headers allow for two delays). It also simplifies our correctness arguments.

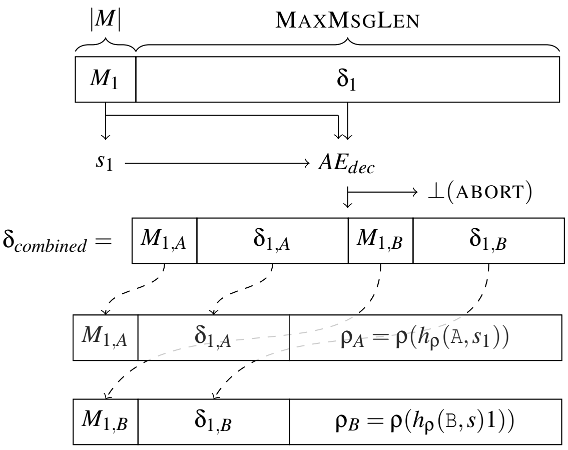


Figure 11: Processing of a MultiSphinx message at the multiplication node n_1 resulting in two outgoing messages that are send then re-queued for processing.

Algorithm 6 Creating a MultiSphinx packet to be routed through hops $n_0, n_1, n_{2,A}, n_{2,B}$ to nodes n_A, n_B.

1: **procedure** $\textsc{Create}(\delta_A, \delta_B, n_0, n_1, n_{2,A}, n_{2,B}, n_A, n_B)$
2: **assert** $|\delta_A| = |\delta_B| = (\textsc{MaxMsgLen} - \textsc{HdrLen})/2$
3: ▷ Secrets for hops from sender to multiplier node n_1
4: $s_0, s_1, \leftarrow \textsc{ComputeSecrets}(n_0, n_1)$
5: ▷ Encrypt from recipient n_A to multiplier node n_1
6: $s_{1,A}, s_{2,A}, s_A \leftarrow \textsc{ComputeSecrets}(n_{1,A}, n_{2,A}, n_A)$
7: $\delta_{3,A} \leftarrow \mathrm{C}(\mathrm{KDF}(s_A, \texttt{cipher})) \oplus \delta_A$
8: $\delta_{2,A} \leftarrow \mathrm{C}(\mathrm{KDF}(s_{2,A}, \texttt{cipher})) \oplus \delta_{3,A}$
9: $\delta_{1,A} \leftarrow \mathrm{C}(\mathrm{KDF}(s_{1,A}, \texttt{cipher})) \oplus \delta_{2,A}$
10: ▷ Add pseudo-random padding and compute padded payloads $\delta'_{...}$ along decryption path
11: $\delta'_{1,A} \leftarrow \delta_{1,A} \| \rho(h_\rho(\texttt{A}, s_1))$
12: $\delta'_{2,A} \leftarrow \mathrm{C}_{dec}(\mathrm{KDF}(s_{1,A}, \texttt{cipher})) \oplus \delta'_{1,A}$
13: $\delta'_{3,A} \leftarrow \mathrm{C}_{dec}(\mathrm{KDF}(s_{2,A}, \texttt{cipher})) \oplus \delta'_{2,A}$
14: ▷ Compute headers and full MACs
15: $M_{3,A} \leftarrow \textsc{CreateHeader}(s_A, *)$
16: $M_{3,A}.\gamma \leftarrow \mathrm{MAC}(\mathrm{KDF}(s_A, \texttt{mac}), \delta'_{3,A} \| M_{3,A}).\beta)$
17: $M_{2,A} \leftarrow \textsc{CreateHeader}(s_A, n_{3,A}, M_{3,A})$
18: $M_{2,A}.\gamma \leftarrow \mathrm{MAC}(\mathrm{KDF}(s_{2,A}, \texttt{mac}), \delta'_{2,A} \| M_{2,A}).\beta)$
19: $M_{1,A} \leftarrow \textsc{CreateHeader}(s_A, n_{2,A}, M_{2,A})$
20: $M_{1,A}.\gamma \leftarrow \mathrm{MAC}(\mathrm{KDF}(s_{1,A}, \texttt{mac}), \delta'_{1,A} \| M_{1,A}).\beta)$
21: **Repeat lines** $6-20$ **for** B
22: ▷ From sender to multiplication node
23: $\delta_{combined} = M_{1,A} \| \delta_{1,A} \| M_{1,B} \| \delta_{1,B}$
24: $M_1 \leftarrow \textsc{CreateHeader}(s_1)$
25: $\delta_1,\ M_1.\gamma \leftarrow AE_{enc}(s_1, \delta_{combined}, M_1.\textsc{metadata}))$
26: $M_0 \leftarrow \textsc{CreateHeader}(s_0, n_1, M_1)$
27: $\delta_0,\ M_0.\gamma \leftarrow AE_{enc}(s_0, \delta, M_0.\textsc{metadata}))$
28: **return** (M_0, δ_0)

Algorithm 7 Processing of an incoming packet at mix node n at mix layer l with secret key x^n.

1: **procedure** $\textsc{Process}(packet)$
2: $(M, \delta) \leftarrow packet$
3: $s \leftarrow (M.\alpha)^{x_n}$
4: **if** $h_\tau(s) \in tags$ **then abort**
5: $tags \leftarrow tags \cup \{h_\tau(s)\}$
6: $\delta' \leftarrow AE_{dec}(s, \delta, M.\gamma, M.\beta)$
7: **if** $\delta' = \perp$ **then abort**
8: $n', delay, M' = \textsc{ProcessHeader}(M)$
9: **if** $l \neq 1$ **then**
10: $\textsc{QueueForSend}(n', (M', \delta'), delay)$
11: **else**
12: $M_{1,A}, \delta_{1,A}, M_{1,B}, \delta_{1,B} \leftarrow \delta'$ ▷ $\delta' = \delta_{combined}$
13: $\rho_A, \rho_B \leftarrow \rho(h_\rho(\texttt{A}, s)), \rho(h_\rho(\texttt{B}, s))$ ▷ $s = s_1$
14: ▷ Process separately to allow independent delays
15: $\textsc{process}(M_{1,A} \| \delta_{1,A} \| \rho_A)$
16: $\textsc{process}(M_{1,B} \| \delta_{1,B} \| \rho_B)$

B Algorithms

Algorithm 8 The fault-tolerant Rollercoaster callback handler and send methods (signatures are checked implicitly).

```
procedure SendToGroup(groupid, payload)
    S ← GenSchedule(msg.source, msg.groupid)
    for recipient ∈ {direct children of self in S} do
        msg ← NewMessage()
        msg.groupid ← groupid
        msg.nonce ← FreshNonce()
        msg.{source, sender, role} ← self
        msg.payload ← payload
        ScheduleForSend(recipient, msg)

procedure OnPayload(msg)
    ApplicationHandle(msg.payload)
    if msg was received while offline then return
    if msg was not seen before then
        S ← GenSchedule(msg.source, msg.groupid)
        for x ∈ {direct children of msg.role in S} do
            msg′ ← CopyMessage(msg)
            msg′.sender ← self
            msg′.role ← x
            ScheduleForSend(x, msg′)
    ScheduleForSend(msg.source, GenAck(msg))

procedure OnAck(msg)
    assert (msg.source = self)
    CancelTimeout(msg, msg.role, msg.sender)

    ▷ Called when a message leaves the payload queue
procedure OnMessageIsSent(msg)
    S ← GenSchedule(msg.source, msg.groupid)
    for x ∈ {recursive children of msg.role in S} do
        timeout ← EstimateTimeout(S, x)
        AddTimeout(msg, x, timeout)

procedure OnTimeout(msg, recipient_failed)
    S ← GenSchedule(msg.source, msg.groupid)
    if not IsForwardingNode(S, msg.role) then
        return
    for x ∈ {recursive children of msg.role in S} do
        CancelTimeout(msg, msg.role, msg.sender)
            ▷ timeout will be recreated when re-try is sent
    recipient′ ← NextRecipient(S, recipient_failed)
    ScheduleForSend(recipient′, msg)
    msg.role ← ∅            ▷ Re-try to failed node w/o role
    ScheduleWithExpBackoff(recipient_failed, msg)
```

Algorithm 9 Methods explaining how the timeout information is stored and updated.

```
procedure OnInit
    self.sessions = [·]          ▷ missing keys default to {}

procedure AddTimeout(msg, role, recipient, timeout)
    CancelTimeout(msg, role, recipient)
    id ← (msg.groupid, msg.nonce)
    entry ← (role, recipient, timeout)
    self.sessions[id] ← self.sessions[id] ∪ {entry}

procedure CancelTimeout(msg, role, recipient)
    id ← (msg.groupid, msg.nonce)
    session = self.sessions[id]
    self.sessions[id] ← {x ∈ self.sessions[id] | x.role ≠
role ∧ x.recipient ≠ recipient}
```

Algorithm 10 Determines whether node *node* is a forwarding node with regards to schedule S.

```
procedure IsForwardingNode(S, node)
    source ← S[0][0][0]
    if node = source then
        return false
    for t = 1 until |S| do
        R ← S[t]
        for (sender, _) in R do
            if node ≠ source and node = sender then
                return true
    return false
```

C Rollercoaster Eventual Delivery

We make the following assumptions for the remainder of Appendix B: All network links are fair-loss and deliver messages with probability > 0. Every layer of the mix network has at least one mix node that correctly forwards messages. Any node can go offline at any time and all nodes except *source* can be Byzantine-faulty. We use the following definitions for the remainder of Appendix B: Let *source* be a node, which does not go offline permanently. Let S be a schedule that includes *source* as its root and all group members as its internal and leaf nodes.

Lemma 1. *Every message which is sent along a randomly chosen path of mix nodes has probability* > 0 *to be delivered.*

Proof. Since there is a mix node that correctly forwards messages in every layer, there is a non-zero probability of choosing mix nodes in all three layers which all correctly forward messages. Since every network link has a non-zero probability of delivering a message, the entire route consisting of multiple network links has a non-zero probability of delivering the message. Therefore, sending a message along a randomly chosen path has probability > 0 of being delivered. $\square$

Lemma 2. *Every direct payload sent by a node* sender*, which does not go offline permanently, to a node* recipient *is eventually delivered.*

Proof. We first consider the case (a) that the delivery of the payload to *recipient* is successful and the delivery of the ACK to *sender* is successful. In this case *sender* can be certain that the payload was delivered to *recipient*, because only *recipient* can compute the correct signature in the ACK. Therefore, the cancellation of the timeout by *sender* is safe.

We now consider the case (b) that the delivery of the payload to *recipient* failed or the delivery of the ACK to *sender* failed. Any of the two failures causes the timeout at the *sender* to expire. As a result *sender* sends the same payload again using a new random path and setting a new timeout. Since sending the payload to *recipient* and sending the ACK to *sender* have non-zero probability of success (Lemma 1), there will eventually be an execution where both succeed.

We now consider the case (c) that *recipient* is offline. For *sender* this situation is indistinguishable from case (b). Therefore, *sender* will retry until *recipient* returns online and sends an ACK.

We now consider the case (d) that *sender* is offline. If the payload has not been sent yet, it will be sent when *sender* returns online. In case (a) *sender* will observe the ACK message when it returns online. In cases (b) and (c) the sender will re-try once it returns online and the timeout expired. Since *sender* will not go offline permanently, transition to cases (a)-(c) will happen eventually. $\square$

Lemma 3. *Any payload sent by* source *is eventually delivered to all direct children of* source *in S.*

Proof. For direct children of *source* in S, all payloads are sent as direct messages. From Lemma 2 it follows that these are delivered eventually. $\square$

Lemma 4. *Any payload sent by* source *is eventually delivered to all indirect children of* source *in S.*

Proof. Let x be an arbitrary indirect child of *source* in S.

We first consider the case (a) that the payload was delivered through the forwarding node(s) to x and the ACK message was delivered to *source*. It is safe for *source* to cancel the respective timeout, as the payload was delivered to x.

We now consider the case (b) that x is a direct child of p which is a direct child of *source*. If *source* receives p's ack, but not x's ack, then x becomes a direct child of *source*. This case is considered in Lemma 3. Otherwise, p timed-out before x, because p's timeout is strictly smaller than the one of x. This means that *source* assigns the role of p to another node p' and sets new timeouts for p' and x. If p' fails, this is repeated for multiple rounds. The list of replacement nodes shrinks by ≥ 1 every round as previously failed nodes will not be considered again. Therefore, eventually forwarding succeeds or *source* (which is part of the replacement list) becomes p' making x a direct recipient. This case is considered in Lemma 3.

We now consider the case (c) that x is a child in a tree path $[source, p_1, p_2, \ldots, x]$. Let p_j with $j \geq 1$ be the parent closest to x whose timeout expires. This implies that all p_i with $i < j$ successfully acknowledged to *source* since their timeouts are strictly smaller. The role of p_j will be assigned to another node p'_j and new timeouts are set. If p'_j fails, this is repeated for multiple rounds. Therefore, eventually forwarding succeeds or *source* becomes p'_j which reduces the length of the path from *source* to x by at least one. Therefore, eventually forwarding succeeds or the path length is reduced so far that case (b) applies. $\square$

Theorem 5. *Let* source *be a node that does not go offline permanently, and that is not Byzantine-faulty. All payloads sent by* source *are eventually delivered to all group members.*

Proof. For any schedule, every group member is either a direct child or an indirect child of *source*. Therefore, the result follows directly from Lemma 3 and Lemma 4. $\square$

Theorem 6. *Let* source$'$ *be a node that may be Byzantine-faulty. Let X be any subset of group members that are not Byzantine-faulty and that do not go offline permanently. If one of the nodes in X receives a payload from* source$'$*, all other nodes in X will eventually receive the payload.*

Proof. Let $x \in X$ be the node that has received a payload message m from *source*$'$, and let x' be any other member of X. We then show that x' eventually receives m.

As described in Section 5.2.1, every node periodically computes a hash of the payloads it has received and sends it to a randomly selected group member; thus, x eventually computes a hash over a message history including m and sends it to x', and by Lemma 1 this message is eventually received by x' (possibly after several attempts). If x' has already received m, we are done. If x' has not yet received m, and assuming the hash function is collision-resistant, then there is no message history known to x' that results in the same hash value, and therefore the hash sent by x is unknown to x'.

x' responds to the unknown hash by sending a request to x, asking it to send any messages that x' is missing. A simple but inefficient algorithm would be for x to resend all payload messages it has ever received to x'. A more efficient approach uses a reconciliation protocol to determine which messages are known to x but unknown to x', and to resend only those messages. Several such reconciliation protocols are known [14]. Whatever protocol is used, it will eventually complete (by Lemma 1), and therefore x' will eventually receive m from x, as required. $\square$

D MultiSphinx Security Proof

This appendix provides proofs for the MultiSphinx security and anonymity claims in Section 5.4.2. We discuss the resistance against a global passive adversary (D.1) that might collude with corrupted mix nodes (D.2). Finally, we show that our claims also extend to active attacks (D.3) as described in the Loopix paper.

D.1 Against a Global Passive Adversary

We will show that a global passive adversary (GPA) that can monitor the entire network traffic does not gain any advantage when MultiSphinx messages (A.2) are used compared to regular Sphinx messages (A.1).

Indistinguishability of MultiSphinx and Sphinx messages for GPA

We first show that a global passive adversary (GPA) cannot tell apart MultiSphinx messages and regular Sphinx messages. By doing this we reduce our security claims to those of the original Loopix paper. We treat all encryption and pseudo-random functions (PRFs) as random oracles as it is done in the original Sphinx paper.

Lemma 7. *MultiSphinx messages from the sender to the multiplication node are indistinguishable from Sphinx messages for a GPA.*

Proof. The headers of all MultiSphinx messages from the sender to the multiplication node are constructed using the same methods as regular Sphinx messages. At the same time the Sphinx header is "[...] hiding the number of hops a messages has travelled so far, as well as the actual number of mixes on the path of a message" [11, p.2]. Therefore, the headers are indistinguishable between the two message types. To an adversary who does not know the key the encrypted payload is indistinguishable from random bits for both message types as per definition of the random oracle. Therefore, the encrypted payloads are indistinguishable between the two message types. □

Lemma 8. *MultiSphinx messages from the multiplication node to other nodes are indistinguishable from Sphinx messages for a GPA.*

Proof. The proof for the header bytes follows analogously to the proof above. The payloads of MultiSphinx messages leaving the multiplication node consist of the encrypted payload concatenated with the pseudo-random padding (the output ρ_i of the PRF). As per the definition of the random oracle, both bit strings are indistinguishable from random noise. Therefore, the entire payload is indistinguishable. This is also true for all following hops, as all messages (before the recipient unpacks the innermost message) are ciphertexts in the random oracle model. □

Lemma 9. *All MultiSphinx messages are indistinguishable from Sphinx messages for a GPA.*

Proof. Since Loopix is using a stratified topology, all message paths will go through a mix node at the multiplication layer. This means that each message (and edge) of the path is either before or after a multiplication node. Lemma 7 and Lemma 8 cover both cases. □

Unlinkability of Messages at Multiplication Node

Theorem 1 of in the Loopix paper [3] analyses the probability that an adversary can link a single message leaving a mix node to one of the previously arriving messages. Our argument follows the same structure to show MultiSphinx maintains the unlinkability property described in the Loopix paper. The MultiSphinx protocol does not affect mix nodes in the first and third layers since they retain a one-to-one relation between incoming and outgoing messages. Therefore the analysis in the Loopix paper holds unchanged. We diverge from the original notation by using κ instead of k to avoid confusion with the k parameter used for our schedule generation.

Theorem 1 in the Loopix paper defines the observation scenario $o_{n,\kappa,l}$ for a passive adversary: first, the attacker observes a set of n messages arriving at a previously empty mix (multiplying into pn messages internally); then a total of $(pn-\kappa)$ messages are emitted by the mix before another set of l messages arrive at the mix; finally, a single message m leaves the mix node and the adversary seeks to correlate this message m with any of the $n+l$ messages observed arriving at the mix node.

Lemma 10. *Let m_1 be any of the initial n messages arriving at the p-restricted MultiSphinx multiplication mix node in scenario $o_{n,\kappa,l}$. Let m_2 be any of the l messages that arrive later. The probability that the outgoing message m was an inner message of either m_1 or m_2 is:*

$$Pr(m \in m_1) = \frac{\kappa}{n(\kappa + pl)}, \tag{7}$$

$$Pr(m \in m_2) = \frac{p}{\kappa + pl}. \tag{8}$$

Proof. With the arrival of n messages and their multiplication the mix node holds pn messages. After emitting $(pn-\kappa)$ messages the mix holds κ messages. The arrival of the l messages leads to a total of $\kappa + pl$ messages from which m is chosen.

The probability that m is an inner message of any of the initial n messages is $\frac{\kappa}{\kappa+pl}$. The requirement that it was an inner message of one particular message m_1 of that batch which leads to: $\frac{1}{n} \cdot \frac{\kappa}{\kappa+pl} = \frac{\kappa}{n(\kappa+pl)}$.

The probability that m is an inner messages of any of the later l messages is $\frac{pl}{\kappa+pl}$. The requirement that it was an inner message of one particular message m_2 of that batch which leads to: $\frac{1}{l} \cdot \frac{pl}{\kappa+pl} = \frac{p}{\kappa+pl}$.

□

These results are qualitatively the same as in the Loopix paper: "[...] continuous observation of a Poisson mix leaks no additional information other than the number of messages present in the mix" [3, p.1207]. As the original paper's argument builds upon the probabilities in its Theorem 1. It also holds for MultiSphinx.

Theorem 11. *A global passive adversary (GPA) that monitors all network traffic does not gain any advantage when MultiSphinx messages are used instead of Sphinx messages.*

Proof. An observer cannot distinguish any MultiSphinx message from a regular Sphinx message (Lemma 9). The difference in processing at the multiplication node also does not provide any advantage (Lemma 10). Furthermore, the multiplication factor p is globally fixed and therefore cannot leak any information about group sizes. All other mix nodes operate exactly as they do with normal Sphinx messages. To an observer all clients create indistinguishable packets at a rate independent of actual communication. Therefore, an observer does not gain any advantage over regular Sphinx messages. □

D.2 Against Corrupt Nodes

We now consider the ability of an adversary who controls a subset of mix nodes along the message path. The adversary can inspect the internal state of the corrupted mix node including short-term and long-term secrets. We exclude active attacks for now, as they are covered in D.3. This is sometimes called an honest-but-curious mix node model.

Theorem 12. *An adversary controlling an honest-but-curious mix node on the first or third layer processing MultiSphinx messages does not gain any advantage compared to regular Sphinx messages for any of the security notions discussed in the original Loopix paper: sender-recipient third-party unobservability, sender online unobservability, sender anonymity, receiver unobservability, and receiver anonymity.*

Proof. From Theorem 11 it follows that the adversary does not gain an advantage from monitoring the in-coming and out-going messages. Also, Algorithm 7 shows the operation of these nodes is identical to regular Sphinx nodes. □

Theorem 13. *An adversary controlling an honest-but-curious multiplication mix node on the second layer processing MultiSphinx messages does not gain any advantage compared to regular Sphinx messages for any of the security notions discussed in the original Loopix paper: sender-recipient third-party unobservability, sender online unobservability, sender anonymity, receiver unobservability, and receiver anonymity.*

Proof. A p-restricted MultiSphinx multiplication node on the second layer does not learn more information about the sender or recipient than a regular Sphinx node on the second layer: it knows its preceding node on the first layer and it knows for each message the succeeding node in the third layer. The addition of deterministic pseudo-random padding $\rho_{\{A,B,\ldots\}}$ does not reveal any information about the sender or recipient as it is derived from a shared secret that is known in the regular Sphinx operation as well. □

Definition 14. *In an anonymity system having* ***group existence anonymity*** *an adversary cannot decide whether a group of a given size is communicating.*

This property does not hold for naïve multicast: a multiplication node observing a message splitting into x other messages can deduce an increased likelihood that a group of that size exists.

Theorem 15. *An adversary controlling an honest-but-curious mix node processing p-restricted MultiSphinx cannot decide group existence with probability better than random chance.*

Proof. All MultiSphinx messages are constructed independently of the underlying Rollercoaster algorithm (see Figure 4) and always have the same number of p wrapped messages. None of the observable properties change if there is an actively communicating group of certain size or not. □

Definition 16. *In an anonymity system having* ***group membership anonymity*** *an adversary cannot determine whether two users are members of the same group or not.*

Theorem 17. *Assume a system using 3 mix layers and p-restricted multicast at the middle layer. Assume an adversary, who controls an honest-but-curious multiplication node, and also controls c out of n mix nodes in the third layer. Let $\mathcal{U}$ be the set of all users evenly distributed among all providers $\mathcal{P}$. Then this adversary cannot decide group membership anonymity with probability better than $\left(1-\left(\frac{n-c}{n}\right)^{p-1}\right) \cdot \frac{|\mathcal{P}|^2}{|\mathcal{U}|^2}$ for a given group message.*

Proof. We assume (to the advantage of the adversary) that MultiSphinx messages contain either payloads to members of the same group or cover traffic. In the real system MultiSphinx messages might contain a mixture of both making the adversary's job more difficult. We also ignore (to the advantage of the adversary) any Loop messages injected by the mix nodes.

We first analyse the chance of the adversary controlling at least 2 of the independently chosen p mix nodes from the third mix layer, as they need both to link the messages.

$$
\begin{aligned}
Pr(X \geq 2) &= 1 - Pr(X=0) - Pr(X=1) \\
&= 1 - \left(\frac{n-c}{n}\right)^p - p \cdot \frac{c}{n} \cdot \left(\frac{n-c}{n}\right)^{p-1} \\
&= 1 - \left(\frac{n-c}{n} + \frac{p \cdot c}{n}\right) \cdot \left(\frac{n-c}{n}\right)^{p-1} \\
&= 1 - \frac{n+(p-1)c}{n} \cdot \left(\frac{n-c}{n}\right)^{p-1} \\
&\leq 1 - \left(\frac{n-c}{n}\right)^{p-1}
\end{aligned}
$$

Because the adversary controls both a multiplication node and multiple layer-3 nodes, they can trace messages that resulted from the same multiplication up to their delivery to the recipients' providers P_1 and P_2. If the traced messages contained payload of the same group, the adversary now knows that there exist users U_1 and U_2 that are likely to be members of the same group and their respective providers are P_1 and P_2. However, without also controlling the providers, the adversary does not know the identity of U_1 and U_2.

The likelihood of correctly guessing the actual recipient for a given provider is $Pr_{provider \to user} = \frac{|\mathcal{P}|}{|\mathcal{U}|}$.

Therefore, the likelihood that an attacker, who controls a multiplication node, correctly deduces two recipients of a message that belong to the same group is:

$$
\begin{aligned}
Pr_{win} &= Pr(X \geq 2) \cdot (Pr_{provider \to user})^2 \\
&\leq (1 - \left(\frac{n-c}{n}\right)^{p-1}) \cdot \frac{|\mathcal{P}|^2}{|\mathcal{U}|^2}
\end{aligned}
$$

□

The following example applies Theorem 17 to a specific scenario: We assume there are $|\mathcal{U}| = 1000$ users evenly distributed among $|\mathcal{P}| = 10$ providers using p-restricted MultiSphinx with $p = 2$. We assume that the adversary controls $c_3/n_3 = 20\%$ of multiplication nodes in the third layer. We also assume that the adversary controls $c_2/n_2 = 20\%$ of all multiplication nodes in the second layer (i.e. Theorem 11 applies to only 20% of all messages). These numbers mean that for each group message sent through the network the adversary has a chance of $4 \cdot 10^{-6}$ to correctly name two users who are members of that group. For an adversary controlling $\frac{c_2}{n_2} = \frac{c_3}{n_3} = 50\%$, the probability increases to $2.5 \cdot 10^{-5}$. We leave more precise statistical analysis (leading to a lower upper boundary) for future work.

D.3 Against Global Active Adversary

We now consider a global active adversary (GAA) that can do everything the observer from the previous sections can do. In addition the GAA can also inject/drop network messages and participate as a limited number of users. These abilities match those provided in the Loopix paper.

Lemma 18. *MultiSphinx messages are resistant to tagging attacks.*

Proof. The improved Sphinx construction protects the integrity of the entire message [15]. This includes the deterministic pseudo-random padding ρ_i in our construction. Therefore, modifications to the messages are detected by the mix nodes and such messages will not be processed. □

Theorem 19. *A global active attacker (GAA) that monitors all network traffic and can modify messages does not gain any advantage when MultiSphinx messages are used instead of Sphinx messages.*

Proof. Mix nodes of all levels use the same protections against active attacks as those in the Loopix paper – namely: integrity check of messages against adversarial tagging, loop traffic against n-1 attacks, and message tags against replay attacks. Lemma 18 shows that all MultiSphinx messages are resistant against tagging attacks as well. Therefore, MultiSphinx provides no advantage to an GAA compared to normal Sphinx messages. □

E Reproduced Latency Distribution

To check the general soundness of our simulation we reproduced the latency distribution provided by the Loopix paper [3, Figure 11]. For their experimental setup with $\lambda_\mu = 2$ the original authors suggest fitting a Gamma distribution with mean 1.93 and standard deviation 0.87. This translates into a shape-scale-parameterised Gamma distribution with parameters $\Gamma_{original}(k \approx 4.95, \theta \approx 0.39)$.

When we compare our data ($n \geq 18000$ measurements) against $\Gamma_{original}$ the fit is not perfect (see Figure 12). The discrepancy can be explained by the fact that the original paper's $\Gamma_{original}$ was determined experimentally and is affected by imprecision caused by their measurement and implementation. However, we can analytically determine the distribution of the sum of the four independent exponential distributions $exp(\lambda_\mu)$: one for the ingress provider and three for the mix nodes. This distribution is $\Gamma_{theory}(k = 4, 0.5)$ and the data from our simulation fits it very well (see Figure 13).

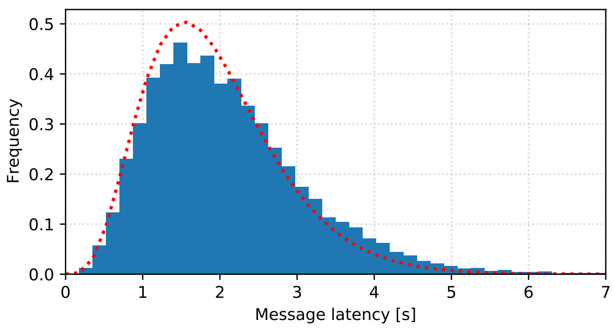


Figure 12: Distribution of latency measured by our simulator and the Gamma distribution $\Gamma_{orginial}$ (dotted red line) from the original paper.

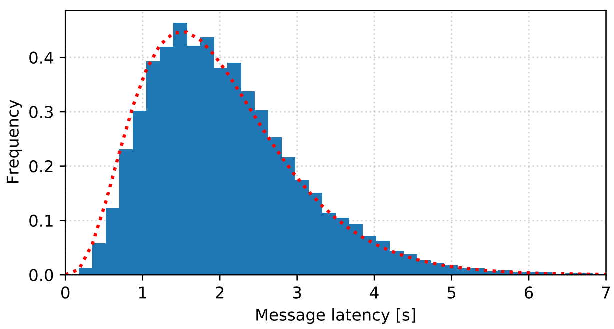
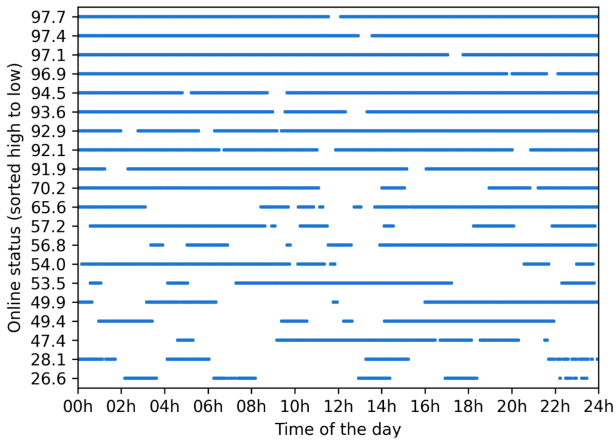


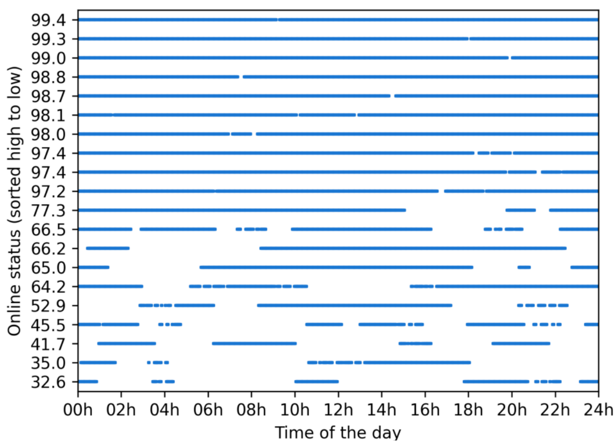
Figure 13: Distribution of latency measured by our simulator and the Gamma distribution Γ_{theory} (dotted red line) determined analytically.

F Visualisation of Offline Models

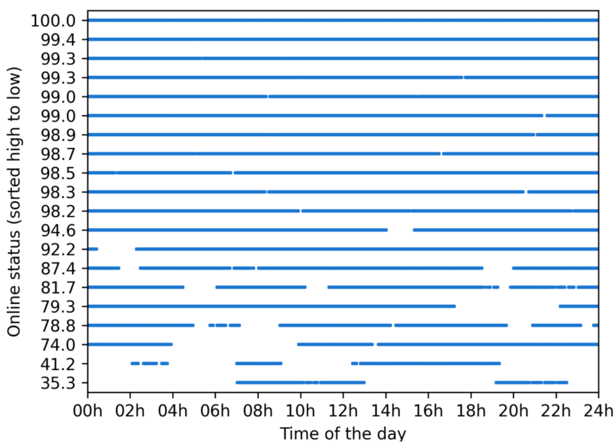
The following figure shows a sample of 20 nodes for the original model with an average online ratio of 65.05%. When a node is online it is marked with a blue dot. The percentage provided next to the Y-axis shows the total fraction of the 24h time span that a node is online.



The following figure shows a sample of 20 nodes for the extrapolated model with an average online ratio of 80.01%:



The following figure shows a sample of 20 nodes for the extrapolated model with an average online ratio of 88.45%:



G p-Restricted Multicast Simulations

In Figure 14 we present additional online and offline scenarios for different sending rates and branching factors without (left) and with (right) p-restricted multicast. See Section 6.5.

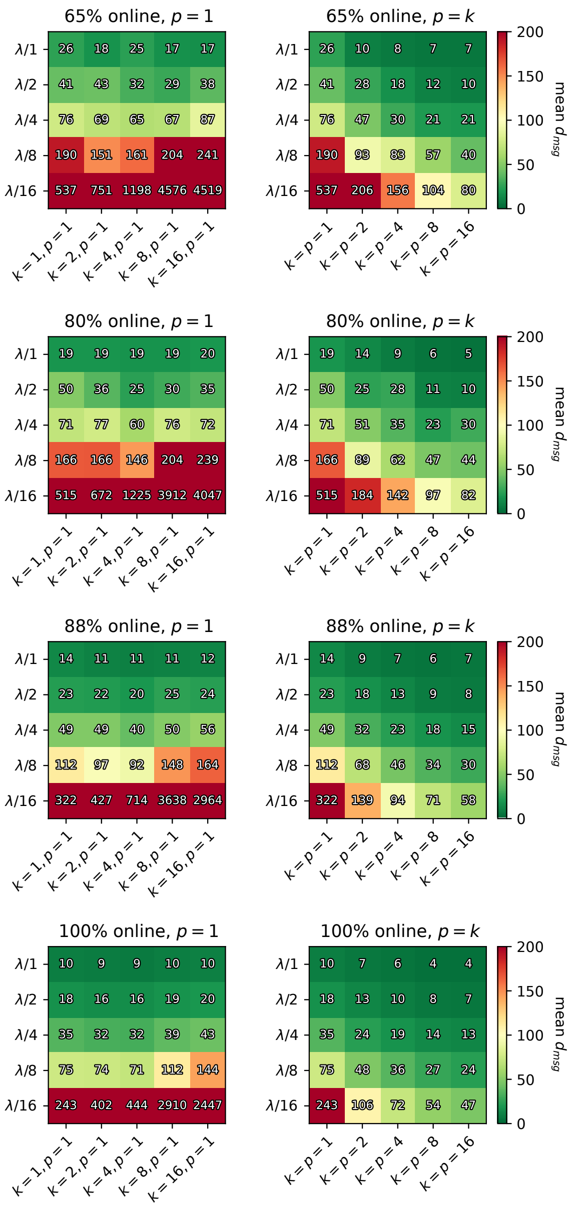


Figure 14: Heatmaps showing the mean message latency for reduced sending rates (y-axis) and different Rollercoaster parameters (x-axis). In the left graph only the logical multiplier factor k is increased. In the right graph the multicast factor p is increased at the same time. Group size 128.

H Histograms

Figures 15 and 16 (next page) provide additional illustrations and express the underlying behaviour of the different strategies. They serve as additional evidence that our simulator functions as described in the paper. The histograms also show the results that are provided in the box charts in Figures 5 and 6.

The distribution of the online naïve sequential unicast strategy in Figure 15 has a very wide body that is an effect of the queuing time in the payload buffer of the sender. Its height depends on the payload rate λ_p. On the other hand, the Rollercoaster graphs show a much tighter distribution with a much higher peak (124k compared to 11k for unicast).

For the offline scenarios in Figure 16 the distribution for the naïve sequential unicast strategy is almost the same as in Figure 15. The non-fault tolerant Rollercoaster strategy fails dramatically and is dominated by extreme outliers – with p_{99} growing to multiple hours! Adding the basic fault tolerance improves the p_{90} percentile and the mean values approach a reasonable order of magnitude. However, nodes coming back online continue to be overwhelmed since they first work through old messages in their mailbox. The fix, as described in the paper is to drop any forwarding messages received while the node was offline. Doing so means that the fault-tolerant Rollercoaster strategy provides excellent results with mean, p_{90}, and p_{99} being less than a minute and better than unicast can provide. The individual peaks visible are the result of timeouts and subsequent re-transmissions.

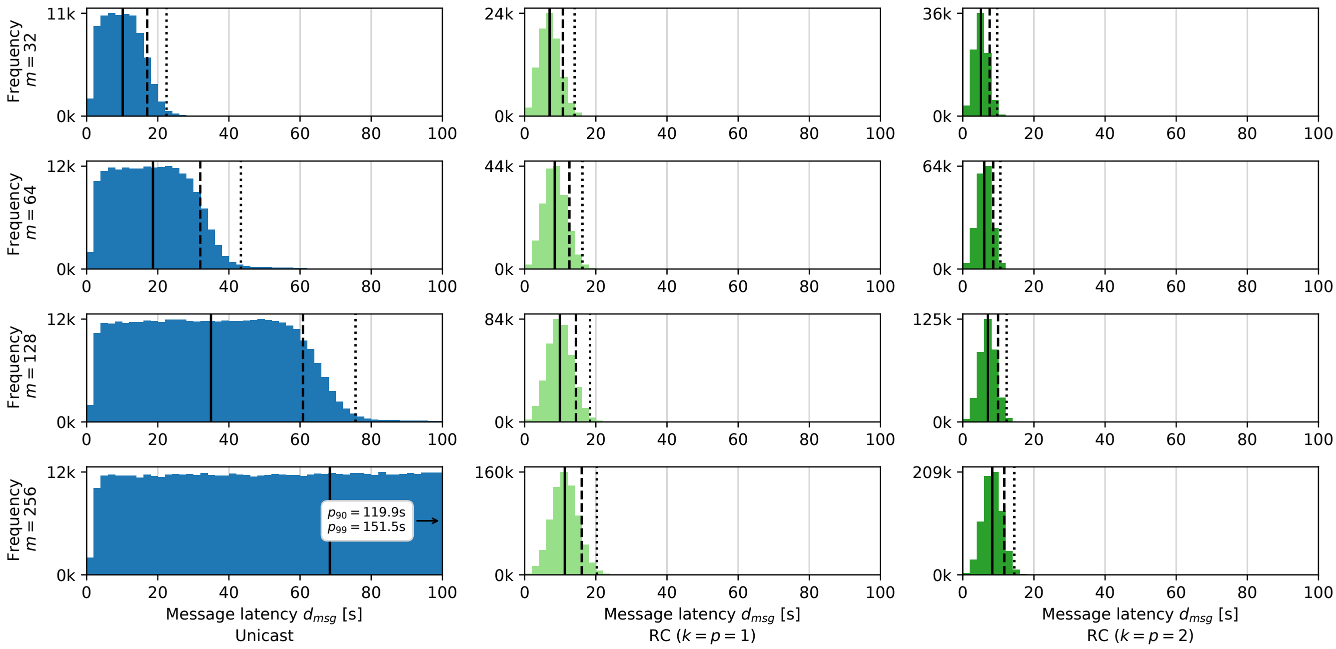


Figure 15: The distribution of message latency of naïve sequential unicast and Rollercoaster (RC) as perceived by the participating group members in a simulation where all users are online all the time. The solid line marks the mean latency whereas the dashed (and dotted) lines mark the p_{90} (and p_{99}) latency. In this figure the y-axes are not linked in order to provide higher fidelity.

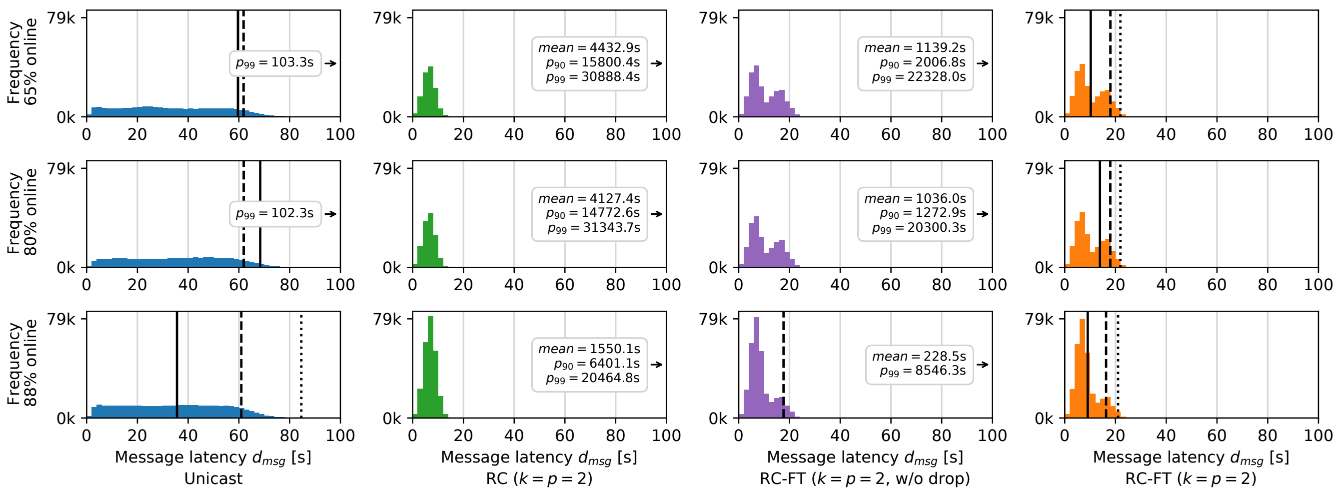


Figure 16: The distribution of message latency d_{msg} for sequential unicast and different Rollercoaster configurations. The group size is 128 and the rows show different offline scenarios. The left graphs show (left-ro-right): (i) Naïve unicast, (ii) Rollercoaster without fault-tolerance for $k = p = 2$, (iii) Rollercoaster with fault-tolerance but not ignoring messages that arrived while offline, (iv) our final Rollercoaster algorithm with fault-tolerance and all optimisations. The solid line marks the mean latency whereas the dashed (and dotted) lines mark the p_{90} (and p_{99}) latency. The mean can be larger than the p_{90} if there are few but large outliers.