Divide and Funnel: a Scaling Technique for Mix-Networks

Debajyoti Das
imec-COSIC, KU Leuven
debajyoti.das@esat.kuleuven.be

Sebastian Meiser
Visa Research
smeiser@visa.com

Esfandiar Mohammadi
University of Luebeck
esfandiar.mohammadi@uni-luebeck.de

Aniket Kate
Purdue University
aniket@purdue.edu

Abstract

While many anonymous communication (AC) protocols have been proposed to provide anonymity over the internet, scaling to a large number of users while remaining provably secure is challenging. We tackle this challenge by proposing a new scaling technique to improve the scalability/anonymity of AC protocols that distributes the computational load over many nodes without completely disconnecting the paths different messages take through the network. We demonstrate that our scaling technique is useful and practical through a core sample AC protocol, *Streams*, that offers provable security guarantees and scales for a million messages. The scaling technique ensures that each node in the system does the computation-heavy public key operation only for a tiny fraction of the total messages routed through the Streams network while maximizing the mixing/shuffling in every round.

We demonstrate Streams' performance through a prototype implementation. Our results show that Streams can scale well even if the system has a load of one million messages at any point in time. Streams maintains a latency of 16 seconds while offering provable "one-in-a-billion" unlinkability, and can be leveraged for applications such as anonymous microblogging and network-level anonymity for blockchains. We also illustrate by examples that our scaling technique can be useful to many other AC protocols to improve their scalability and privacy, and can be interesting to protocol developers.

1 Introduction

Starting with Chaum [17], anonymous communication (AC) protocols [5, 6, 9, 18, 20, 22, 23, 30, 34, 38–41, 50–55, 58, 60, 65, 66] strive to be both practically applicable with minimal computational and communication overhead while providing strong anonymity properties. Some protocols, like Tor [29,30], focus more on low latency and low bandwidth overhead, while others, like dining cryptographers' networks [18] focus on easily provable sender anonymity. Most AC protocols fall in between those two, either guaranteeing or providing an argument for the anonymity being offered while attempting to minimize the overhead introduced through delay, noise messages, or user coordination/preprocessing.

AC protocols need to ensure that messages cannot be traced from source to sink, i.e., from sender to recipient; the typical way to achieve this is by having messages mingle with other messages in a way that is hard to disentangle by a curious observer or malicious protocol party. This property of message indistinguishability, often called "mixing", mainly occurs when messages meet in an honest protocol party that in turn both performs cryptographic operations on them and outputs them in a shuffled order to break a potential linking between incoming and outgoing messages.[1] However, an adversary can additionally leverage differences in user behavior (e.g, if Alice is active only at a specific time of the day) to guess who might have sent a message. Even a trusted-third-party anonymizer can not defend against such leakage. Some protocols [23, 41, 50, 52, 53, 58, 60, 65, 66] defend against that by restricting/enforcing how the protocol clients can behave. We consider that problem to be orthogonal and focus on the "mixing" problem.

To ensure that messages appear to mix goes hand in hand with saturating network links between parties and funneling messages through a small number of mixing nodes. In this paper, we present a novel method to scale up AC protocols by allowing them to distribute computation over an arbitrarily large set of parties while preserving almost the same degree of link saturation and the chance for messages to meet. Our method replaces existing protocol parties with computation-light "funnel" nodes that do not need to perform any public-key cryptographic operation on individual messages. An arbitrary amount of other protocol parties perform the required cryptographic computations for them before sending the processed messages to their next destination.

We demonstrate the applicability of our scaling method on a core AC protocol, *Streams*, for which we prove a strong

[1] The two messages in question don't have to physically meet at an honest party. Indeed, the property can hold when two messages each meet some other, say, noise messages, that then end up confusing the paths being taken.

mixing property. *Streams* might be of independent interest for certain applications (see Section 10.2), as it scales to a million messages and achieves practical end-to-end latency with close-to-optimal mixing against honest-but-curious global attackers with a limited amount of honest-but-curious nodes and clients and strong background knowledge.

With a prototype implementation, we demonstrate that *Streams* can scale for a million messages with an end-to-end latency of 16 seconds for each message, for a fraction of 10% compromised nodes in the system, while offering pairwise message unlinkability up to $\delta \leq 2^{-30}$ ($\approx 10^{-9}$). Our proofs formally show mixing occurs in the presence of a global passive adversary that (passively) compromises a fraction of the protocol parties. We extend the protocol's security to defend against active attacks by incorporating recently proposed defenses and heuristics from prior work [58]. (c.f. Section 6.)

We find *Streams* to be useful for applications such as blockchain access privacy and anonymous microblogging. Moreover, we find our scaling method for distributing computations over protocol parties to be not only relevant to *Streams* but also encouraging for protocol designers interested in scaling up other AC protocols. As three representative examples, we consider the prominent Loopix, Karaoke, and Vuvuzela protocols, and describe how our scaling technique of divide and funnel can enhance their mixing properties.

2 Problem statement and System Overview

2.1 System Model

We consider a typical mix network based architecture [50, 58] allowing users to send messages anonymously using an overlay network of mix nodes. Our main objective is to demonstrate our scaling technique by designing an end-to-end provably secure and scalable AC protocol.

Similar to provably-secure mix-net systems such as [19, 50–52, 65, 66], we assume that protocols parties work in rounds. However, only the nodes need to synchronize their rounds; the system's clients do *not* need to be aware of such rounds or synchronization. Even for the nodes, as we further elaborate in Section 4.4, synchronization only improves anonymity, while safety and liveness of the protocol remain unaffected even if nodes are unsynchronized.

2.2 Attacker Model and Security Goals

Just as many other AC protocols [5, 12, 50–52, 55, 65, 66], *Streams* is designed as a core building block for anonymous communication. In this work, we prove *Streams*'s anonymity properties against a static passive (i.e., honest-but-curious) attacker. In particular, we consider passive global network attackers, passive statically compromised (i.e., honest but curious) nodes, and passive compromised clients. To precisely characterize the security guarantee that *Streams* offers, we utilize a property that we call *pairwise unlinkability*: the attacker shall not be able to figure out which of two messages entering a system at a similar time corresponds to which of the two same messages leaving the system at a later point.

This notion of unlinkability is closely related to other prominent anonymity notions; in Section 3.1 we relate it to sender anonymity [11] (which of two potential senders has sent a specific message?), relationship anonymity [11] (who is in communication with who?), and unobservability (is a given sender actively sending a message or idle?).

Even though our formal analysis focuses on global passive adversaries, *Streams* incorporates integrity protection using the standard cryptographic methods [25]. In Section 6 we discuss how our protocol can defend against active attacks.

Non-goals. Similar to [5, 12, 19, 20, 22, 23, 34, 50–52, 55, 58, 60], we do not consider side-channel attacks — the detailed analyses of fingerprinting of web-browsing and other side-channels that might arise in specific application scenarios are left for future work. As with other AC protocols [5, 12, 19, 20, 22, 23, 34, 50–52, 55, 58, 60, 65, 66] in the literature, we do not consider an adversary whose sole purpose is to launch denial-of-service (DoS) attacks. Attacks like targeted flooding to degrade the performance of a node is out of scope for this work — any technique that can be deployed against such attacks for other protocols can be deployed for *Streams* as well. However, we deploy countermeasures against DoS-anonymity attacks (loop messages) from the literature [58].

2.3 Protocol Idea

Assumptions. We assume that each pair of nodes maintain SSL/TLS session between them. Additionally, we leverage the *Sphinx* packet format [25] that provides end-to-end encryption for all messages, and a node does not learn the path length and the relay position of the node on the path of a packet. Towards dealing with an adaptive adversary, we assume randomness beacons [21, 31] available to all the protocol parties.

Scaling Technique. *Streams* tries to keep all messages in the system together across different rounds, which allows them to mix better and after fewer rounds. In a given round, all clients and all nodes funnel all their messages (except the messages that are to be delivered in that round) through a single designated node that we call *funnel node* for that round. However, our aim is to scale the system for millions of messages; using only one node per round to perform the public-key cryptographic operations for all messages is infeasible.

We make this extreme form of funneling and mixing feasible by distributing the computation-heavy cryptographic operations among all available nodes; yet we use one dedicated, randomly chosen funnel node per round to optimize mixing. We separate each round into a compute phase and a funnel phase. In the compute phase the stream of message fans out to all nodes before merging again (in the funnel phase) in

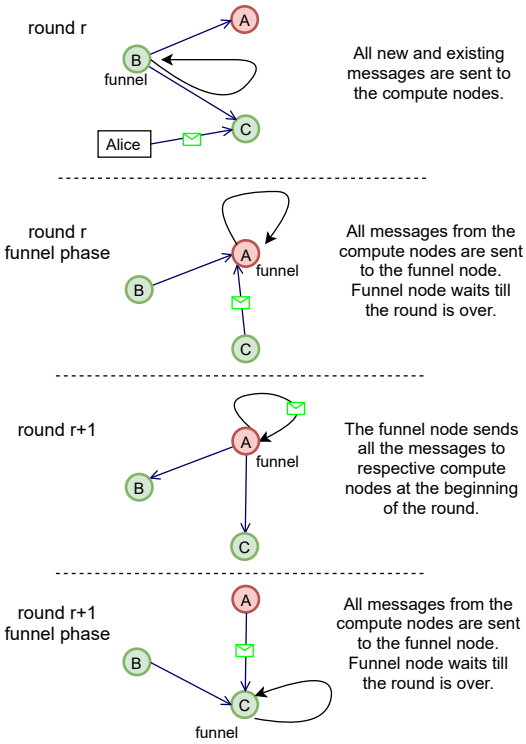


Figure 1: Routing Strategy in *Streams* for the funnel node sequence $\{B,A,C\}$ when Alice sends an onion packet with onion layers for nodes $\{C,A,\dots\}$

the next dedicated and randomly chosen funnel node. As a result, *Streams* can be scaled up considerably while providing strong pairwise unlinkability for all messages that are jointly kept in the system for long enough to meet and mix.

Routing Strategy. *Streams* routes messages through a series of compute and funnel phases, so that each message alternates between compute nodes and funnel nodes. To send a message msg, a user Alice chooses a path consisting *only* of compute nodes and onion encrypts msg for those compute nodes; she sends msg to the first of those nodes. The protocol parties (nodes) collectively choose the funnel node that collects and shuffles packets in a round. Fig. 1 pictorially illustrates the routing strategy in *Streams*. Section 8.2 demonstrates that the compute phase is significantly shorter than the funnel phase.

The nodes only need to agree on the next funnel node, which yields another advantage: for every message the client picks the compute nodes on the path of the message to construct the onion packet without knowing the funnel nodes. The system could use a round-robin scheduling for funnel nodes; we choose to use randomness beacons instead so that the adversary cannot strategically compromise funnel nodes.

2.4 Properties Achieved by *Streams*

Below we summarize the key properties achieved by our scaling technique and the design of *Streams*:

1. *Streams* provides strong pairwise unlinkability of messages, only requiring a modest latency overhead: for any latency in $\omega(\log \eta)$, the probability that two messages are pairwise unlinkable is overwhelming in the security parameter η, even when a constant fraction of nodes are (passively) compromised. We formally prove our security claims about *Streams* in Section 5.
2. *Streams* is designed for scalability and can manage up to one million messages in the system at any given time, while being reasonably practical (several seconds of end-to-end latency and low communication overhead for users).
3. *Streams* allows the clients to be oblivious to the round synchronizations among the nodes; they can send their messages whenever they want.
4. *Streams* is resilient to hiccups in synchronization (up to a certain extent) among local clocks of the nodes without any external intervention.
5. *Streams* provides message integrity and defends against anonymity attacks based on packet drops (c.f. Section 6).

2.5 Comparison with the State of the Art

In this section we give an overview over existing anonymous communication (AC) protocols and discuss their main differences to *Streams*. The Tor network [30] offers low latency and low communication overhead; yet low latency together with low communication overhead has been shown to be a conceptual problem for anonymous communication [26, 27], as has been illustrated with traffic analysis against Tor [13, 35, 46]. We do not attempt to compete with Tor's near real-time latency; instead, we add just enough latency (several seconds) to achieve provable unlinkability.

The recently proposed AC protocol Loopix [58] combines a stratified mix network architecture with exponentially distributed delays to achieve a flexible mixing protocol that does not require fixed rounds; Loopix offers tuneable parameters that balance between latency overhead and the required traffic volume to offer protection against traffic analyses, without providing a formal proof on the degree of pairwise unlinkability for any given parameter set. It is not clear whether Loopix offers strong pairwise unlinkability for interesting latency numbers. Additionally, Loopix scales for many users by employing multiple paths, which in turn reduces the chance of two messages mixing with each other. In Section 10.1, we discuss how Loopix could improve their mixing degree while scaling up by utilizing our scaling approach.

Karaoke [52] and Stadium [65] can scale to millions of users. To do so, both protocols employ a network topology that leverages a number of noise messages in $\Theta(|\text{servers}|^2)$ to achieve link saturation, which yields a property similar to pair-

wise unlinkability. With *Streams* we follow a different path: we achieve optimal link saturation using the TLS-only funnel nodes, thus reducing the amount of noise messages required. In Section 10.1, we discuss how to scale up these protocols with our approach without this explosion of noise messages. Most importantly, however, these two protocols strive for a weaker anonymity property based in differential privacy and, in contrast to *Streams*, not statistical indistinguishability (even for ideal cryptographic operations). Differentially private AC protocols allow an attacker to develop a strong suspicion about who sent a message, which we strive to avoid. Vuvuzela [66] also aims for differential privacy guarantees. In addition, in Vuvuzela each node processes all messages in the system, leading to a large latency (around 37 seconds with one million messages in the system, with a chain of only 3 nodes). The trust assumption based on having only three nodes in the system makes a direct comparison challenging; with *Streams* we opt for a larger number of protocol parties to avoid having to place trust in a small set of nodes.

Stadium, Vuvuzela, and Karaoke require all clients to send messages together before processing a batch to accomplish two goals: (1) avoid leakage from user behaviour (e.g., if Bob is not active when a message is sent, Bob can't have been the sender); (2) servers in the first layer add noise messages to the batch to help their mixing process. This kind of synchronization among millions of clients is very difficult to achieve in a real world scenario. The design of *Streams* does not require the help of noise messages from the servers to achieve mixing. Moreover, *Streams* disentangles the mixing property from user behavior. Hence, we only require the servers to have synchronized clocks but not the clients to use a synchronized usage pattern. We explain in Section 3.1 how our mixing property combined with such restricted client behavior directly implies the traditional anonymity notions achieved by the above protocols. Not requiring the messages to be processed in batches is a design advantage in itself; to the best our knowledge, only Loopix [58] aims for that property before *Streams*.

Atom [50] is a closely related protocol both in terms of the guarantees provided and the explicit focus on scaling: atom's random permutations are similar to pairwise unlinkability and are presented together with a technique for horizontal scaling. Atom's horizontal scaling techniques can handle a million short messages within a reasonable amount of time (around 28 minutes). In contrast, *Streams* offers a significantly shorter end to end latency in the order of a few seconds. In Table 1 we compare *Streams* with Atom, Stadium, and Karaoke in terms of the number of communication hops required to achieve the security guarantees they provide. In the second row of the table, we estimate the number of hops required by Atom to achieve $\delta < 2^{-30}$ with 10% corrupted nodes. In Section 8.3, we additionally present a detailed performance comparison with Atom, Stadium, and Karaoke.

Ando et al. [8] propose a butterfly topology to achieve scalability and prove a mixing property. Butterfly networks are extremely effective in shuffling messages in the absence of compromised parties, but are not resistant in their presence. Even individual compromised nodes can jeopardize the mixing properties of the network. To overcome this challenge, Ando et al. propose to use $\Omega(\log^2(\eta))$ many rounds for the security parameter η in order to provide mixing in the presence of passively compromised nodes. As *Streams* funnels messages through single nodes it can resist passively compromised nodes with just $O(\log(\eta))$ rounds.

Table 1: End-to-end latency offered by *Streams* compared to other provably secure protocols when they handle a total system load of one million messages. In the first two rows, NIZK means non-interactive zero knowledge proofs of knowledge.

Protocol	#hops	Security	Defense against active attacks
Atom [50]	320	$\delta \leq 2^{-64}$	NIZK
Atom (*est.*)	120	$\delta \leq 2^{-30}$	NIZK
Stadium [65]	9	DP	verifiable shuffle
Karaoke [52]	14	DP	server noise + bloom filter
Loopix [58]	≥ 3	unknown	loop messages
Streams	32	$\delta \leq 2^{-30}$	loop messages

Dining cryptographers' networks (DC-nets) [18] and its successors [23, 24, 41, 60, 67] offer easily provable sender anonymity, low latency, and can be scaled up to a certain extent via the any-trust assumption on the servers; however, these protocols inherently demand to fix the users participating in a protocol-round in advance, and expect those users to agree on pair-wise symmetric keys and cannot manage churn easily.

The line of two-party or multi-party computation as a service towards offering anonymity [5, 6, 22, 34, 55] promises strong guarantees in the presence of compromised servers. As these protocols are communication-heavy, the number of MPC-parties cannot easily be scaled up; hence, spreading the trust over a large number of servers is more challenging here than it is for mixnets, such as *Streams*.

3 Security Definition And Background

Anonymity properties, such as sender anonymity or relationship anonymity, depend heavily on the behavior of clients and their choices for the overall message latency: Even if the protocol in question implements a trusted third party it cannot hide which clients are sending messages at which time; moreover, if Alice and Bob send messages at different times, but the overall latency of each of those messages is drawn from the same (independent) distribution, then the recipients of said messages as well as a passive observer can learn information about the potential sender simply by analyzing the timing.

To avoid dealing with these client-dependent and distribution dependent aspects of anonymity we here focus on the

degree of mixing provided by AC protocols. To this end we follow the examples of indistinguishability-based anonymity notions. We formalize the question if the adversary could distinguish whether or not two messages, that spent at least a given amount of shared time in the protocol, could have been swapped along the way. This property is close to message swapping properties from the literature, such as tail indistinguishability by Kuhn et al. [49]. We assume an honest-but-curious global network level attacker that can eavesdrop on a fraction of the nodes (statically chosen), and has strong background knowledge about the behavior of the clients, formally the attacker controls all but two users.

Definition 1 (Pairwise unlinkability)**.** *A protocol provides* pairwise unlinkability *of messages over time t up to probability δ for $0 \leq \delta < 1$ if any pair of messages $(u_0, m_0, t_{s,0}, t_{f,0}, R_0)$ and $(u_1, m_1, t_{s,1}, t_{f,1}, R_1)$ for honest u_0, u_1, where u is the sender of the message, m the content, t_s the time the message enters the system, t_f the time the message leaves the system, and R the receiver of the message, with $\min(t_{f,0}, t_{f,1}) - \max(t_{s,0}, t_{s,1}) \geq t$ cannot be distinguished from the pair $(u_1, m_0, t_{s,1}, t_{f,0}, R_0)$ and $(u_0, m_1, t_{s,0}, t_{f,1}, R_1)$ with an advantage greater than δ by any efficient global passive adversary $\mathcal{A}$ passively and statically compromising at most c nodes.*

A global passive adversary here is a machine that can observe all network traffic between parties, but not alter it directly. Passive and static corruption means that the adversary can choose a set of c out of all K nodes at the beginning of the protocol; the adversary then has access to the internal states of these c nodes, including all of their keys and random choices. Efficient here means that the adversary is restricted in its computational power to run at most a number of steps polynomial in the length of its input, which effectively just rules out that it can break the underlying cryptographic primitives of the protocol. Informally, we say that the two messages *are shuffled* from the adversary's point of view if a protocol achieves pairwise unlinkability. We say that a protocol achieves *strong* pairwise unlinkability if δ is negligible in a security parameter η. Ideally, we want our protocol to achieve strong pairwise unlinkability. We discuss below how pairwise unlinkability relates to sender anonymity and relationship anonymity.

Pairwise properties and more than two parties. Note that pairwise unlinkability for all pairs of messages is a strong property that holds for all pairs of messages at the same time. The property also naturally extends to more than two messages. Practically, distinguishing between two cases where more than two messages are different/swapped is often easier than just distinguishing a single pair. Formally, we can bound the advantage by increasing δ. Consider a case where there are three messages $(u_0, m_0, t_{s,0}, t_{f,0}, R_0)$, $(u_1, m_1, t_{s,1}, t_{f,1}, R_1)$, and $(u_2, m_2, t_{s,2}, t_{f,2}, R_2)$ with $\min(t_{f_0}, t_{f_1}, t_{f_2}) - \max(t_{s_0}, t_{s_1}, t_{s_2}) \geq t$. We know that the adversary cannot distinguish this case from $(u_1, m_0, t_{s,0}, t_{f,0}, R_0)$, $(u_0, m_1, t_{s,1}, t_{f,1}, R_1)$, and $(u_2, m_2, t_{s,2}, t_{f,2}, R_2)$ with advantage greater than δ. Moreover, we know that the adversary cannot distinguish that case from $(u_2, m_0, t_{s,0}, t_{f,0}, R_0)$, $(u_0, m_1, t_{s,1}, t_{f,1}, R_1)$, and $(u_1, m_2, t_{s,2}, t_{f,2}, R_2)$ with advantage greater than δ. Thus, the adversary cannot be able to distinguish the first and last of those cases with advantage greater than $2 \cdot \delta$. This argument extends naturally to any larger set of messages that are being swapped around simultaneously.

3.1 Pairwise Unlinkability and Anonymity

Our notion of pairwise unlinkability is conceptually closely related to *tail indistinguishability* by Kuhn et al. [49]. The main difference is that in their definition packets are required to meet in a node that processes them cryptographically. In this sense pairwise unlinkability follows a previous notion of unlinkability of Kate et al. [48], but extends it with the explicit time when a message enters and leaves the system.

Sender anonymity. One common anonymity notion, *sender anonymity*, states that the recipient of a message cannot distinguish whether the message originated in one sender over another sender, even for a pair of potential senders of the adversary's choice. This notion closely resembles pairwise unlinkability with one key difference: sender anonymity typically talks about a single challenge message, not about a pair of messages; this can be overcome by requiring a degree of bandwidth overhead, such as ensuring all senders communicate regularly and can send dummy messages to confuse the adversary. However, even requiring dummy messages to be sent, an adversary might still deduce the challenge sender from timings alone.

If, say, the adversary observes Alice sending a message in round t and Bob sending a message in round $t+2$, the arrival time of the challenge message together with the distribution of the latency might tell the adversary who of them is more likely to have sent the challenge message. In the simplest example, for a constant latency, the adversary could immediately exclude one of them from being the challenge sender.

Note that this apparent attack is independent of how a protocol achieves anonymity and even applies if the messages are kept in a trusted third party for the same amount of time.

Relationship anonymity. A similar notion states that if two senders send one message each to two receivers, a third party is unable to decide which sender talks to which receiver significantly better than purely guessing. Loopix calls this property *Sender-Receiver Third-party Unlinkability*. Given that the two messages in question are sent in the same round and that both senders choose a sufficiently large latency from the same distribution, pairwise unlinkability immediately implies this anonymity property.

If the challenge senders send their messages h rounds apart, we achieve a weaker, quantitative form of this property (akin to differential privacy), where the degree of anonymity depends on the round difference h and the latency distribution.

ℓ	Maximum latency allowed for a message
L	Minimum required latency of a message
I	Set of all nodes
I_h	Set of all honest nodes
K	Total number of nodes $\lvert I\rvert$
c	Number of compromised nodes $\lvert I - I_h\rvert$
O	An onion packet
η	The security parameter
δ	the adversarial advantage
$\xleftarrow{\$} [b,c]$	Draw uniformly at random from $[b,c]$

Figure 2: Protocol and system parameters for *Streams*

Unobservability. The notion of *unobservability* states that the adversary cannot distinguish between a sender actively engaged in communication and the sender being idle. This property too is highly dependent on the sending patterns of clients and whether or not they introduce noise messages regularly. If we require all senders to send dummy messages whenever they don't have a real message to send, then this property follows from pairwise unlinkability.

4 Protocol Description

Here we present the system setup, the protocol design of *Streams*, and how the separation of duty into compute and funnel phases helps scale for a large number of users. In Appendix A, we present a formal specification of the protocol in Universal Composability (UC) framework.

4.1 System Setup

We consider a set $\mathcal{S}$ of users communicating to a set $\mathcal{R}$ of recipients through a set I of intermediate nodes (or just 'nodes'). In real life, the same user can act as sender as well as recipient, however, we consider the sender role and recipient role as two separate logical entities. Each sender is denoted by u_i where $i \in \{1,\ldots,\mathsf{N}\}$ and $|\mathcal{S}| = \mathsf{N}$. Similarly, each recipient is denoted by R_i where $i \in \{1,\ldots,\mathsf{N}'\}$ and $|\mathcal{R}| = \mathsf{N}'$.

We consider global passive adversaries that can statically compromise up to c nodes out of a total of $\mathsf{K} = |I|$ nodes. In this section, we only consider passive corruption, which means that the compromised protocol parties still follow the protocol specifications, however the adversary has access to all the internal states of a compromised party. In Section 6 we discuss the necessary adaptations for the protocol against an active adversary.

Our protocol uses a round-based communication model and synchronized clocks. In Section 9.1, we discuss how our results can be extended to loosely synchronized clocks.

We consider the availability of a public key infrastructure (PKI) to all the users and nodes. For each party (client or node) P there exist a private public key pair $(\mathtt{sk}_P, \mathtt{pk}_P)$. For a party P to send a message to a party Q, P needs to know the public key $\mathtt{pk}_Q$ of Q. We assume such a PKI system can be instantiated using a one time setup similar to [19, 20, 30, 40, 58] — the exact procedure is out of scope for this work. We summarize all the system parameters in Figure 2.

For our formal security proofs, we consider static corruption by the adversary. However, we make use of randomness beacons to defend our protocol against dynamic corruption.

Randomness Beacon. We assume that each protocol party (including the adversary) has access to an incorruptible randomness beacon. In particular, future values of this beacon are not known to the adversary.

A randomness beacon [59] emits a new *random* value at intermittent intervals such that the emitted values are bias-resistant, i.e., no entity can influence a future beacon value, and unpredictable, i.e., no entity can predict future beacon value. NIST's Randomness Beacons project [21] and the emerging Drand Organization [31] are two prominent ready to use Internet-based instantiations of randomness beacon, while several other protocols [14, 15, 43, 61, 64] and implementations [1, 2, 42] are also available.

To focus on the building blocks that we provide, we abstract away from the cryptographic details of those constructions and assume such an ideal randomness beacon. It outputs each time a ℓ-long substring of an infinite random string beacon; using that ℓ-length string a protocol party can derive the funnel nodes for the next ℓ rounds. Formally, we only require the randomness beacon to be unpredictable before the protocol starts, as our adversary can only statically compromise parties; the beacon, however, can also be leveraged for resistance against dynamic corruption.

4.2 The Core Protocol

First we present the core protocol that does not scale well with the number of users. We then describe our complete protocol with horizontal scaling in Section 4.3.

Packet format. We use the Sphinx packet design [25, 47] to ensure that all messages are end-to-end encrypted; we call them "onion packets". The Sphinx packet design also guarantees that an intermediate node, just by looking at a packet, does not learn any information beyond the routing information needed to forward the message to the next node — it hides the path length and the relay position of the node on the path, and the node does not learn anything other than the next nodes on the path.

Clients. Whenever a client wants to send a message m, she decides the delay d for every message by picking a number from $[\frac{\ell}{2}, \ell - 1]$ following a distribution D. In general D can be any discrete probability distribution; however, in a typical setting we assume D to be uniform in $[\frac{\ell}{2}, \ell - 1]$.

The client derives the path of a packet based on the string returned by the randomness beacon. For a given round r if the

SendMessage(msg):

$d \leftarrow DelayDistribution\left(\frac{\ell}{2}, \ell - 1\right)$

$\{x_1, \ldots, x_d\} \xleftarrow{\$} I^d$; $p := \{x_1, \ldots, x_d, R\}$

Construct a Sphinx onion packet O with path p and content msg

Send the onion packet O to x_1.

Figure 3: Client Protocol Design Π_{client} with Scaling

randomness beacon returns the string $\{x_r, x_{r+1}, \ldots, x_{r+\ell}\}$, any onion packet constructed at round r will be constructed for the path of nodes $\{x_r, \ldots, x_{r+d}, R\}$ for a delay d and intended recipient R. The client will send the onion packet to node x_r at round r.

All other clients as well as nodes send their packets to the node x_r at round r, since the randomness beacon returns the same node $\{x_r, x_{r+1}, \ldots, x_{r+\ell}\}$ to all the parties (clients, nodes, and the adversary) at round r.

Nodes. The nodes act similar to onion routers [30, 40] except a node in our protocol accepts packets only in the rounds indicated by the randomness beacon. More formally, when a node receives a packet in round r, it checks if x_r matches its own id for a string $\{x_r, x_{r+1}, \ldots, x_{r+\ell}\}$ returned by the randomness beacon — if not, it rejects the packet. If x_r matches its id, the node with onion packets peels a layer of onion for each packet and forwards them to the next destination.

QUEUE := a queue where the node stores incoming messages.
nodeID := a unique ID in $[0, \mathsf{K} - 1]$

IncomingMessage (**onion packet** O):

ADD O to QUEUE

NewRound (**round number** r):

funnel := Query the randomness beacon for the current round r
if nodeID = funnel **then**
  tempQ := Shuffle and copy the elements from QUEUE
  while tempQ is not empty **do**
    O := dequeue the first element from tempQ
    Forward O to node x over TLS for $O = \{x, O'\}$
  end while
else
  Wait until all messages are received from funnel for round r
end if
tempQ := Copy all the elements from QUEUE
nextFunnel := Query randomness beacon for next round $(r+1)$
while tempQ is not empty **do**
  O := dequeue an element from tempQ
  $\{x, O'\}$:= Remove one onion layer from O
  Forward onion $\{x, O'\}$ to nextFunnel over TLS
end while

Figure 4: Node Protocol Design with Scaling

4.3 Horizontal Scaling With Compute and Funnel Phases

One major bottleneck in the above protocol is the processing power of the nodes — the total number of clients the system can serve is restricted by the processing power of the weakest node. We propose to separate duties such that, instead of one node processing all the onion packets in a round, many nodes come together to share the processing load. Each round is then separated into a *compute phase*, where the task of onion decryption to prohibit linking is distributed over all nodes, and a *funnel phase*, where a randomly chosen node collects and mixes all the messages. The compute phase does not have a fixed time span. Immediately after processing, each compute node directly forwards the packet to the next funnel node. The funnel node uses the full time of each round. At the end of the round, the funnel node forwards the shuffled packets to the respective subsequent compute node. As an additional advantage, now the clients do not need to be aware of the funnel node synchronization; because the packets are onion encrypted only for the compute nodes, not for the funnel nodes — which we explain in detail shortly. Figure 1 illustrates these compute and funnel phases.

We assume persistent TLS connections between each pair of nodes, so that they have an authenticated and encrypted channel between them. Therefore, even a global network level adversary cannot see the content of a packet passed between two nodes unless one of them is compromised; however the adversary can observe that a packet is passed between them.

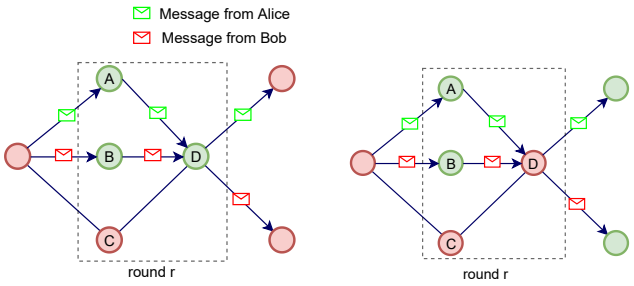
The separation into funnel or compute node is conceptual: the same node can act as a funnel node or a compute node in different rounds. The funnel nodes are picked using the string $\{x_r, x_{r+1}, \ldots, x_{r+\ell}\}$ emitted by the randomness beacon. Each client picks the compute nodes uniformly at random (with replacement) from all available nodes for each hop of an onion packet independent of any other packet or any other hop of the same packet, and constructs the onion packet only for the compute nodes. All the packets in every round go to the designated funnel node; the funnel node shuffles all the received packets and forwards them (without any cryptographic operation) to the compute nodes based on the next node information in the Sphinx packet header. Then the compute node removes one layer of a received onion packet, and forwards the packet immediately to the next designated funnel node.

Funnel nodes act as mix nodes for the messages. All messages will meet in the same funnel nodes as their paths are coordinated by the randomness beacon. Specifically, a node only acts as a funnel node if the randomness beacon determines that it is the funnel node for the current round. The funnel node is a bottleneck of the system in terms of network

bandwidth; however, the funnel node does not have to perform any cryptographic computations, allowing the system to scale up to the full bandwidth capabilities.

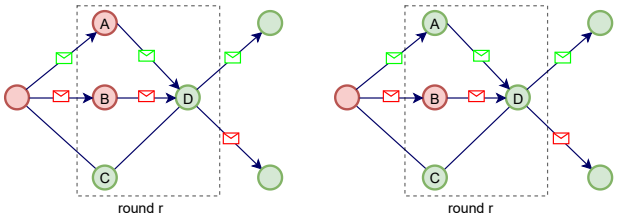
Compute nodes act similar to onion routers [30, 40] — in every round a compute node with onion packets peels a layer of onion for each packet and immediately forwards them to the next designated funnel node.

The pseudocode representations of the protocols run by each honest client and each honest node in the presence of global passive adversaries (that can additionally passively compromise some protocol parties) are presented in Fig. 3 and Fig. 4 respectively. Additionally, we present the defenses for our protocol against active attacks in Section 6.



(a) Messages from Alice and Bob *mix* with each other in a round when they both pass through honest compute nodes and then an honest funnel node D.

(b) Message from Alice and Bob *do not mix* with each other in a round if the funnel node D is not honest even though they pass through honest compute nodes.



(c) Messages from Alice and Bob *do not mix* with each other in a round if none of them passes through honest compute nodes in the same round.

(d) Messages from Alice and Bob *do not mix* with each other in a round if both of them do not pass through honest compute nodes in the same round.

Figure 5: Cases where two messages mix (or not).

It is important to observe that the following event is equivalent to two messages going through an honest mixnode in the core protocol: two messages are processed by some honest compute nodes (not necessarily same) in round r, and then both of them go through the same honest funnel node in round $r+1$. In that case, the two messages achieve "mixing" even if the whole network before and after that is compromised. In Figure 5, we pictorially show the possible cases when two messages can mix (or not).

4.4 Client and Node Synchronization

Note that clients in *Streams* need *not* be synchronized: clients choose the path of compute nodes for their messages and then send them to the first such compute node. Thus, the clients don't need to be aware of the succession of funnel nodes and the rounds in which they are used.

If nodes lose their synchronization with the protocol, the messages they forward don't reach the correct funnel nodes anymore. Consequently, anonymity is significantly harmed. This can be detected by funnel nodes (as the wrong nodes receive messages) and the offending compute node can be fixed or removed from the protocol. While anonymity is reduced, the liveness of the protocol is untouched: the affected messages can still be sent to the next compute node specified on its path that will hopefully be synchronized and allow the message to rejoin the stream. In Section 9.1 go into further depth on the topic of loose synchronization.

5 Security Analysis

In this section we analyze the security of our protocol *Streams* against a global passive adversary that can passively compromise (the compromised parties still follow the protocol) some portion of the nodes. We discuss the required integrity measures for our protocol against active adversaries in Section 6.

In Appendix B we present a detailed security proof that first shows that *Streams* (as presented in Appendix A) can be abstracted by an ideal functionality $\mathcal{F}_{Streams}$ and second proves pairwise unlinkability for $\mathcal{F}_{Streams}$. Here we present the core security properties achieved by *Streams* and their implications.

5.1 Pairwise Unlinkability of *Streams*

First we argue that *Streams* provides pairwise unlinkability of messages for $\delta < \gamma^L$ over L rounds for a constant fraction γ.

Theorem 1 (Security of *Streams*)**.** *Assuming a secure public-key encryption scheme, a secure TLS functionality, an incorruptible randomness beacon, and given a constant fraction* $\frac{c}{K} < 1$*, Streams provides pairwise unlinkability of messages over L rounds up to probability* δ *as in Definition 1, where* $\delta < \gamma^L$ *for* $\gamma = 1 - \left(\frac{K-c}{K}\right)^3$.

The key idea is that two messages get shuffled if they go through an honest funnel node right after going through honest compute nodes (c.f. Figure 5). It is not necessary that they pass through the same honest compute node, however, they need to pass through some honest compute nodes in the same round just before passing through an honest funnel node. If the messages stay in the system long enough (L is sufficiently high), with high probability there will be at least one such honest sequence so that they can mix.

As a consequence of the above theorem, for all $L \in \omega(\log \eta)$ for a security parameter η the adversarial advantage δ is negligible, and all pairs of messages that stays together in the protocol for at least L rounds, get shuffled with overwhelming probability.

$\gamma = 1 - \left(\frac{K-c}{K}\right)^3$ is conceptually the proportion of compute node and funnel node pairs in a round where the two messages cannot mix. In Figure 6a we plot the relationship between γ and the fraction $\frac{c}{K}$ of compromised parties.

If we want to have the same level of concrete security as without compute and funnel nodes, we need to increase latency, or with similar latency the protocol can only be resilient against lesser fraction of compromised nodes. However, one advantage of this construction is that the cost or overhead does not increase linearly with the number of clients, or more importantly, does not even depend on the number of clients. In Fig. 6b, we compare the latency overhead needed for our protocol (with compute and funnel separation) to achieve different level of security with that of our core protocol without compute and funnel separation (refer to Theorem 2). For $\frac{c}{K} \leq 0.2$ the number of rounds only doubles with the separation of duties to achieve the same level of security.

Even though, the separation into compute and funnel phases provides scalability at the cost of security, the δ value still decreases exponentially with latency. We show the relationship between them in Fig. 6c.

5.2 Pairwise Unlinkability of Core Protocol

We also want to analyze the scenario where we do not need to scale horizontally, core protocol described in Section 4.2 is sufficient (e.g., the nodes are as powerful as network routers or the total number of users is less than few thousands).

Theorem 2 (Pairwise unlinkability of the Core protocol)**.** *Assuming a secure public-key encryption scheme, an incorruptible randomness beacon, and given a constant fraction* $\frac{c}{K} < 1$*, the core protocol described in Section 4.2 provides pairwise unlinkability of messages over L rounds up to probability* δ *as in Definition 1, where* $\delta < \left(\frac{c}{K}\right)^L$.

The above theorem also gives us an important insight about how much security is degraded to achieve scalability through our funnel and compute nodes.

6 Defense Against Active Attackers

Resiliency of anonymous communication against active attacks is well studied in the literature [7, 63] and not the main focus of this work. We leverage existing techniques to protect packet integrity and to prevent a total loss of anonymity due to packet dropping. Our scaling methodology is generally orthogonal to the protection against active attacks and we invite the consideration of further protective measures against active attacks. Concretely, we use exactly the same strategy as Loopix [58] to defend against active attacks. Our protocol already makes use of the Sphinx [25] packet format, which comes with confidentiality (including padding) and message integrity, and allows for defense against replay attacks and tagging attacks (see below). Additionally, following Loopix, we incorporate messages (called *loop messages*) that users send to themselves to detect and combat packet drops by an active adversary. We consider the following relevant attacks:

(n-1) Attacks [62]. In such attacks, the adversary blocks all but one target message to a node in order to follow the target message. In case the adversary decides to drop messages from an honest user Alice, Alice will likely not receive her own loop messages back and she will know that the system is under attack. Alice can then spread the word through some public medium so that other users can stop using the system.

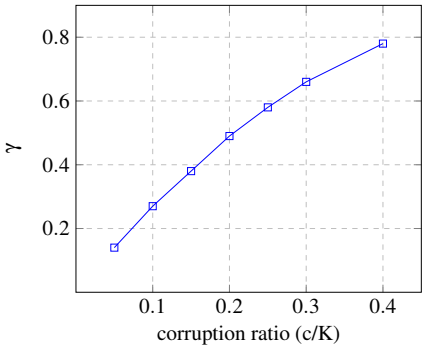
If Alice sends λ loop messages for every real message, the adversary can drop a message from Alice with a probability of $\frac{\lambda}{1+\lambda}$ that Alice will detect the attack (since loop messages are indistinguishable from real messages). Therefore, if the adversary drops k messages from Alice, Alice will detect the attack with probability $1 - \left(\frac{1}{1+\lambda}\right)^k$. As a result, the probability of detection increases drastically with increasing k.

Note that we do not require any specific usage pattern from the users to enable this defense, we only require that they add λ additional messages per real message whenever they are using the system. Additionally, λ can be any constant number, however for our system we conservatively choose $\lambda = 1$. [2]

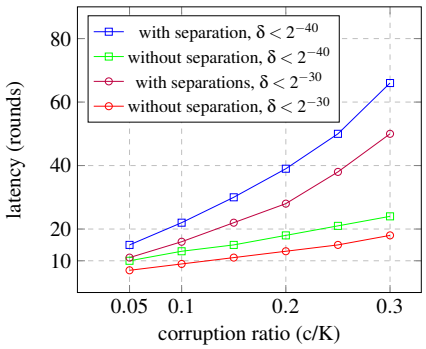
Replay Attacks. Replay attacks are detected and prevented using the replay detection tag implemented in the Sphinx packet header. The *replay detection tag* allows a node to verify if a packet has already been seen or not; if the packet is a replay it is dropped. In our system, we verify the replay detection tag only on the *compute* nodes. For a funnel node, the traffic from a compute node to the funnel node is protected by TLS/SSL, and hence, protected from replay attacks if the compute node is honest. In case the compute node is adversarial, the funnel node anyway cannot ensure *mixing* for the packets coming from that compute node. The subsequent honest compute node could then figure out whether a replay attack occurs, using the replay detection tags.

Tagging Attacks. The Sphinx packet structure also defends against tagging attacks — if the adversary tries to tag a message, the Sphinx packet verification will fail and the packet will be dropped. If a loop message is dropped, the corresponding user will detect the attack.

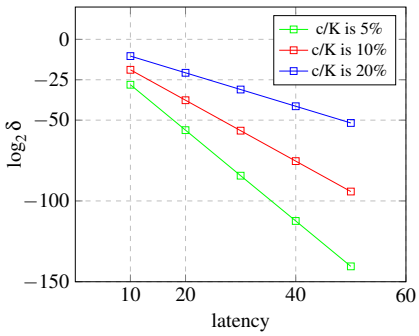
[2] This value corresponds to roughly four times the number of loop messages per other message when compared to Loopix; it can be argued that Loopix' randomized and strictly controlled message sending patterns might confuse some adversaries – we leave these considerations and the exact choice of λ to system designers.



(a) corruption ratio $\frac{c}{K}$ vs. the probability γ of two messages not mixing in a randomly chosen funnel node and its predecessors (compute nodes) in a pair of paths.



(b) corruption ratio $\frac{c}{K}$ vs. the required latency (in rounds) to achieve different levels of security — viz, one in a billion ($\delta < 2^{-30}$) security, and one in a trillion ($\delta < 2^{-40}$) security.



(c) latency (in rounds) vs. $\log \delta$ for different values of corruption ratio $\frac{c}{K}$ (e.g. 5%, 10%, 20%). The linear decrease in $\log \delta$ means exponential decrease in the δ value.

Figure 6: Effective security of introducing compute and funnel phases

7 Implementation

We have developed a proof-of-concept implementation of *Streams* in approximately 5000 lines of Go code (v1.15). Our implementation[3] builds upon the existing Loopix implementation [28] for the cryptographic and sphinx packet realizations[4]; we then add our own implementation for funnel and compute nodes. We make following system considerations.

Synchronization. For the prototype implementation we consider a global clock that every protocol party (clients and nodes) follows. For real deployments, the global clock can be replaced with local clocks in combination with the idea of loose synchronization technique described in Section 9.1.

About Rounds. We decide the round duration based on the load on the system, or more specifically, how many onion packets are there in the system at any given point of time. We are going to demonstrate later that the amount of time it takes to process onion packets is proportional to the number of packets in our system.

Random Shuffle. We implement the Fisher–Yates shuffle [32] to achieve in-memory shuffle of n elements with $\Theta(n)$ computational complexity. This algorithm requires a continual source of randomness — each funnel node uses a locally stored random number table for that purpose. Moreover, shuffling can be preprocessed by shuffling the indexes; when a message comes, it is directly stored in the shuffled position of the index.

[3]The (anonymized) implementation is available at the following link: https://drive.google.com/drive/folders/10EjJGqo0ZzHrLm5V78dSRDdOhR1FQHd0

[4]However, we want to emphasize that our system design is fundamentally different from Loopix, we only use their library to implement cryptographic and sphinx packet operations.

8 Performance Evaluation

In this section we evaluate the performance of *Streams* using our prototype implementation.

8.1 Processing Capacity of Compute Nodes

We first evaluate how many onion packets can be processed by a single compute node in a given amount of time. To that end, we run a standalone compute node, then give the node different number of onion packets to process, and measure the time spent to process those packets. For this experiment, we run a compute node on a machine which has 48 Intel Xeon Silver 4116 processors (3 GHz) with 128 GB RAM.

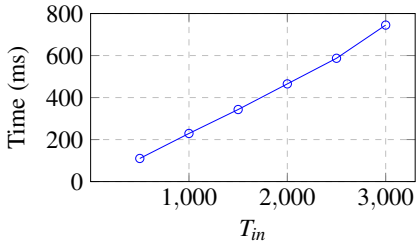


Figure 7: number of packets (T_{in}) sent to a compute node per round (x-axis) vs. time taken in milliseconds (y-axis) for the compute node to process those packets.

In Figure 7, we plot a graph between number of onion packets given to a compute node vs. time taken to process those packets. We take the measurements by repeating the experiment 10 times and taking an average of those.

We note that our crypto library is implemented in the high level language Go. A more optimized implementation in a low-level language can improve the packets processing speed significantly at compute nodes.

8.2 Processing Capacity of Funnel Nodes

Now we evaluate how our funnel nodes can scale for different numbers $(200K, 400K, \cdots, 1M)$ of onion packets even with slow compute nodes. Since, the number of compute nodes is expandable, the funnel nodes dictate the round duration. We want to measure for funnel nodes how much time is taken by the TLS layer to process different number of packets, as well as how much time is taken to run the the shuffle algorithm.

Experimental Setup. To evaluate the performance/scalability of funnel node, we run a standalone funnel node and send varying number of onion packets to that funnel node and measure the time taken for the following two operations: 1. process different number of packets by the TLS layers; 2. run the Fisher–Yates shuffle algorithm for those packets. We run the funnel node program on a machine with 48 Intel Xeon Silver 4116 processors (3 GHz) with 128 GB RAM. We allow the system to spawn up to 100 threads for parallel processing.

Results. We plot our findings in Figure 8. All the measurements are average of 10 runs approximated to the nearest integer. Since we achieve in-memory shuffle using Fisher–Yates algorithm, the observed shuffle time is only around 10 milliseconds even for 1 million packets (c.f. Fig. 8a), even though the whole shuffle protocol runs in a single thread.

On the other hand, processing 1M packets over TLS takes around 112 milliseconds even though we spawn up to 100 threads for TLS processing. The overhead involves AES encryption/decryption for TLS, and handling multiple TLS threads to ensure one thread does not overwrite a packet from another thread. This experiment shows that the dominant factor in deciding the round duration is TLS processing. This overhead can be further improved by having a more optimized implementation to handle the TLS threads.

Memory and Network Overhead. When we run the above benchmarks for the funnel node, we also measure the memory usage of the process and estimate the throughput requirements (amount of data received by the funnel node over the network for those many packets) based on each packet size (2056 Bytes) — we plot them in Fig. 8c. We can observe that, for 1M packets the memory utilization by the server process remains below 3 GB, however, the network throughput requirement can become a bottleneck (receives a total of 2.1 GB in messages). If we choose round duration to be 1 second, to handle 1M packets the responsible funnel node requires a burst network capacity of around 17 Gbps.

There are service providers [33, 57] that supports up to 40 Gbps network speed. Streams nodes do not continuously need a high network capacity, it suffices it they have a high burst capacity and a moderate average capacity. Also, Amazon EC2 c5.18xlarge instances can support 25 Gbps [10] speed. In the future (or) if the system needs to support more than 1M messages per round, the servers can setup Multipath-TCP [4, 45, 56], in order to fulfill the GBps requirement. Moreover, the GBps bandwidth prices fell drastically in the past and there is no reason to expect any change to that trend.

8.3 End-to-end Latency Evaluation

We evaluate the end-to-end latency offered by our protocol in the following way: we choose our round duration to be 1 second which can easily incorporate the TLS processing time at the funnel, the communications between compute and funnel nodes, and the overhead of round synchronization (we discuss about it in Section 9.1); and from Fig. 6b we know the number of rounds required to achieve $\delta \leq 2^{-30}$. If we consider that almost 10% of the total nodes are compromised, we need $\ell = 16$. The makes the end-to-end latency for a message in our protocol to 16 seconds.

If the system can process 1 million messages every round, it can allow about 625K new messages on average in every round (each message stays in the system for $\ell = 16$ rounds). With noise message ratio $\lambda = 1$, the system can support 312K new real messages per round.

Comparison With Other Protocols. If we compare the performance of *Streams* with other provably secure protocols that can serve similar number of total messages in the system, only Karaoke performs better than *Streams* in terms of latency, but at the cost of anonymity (only provides differential privacy with $\varepsilon = \ln 2$ with an end-to-end latency of 6 seconds).[5] Our protocol outperforms Stadium with a significant margin (Stadium has an end-to-end latency of 68 seconds while providing only DP guarantees). While comparing, we consider the total number of messages that can be handled by a system rather than the total number of users. Because, in protocols like Karaoke or Stadium, the users need to send their messages at the beginning of an epoch, and wait for those messages to get delivered before starting the next epoch. In contrast, *Streams* users can send messages whenever they want.

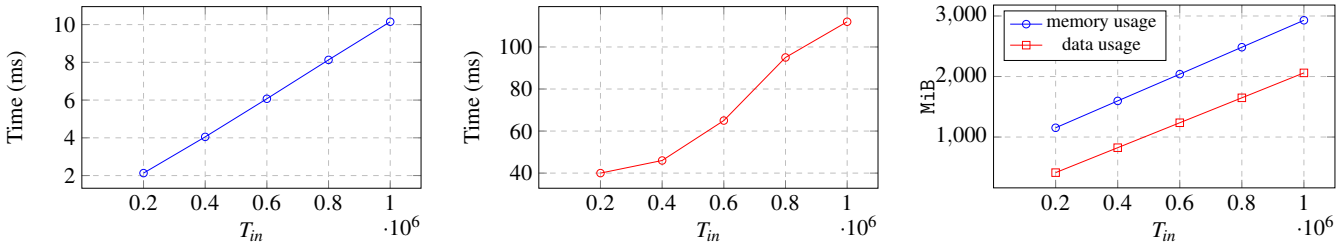
Atom provides much stronger security guarantees ($\delta < 2^{-64}$) than *Streams*, however, *Streams* has much lower end-to-end latency than Atom. We estimate the end-to-end latency of Atom (in their trap message scenario with no churn) from their measurements [50], for $\delta \leq 2^{-30}$ and $\frac{c}{K} = 10\%$ — the estimated end-to-end latency for each message is around 630 seconds, which is much higher than *Streams*.

9 Resiliency Improvements

9.1 Resiliency for Loose Synchronization

In our protocol description in Section 4 we assumed that all the protocol parties are perfectly synchronized. However, it

[5] In our comparison, we are gracious and do not count the worst-case waiting time for sending a message in Karaoke and Stadium, which leads to a factor of 2 in terms of latency.



(a) Number of packets (T_{in}) sent to a funnel node (x-axis) vs. the amount of times in microseconds (y-axis) to run Fisher Yates shuffle for the given number of packets.

(b) number of packets (T_{in}) sent to a funnel node via TLS connections (x-axis) vs. the amount of times in milliseconds (y-axis) to process those packets.

(c) number of packets (T_{in}) sent to a compute node per round vs. the resources consumed in terms of maximum memory usage by a funnel server and the amount of data received.

Figure 8: Onion packets processing at funnel nodes

can be challenging to maintain such synchronization continuously. Here we discuss how to relax that assumption by allowing each protocol party to follow their own local clock.

We assume that the maximum difference between two local clocks of the nodes is bounded by μ milliseconds. The clients do not need to keep track of rounds at all, and can send messages to the system whenever they want. A client sends an onion packet to the first compute node on the onion path. The compute node based on its local clock can decide which funnel node to forward the packets to. As long as μ is lower than a few hundred milliseconds, we can add μ in the computation of round duration to handle the synchronization gap among the nodes. We can still have a reference global clock which the nodes can synchronize their local clocks with from time-to-time. We only need equivocation protection from that global clock, the protocol does not depend on that clock for the anonymity property.

However, suppose nodes (at most 10% for example) in the system have a difference of more than few hundred milliseconds with the reference global clock. Such unsynchronized compute nodes can send packets to wrong funnel nodes. If a node receives an onion packet that it is not supposed to receive (probably a dishonest or badly synchronized compute node has sent the packet to a wrong funnel node), the node just forwards the packet to the correct compute node (according to the onion packet header) at the end of the round. Thus, the protocol still functions properly, although the latency needs to be increased based on the amount of such unsynchronized (and compromised) nodes to provide same level of mixing.

With the above modified approach, a node does not have to derive at which round it should act as a funnel node or compute node. For all the onion packets (according to the onion headers) if it is the intended compute node, it acts as a compute node; for all the rest of the packets it acts as a funnel node and forwards them to the next corresponding compute nodes at the end of the round.

9.2 Denial-of-Service (DoS) Attacks Against Funnel Nodes

Although our formal security analysis does not consider DoS attacks, our design of having one funnel node per round introduces a single point failure against such attacks. If a powerful adversary is able to redirect the DoS attack to the next funnel node within a span of one round, and can keep doing that, it will be able to block the whole system.

Defense. To defend against such attacks, we utilize the fact that each pair of nodes have a persistent TLS connection between them. If the funnel node is under attack, when the compute nodes send packets to the funnel they will not receive any TCP/IP acknowledgments for the dropped packets. Here we only consider the scenario where the funnel is under attack, and not the scenario where a malicious funnel intentionally drop packets. We discuss about that scenario in Section 9.3.

In our defense strategy, whenever the compute nodes detect such an attack against the funnel nodes, the compute nodes will shuffle the packets they locally have and directly forward them to the next compute nodes at the end of the round. This will compromise anonymity but provide availability for the system when the funnel nodes are under attack.

Assuming limited capacity of the adversary to DoS the funnel nodes, we can consider that only a constant fraction ι of funnel nodes will under attack; and the anonymity provided by the protocol will be lower bounded by that when a total of $\iota + c$ nodes are compromised[6].

If the adversary is really strong and can attack all the funnel nodes one after another, the overall anonymity of the system reduces to the anonymity provided by Karaoke but without the noise messages from the servers. However, considering a constant fraction of honest users, differential privacy bounds similar to Karaoke [52] can be derived. We consider the exact bounds in such scenarios to be out of scope for this work, and leave it for future work.

[6]If a funnel node is compromised, the protocol anyway does not achieve any *mixing* on that funnel node; and it is equivalent to directly forwarding the message to the next compute node.

9.3 Avoiding Single Points of Failure

The scaling technique presented here and exemplified within *Streams* regularly funnels all packets through a single node (hence the term funnel nodes). While this allows the maximal number of packets to mix, it also inherently creates single points of failure: a funnel node can get overwhelmed and process packets more slowly than anticipated or it could even fail outright. We could argue that this is not entirely new for anonymous communication protocols, but that does not alleviate the concern in all cases.

We consider a comprehensive analysis of different modes of node failures and resiliency assumptions out of scope for this paper and thus only present brief considerations. Rare and short delays and slowdowns of funnel nodes can be considered as issues with synchronization (see above). If the risk of total failure or repeated and significant delays is deemed too high, then one or more redundant funnel nodes can be introduced as follows: every round clients and compute nodes alike send a copy of each packet to each of the two or more funnel nodes selected for that round. In the next round each compute node gets the messages from all funnel nodes and can verify that they worked properly. Since clients choose the sequence of compute nodes, the packets from different funnels still visit the same compute nodes.

Implementing this method naturally increases the overall network load (as twice as many packets are sent to and form compute nodes); this does not lead to new bottlenecks, as compute nodes are limited mostly by the complexity of computation and should be able to handle an increase in network load, while the load for funnel nodes is unchanged. However, redundant funnel nodes slightly decrease the guarantees provided: our analysis holds as provided, except that if any one of the two funnel nodes is compromised then we are in the case where "the funnel node is compromised". This effectively decreases the security provided slightly. [7] We leave further analyses and considerations to future work.

10 Application Considerations

10.1 Applicability of Our Scaling Technique

One common bottleneck in traditional mixnet based systems is the processing power of the nodes — the total number of users the system can serve is restricted by the processing power of the weakest node. To avoid this issue, many systems [52, 58, 65, 66] employ parallel mixnets to scale with the number of users. However that approach does not always provide provable unlinkability. Karaoke [52], Stadium [65], Vuvuzela [66] achieve anonymity in the differential privacy sense at the expense of latency overhead and the chance of mixing degrades with the number of parallel paths.

Compared to those protocols, the latency required to maintain the same level of anonymity with our scaling technique grows more gracefully with the number of users (does not grow at all up to 1 million messages). Other protocols can in fact make use of our scaling technique of splitting the mixing and computing responsibilities to improve their scalability/privacy properties. Below we describe how our scaling technique can be used to improve some example protocols:

Loopix [58]. When Loopix needs to scale for many users it employs multiple paths, which in turn reduces the chance of two messages mixing with each other. Instead Loopix can split the responsibilities in the following way: the randomized delay and mixing of messages happens at a funnel node, while the onion decryption happens at a compute node. This separation of duties would not introduce fixed-length rounds to Loopix, thus allowing Loopix to keep the desired asynchronous model. To keep a comparable level of security as well as the overall structure of the protocol, the paths chosen by clients now include both funnel nodes and compute nodes. In exchange, the separation of duties could drastically reduce the requirement to expand the number parallel paths for Loopix, and hence, could guarantee better mixing.

Karaoke [52]. Since Karaoke already works in rounds, it is easier for Karaoke to adopt our scaling technique. Depending on the number of messages that have to be processed per round, the nodes would choose one or more funnel nodes after the compute phase (e.g., one funnel node per million of messages). If the number of users in the system exceeds several million, the messages can be randomly distributed among a few funnel nodes, e.g., by using a hash function with the message and the random number from the randomness beacon as input. In that case, each funnel node will achieve shuffling for the subset of messages it receives. As this subset would be randomly chosen, over log-many rounds pairwise shuffling will occur. As a result, this separation of duties increases the chance of mixing, and it reduces the number of parallel paths. Reducing the number of in parallel paths in turn further improves the required number of round until messages mix, i.e., until mixing can be proven.

Vuvuzela [66]. Vuvuzela employs a single chain of nodes, and can directly enjoy the benefits of efficient scaling using our technique. The extension would be very similar to how *Streams* employs the scaling technique in this paper — each Vuvuzela node can be replaced with a funnel node and a bunch of compute nodes. By employing many compute nodes, Vuvuzela can significantly reduce the time required to process packets in a round, and thus reduce the overall end-to-end latency by a significant factor, or scale for more number of users with similar end-to-end latency, while maintaining high level of anonymity.

[7] In Theorem 3, where we show that there is a constant $0 < \gamma < 1$, we choose $\gamma = 1 - \left(\frac{K-c}{K}\right)^3$. This changes to $\gamma = 1 - \left(\frac{K-c}{K}\right)^{3+\tau}$ if we use τ redundant funnels for a total of $\tau + 1$ funnel nodes per round. The overall complexity remains; *gamma* is just a larger constant.

10.2 Application Scenarios of *Streams*

Our performance analysis clearly demonstrates that *Streams* can scale well with a large number of users. Beyond the traditional mixnet applications such as anonymous e-mailing, we find *Streams* to be useful to applications such as network-level anonymity for publishing blockchain transaction [44], and anonymous microblogging.

In cryptocurrency networks such as Bitcoin and Ethereum, when users publish their transactions to the networks, the network-level adversaries can easily link these transactions to their respective users' IP addresses. Deanonymization attacks in the blockchain space enables targeting users with substantial stakes/coins. Unlike currently considered/employed solutions in this space such as Tor, Dandelion++ [36], *Streams* can offer provable security guarantees without introducing an unacceptable delay: as the consensus process already takes up to a few minutes in these public blockchain environments, latency delay of a few second for provable network anonymity can be acceptable.

Similar to Atom [50], Riposte [22] and Express [34], *Streams* with its latency of only a few seconds can also be useful for scenarios where users want to broadcast short messages in a provably anonymous manner. Such an anonymous microblogging service can be particularly useful to whistleblowers and protesters against authoritarian regimes, and freedom of expression in general.

11 Conclusion and future work

In this paper we introduced a scaling technique for anonymous communication protocols that distributes cryptographic computations while still allowing messages to meet and mix. We demonstrated the applicability of our scaling technique with the sample protocol *Streams* by scaling it for a million messages while keeping the end-to-end latency as low as 16 seconds, and guaranteeing good pairwise unlinkability of messages with 10% compromised nodes in the system. The property that the users are not required to be synchronized with rounds is an added advantage of *Streams* (and our scaling technique in general) over other round-based protocols like Karaoke, Atom, or Stadium. Our scaling technique can be leveraged by protocol designers because of its relevance for other protocols (demonstrated through the examples of Loopix, Vuvuzela, and Karaoke) to improve scalability and mixing guarantees through network link saturation.

Our scaling technique also comes with its own limitations — (i) beyond 1M messages the Gbps connection speed requirements of the nodes can become a bottleneck; (ii) the nodes still need to have some level of round synchronization among themselves. An interesting future work can be to evaluate the exact system parameters to achieve a desired level of security when multiple funnel nodes are employed in parallel to scale the system beyond several millions of messages.

References

[1] blockchain oracle service, enabling data-rich smart contracts. https://provable.xyz/, accessed 2021.

[2] Generate random numbers for smart contracts using chainlink vrf, accessed 2021.

[3] RFC 8446. The transport layer security (tls) protocol version 1.3. https://tools.ietf.org/html/rfc8446, accessed April 2021.

[4] RFC 8684. Tcp extensions for multipath operation with multiple addresses. https://datatracker.ietf.org/doc/html/rfc8684, accessed April 2021.

[5] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder–scalable, robust anonymous committed broadcast. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1233–1252, 2020.

[6] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous Messaging via Secure Multiparty Computation. In *Proceedings of the 26th USENIX Security Symposium*, pages 1217–1234. USENIX Association, 2017.

[7] Mashael Alsabah and Ian Goldberg. Performance and security improvements for tor: A survey. *ACM Comput. Surv.*, 49(2), September 2016.

[8] Megumi Ando, Anna Lysyanskaya, and Eli Upfal. Practical and Provably Secure Onion Routing. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 144:1–144:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[9] Megumi Ando, Anna Lysyanskaya, and Eli Upfal. On the complexity of anonymous communication through public networks. *CoRR*, abs/1902.06306, 2019.

[10] Amazon AWS. Amazon EC2 instance network bandwidth. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html, Accessed January 2021.

[11] Michael Backes, Aniket Kate, Praveen Manoharan, Sebastian Meiser, and Esfandiar Mohammadi. AnoA: A Framework For Analyzing Anonymous Communication Protocols. In *Proc. 26th IEEE Computer Security Foundations Symposium (CSF 2013)*, pages 163–178, 2013.

[12] Ludovic Barman, Italo Dacosta, Mahdi Zamani, Ennan Zhai, Apostolos Pyrgelis, Bryan Ford, Joan Feigenbaum,

and Jean-Pierre Hubaux. Prifi: Low-latency anonymity for organizational networks. *Proceedings on Privacy Enhancing Technologies*, 2020:24–47, 10 2020.

[13] K. S. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. C. Sicker. Low-resource routing attacks against tor. pages 11–20, 2007.

[14] Adithya Bhat, Nibesh Shrestha, Aniket Kate, and Kartik Nayak. Randpiper - reconfiguration-friendly random beacons with quadratic communication. *IACR Cryptol. ePrint Arch.*, (1590), 2020.

[15] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[16] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. pages 136–145, 2001.

[17] David Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 4(2):84–88, 1981.

[18] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[19] David Chaum, Debajyoti Das, Farid Javani, Aniket Kate, Anna Krasnova, Joeri de Ruiter, and Alan T. Sherman. cmix: Mixing with minimal real-time asymmetric cryptographic operations. In *15th International Conference on Applied Cryptography and Network Security 2017*, 2017.

[20] Chen Chen, Daniele E. Asoni, David Barrera, George Danezis, and Adrian Perrig. HORNET: High-speed onion routing at the network layer. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 1441–1454, 2015.

[21] Information Technology Laboratory Computer Security Division. Interoperable randomness beacons: Csrc, accessed April 2021.

[22] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015.

[23] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable Anonymous Group Messaging. pages 340–350, 2010.

[24] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively Accountable Anonymous Messaging in Verdict. In *Proc. 22nd USENIX Security Symposium*, pages 147–162, 2013.

[25] George Danezis and Ian Goldberg. Sphinx: A Compact and Provably Secure Mix Format. pages 269–282, 2009.

[26] D. Das, S. Meiser, E. Mohammadi, and A. Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 108–126, May 2018. extended version under https://eprint.iacr.org/2017/954.

[27] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Comprehensive anonymity trilemma: User coordination is not enough. *Proceedings on Privacy Enhancing Technologies*, 2020:356–383, 07 2020.

[28] Deepmind. Anonymous messaging using mix networks. https://github.com/deepmind/loopix-messaging, Accessed January 2021.

[29] R. Dingledine and N. Mathewson. Tor Protocol Specification. https://gitweb.torproject.org/torspec.git/, Accessed March 2021.

[30] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 21, USA, 2004. USENIX Association.

[31] Drand. Drand - a distributed randomness beacon daemon, Accessed April 2021.

[32] Manuel Eberl. Fisher-yates shuffle. *Arch. Formal Proofs*, 2016, 2016.

[33] Hurricane Electric. Hurricane Electric internet services. http://he.net/ip_transit.html, Accessed July 2021.

[34] Saba Eskandarian, Henry Corrigan-Gibbs, M. Zaharia, and D. Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. *ArXiv*, abs/1911.09215, 2019.

[35] N. S. Evans, R. Dingledine, and C. Grothoff. A Practical Congestion Attack on Tor Using Long Paths. pages 33–50, 2009.

[36] Giulia C. Fanti, Shaileshh Bojja Venkatakrishnan, Surya Bakshi, Bradley Denby, Shruti Bhargava, Andrew Miller, and Pramod Viswanath. Dandelion++: Lightweight cryptocurrency networking with formal anonymity guarantees. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2):29:1–29:35, 2018.

[37] Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi, and Jörg Schwenk. Universally composable security analysis of tls. In Joonsang Baek, Feng

Bao, Kefei Chen, and Xuejia Lai, editors, *Provable Security*, pages 313–327. Springer Berlin Heidelberg, 2008.

[38] Nethanel Gelernter, Amir Herzberg, and Hemi Leibowitz. Two cents for strong anonymity: the anonymous post-office protocol. 01 2017.

[39] Sharad Goel, Mark Robson, Milo Polte, and Emin Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. 2003. https://www.cs.cornell.edu/people/egs/herbivore/herbivore.pdf.

[40] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Onion Routing. *Commun. ACM*, 42(2):39–41, 1999.

[41] Philippe Golle and Ari Juels. Dining cryptographers revisited. In *Proc. of Eurocrypt 2004*, 2004.

[42] Mads Haahr. True random number service, Accessed April 2021.

[43] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[44] Ryan Henry, Amir Herzberg, and Aniket Kate. Blockchain access privacy: Challenges and directions. *IEEE Security Privacy*, 16(4):38–45, 2018.

[45] Benjamin Hesmans and Olivier Bonaventure. Tracing multipath tcp connections. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 361–362, New York, NY, USA, 2014. Association for Computing Machinery.

[46] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. Users get routed: Traffic correlation on tor by realistic adversaries. In *Proc. ACM SIGSAC conference on Computer & communications security 2013*, pages 337–348, 2013.

[47] Aniket Kate and Ian Goldberg. Using Sphinx to Improve Onion Routing Circuit Construction. pages 359–366, 2010.

[48] Aniket Kate, Greg M Zaverucha, and Ian Goldberg. Pairing-based onion routing with improved forward secrecy. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):1–32, 2010.

[49] Christiane Kuhn, Martin Beck, and Thorsten Strufe. Breaking and (Partially) Fixing Provably Secure Onion Routing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 168–185. IEEE, 2020.

[50] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 406–422, New York, NY, USA, 2017. Association for Computing Machinery.

[51] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An Efficient Communication System With Strong Anonymity. In *Proc. Privacy Enhancing Technologies Symposium (PETS 2016)*, pages 115–134, 2016.

[52] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 711–725, Carlsbad, CA, October 2018. USENIX Association.

[53] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: Strong metadata security for voice calls. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 211–224, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A Scalable, Traffic Analysis Resistant Anonymity Network for VoIP Systems. In *Proc. ACM Conference on Special Interest Group on Data Communication (SIGCOMM 2015)*, pages 639–652, 2015.

[55] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 887–903, 2019.

[56] IP networking lab. MultiPath TCP - Linux Kernel implementation. https://www.multipath-tcp.org/, Accessed January 2021.

[57] OVHcloud. Customizable public bandwidth to reach your full potential. https://www.ovhcloud.com/de/bare-metal/bandwidth/, Accessed July 2021.

[58] Ania Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *Proc. 26th USENIX Security Symposium*, 2017.

[59] Michael O Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409. IEEE, 1983.

[60] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. P2P Mixing and Unlinkable Bitcoin Transactions. In *Proc. 25th Annual Network & Distributed System Security Symposium (NDSS)*, 2017.

[61] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. Hydrand: Practical continuous distributed randomness. In *2020 IEEE Symposium on Security and Privacy (SP). IEEE*, 2020.

[62] Andrei Serjantov, Roger Dingledine, and Paul Syverson. From a trickle to a flood: Active attacks on several mix types. volume 2578, 02 2003.

[63] Fatemeh Shirazi, Milivoj Simeonovski, Muhammad Rizwan Asghar, Michael Backes, and Claudia Díaz. A survey on routing in anonymous communication protocols. *ACM Comput. Surv.*, 51(3):51:1–51:39, 2018.

[64] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.

[65] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. pages 423–440, 10 2017.

[66] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP 2015)*, 2015.

[67] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in Numbers: Making Strong Anonymity Scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182, 2012.

A Protocol Description in UC Framework

We described the design of *Streams* in Section 4. Here we formally present the protocol design of *Streams* in the UC-framework.

A.1 Model

We use a hybrid world UC model [16] to represent our protocol – where the protocol has access to some additional ideal (hybrid) functionalities that is available to the protocol as well as the adversary. A protocol party (an honest user or node) or the adversary can access such a functionality through an incorruptible ITI $\mathcal{F}$ that provides certain ideal guarantees, e.g., clock time, key registration etc. More specifically, our formalization uses four such functionalities: a round-based communication functionality $\mathcal{F}_{round}$, a globally available randomness beacon $\mathcal{F}_{CRF}$, a key registration functionality $\mathcal{F}_{RKR}$, and a secure channel functionality $\mathcal{F}_{SCS}$. The environment $\mathcal{E}$ can access those ideal functionalities either through the protocol parties or through the adversary.

Array Rounds := {**false**, ..., **false**} // Array of length $\mathsf{K}+\mathsf{N}+\mathsf{N}'$
Round := 0; PartiesIncremented := 0
QUEUE = a queue where the incoming messages are stored

QueryRound() from $\mathcal{A}$ or $\mathcal{F}_{CRF}$ or party i:

return {Round, Rounds}

RequestRound() from party i:

return Rounds[i]

NextRound() from party i:

if Rounds[i] = **true** **then**
  return "invalid action"
else
  Rounds[i] ← **true**; PartiesIncremented += 1
end if
if PartiesIncremented = $\mathsf{K}+\mathsf{N}+\mathsf{N}'$ **then**
  Round += 1; PartiesIncremented ← 0
  Reset Rounds[j] := **false** $\forall j : 0 \leq j < \mathsf{K}+\mathsf{N}+\mathsf{N}'$
  Forward all elements of QUEUE to $\mathcal{A}$; empty QUEUE
end if

Upon receiving msg $(P, P_{next}, O, \text{round})$ **from** *Streams*

if round = Round **then**
  ADD $(P, P_{next}, O, \text{Round})$ to QUEUE
end if

Figure 9: Round Functionality $\mathcal{F}_{round}$

Round Functionality $\mathcal{F}_{round}$. We introduce a hybrid functionality $\mathcal{F}_{round}$ (see Figure 9) to enforce rounds on the protocol parties. We ensure that the environment $\mathcal{E}$ activates the honest parties in every round. $\mathcal{F}_{round}$ ensure, though, that the environment $\mathcal{E}$ cannot activate a protocol party multiple times in the same round by keeping track of the *Rounds*[i] flag for each party i (including both clients and nodes). Additionally, it ensures that all the network packets intended to send for a given round is not send before or after that round to an honest protocol party. As a consequence, the environment can stop the entire protocol at anytime. As then no messages would be delivered anymore, stopping the entire execution does not leak any information to the environment.

Randomness Beacon Functionality $\mathcal{F}_{CRF}$. We assume that each protocol party (including the adversary) has access to an

crf = an infinitely long random string.

GetFunnels(ℓ):

round, $\leftarrow$ QueryRound()
return $\{\mathsf{crf}[\mathtt{round}] \mod \mathsf{K}, \ldots, \mathsf{crf}[\mathtt{round}+\ell-1] \mod \mathsf{K}\}$

Figure 10: Randomness Beacon Functionality $\mathcal{F}_{CRF}$

incorruptible randomness beacon. In particular, future values of this beacon are not known to the adversary. We model this beacon with an ideal functionality $\mathcal{F}_{CRF}$ (see Fig. 10) that outputs each time a ℓ-long substring of an infinite random string beacon. Using that ℓ-length string a protocol party can derive the next ℓ funnel nodes.

Key registration functionality $\mathcal{F}_{RKR}$. The key registration functionality $\mathcal{F}_{RKR}$ is solely used by the subprotocol Π_{sub} from [49], which handles all cryptographic operations. Π_{sub} is treated in a black-box manner throughout this section. For completeness, we provide a description of Π_{sub} and $\mathcal{F}_{RKR}$ in Appendix D.

Secure Channel Functionality $\mathcal{F}_{SCS}$. We also use the secure communications sessions functionality $\mathcal{F}_{SCS}$ from the work of Gajek et al. [37, Figure 4]. They show that $\mathcal{F}_{SCS}$ abstracts the TLS [3] protocol. It is crucial to note here that all the protocol parties in our model work in rounds, and therefore, $\mathcal{F}_{SCS}$ as well forwards all the messages to the $\mathcal{F}_{round}$ functionality instead of the environment; the $\mathcal{F}_{round}$ functionality in turn forwards those messages to the environment when the round ends.

Packet format. We use the Sphinx packet design [25, 47] to ensure that all messages are end-to-end encrypted; we call them "onion packets". The Sphinx packet design also guarantees that an intermediate node, just by looking at a packet, does not learn any information beyond the routing information needed to forward the message to the next node — it hides the path length and the relay position of the node on the path, and the node does not learn anything other than the next nodes on the path. The security properties of the packet design is already incorporated in the onion subprotocol Π_{sub} that we use from the work of Kuhn et al. [49].

A.2 The Core Protocol

First we present the core protocol that does not scale well with the number of users. Then in Appendix A.3 we describe our complete protocol with horizontal scaling. Our protocol has two kinds of parties — clients and nodes. So we define our protocol in two parts as well — clients and nodes. Additionally, the protocol parties as well as the adversary have access to the hybrid functionalities as described above (in Appendix A.1).

Clients. Whenever a client wants to send a message m, she decides the delay d for every message by picking a number from $[\frac{\ell}{2}, \ell-1]$ following a distribution D. In general D can be any discrete probability distribution; however, in a typical setting we assume D to be uniform in $[\frac{\ell}{2}, \ell-1]$.

The client derives the path of a packet based on the string returned by the randomness beacon. For a given round r if $\mathcal{F}_{CRF}$ returns the string $\{x_r, x_{r+1}, \ldots, x_{r+\ell}\}$, any onion packet constructed at round r will be constructed for the path of nodes $\{x_r, \ldots, x_{r+d}, R\}$ for a delay d and intended recipient R. The client will send the onion packet to node x_r at round r.

All other clients as well nodes sends their packets to the node x_r at round r, since $\mathcal{F}_{CRF}$ returns the same node $\{x_r, x_{r+1}, \ldots, x_{r+\ell}\}$ to all the parties (clients, nodes, and the adversary) at round r.

Nodes. The nodes act similar to onion routers [30, 40] except a node in our protocol accepts packets only in the rounds indicated by the randomness beacon. More formally, when a node receives a packet in round r, it checks if x_r matches its own id for a string $\{x_r, x_{r+1}, \ldots, x_{r+\ell}\}$ returned by $\mathcal{F}_{CRF}$ — if not, it rejects the packet. If x_r matches its id, the node with onion packets peels a layer of onion for each packet and forwards them to the next destination.

A.3 Horizontal Scaling With Compute and Funnel Phases

One major bottleneck in the above protocol is the processing power of the nodes — the total number of clients the system can serve is restricted by the processing power of the weakest node. We propose to separate duties such that, instead of one node processing all the onion packets in a round, many nodes come together to share the processing load. Each round is then separated into a *compute phase*, where the task of onion decryption to prohibit linking is distributed over all nodes, and a *funnel phase*, where a randomly chosen node collects and mixes all the messages. In UC-realization of our protocol, we split those phases into separate rounds, to avoid having two sequential communications within a single round. Therefore, one single round of our original protocol maps to two rounds in the UC-version. In the compute round, each compute node processes the packets with them and forwards them to the designated funnel node, then in the funnel round, the funnel node shuffles all the messages that it received in the last round, and forwards them to the respective subsequent compute nodes.

The funnel nodes are picked using the string $\{x_r, x_{r+1}, \ldots, x_{r+\ell}\}$ emitted by the randomness beacon $\mathcal{F}_{CRF}$. Each client picks the compute nodes uniformly at random (with replacement) from all available nodes for each hop of an onion packet independent of any other packet or any other hop of the same packet. All the packets in every even round go to the next designated funnel node. Then in the next (odd) round, the funnel node shuffles all the received packets and forwards them

QUEUE = a FIFO queue.

SendMessage(msg, R) **from party** i
  $r \leftarrow QueryRound()$
  if round $\neq r$ **then**
    reject packet and exit
  end if
  ADD (msg, R) to QUEUE

Upon new round from $\mathcal{E}$**:**
  boolean flag := **RequestRound**() // defined in $\mathcal{F}_{round}$
  if flag $\neq$ **true then**
    return "invalid action"
  end if
  if round mod 2 = 0 // compute round **then**
    while QUEUE is not empty **do**
      $(msg, R) \leftarrow$ dequeue QUEUE;
      $d \leftarrow DelayDistribution\left(\frac{\ell}{2}, \ell - 1\right)$
      $\{x_1, \ldots, x_d\} \xleftarrow{\$} I^d$; $p := \{x_1, \ldots, x_d, R\}$
      call `Process_new_onion`$(self, msg, d+1, p)$ from Π_T
    end while
  end if
  $NextRound()$

Upon receiving a message msg **from** Π_{sub}**:**
  Output "Message msg received" to $\mathcal{E}$

Figure 11: Client Protocol Design Π_{client} as described in Appendix A.3

Π_T:`Process_new_onion`$(self, msg, d+1, p)$
  Call `Process_new_onion`$(self, msg, d+1, p)$ from Π_{sub}.
  Intercept the network packet $packet$ and send it to Π_{rer}.

Π_T: `Forward_Onion`(O)
  Call `Forward_Onion`(O) in the subprotocol Π_{sub}.
  Intercept the network packet $packet$ and send it to Π_{rer}.

Π_{rer}**: Upon a packet** $packet$
  $\_, funnel \leftarrow$ GetFunnels(2) // Select next funnel node
  Send $packet$ over $\mathcal{F}_{SCS}$ to $funnel$.

Figure 12: Π_T and Π_{rer}

(without any cryptographic operation) to the compute nodes based on the next node information in the Sphinx packet header. Then, again in the even round, the compute node removes one layer of the onion packet, and forwards the packet immediately to the next designated funnel node.

The packets are onion encrypted only for the compute nodes, not for the funnel nodes. Additionally, we assume authenticated and encrypted channel between each pair of nodes which is realized by the $\mathcal{F}_{SCS}$ functionality. The pseudocode representations of the protocols run by each honest client and each honest node are presented in Fig. 11 and Fig. 13 respectively.

B Security Analysis (extended)

In this section we formally analyze the security of our protocol *Streams* against a global passive adversary that can statically and passively compromise (the compromised parties still follow the protocol) some portion of the nodes.

For proving security, we first prove that an intermediary representation, a UC ideal functionality $\mathcal{F}_{Streams}$, of *Streams* that does not rely on cryptographic operations but on shared memory. This ideal functionality $\mathcal{F}_{Streams}$ is carefully crafted such that all attacks on *Streams* can be mounted on $\mathcal{F}_{Streams}$ as well, against a wide range of attacker capabilities. In a second step, we prove pairwise unlinkability for the faithfully abstracted ideal functionality, which in turn implies pairwise unlinkability for *Streams*.

We say that the ideal functionality $\mathcal{F}$ *is realized by* a protocol Π if all attacks (within the execution model) that can be mounted on Π can be translated to attacks on Π, for a wide range of attacker capabilities. An ideal functionality, like $\mathcal{F}_{Streams}$, can abstract away from cryptographic details while faithfully modeling all weaknesses of the protocol.

B.1 An Ideal Functionality for AC Protocols

The ideal functionality $\mathcal{F}_{Streams}$ basically acts as as trusted third party to whom users tell that they would like to anonymously send a message. This trusted third party leaks as much information as *Streams* would leak. Due to the regular meeting points at funnel nodes and TLS protection, $\mathcal{F}_{Streams}$ does not need to leak which onion is sent from which compute node to which compute node, as long as the party that sends the onion and the next subsequent funnel node are honest. Removing the very leakage of which onion is sent to whom enables us to prove a strong shuffling property for Theorem 3, pairwise unlinkability with overwhelming probability (see Definition 1).

Formally, the ideal functionality $\mathcal{F}_{Streams}$ provides API calls for when clients want to send a message and they react to network messages. Moreover, as we consider a round-based protocol and the UC-framework is a sequential activation

INPUT_QUEUE = a queue where incoming messages are stored
OUTPUT_QUEUE = a queue where outgoing messages are stored
nodeID := a unique ID in $[0, \mathsf{K}-1]$

Upon input message (onion packet O):

ADD O to INPUT_QUEUE

Upon new round from $\mathcal{E}$:
boolean flag := **RequestRound**() // defined in $\mathcal{F}_{round}$
if flag $\neq$ **true then**
return "invalid action"
end if
$funnel$:= GetFunnels(1)
if round $\mod 2 = 1$ AND nodeID $= funnel$ **then**
Π_{funnel}
else if round $\mod 2 = 0$ **then**
Π_{worker}
end if
swap INPUT_QUEUE and OUTPUT_QUEUE.
NextRound()

Π_{funnel}**:**
Shuffle OUTPUT_QUEUE
while OUTPUT_QUEUE is not empty **do**
$O \leftarrow$ dequeue the first element from OUTPUT_QUEUE

Forward O to $\mathcal{F}_{SCS}$
end while

Π_{worker}**:**
while OUTPUT_QUEUE is not empty **do**
$O \leftarrow$ dequeue the first element from OUTPUT_QUEUE

call *Forward_Onion*(O) from the subprotocol wrapper Π_T
end while

Figure 13: Node Protocol Design as described in Appendix A.3

inputBuffer[] an array of queues to store messages for nodes
crf = an infinitely long random string
queue = a hashmap
round := 0; *newRound*[] := {false, false, ...};
partyCount := 0

Upon new round from $\mathcal{E}$ for party P:
if *newRound*(P) = **true then**
return "invalid action"
end if
set *newRound*(P) := **true** ; *partyCount*+ = 1
if round is odd (funnel round) AND P is a client **then**
$(m, R, t) \leftarrow$ dequeue *inputBuffer*[P]
$d \leftarrow$ *DelayDistribution*($\frac{\ell}{2}, \ell - 1$); $\{x_1, \ldots, x_d\} \xleftarrow{\$} I^d$
if $!\exists x \in \{x_1, \ldots, x_d\}$ such that $x_a \in \mathtt{I_h}$ **then**
Send $(m, x_1, \ldots, x_d)$ to $\mathcal{S}$
else
let x_a := the **first** honest party on the path $\{P, x_1, \ldots, x_d\}$
Generate a random message q
Send $(q, x_1, \ldots, x_a)$ to $\mathcal{S}$
store $(q, x_a, m, x_{a+1}, \ldots, x_d, R)$ in *queue*(round $+ a$)
end if
end if
if round is even (compute round) AND *partyCount* $=$ N $+$ K **then**
SendInformation()
end if
NextRound(P)

Upon input message (m, R, t) from $\mathcal{E}$ for party P:
if round $\neq t$ **then**
reject packet and exit
end if
Add (m, R, t) in *inputBuffer*[P]

Upon receiving a message m for party P:
Output m to $\mathcal{E}$

Figure 14: Ideal functionality $\mathcal{F}_{Streams}$

SendInformation()
  $y := \mathsf{crf}[\mathtt{round}+1]\%\mathsf{K}$
  for each $(q, x_a, m, x_{a+1}, \ldots, x_d, R) \in queue(\mathtt{round})$ **do**
    Remove $(q, x_a, m, x_{a+1}, \ldots, x_d, R)$ from $queue(\mathtt{round})$
    $link := q$
    if $y \in I_h$ **then**
      $link := \perp$
    end if
    let x_ϕ be the **next** honest node on the path $\{x_{a+1}, \ldots, x_d\}$
    if there is **no** such x_ϕ **then**
      Add $(link, m, x_{a+1}, \ldots, x_d, R)$ in a temporary queue Q
    else
      generate a random message q'
      Add $(link, q', x_{a+1}, \ldots, x_\phi)$ in Q
      Add $(q', x_\phi, m, x_{\phi+1}, \ldots, x_d, R)$ to
        $queue(\mathtt{round}+\phi-a)$
    end if
  end for
  Shuffle the elements of Q and send them to $\mathcal{S}$

Figure 15: Leakage from the ideal functionality $\mathcal{F}_{Streams}$

framework (to simplify the analysis), we formally need a "new round" API call.

The ideal functionality expects input messages of the form (msg, R, t). As the protocol works in rounds, $\mathcal{F}_{Streams}$ stores the input messages in an input queue. Upon the "new round"-command, an element from the input queue is processed. When processing an input, the ideal functionality $\mathcal{F}_{Streams}$ checks which message only has compromised parties $x_i \notin \mathtt{I_h}$ on its path. For those cases, the ideal functionality leaks the message to the simulator $\mathcal{S}$. Otherwise, $\mathcal{F}_{Streams}$ provides a temporary identifier (in the form of a random integer) to $\mathcal{S}$ in place of a message, along with the segment of the path until the next honest compute node. When the round corresponding to that compute node comes, $\mathcal{F}_{Streams}$ again provides a new temporary identifier along with the next segment of path. When $\mathcal{F}_{Streams}$ switches the temporary identifiers, if there are not honest funnel before or after the honest compute node, it provides the mapping between the old and the new identifiers to allow $\mathcal{S}$ to link between packets. Here we slightly over-approximate the leakage by not distinguishing between honest and compromised recipients, because in some protocol setting (anonymous broadcast) or anonymity notion (sender anonymity) the adversary can anyway see the message in plaintext once it comes out of the protocol. We formally present the ideal functionality in Fig. 15.

B.2 Pairwise Unlinkability of $\mathcal{F}_{Streams}$

Next we show that, at the expense of latency, our ideal functionality provides pairwise unlinkability — if two messages stay together in $\mathcal{F}_{Streams}$ for a sufficiently long time (polylogarithmic in the security parameter), they get shuffled.

Theorem 3 (Pairwise unlinkability of $\mathcal{F}_{Streams}$)**.** *If the amount of compromised nodes is a constant fraction $\frac{\mathsf{c}}{\mathsf{K}} < 1$, $\mathcal{F}_{Streams}$ provides pairwise unlinkability of messages over $\mathfrak{L}$ rounds up to probability δ as in Definition 1, where $\delta < \gamma^{\mathfrak{L}/2}$ with $\gamma = 1 - \left(\frac{K-c}{K}\right)^3$.*

We postpone the full proof in Appendix C.2. Note that, $\mathfrak{L}$ rounds in the UC-framework version of our protocol translates to $L = \mathfrak{L}/2$ rounds in the original protocol. If $\mathfrak{L}$ in the above theorem is polylogarithmic, δ becomes negligible, which gives us the following corollary.

Corollary 1. *Given a constant fraction $\frac{\mathsf{c}}{\mathsf{K}}$, in the presence of any adversary $\mathcal{S}$, if two arbitrary messages stay together in the protocol $\mathcal{F}_{Streams}$ for $\mathfrak{L} \in \omega(\log \eta)$ rounds they are shuffled with an overwhelming probability.*

B.3 Abstraction Proof for *Streams*

Recall that formally *Streams* runs in the $\mathcal{F}_{CRF}, \mathcal{F}_{RKR}, \mathcal{F}_{SCS}, \mathcal{F}_{round}$ hybrid model. Our ideal functionality $\mathcal{F}_{Streams}$ absorbs the hybrid functionalities $\mathcal{F}_{CRF}$, $\mathcal{F}_{RKR}$ and $\mathcal{F}_{SCS}$ completely. However, we keep the $\mathcal{F}_{round}$ functionality untouched.

Note that, the realization of the ideal functionalities $\mathcal{F}_{CRF}, \mathcal{F}_{RKR}, \mathcal{F}_{SCS}$ in the UC-world translates to secure TLS/SSL, secure public-key encryption scheme, and incorruptible randomness beacon in the real world.

Theorem 4. *For any subprotocol Π_{sub} in the $\mathcal{F}_{RKR}$-hybrid model that UC realizes $\mathcal{F}_{sub}$, the anonymity protocol Streams from Appendix A using the subprotocol Π_{sub} in the $\mathcal{F}_{CRF}$, $\mathcal{F}_{RKR}$, $\mathcal{F}_{SCS}$, $\mathcal{F}_{round}$-hybrid model UC-realizes $\mathcal{F}_{Streams}$ in the $\mathcal{F}_{round}$-hybrid model.*

Kuhn et al. [49] show that under standard cryptographic assumptions there is a protocol Π_{sub} in the $\mathcal{F}_{RKR}$-hybrid model that UC realizes $\mathcal{F}_{sub}$. The key idea is to utilize (in a black-box reduction) the UC-realization proof of Π_{sub} such that in the proof the subprotocol's ideal functionality $\mathcal{F}_{sub}$ can be considered. This ideal functionality $\mathcal{F}_{sub}$ is used to abstract away from any cryptographic operations. The second key insight is that the attacker (and the simulator) can perfectly predict how many onions are in the protocol and when each party sends a message. So, only if the recipient is compromised or a message is sent to a client (or the input buffer is full) information is leaked from the protocol. In those cases, the ideal functionality $\mathcal{F}_{Streams}$ indeed leaks information such that the simulator can faithfully (and indistinguishably) simulate the network traffic. The full proof is postponed to Appendix C.

B.4 Pairwise Unlinkability of *Streams*

Since $\mathcal{F}_{Streams}$ provides pairwise unlinkability of messages for $\delta < \gamma^{\mathfrak{L}/2}$ over $\mathfrak{L}$ rounds for a constant fraction γ, as a corollary

to Theorem 4 and Theorem 3 we can state the following security theorem for our protocol *Streams*.

Theorem 5 (Security of *Streams*)**.** *For any subprotocol Π_{sub} in the $\mathcal{F}_{RKR}$-hybrid model that UC realizes $\mathcal{F}_{sub}$, given a constant fraction $\frac{c}{K} < 1$, Streams (using Π_{sub}) provides pairwise unlinkability of messages over $\mathfrak{L}$ rounds up to probability δ as in Definition 1, where $\delta < \gamma^{\mathfrak{L}/2}$ where $\gamma = 1 - \left(\frac{K-c}{K}\right)^3$.*

As a consequence, for all $\mathfrak{L} \in \omega(\log \eta)$ for a security parameter η the δ is negligible, and all pair of messages that stays together in the protocol for at least $\mathfrak{L}$ rounds, get shuffled with overwhelming probability. Recall that $\mathfrak{L}$ rounds of the protocol in the UC-framework translates to $L = \mathfrak{L}/2$ rounds in the original protocol.

B.5 Pairwise Unlinkability for the Core Protocol

We also want to analyze the scenario where we do not need to scale horizontally, core protocol described in Appendix A.2 is sufficient (e.g., the nodes are as powerful as network routers or the total number of users is less than few thousands).

Theorem 6 (Pairwise unlinkability of the Core protocol)**.** *For any subprotocol Π_{sub} in the $\mathcal{F}_{RKR}$-hybrid model that UC realizes $\mathcal{F}_{sub}$, given a constant fraction $\frac{c}{K} < 1$, the core protocol (using Π_{sub}) described in Appendix A.2 provides pairwise unlinkability of messages over $\mathfrak{L}$ rounds up to probability δ as in Definition 1, where $\delta < \left(\frac{c}{K}\right)^{\mathfrak{L}}$.*

The above theorem also gives us an important insight about how much security is degraded to achieve scalability through our funnel and compute nodes. Note that the number of rounds $\mathfrak{L}$ in the UC-framework version of the core protocol translates to $\mathfrak{L}$ rounds in original code protocol described in Section 4.2 as well, since there is no separate funnel and compute phase. Similar to other proofs in this paper, we postpone this proof to Appendix C.

C Postponed proofs

C.1 *Streams* UC-realizes Ideal Functionality $\mathcal{F}_{Streams}$

We stress that it makes our result stronger that we cast our ideal functionality in the $\mathcal{F}_{round}$-hybrid model. It is straightforward to let $\mathcal{F}_{Streams}$ additionally absorb $\mathcal{F}_{round}$.

Theorem 4. *For any subprotocol Π_{sub} in the $\mathcal{F}_{RKR}$-hybrid model that UC realizes $\mathcal{F}_{sub}$, the anonymity protocol Streams from Appendix A using the subprotocol Π_{sub} in the $\mathcal{F}_{CRF}$, $\mathcal{F}_{RKR}$, $\mathcal{F}_{SCS}$, $\mathcal{F}_{round}$-hybrid model UC-realizes $\mathcal{F}_{Streams}$ in the $\mathcal{F}_{round}$-hybrid model.*

Proof. We show the theorem via a series of game hops, starting with the protocol *Pi* and an arbitrary network adversary $\mathcal{A}$. With delta changes in each game, in the final game we end up with the ideal functionality $\mathcal{F}_{Streams}$ and a simulator $\mathcal{S}$. As $\mathcal{F}_{CRF}$ does not send messages to the environment and is not accessible to the environment, it can be easily absorbed by the ideal functionalities. For brevity, we hence neglect it in the subsequent argumentation.

Game 1. *In this game, we consider the original protocol Streams execution with the network attacker $\mathcal{A}$ and the environment $\mathcal{E}$. The protocol follows the code in Figure 11 and Fig. 13.*

Now, we design a game and a protocol where the subprotocols associated with onion processing are replaced with ideal functionality from [49].

Game 2. *Instead of calling the protocol subroutines from Π_{sub}, our protocol Streams now calls the ideal functionality $\mathcal{F}_{sub}$ from Kuhn et al. [49] (for completeness also in the appendix Figure 17). Moreover, the attacker is replaced by a variant of the simulator S_{sub} from Kuhn et al.*

- ***The simulator*** *S^*_{sub} behaves like the simulator S_{sub} in the paper of Kuhn et al. [49] except that acts on one kind of message differently to S_{sub}: if an $\mathcal{F}_{SCS}$ instance sends a message $p := (\texttt{"sent"}, P_i, Map, \mathsf{size})$, where "sent" is a string, P_i is the sender of a packet, Map is the next funnel in the protocol, size is the size of a packet. In that case, S^*_{sub} sends the message p directly to A_d, which is running inside S_{sub}. As $\mathcal{A}_d$ is stateless, these extra messages do not change $\mathcal{A}_d$'s behaviour.*

The protocol Streams still follows the code in Figure 11 and Fig. 13, except that it called $\mathcal{F}_{sub}$ instead of Π_{sub}.

Claim 1. *There is a simulator S_{sub} such that Game 1 is indistinguishable from Game 2.*

Proof of Claim . Analogously to UC's completeness theorem, it suffices to consider the dummy attacker $\mathcal{A}_d$ that forwards all messages to the environment and only acts on the environment's orders.[8] We replace $\mathcal{A}_d$ with a simulator S^*_{sub} that almost behaves like the simulator S_{sub} in the paper of Kuhn et al. [49], which internally runs $\mathcal{A}_d$. S^*_{sub}, however, has to also present an indistinguishable view for $\mathcal{E}$; hence, it has to forward all $\mathcal{F}_{SCS}$ notifications to the environment, just as $\mathcal{A}_d$ would do.

Next, we show that Game 1 (running Π_{sub}, $\mathcal{F}_{RKR}$, and $\mathcal{A}_d$) and Game 2 (running $\mathcal{F}_{sub}$ and S^*_{sub}) are indistinguishable. We show that, if Game 1 is distinguishable from Game 2, then Π_{sub} does not UC realize $\mathcal{F}_{sub}$, which contradicts [49]. $\mathcal{F}_{RKR}$ is faithfully simulated within S_{sub}; hence, it behaves exactly the same in these two interactions.

[8] For any other attacker $\mathcal{A}$ and each environment $\mathcal{E}$, there is an environment $\mathcal{E}'$ that internally emulates the interaction between $\mathcal{E}$ and $\mathcal{A}$. $\mathcal{E}'$ interacts with the dummy attacker $\mathcal{A}_d$ and produces the same view.

Towards contradiction, assume that Game 1 is distinguishable from Game 2. Given an environment $\mathcal{E}$ that can distinguish Game 1 from Game 2, we construct an environment $\mathcal{E}_{sub}$ that can distinguish Π_{sub} and $\mathcal{F}_{RKR}$ interacting with the dummy attacker $\mathcal{A}_d$ from $\mathcal{F}_{sub}$ interacting with S_{sub}. $\mathcal{E}_{sub}$ internally runs $\mathcal{E}$. $\mathcal{E}_{sub}$ has to ensure that $\mathcal{E}$ believes that it is in Game 1 or Game 2, respectively. Hence, $\mathcal{E}_{sub}$ has to ensure that $\mathcal{E}$ gets the same messages as in Game 1 and Game 2, respectively. So, we have to make sure that $\mathcal{E}$ sees the same the funnel-protocol communication and the notification messages from $\mathcal{F}_{SCS}$ for each packet that are handed through from $\mathcal{A}_d$ in Game 1. The funnel-protocol communication can be achieved by $\mathcal{E}_{sub}$ running the funnel-protocol instances. In Game 1 and Game 2, the $\mathcal{F}_{SCS}$ notification messages are sent whenever a funnel or a compute node instance communicates with $\mathcal{F}_{round}$. Hence, $\mathcal{E}_{sub}$ has to ensure that $\mathcal{E}$ gets these notification messages at the correct time, which can do as it internally runs $\mathcal{F}_{SCS}$.

- **The environment** $\mathcal{E}_{sub}$ internally runs $\mathcal{F}_{SCS}$, $\mathcal{F}_{round}$, and $\mathcal{E}$. Let $Int_{1,sub}$ be the interaction between Π_{sub} and $\mathcal{F}_{RKR}$ from [49] and the dummy attacker $\mathcal{A}_d$ with an environment (in our case $\mathcal{E}_{sub}$), and let $Int_{2,sub}$ be the interaction between $\mathcal{F}_{sub}$ and the simulator S_{sub} from [49]. As $\mathcal{E}_{sub}$ does not know whether it is interacting with $Int_{1,sub}$ or $Int_{2,sub}$, we describe its behavior agnostic to $b = 1$ or $b = 2$ with $Int_{b,sub}$.
  - Upon receiving a message a party from $Int_{b,sub}$ (i.e., from a Π_{sub} instance or $\mathcal{F}_{sub}$), run Π_{client} and forward the response to $\mathcal{E}$.
  - Upon receiving a message over the network from $Int_{b,sub}$ (i.e., from $\mathcal{A}_d$ or S_{sub}), forward the message to $\mathcal{F}_{SCS}$ and faithfully (as in Game 1) compute the interaction between $\mathcal{F}_{round}$, the funnel instances, and $\mathcal{E}$.
  - Upon receiving a message from $\mathcal{E}$ for the network attacker, directly forward this message to the network attacker in $Int_{b,sub}$ (i.e., to $\mathcal{A}_d$ or $\mathtt{S}_{sub}$).
    - Upon receiving a notification message from the internally emulated $\mathcal{F}_{SCS}$, forward it to $\mathcal{E}$. (We stress that S^*_{sub} is split into this interaction and the part that is run in Int_{sub}.)

For each $b \in \{1,2\}$, we have to show that for $\mathcal{E}$ the interaction within $\mathcal{E}_{sub}$, which in turn interacts with $Int_{b,sub}$, is indistinguishable from the interaction with Game b. For $b = 1$, the interaction within $\mathcal{E}_{sub}$ solely differs in the order in which the notification message from $\mathcal{F}_{SCS}$ arrives. As these messages first reach $\mathcal{E}_{sub}$ before reaching $\mathcal{E}$, $\mathcal{E}_{sub}$ can successfully reverse the order again (see above) and constructs a perfect view for $\mathcal{E}$.

For $b = 2$, $\mathcal{E}_{sub}$ internally emulates Game 1 (except for Π_{sub}). We show that for $b = 2$ nevertheless the view of $\mathcal{E}$ when being emulated within $\mathcal{E}_{sub}$ is indistinguishable from the view when interacting in Game 2. Recall that the only difference between Game 1 and Game 2 is that Π_{sub} is replaced by $\mathcal{F}_{sub}$, and $\mathcal{A}_d$ is replaced by $\mathtt{S}_{sub}$. As $\mathcal{F}_{sub}$ is changed by $\mathcal{E}_{sub}$, it suffices to analyze whether the message transcript to $\mathtt{S}_{sub}$ is indistinguishable for $\mathtt{S}_{sub}$ and whether the transcript from $\mathtt{S}_{sub}$ (through $\mathcal{E}_{sub}$) is indistinguishable for $\mathcal{E}$.

Whenever by $Int_{2,sub}$ a message is sent by $\mathtt{S}_{sub}$ to $\mathcal{E}_{sub}$, this messages first goes through the internally emulated instances of the $\mathcal{F}_{SCS}, \mathcal{F}_{round}$, and Π_{funnel} protocols. These protocols solely forward messages, and of these only $\mathcal{F}_{SCS}$ sends a notification to the network attacker $\mathcal{A}_d$. In this case, as defined above, $\mathcal{E}_{sub}$ directly forwards the notification to $\mathcal{E}$; this is exactly the same that would happen in Game 2. All other messages are forwarded and, as in Game 2, potentially sent to $\mathcal{E}$. Hence, whenever in Game 2 a message is sent to $\mathcal{E}$ also in $\mathcal{E}_{sub}$'s internal emulation (if $b = 2$) a message is sent to $\mathcal{E}$.

Next, we consider the case where for $b = 2$ a message is sent by $\mathcal{E}$ (while it is being internally emulated by $\mathcal{E}_{sub}$) to the network attacker, which would in Game 2 be $\mathtt{S}^*_{sub}$. As defined above, in this case, $\mathcal{E}_{sub}$ sends the message directly to the network attacker in $Int_{b,sub}$. As $b = 2$, the network attacker is $\mathtt{S}_{sub}$. Hence, the message transcript (from $\mathtt{S}_{sub}$'s point of view) is exactly the same as in Game 2.

If Int_{sub} is the interaction with Π_{sub}, $\mathcal{F}_{RKR}$, and $\mathcal{A}_d$, $\mathcal{E}_{sub}$ ensures that $\mathcal{E}$ has exactly the same view as in Game 1. If Int_{sub} is the interaction with $\mathcal{F}_{sub}$ and S_{sub}, $\mathcal{E}_{sub}$ ensures that $\mathcal{E}$ has exactly the same view as in Game 2. Hence, by assumption, with the translation of $\mathcal{E}_{sub}$ the submachine $\mathcal{E}$ can distinguish the interaction with Π_{sub}, $\mathcal{F}_{RKR}$, and $\mathcal{A}_d$ from the interaction with $\mathcal{F}_{sub}$ and S_{sub}.

For any poly-bounded $\mathcal{E}$, $\mathcal{E}_{sub}$ acts as a poly-bounded environment in the UC game. Yet, Kuhn et al. [49] proved that there is no poly-bounded environment that can distinguish these two interactions, which is a contradiction. Hence, Game 1 and Game 2 are indistinguishable.

◇

Game 3. *We replace Π_{worker}, Π_{client}, $\mathcal{F}_{SCS}$, $\mathcal{F}_{sub}$ with the ideal functionality $\mathcal{F}_{Streams}$. The simulator S^*_{sub} is replaced by a simulator S_f. The simulator S_f internally runs S^*_{sub} but translates the format of the output of $\mathcal{F}_{Streams}$ to the format output by $\mathcal{F}_{sub}$. We stress that as we are in the hybrid $\mathcal{F}_{round}$-model, $\mathcal{F}_{round}$ remains in the ideal world as it was in the previous games.*

Claim 2. *With the simulator S_f, Game 3 is indistinguishable from Game 2.*

Proof of Claim . For the analysis, we divide the execution in overlapping sub-sequences of the form compute node, funnel, compute node (overlapping at the last funnel). For those subsequences where the funnel is malicious or the first compute node is malicious, $\mathcal{F}_{Streams}$ has exactly the same leakage as $\mathcal{F}_{sub}$, except that the format of the leakage is translated. If one of the funnels is honest and the first compute nodes is honest, though, $\mathcal{F}_{Streams}$, in contrast to $\mathcal{F}_{sub}$, does not leak which compute node sends (the ideal abstraction of) an onion

to which other compute node. Next, we argue that this leakage is also hidden in Game 2, as $\mathcal{F}_{SCS}$ and the funnels hide this information.

As the first compute node is honest, it does not leak to the network attacker to whom the onion is sent. If the first funnel is honest, it does not leak the link between the two compute nodes to the network attacker. As $\mathcal{F}_{SCS}$ only notifies the network attacker that some messages was sent and as the funnels shuffle the messages of each round, the network attacker does not learn by whom an onion was sent. ◇

Therefore, for simulator $\mathcal{S}$ our protocol *Streams* UC-realizes the ideal functionality $\mathcal{F}_{Streams}$. □

C.2 Security Analysis Proofs

Theorem 3. If the amount of compromised nodes is a constant fraction $\frac{c}{K} < 1$, $\mathcal{F}_{Streams}$ provides pairwise unlinkability of messages over $\mathfrak{L}$ rounds up to probability δ as in Definition 1, where $\delta < \gamma^{\mathfrak{L}/2}$ for some constant constant fraction $0 < \gamma < 1$.

Proof. Recall that, we assume that each node in a round is chosen uniformly at random (funnel nodes by randomness beacon and compute nodes by the clients) with replacement, and independent of all other rounds. Conceptually, a careful strategy where nodes are chosen by avoiding repetition as much as possible can provide better security guarantees. For the ease of analysis, we make such weaker assumption.

If two messages remain in $\mathcal{F}_{Streams}$ for $\mathfrak{L}$ rounds, they are shuffled if both of those two messages have honest compute nodes on their path in some round r, and then an honest node is picked as the funnel node in round $r+1$. If that happens, a shuffled list of newly generated temporary identifiers are given to $\mathcal{S}$ on behalf of those messages. In Figure 5, we pictorially show the possible cases when two messages can mix (or not).

Let a be the probability of a randomly picked node being honest; $a = \frac{K-c}{K}$ Since the funnel node is picked uniformly at random (with replacement, and independent of all other nodes), the probability of the funnel node being compromised is $\frac{c}{K} = (1-a)$, and being honest is a. Similarly, each compute node on the path of a message is selected uniformly at random (with replacement, and independent of all other nodes). Therefore, the probability of an compute node being honest is also a.

Therefore, the probability that the two messages mix in for a given pair of compute node in round r and funnel node in round $(r+1)$ is a^3. And, the probability that they don't mix in those pair of rounds is $\gamma = (1-a^3)$. If two messages stay in the system together for $\mathfrak{L}$ rounds, they do not mix with probability at most $\delta < \left(1-a^3\right)^{\frac{\mathfrak{L}}{2}}$.

γ is conceptually the proportion of sequences of nodes where the two messages cannot mix. In Figure 6a we plot the relationship between γ and $\frac{c}{K}$. For a constant $\frac{c}{K}$, γ is constant; and hence $\delta < \gamma^{\mathfrak{L}/2}$ where $\gamma = \left(1-a^3\right)$ □

Theorem 6. Given a constant fraction $\frac{c}{K}$, against any adversary $\mathcal{S}$, if two messages stay together for $\mathfrak{L} \in \omega(\log \eta)$ rounds in the core protocol described in Appendix A.2 they are shuffled with a probability $(1-\delta)$, where $\delta < \left(\frac{c}{K}\right)^{\mathfrak{L}}$.

Proof Sketch. We skip the detailed proof as the proof methodology is very similar to the proof with compute and funnel nodes. The UC proof becomes much easier if we do not have to distinguish between compute and funnel nodes. And for the combinatorial argument there is one key difference: instead of the funnel node in the funnel round and the two compute nodes in the immediate next compute round, there is only one node and just one round.

Therefore, instead of representing each pair of rounds with three coin tosses in the compute and funnel node scenario, we have exactly one coin toss per round for the core protocol with success probability $a = \frac{c}{K}$. And hence, we can define an ideal functionality $\mathcal{F}_{core}$ as described in Fig. 16, which shuffles the messages in the system whenever they encounter an honest node on the path. To provide a simulator for $\mathcal{F}_{core}$ we use the exact same simulator $\mathcal{S}_{sub}$ as the one we use in the proof of Theorem 4, with only one minor modification that $\mathcal{S}_{sub}$ directly forwards all the network messages to the round functionality.

From the ideal functionality it is simple to show that the probability of **not** finding an honest node in a path of length $\mathfrak{L}$ is upper bounded by $\delta \leq \left(\frac{c}{K}\right)^{\mathfrak{L}}$. Therefore, if two arbitrary messages stay in the protocol for at least $\mathfrak{L}$ rounds, they are shuffled with probability at least $1-\delta$. □

D Existing functionalities

Ideal Functionality for Onion Routing: We borrow the ideal functionality $\mathcal{F}_{sub}$ for onion routing from the work of Kuhn et al. [49, Algorithm 1]. We present the ideal functionality in Figure 17 for completeness. Kuhn et al. [49, Appendix E] also presents a modified version of Sphinx [47] that realizes the ideal functionality $\mathcal{F}_{sub}$. We use the same modified version of Sphinx as our Π_{sub} in the current work.

We aim for anonymous broadcast to the network. In our ideal functionality, messages from our last node are sent to the environment instead of delivering them to an explicit receiver. The delivery of messages occurs through the environment which controls the network functionality.

Secure Communications Sessions: We use the ideal functionality $\mathcal{F}_{scs}$ from the work of Gajek et al. [37, Figure 4] to realize secure communications sessions. Their work shows that the TLS protocol [37, Figure 5] UC-realizes the ideal functionality $\mathcal{F}_{scs}$.

inputBuffer[] an array of queues to store messages for nodes
crf = an infinitely long random string; *queue* = a hashmap
round $:= 0$, *newRound*[] $:= \{$false, false, $\ldots\}$;
partyCount $:= 0$

Upon new round from $\mathcal{E}$ for party P:
- **if** *newRound*$(P) =$ **true then**
  - return "invalid action"
- **end if**
- set *newRound*$(P) :=$ **true** ; *partyCount*$+= 1$
- **if** P is a client **then**
  - $(m,R,t) \leftarrow$ dequeue *inputBuffer*$[P]$
  - $d \leftarrow$ *DelayDistribution*$(\frac{\ell}{2}, \ell - 1)$;
  - $\{x_1,\ldots,x_d\} \xleftarrow{\$}$ $\{\mathsf{crf}[\mathtt{round}]\%\mathsf{K},\ldots,\mathsf{crf}[\mathtt{round}+d]\%\mathsf{K}\}^d$
  - **if** $!\exists x \in \{x_1,\ldots,x_d\}$ such that $x_a \in \mathtt{I_h}$ **then**
    - Send $(m,x_1,\ldots,x_d)$ to $\mathcal{S}$
  - **else**
    - let $x_a :=$ the **first** honest party on the path $\{P,x_1,\ldots,x_d\}$
    - Send $(q,x_1,\ldots,x_a)$ to $\mathcal{S}$ where $q \xleftarrow{\$} \mathcal{M}$
    - store $(q,x_a,m,x_{a+1},\ldots,x_d,R)$ in *queue*$(\mathtt{round}+a)$
  - **end if**
- **end if**
- **if** *partyCount* $= \mathsf{N}+\mathsf{K}$ **then**
  - SendInformation()
- **end if**
- NextRound(P)

Upon input message (m, R, t) from $\mathcal{E}$ for party P:
- *inputBuffer*$(P)+= 1$
- **if** round $\neq t$ **then**
  - reject packet and exit
- **end if**
- Add (m,R,t) in *inputBuffer*$[P]$

Upon receiving a message m for party P:
- Output "Message m received" to $\mathcal{E}$

SendInformation()
- **for** each $(q,x_a,m,x_{a+1},\ldots,x_d,R) \in$ *queue*(round) **do**
  - Remove $(q,x_a,m,x_{a+1},\ldots,x_d,R)$ from *queue*(round)
  - let x_ϕ be the **next** honest node on the path $\{x_{a+1},\ldots,x_d\}$
  - **if** there is **no** such x_ϕ **then**
    - Add $(m,x_{a+1},\ldots,x_d,R)$ in a temporary queue Q
  - **else**
    - Add $(q',x_{a+1},\ldots,x_\phi)$ in Q for $q' \xleftarrow{\$} \mathcal{M}$
    - Add $(q',x_\phi,m,x_{\phi+1},\ldots,x_d,R)$ to *queue*$(\mathtt{round}+\phi-a)$
  - **end if**
- **end for**
- Shuffle the elements of Q and send them to $\mathcal{S}$

Figure 16: Ideal functionality $\mathcal{F}_{core}$ for the Core Protocol

Data structure:
Bad: Set of Corrupted Nodes
L: List of Onions processed by adversarial nodes
B_i: List of Onions held by node P_i
// Notation:
// $\mathcal{S}$: Adversary (resp. Simulator)
// $\mathcal{Z}$: Environment
// $\mathcal{P} = (P_{o_1},\ldots,P_{o_n})$: Onion path
// $O = (sid,P_s,P_r,m,n,\mathcal{P},i)$: Onion = (session ID, sender, receiver, message, path length, path, traveled distance)
// N: Maximal onion path length

On message Process_New_Onion$(P_r,m,n,\mathcal{P})$ *from* P_s
- // P_s creates and sends a new onion (either instructed by $\mathcal{Z}$ if honest or $\mathcal{S}$ if corrupted)
- **if** $|\mathcal{P}| > N$; // selected path too long **then**
  - Reject
- **else**
  - $sid \leftarrow^R$ session ID ; // pick random session ID
  - $O \leftarrow (sid,P_s,P_r,m,n,\mathcal{P},0)$; // create new onion
  - Output_Corrupt_Sender$(P_s,sid,P_r,m,n,\mathcal{P},$start$)$
  - Process_Next_Step(O)

Procedure Output_Corrupt_Sender$(P_s,sid,P_r,m,n,\mathcal{P},temp)$
- // Give all information about onion to adversary if sender is corrupt
- **if** $P_s \in$ *Bad* **then**
  - Send "*temp* belongs to onion from P_s with $sid,P_r,m,n,\mathcal{P}$" to $\mathcal{S}$

Procedure Process_Next_Step$(O = (sid,P_s,P_r,m,n,\mathcal{P},i))$
- // Router P_{o_i} just processed O that is now passed to router $P_{o_{i+1}}$
- **if** $P_{o_j} \in$ *Bad for all* $j > i$ **then**
  - Send "Onion from P_{o_i} with message m for P_r routed through $(P_{o_{i+1}},\ldots,P_{o_n})$" to $\mathcal{S}$
  - Output_Corrupt_Sender$(P_s,sid,P_r,m,n,\mathcal{P},end)$
- **else**
  - // there exists an honest successor P_{o_j}
  - $P_{o_j} \leftarrow P_{o_k}$ with smallest k such that $P_{o_k} \notin$ Bad
  - $temp \leftarrow^R$ temporary ID Send "Onion *temp* from P_{o_i} routed through $(P_{o_{i+1}},\ldots,P_{o_{j-1}})$ to P_{o_j}" to $\mathcal{S}$
  - Output_Corrupt_Sender$(P_s,sid,P_r,m,n,\mathcal{P},temp)$ Add $(temp,O,j)$ to L

On message Deliver_Message$(temp)$ *from* $\mathcal{S}$
- // Adversary $\mathcal{S}$ (controlling all links) delivers onion belonging to *temp* to next node
- **if** $(temp,\_,\_) \in L$ **then**
  - Retrieve $(temp,O = (sid,P_s,P_r,m,n,\mathcal{P},i),j)$ from L
  - $O \leftarrow (sid,P_s,P_r,m,n,\mathcal{P},j)$**if** $j < n+1$ **then**
    - $temp' \leftarrow^R$ temporary ID Send "$temp'$ received" to P_{o_j} Store $(temp',O)$ in B_{o_j}
  - **else**
    - **if** $m \neq \perp$ **then**
      - Send "Message m received" to P_r

On message Forward_Onion$(temp')$ *from* P_i
- // P_i is done processing onion with $temp'$ (either decided by $\mathcal{Z}$ if honest or $\mathcal{S}$ if corrupted)
- **if** $(temp',\_) \in B_i$ **then**
  - Retrieve $(temp',O)$ from B_i Remove $(temp',O)$ from B_i
  - Process_Next_Step(O)

Figure 17: Ideal functionality $\mathcal{F}_{sub}$ for onion routing [49].