Breaking and (Partially) Fixing Onion Routing with Fragmentation

Daniel Schadt
daniel.schadt@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Christoph Coijanovic
christoph.coijanovic@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Thorsten Strufe
thorsten.strufe@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract

Mix networks are a cornerstone of anonymous communication, protecting users' relationships by relaying their messages through a series of mix nodes. To accommodate large payloads, deployed systems rely on message fragmentation, but this seemingly benign feature unwittingly opens a subtle door for adversaries.

In this paper, we show that fragmentation enables adversaries to tag messages by suppressing single fragments and thereby break sender-recipient unlinkability, striking at the core privacy guarantee of mix networks. We demonstrate the practicality of this attack on Nym, a real-world deployed mix network. To address this threat, we design a lightweight mitigation that incurs only little overhead in both packet size and processing time. We provide a formal model of fragmentation in mix networks and prove that our mitigation restores unlinkability in this model.

Keywords

message fragmentation, mix network, onion routing

1 Introduction

Communication metadata, such as *who talks to whom* or *when do people talk*, is very sensitive: The fact that Alice spoke to an investigative journalist just before a major story was published makes her suspicious of whistleblowing. Anonymous Communication Systems (ACS) aim to enable communication between different users, Alice and Bob, without disclosing such metadata to observers or the messaging servers.

Tor [14] is the ACS with by far the largest userbase.[1] It is based on relaying messages through several intermediaries and layered encryption between the sender and each of them. Since Tor is susceptible to traffic analysis and timing analysis, and does not protect against global adversaries [17], there has recently been a push to deploy ACS designs that offer stronger privacy protection. The most prominent example of this is Nym,[2] which positions itself as an alternative to Tor. Nym is based on the Loopix [21] mix network design. In a mix network, each intermediary additionally shuffles or delays messages before forwarding to obfuscate timing patterns [9].

[1] https://metrics.torproject.org/
[2] https://nym.com

Proceedings on Privacy Enhancing Technologies 2026(4), 272–289

https://doi.org/10.56553/popets-2026-0120

To prevent packets from being traced through the network, mix networks require that all packets be the same size. However, since users may want to send messages of different sizes, deployed systems use fragmentation, whereby one large message is split into multiple packets. While this seems like a straightforward solution, fragmentation enables a new attack breaking the mix network's privacy guarantees: Consider Alice, who sends a fragmented message to a malicious receiver, Mallory. If Mallory causes one of the fragments to be dropped while it can still be linked to Alice, then Mallory will receive the remaining fragments but notice that there is a fragment missing. This allows her to confirm Alice as the sender.

While the possibility of such an attack has been identified in literature [2, 19], it has not been formally described or investigated. Thus, our first major contribution is an analysis of how the dropping of message fragments breaks anonymity in mix networks. We call this attack a *Fragging* attack, and derive conditions that have to be met by the adversary to be able to execute a Fragging attack. We also explain how previous security notions of mix networks did not consider this attack, leading to mix networks being proven secure despite their susceptibility to Fragging attacks: Previous notions only consider self-contained messages without any concept of fragmentation. We hence extend these privacy properties to *fragmented layer unlinkability* (FLU) and *fragment Hiding* (FH), which formalize mix network privacy goals in the presence of message fragmentation.

As our second major contribution, we show that the susceptibility of mix networks to fragmentation-based attacks is not inherent. We propose *Scylla*, a new mix format. Scylla is an adaptation of the Sphinx mix format [12], but has support to (securely) fragment messages built in. It uses an all-or-nothing-transform (AONT) [5] to ensure that all fragments have arrived before revealing sensitive information, thereby turning a fragment-drop from a privacy-leakage into a denial-of-service. By moving the fragmentation from the "application layer" (as is done in Nym) to the "mix layer," we can reason better about its privacy guarantees.

We formally analyze the privacy guarantees of Scylla using our extended privacy properties. We prove that Scylla offers strictly stronger privacy protection than "trivial" fragmentation, as it hides the sender-receiver relationship, even under a Fragging attack. Note that Scylla cannot protect against delay-based tagging of fragments, which can be used to execute attacks similar to Fragging. To also protect against delaying adversaries, Scylla has to be combined with additional techniques [3].

Further, we implement a prototype of Scylla and empirically evaluate its performance in terms of computation time and size overhead. We find that Scylla shows the same performance characteristics as Sphinx, thereby making it a very practical choice.

This paper is structured as follows: In Section 2 we introduce our notation and background knowledge. In Section 3 we describe our system and threat model. In Section 4 we introduce the Fragging attack and analyze how it applies to Nym and Tor. In Section 5 we introduce Scylla as our mitigation against Fragging. In Section 6 we formalize and prove the security of Scylla. In Section 7 we evaluate the performance of Scylla using our prototype. In Section 8 we talk about limitations of our approach. In Section 9 we discuss further design decisions. In Section 10 we summarize related work. Finally, in Section 11 we conclude our findings.

2 Background

In this section, we introduce the general notation, as well as mix networks and the Sphinx mix format.

2.1 Notation

Throughout this paper, we use κ to refer to the security parameter. Breaking our security primitives should require work that scales exponentially in κ.

When working with bitstrings $a, b \in \{0,1\}^*$, we write $a||b$ to mean the concatenation of a and b, $a_{[x..y]}$ to mean the substring of a starting from the x^{th} character and ending before the y^{th} character, and $a \oplus b$ to mean the bitwise xor of a and b (truncated to the shorter of a and b). We write a^x to mean the character a repeated x times.

We write $x \leftarrow^{\text{R}} X$ to mean a uniformly random choice of x from set X.

2.2 Mix networks

Mix networks are a method to implement anonymous communication systems, introduced by Chaum in 1981 [9]. In a mix network, a message is routed across multiple *mix nodes* before it reaches its intended recipient. A mix node obfuscates the link between incoming and outgoing packets in two ways:

First, messages are encrypted multiple times, once for each node along the path. Each node removes the corresponding layer of encryption, making re-identification based on message content impossible. This technique is called *onion encryption*, and messages following this encryption pattern are commonly called *onions*.

Further, each mix node shuffles the messages, thereby making it impossible to trace messages based on the order in which they leave the mix node.

Mix networks operate under the *anytrust* assumption, meaning that one node on the path must be honest in order for the anonymity guarantees to hold. Anonymity is usually guaranteed even against a strong, global active adversary and against malicious recipients.

Kuhn et al. identify two models in which a mix network can be used [18]: In an *integrated system model*, the receiver acts as a mix node themselves, and while routing messages discovers that a message was intended for them. This is in contrast to the *service model*, in which the actual receiver is not part of the ACS, and instead the final mix node delivers the message using a different transport.

2.3 Sphinx

Sphinx is a mix format introduced by Danezis and Goldberg [12], which promises compactness and provable security. It has been used as the basis for other formats (e.g., MultiSphinx [15], PolySphinx [25]) and deployed commercial systems like Nym.

In Sphinx, a packet consists of a header and the payload. The header contains the message's path. It is onion encrypted using a stream cipher and integrity-checked at every hop via a MAC. The payload, which is revealed to the final mix node, contains the message's recipient and the text. It is encrypted using a block cipher, but integrity is not checked at every hop. With this design, Sphinx assumes a *service model*.

The split between header and payload allows Sphinx to generate single-use reply blocks (SURBs). A SURB is a pre-computed header that Alice can provide to Bob. Bob can then use this SURB to reply to Alice without learning Alice's identity himself.

The security of Sphinx is proven using Camenisch and Lysyanskaya's *onion properties* [6]. However, those properties have later been found to be insufficient [18] and a fixed security proof has been presented [27].

2.4 Nym

Nym is a commercially available mix network that aims to provide a general-purpose low-level anonymous channel, which can be used transparently by applications. Nym is based on the Loopix design [21] and uses Sphinx as the message format. In Nym/Loopix, messages are shuffled at mix nodes by delaying them artificially: The sender embeds a delay drawn from an exponential distribution into the packet header for each mix node. Each mix node then waits for the specified amount of time before releasing the packet.

Loopix also adds *loop* messages, which are messages that a user sends through the network back to themselves. This allows active attacks to be detected, because the users notice that expected loop messages are late or missing.

Nym additionally adds a reliable transport mechanism on top of Loopix: Each packet has a SURB attached that must be used by the recipient to acknowledge the reception of the packet. If the sender notices that a packet has not been acknowledged in time, they will re-transmit it. The acknowledgment timeout is scaled by the expected message delay, as the sender knows the delay of their message.

2.5 Tagging attacks

Tagging attacks represent a common attack technique in mix networks [18]. The idea is that the adversary inserts a "tag" before the onion arrives at an honest mix node. Later, they can recognize this tag in the processed onion and therefore trace it.

Tagging attacks are usually prevented by employing hop-by-hop integrity checking such that any modification to an onion is recognized by an honest node, causing the onion to be discarded. Sphinx uses this technique for the header part of an onion, where the MAC ensures integrity.

However, Sphinx uses a second technique to secure the payload: The payload is processed using a wide block cipher, so any modification to the payload will decrypt to random looking bytes and irrevocably destroy the contained information [12]. The adversary can therefore recognize the tag at the final node, but as the recipient's address is embedded within the payload, they can no longer learn information about the recipient or the original message text.

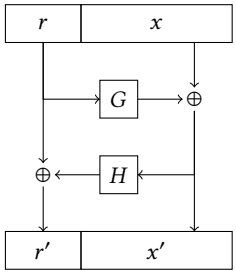

Figure 1: Diagram of the OAEP transform.

2.6 OAEP

The optimal asymmetric encryption padding (OAEP) was introduced by Bellare and Rogaway in the context of RSA encryption [4]. It represents an invertible transformation that is applied to a message before it is encrypted, ensuring non-determinism (and therefore semantic security) as well as non-malleability.

Definition 1 (OAEP [4])**.** Let $G : \{0,1\}^\kappa \to \{0,1\}^n$ and $H : \{0,1\}^n \to \{0,1\}^\kappa$ be two random functions, and $r \xleftarrow{R} \{0,1\}^\kappa$ a random nonce. We then define the OAEP transform as

$$\mathrm{OAEP}(x;r) = (r \oplus H(x \oplus G(r)))||(x \oplus G(r)).$$

If r is not given, we say that OAEP internally draws $r \xleftarrow{R} \{0,1\}^\kappa$.

This transform is also shown in Figure 1. Note that unlike the original definition, we place the random value in front of the transformed message for practical reasons: We will prepend other metadata to the message for mixing purposes, so we consistenly keep the actual payload at the end.

3 System and threat model

We assume a mix network as the underlying ACS. The network consists of *users*, who want to send messages, and *mix nodes*, which shuffle those messages and relay them to further mix nodes. We assume that the network runs in the service model, so that the final mix node forwards a message to its recipient using a non-anonymous transport, for example, email.

We use the term *entry node* or *gateway* to refer to the mix node at the beginning of the path, and the term *exit node* to refer to the final mix node on a path.

We distinguish between a *message*, which is the text that a user wants to send, the *payload*, which is the content of an onion revealed to the final hop, and an *onion*, which is an onion-encrypted payload together with a header containing the onion's path.

We assume that all onions in a mix network must have a fixed size. Therefore the length of a payload is limited. To send shorter messages, users pad their messages. To send longer messages, users split their message into multiple payloads and create multiple onions. Further, the limited size imposes a maximum length on the path that can be encoded, which we denote r.

We assume that the adversary is active and global. In particular, it can listen in on any network link, and modify and drop messages.

We further assume that the adversary can corrupt a subset of all mix nodes, meaning that they share their key material with the adversary, and they adapt their operation according to the adversary's command. The recipient is also assumed to be malicious, meaning that they also share their key material and received messages with the adversary.

We assume that the adversary is limited to polynomial time in κ and as such, it cannot break cryptographic primitives.

The goal of the adversary is to find out the sender-recipient pairs. More concretely, it wants to determine with whom Alice is communicating, given that Alice is honest.

4 The Fragging attack

The dropping bound [19] states that an ACS cannot protect privacy in a setting where a (malicious) receiver Bob *knows* that a message should arrive, and the adversary can drop packets of potential senders: If the adversary drops packets from Alice, and Bob still receives the message, the adversary can exclude Alice as a possible sender – and vice versa, if the message does not arrive, then Alice was likely the actual sender.

The dropping bound was formulated under the assumption of self-contained messages. As such, the attack relies on "external information" (like interpretation of previous message content or fixed schedules) to know whether a message is to be expected. However, we show that in realistic scenarios where fragmentation is applied in mix networks, this information can be gained from within the ACS. To do so, we look at Nym and Tor as two ACS in use, and we see how Fragging applies there. Then, we recap our theoretical model of Fragging.

4.1 Fragging in Nym

Loopix requires all packets to have a fixed size. Therefore, Nym transparently fragments messages to allow for larger messages to be sent.[3] To do so, Nym implements two techniques on top of Sphinx:

The first technique allows for a message to be split into 255 fragments in a `FragmentSet`. Each set has a random ID, and each fragment carries the ID of the set it belongs to, as well as its index within the set and the number of fragments in its set.

The second technique allows for messages that span more than 255 fragments to be grouped into multiple different `FragmentSets`, which are then linked. To do so, each fragment set contains the identifier of the previous and/or next fragment set.

To understand how the Fragging attack affects Nym, particularly in conjunction with their packet acknowledgment and retransmission mechanism, we have implemented a prototype in a small-scale Nym testbed. For this, we have modified the Nym applications[4] to carry out a Fragging attack. We then ran our modified binaries using the Shadow network simulator [16], giving us a self-contained Nym network. Our network consists of four users: Alice, Carol and Dave, who all send a message to Bob. Alice's and Dave's messages have 8 fragments, while Carol's has 9. We instruct Alice's gateway to drop a percentage of her packets, and see how Bob perceives the signal. We leave Carol's and Dave's packets unaffected to serve as a baseline.

[3] https://github.com/nymtech/nym/tree/a4e674c9/common/nymsphinx/chunking/src
[4] https://github.com/nymtech/nym

We expect to see an increase in the time it takes Bob to receive the full fragment set: If Alice's gateway drops a fragment, Alice's client waits for the retransmission timeout before sending it again.

For Carol and Dave as senders, we measure an average of 308 ms between Bob receiving the first fragment and Bob receiving the last fragment of the message. For Alice, dropping a fragment increases this latency to around 2000 ms, as Alice's client waits for the retransmission timeout. When we increase the chance of dropping a packet, this latency increases further, as the same fragment might be dropped multiple times before it eventually arrives. We show these results in Figure 2, in the top-left graph.

We repeat this experiment with increased average per-hop delays, to see if the resulting increase in message latency obscures the signal for Bob. We expect that the delay caused by missing fragments to be larger than the normal message delay, even for bigger average per-hop delays.

For those repetitions, we use the same Nym testbed, but compile Nym with `DEFAULT_AVERAGE_PACKET_DELAY` set to values 200 ms, 350 ms, and 500 ms. We report the results in Figure 2 in the top-right, bottom-left and bottom-right graphs. We can see that the set completion time increases for clients who are not under attack (Carol and Dave), but Alice's set completion times remain distinguishable.

From this proof-of-concept, we expect that the Fragging attack can be performed in Nym. While our testbed does not model network latencies or congestion, we expect those additional delays to not hide the signal: Network delays are generally in the order of 100 ms to 200 ms,[5] whereas the retransmission timeout is in the order of 1.5 s. The resulting scenario will look similar to the case with increased mixing delays, as higher mixing delays, higher network latencies and higher processing times are indistinguishable to the recipient.

We note that the fragmentation happens at the protocol level, transparent to the application. Therefore, any message sent over the network is susceptible to the Fragging attack.

4.2 Fragging in Tor

As Tor is a circuit-based system, defining single logical "messages" in the context of Tor is hard. Instead, Tor defines *streams* of data [14], where the data sent over a stream during its lifetime can be seen as a message. Therefore, Tor fragments a message across many protocol-level *cells* that make up the stream.

Fragging-like attacks in Tor are known and can be carried out in multiple ways [28]: Modifying, inserting and dropping cells in a stream all lead to the circuit being torn down, as the encryption state between sender and exit relay goes out of sync. This allows an adversary to send at least a 1-bit confirmation signal to the end node. If the adversary knows how many cells are to be expected, they can deliberately select the position of the cell to drop, thereby encoding $\log n$ bits of information for the exit relay [20].

The Tor project initially considered such attacks as out-of-scope of its threat model,[6] but has since changed its stance.[7] Their mitigation currently relies on the client noticing abnormal behavior.

[5] See e.g. https://wondernetwork.com/pings/

[6] https://blog.torproject.org/one-cell-enough-break-tors-anonymity/

[7] https://spec.torproject.org/proposals/344-protocol-info-leaks.html

4.3 A Fragging model

We identify the following conditions which have to be met for a successful Fragging attack:

- C1 The adversary must be able to drop packets while knowing their original sender.
- C2 The adversary must be able to detect that a packet is missing.
- C3 The adversary must know the recipient from the subset of packets that have arrived.

C1 is met in Nym and Tor, because a malicious entry node can drop packets. As the packet has not been mixed yet, the sender's IP address is still visible.

C2 is met in Nym as the explicit fragment identifiers in the messages signal whether the set is complete or not. In Tor, C2 is met because a desynchronized decryption state in the client and exit node will signal that a circuit has been tagged.

C3 is met in Nym because other fragments of a fragment set will have arrived at the recipient, even if single fragments have been dropped. Similarly in Tor, a stream is connected to its destination before data is sent.

We note that all three conditions have to be met for Fragging to be possible: If C1 is not met, the adversary cannot embed the tag by dropping a fragment. If C2 is not met, the adversary will not recognize the tag. If C3 is not met, the adversary may recognize the tag, but it will no longer know who the recipient is.

5 A Fragging mitigation

In this section, we will describe our mitigation against the Fragging attack in the form of a new mix format, Scylla.[8] Scylla is an adaptation of Sphinx. It can be used in mix networks where Sphinx is currently used, providing secure fragmentation.

The core idea behind Scylla is to break the third condition for the Fragging attack: Scylla prevents the final hop from learning the message's recipient if a fragment is missing, similar to how Sphinx destroys this information if the payload has been tampered with. To do so, clients apply an all-or-nothing transform (AONT) [23] to the payload (including the recipient's address) before splitting the transformed payload into fragments. This ensures that the final hop must collect all fragments before it learns the recipient or the message.

In Scylla, we use the Optimal Asymmetric Encryption Padding (OAEP) scheme [4] as an AONT [5]. Further, we augment the header with a fragment identifier and fragment index, which allows the final hop to match fragments of the same message.

We acknowledge that a design that has a mix node store fragments for users increases the resource consumption of such nodes and can lead to denial-of-service attacks. We propose to randomize the selection of the final nodes (and not use fixed cascades), so that the load will be evenly distributed. Against malicious clients, techniques such as proof-of-work can be implemented to increase the cost to create fragments. Finally, we want to highlight that Scylla only prevents dropping-based fragging attacks. To also detect the delaying of fragments, it has to be combined with other techniques [3].

[8] Scylla is a Greek mythical creature which is often depicted as a woman with multiple dog heads coming out of her body.

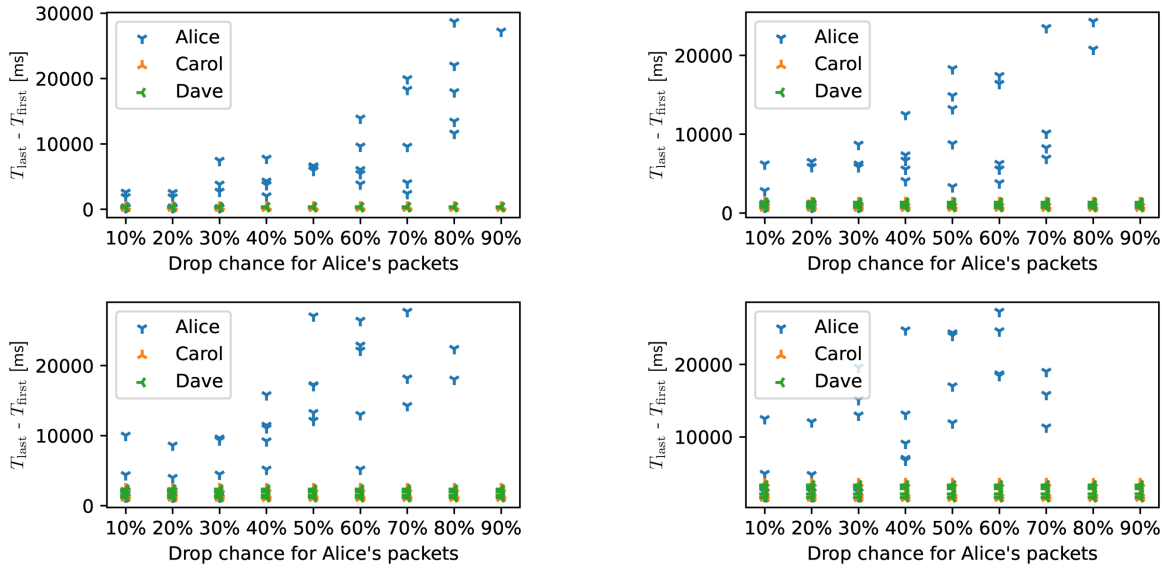

Figure 2: Time until a fragment set was completed, grouped by the sender of the message. Alice's packets were dropped with the given chance (x axis), while Carol's and Dave's packets were unaffected. For each drop chance, we have done 5 experiment runs with different random seeds. The top-left graph represents the scenario with Nym's default average per-hop delay of 50 ms**, top-right represents** 200 ms**, bottom-left represents** 350 ms**, and bottom-right represents** 500 ms**.**

We detail how Scylla works in the sections below, and show an overview in Figure 3.

5.1 Required primitives

Scylla requires the following primitives:

A group $\mathbb{G}$ with generator g and order q in which the Gap Diffie-Hellman Assumption holds.

Hash functions $h_b : \mathbb{G} \times \mathbb{G} \to \mathbb{Z}_q^*$ to generate blinding factors, and $h_\mu, h_\rho, h_\pi : \mathbb{G} \to \{0,1\}^\kappa$ for key derivation.

A pseudorandom generator $\rho : \{0,1\}^\kappa \to \{0,1\}^*$, a message authentication code $\mu : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^\kappa$ and a block cipher $\pi : \{0,1\}^\kappa \times \{0,1\}^l \to \{0,1\}^l$.

We denote three "flag bytes" with arbitrary, distinct, publicly known values, $F_R, F_F, F_D \in \{0,1\}^8$ to indicate whether a mix node should forward a message to the next mix node, whether it acts as the final node, or whether it delivers a reply.

5.2 Message creation

In this section, we describe how Scylla onions are formed. We denote this procedure as $\texttt{CreateOnions}(R, f, P, P^{\succ}, m)$, where R is the recipient, $P_1, \ldots, P_f$ are the paths for the fragments, $P^{\succ}$ is the final hop, and $m_1, \ldots, m_f$ are the message fragments. As the first fragment will contain extra bytes to allow for integrity checking, as well as the recipient's address, we require that $|m_1| + 3\kappa = |m_2| = \ldots = |m_f| = l$. We implicitly treat $P^{\succ}$ to be appended to each P_i, such that all paths end in $P^{\succ}$.

First, the client prepares the *payload*. The payload consists of the authentication tag, the recipient, and the message content. As the content is already given as fragments, the client only needs to prepend the authentication tag and recipient to the first fragment:

$$M_1 = 0^\kappa || R || m_1$$
$$M_i = m_i \text{ for } i > 1$$

The fixed string 0^κ is later used to verify whether decryption has succeeded.

Those fragments will then be transformed using OAEP:

$$A_1, \ldots, A_f = \text{OAEP}(M_1, \ldots, M_f)$$

We note that OAEP will add κ bits to the first fragment, therefore bringing all prepared fragments to the same size l.

Next, the client draws an identifier for this fragment set $F \xleftarrow{R} \{0,1\}^\kappa$ and builds the actual onions. This is done in the same way as Sphinx onions are built [12], but F and the fragment index i are included in each header for the exit node:

First, the client draws f random secrets, $x_i \xleftarrow{R} \mathbb{Z}_q^*$.

Then, for each path P_i, we let $\nu = |P_i|$. The client then computes the shared secrets and blinding factors:

$$\alpha_{i;0} = g^{x_i}, \quad s_{i;0} = y_{P_{i;0}}^{x_i}, \quad b_0 = h_b(\alpha_{i;0}, s_{i;0})$$
$$\alpha_{i;1} = g^{x_i b_0}, \quad s_{i;1} = y_{P_{i;1}}^{x_i b_0}, \quad b_1 = h_b(\alpha_{i;1}, s_{i;1})$$
$$\ldots$$
$$\alpha_{i;\nu} = g^{x_i b_0 \ldots b_{\nu-1}}, \quad s_{i;\nu} = y_{P_{i;\nu}}^{x_i b_0 \ldots b_\nu}, \quad b_\nu = h_b(\alpha_{i;\nu}, s_{i;\nu})$$

Additionally, the client computes the filler strings $\phi_{i;0}$ to $\phi_{i;\nu-1}$:

- $\phi_{i;0} = \varepsilon$
- For $0 < j < \nu$:

$$\phi_{i;j} = \{\phi_{i;j-1} || 0^{2\kappa+8}\} \oplus \{\rho(h_\rho(s_{i;j-1}))_{[(r-j)(2\kappa+8)..(r+1)(2\kappa+8)]}\}$$

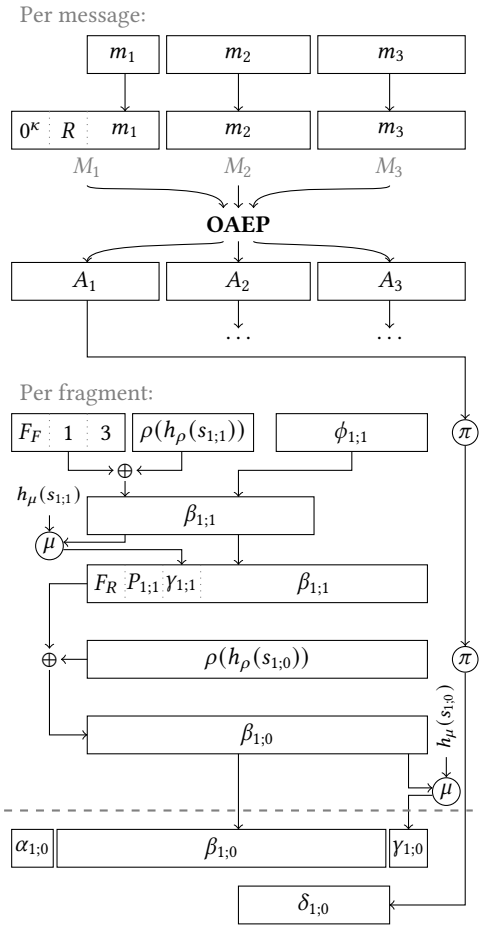

Figure 3: Overview of Scylla message creation with an example message consisting of 3 fragments over 2 hops. Elements below the thick dashed line are the final onion parts.

Finally, the client computes the headers $H_{i;v-1}, \ldots, H_{i;0}$ and payloads $\delta_{i;v-1}, \ldots, \delta_{i;0}$ for the f onions:

$$
\begin{aligned}
H_{i;j} &= (\alpha_{i;j}, \beta_{i;j}, \gamma_{i;j}) \\
\beta_{i;v-1} &= \{(F_F||i||F) \oplus \rho(h_\rho(s_{i;v-1}))\}||\phi_{i;v-1} \\
\beta_{i;j} &= \{F_R||P_{i;j+1}||\gamma_{i;j+1}||\beta_{i;j+1}\}_{[..2r\kappa]} \oplus \rho(h_\rho(s_{i;j})) \\
\gamma_{i;j} &= \mu(h_\mu(s_{i;j}), \beta_i) \\
\delta_{i;v-1} &= \pi(h_\pi(s_{i;v-1}), A_i) \\
\delta_{i;j} &= \pi(h_\pi(s_{i;j}), \delta_{i;j+1})
\end{aligned}
$$

5.3 Onion processing

When a mix node receives an onion $((\alpha, \beta, \gamma), \delta)$, processing starts identical to Sphinx until the node has decrypted the routing information. In detail, the node performs the following steps:

First, the node checks whether it has seen (α, β, γ) before. If so, the node discards the onion.

Then, it computes the shared secret $s = \alpha^x$, where x is the private key of the mix node. Then, it verifies the MAC by checking whether $\gamma = \mu(h_\mu(s), \beta)$. If the values are not equal, the node discards the onion. If the values are equal, processing continues.

Next, the node decrypts the routing information $B = (\beta||0_{2\kappa+8}) \oplus \rho(h_\rho(s))$. The first byte $B_0 = B_{[0..8]}$ then determines how the onion is processed further:

If $B_0 = F_R$, the mix node relays the onion to the next node. In this case, $P = B_{[8..8+\kappa]}$ gives the address of the next hop, $\gamma' = B_{[8+\kappa..8+2\kappa]}$ gives the next MAC and $\beta' = B_{[8+2\kappa..]}$ gives the next routing information. The node then blinds the group element $\alpha' = \alpha^{h_b(\alpha,s)}$. Finally, the node decrypts the payload $\delta' = \pi^{-1}(h_\pi(s), \delta)$. It then relays the new onion $(\alpha', \beta', \gamma'), \delta')$ to P.

If $B_0 = F_F$, the mix node is the final hop on a packet's path. It extracts the fragment index $i = B_{[8..8+\kappa]}$ and the fragment set ID $F = B_{[8+\kappa..8+2\kappa]}$, and it decrypts the payload $\delta' = \pi^{-1}(h_\pi(s), \delta)$. Then, it retrieves the previous fragments for set F, orders them by their indices i and reverts the OAEP transform on their payloads $M = \text{OAEP}^{-1}(\delta'_0, \delta'_1, \ldots)$. Further processing then depends on M:

- If M starts with κ bits of zeroes, the authentication tag matches. The node can then extract the recipient R and message m, and forward the final message to the user.
- If M does not start with κ zeroes, the node saves the collected fragments for later and re-runs this step once another fragment for this set arrives. If no further fragment arrives within a set time, the node assumes that M is orphaned and discards it.

If $B_0 = F_D$, the mix node is the final hop on a reply path. It extracts the destination address from $D = B_{[8..8+\kappa]}$ and the reply ID $I = B_{[8+\kappa..8+2\kappa]}$. It then passes on r and $\pi^{-1}(h_\pi(s)\delta)$ to D.

5.4 SURB and reply creation

Scylla supports replies in a similar way as Sphinx by generating single-use reply blocks (SURBs). Like in Sphinx, the destination address in SURBs is contained in the header instead of the payload, as the recipient cannot be trusted to see this value.

While it might seem like this choice opens up the possibility for Fragging attacks again, we note this is not a problem for replies: The assumption for Fragging is that the adversary can see the recipient address as well as recognize that a fragment is missing. In the backwards direction however, a missing fragment can only be discovered by the original sender, who is honest by assumption.

This allows us to optimize the space needed in Scylla packets: In the forward direction, the header contains the fragment identifier and index, whereas the payload contains the destination address. In the backward direction, we can swap those elements, such that the space for fragment identifier and index are now used for the destination, and the payload contains the fragment meta-information. This is shown in Figure 4.

In order to create a SURB, the client follows the same steps as in message creation (Section 5.2), with the following changes:

- An empty payload is used.
- At the final header, instead of i and F, the client embeds its own address and a random reply identifier $I \leftarrow^R \{0,1\}^\kappa$.
- At the final header, instead of using F_F, it uses F_D.
- The client stores the $s_{1;j}$ indexed by I such that it can later unwrap the payload.

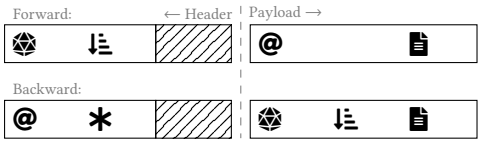

Figure 4: Which information is in the header, and which information is in the payload: @ recipient, ✱ reply ID, fragment set ID, fragment index, message text.

The resulting onion H_0 can then be used as a SURB. Note that in addition to H_0, the SURB user also needs to know the address of the first hop.

In order to use SURBs, Bob first splits his message into fragments $m_1, \ldots, m_f$ with $|m_1| = \ldots = |m_f| = l - 2\kappa$. Then he draws a random fragment set identifier $F \xleftarrow{R} \{0,1\}^\kappa$ and prepares each payload as $M_i = F||i||m_i$. He then attaches each M_i to a SURB, and sends it to the respective first hop.

When Alice receives a reply, she looks up the shared secrets using the reply identifier I. She then computes the following to reconstruct the payload:

$$\delta' = \pi(h_\pi(s_{1;1}), \ldots (\pi(h_\pi(s_{1;v-1}), \delta)))$$

Finally, she can parse the payload as $F||i||m_i$ to collect the fragments for the reply message.

5.5 Interaction with retransmissions

In Section 4.1, we have described Fragging against Nym. Due to Nym's retransmission algorithm, the adversarial recipient always receives all fragments; drops can be detected by the time it takes all fragments to arrive. Scylla alone cannot prevent Fragging in this setting, as the resulting delays in delivery would still be observable. If we want to preserve Nym's reliable delivery while closing the side-channel, we have to also adapt the retransmission algorithm.

Currently, retransmission in Nym is implemented via per-fragment acknowledgments. To match Scylla's all-or-nothing delivery, acknowledgments must cover whole messages instead. It has to be sent upon reception of all fragments, only, once the message has successfully been restored. If a message is not acknowledged in time, *all* of its fragments have to be re-sent under a fresh identifier.

The change to per-message acknowledgments prevents the adversarial receiver from learning the fragment set completion time if it is over a certain threshold. Reliable transmission is preserved at the cost of increased latency at high drop chances and high fragment counts. We discuss the performance impact of this change in greater detail in Section A.

6 Security

In this section, we define and prove the security of our mitigation defined in Section 5. Ideally, we want to give a formalization for onion routing with fragmentation within the Universal Composability (UC) framework [7], like previous work on formal onion routing. However, we argue in Section B why the standard UC model does not permit such a formalization without additional assumptions. Under hybrid models (such as UC with a Common Reference String [8, 10]), realizing this functionality might be possible, but we consider it outside the scope of this work. Instead, we define security in terms of game-based properties and show that Scylla fulfills those properties.

6.1 Security overview

For a mix format that securely supports fragmentation, we identify three requirements:

First, when individual onions pass through an honest mix node, the input and output onions must be unlinkable. This is the same for non-fragmented onions, and it is formalized by the *layer unlinkability* property of Kuhn et al. [18]

Second, when onions arrive at their final hop and their (reassembled) message is passed on, those inputs and outputs must be unlinkable as well. This is a form of *layer unlinkability* adapted for the final hop.

Third, when the final hop receives an incomplete fragment set, it must not learn parts of the message or the intended recipient. This prevents the Fragging attack, as dropping a single fragment causes the whole message to be rendered inaccessible.

As Scylla is modified version of Sphinx and relies on the same key blinding mechanism, we refer to existing security proofs for the first point of "standard" layer unlinkability [27].

For the second and third points, we provide formalizations inspired by Kuhn et al.'s original games and proofs in the next section: For the original games, Kuhn et al. prove that protocols which satisfy their *layer unlinkability* and *tail indistinguishability* properties realize their given ideal functionality for onion routing. We make sensible adaptions to *layer unlinkability* to ensure fragmented onions are not linkable past an honest node, and we replace *tail indistinguishability* with a new property that takes fragmented messages into account.

6.2 Game properties

As UC realization is unattainable, we instead define the security of fragmentation-supporting onion routing directly via games. To do so, we look at the generalized versions of *layer unlinkability* and *tail indistinguishability* from Scherer et al.'s framework [26], as well as the *hiding* notion of commitment schemes.

As our first requirement is already covered by previous formalizations, we do not define a separate game for this. For the two remaining requirements, we define two additional properties:

The first property is an extension to the layer unlinkability (LU) game at the final hop, which ensures that the adversary cannot map incoming onions to outgoing messages, given that the final hop is honest. We call this extended property *fragmented layer unlinkability* (FLU).

The second property formalizes the security given that fragments are missing and the final hop is malicious, ensuring that it cannot learn anything about the final recipient without collecting all fragments. We call this property *fragment hiding* (FH).

Fragmented layer unlinkability. Our extension to layer unlinkability is of formal nature. We adapt the notion to only consider a single relay, and change the challenge so that it contains all fragments of a message. The adversary's task is then to distinguish between two challenge sets, each representing a complete onion.

We begin by adjusting the notation for onion routing to consider fragments. As such, we consider the following definition of an onion routing scheme:

Definition 2 (Onion routing with fragmentation)**.** Given the sets $\mathcal{X}_R$ of recipients, $\mathcal{X}_\text{P}$ of router names, $\mathcal{X}_\text{PK}$ of public keys, $\mathcal{X}_\text{SK}$ of secret keys, $\mathcal{X}_m$ of messages, and $\mathcal{X}_O$ of onions, an onion routing scheme with fragmentation consists of the following algorithms:

- $(\text{PK}_H, \text{SK}_H) \leftarrow \text{Gen}(1^\kappa, p, P_H)$ is a key generation algorithm, which takes the public parameters p as well as the name $P_H \in \mathcal{X}_\text{P}$ of a router. It returns a public key from $\mathcal{X}_\text{PK}$ and a secret key from $\mathcal{X}_\text{SK}$.
- $\mathbb{O} \leftarrow \text{FormOnion}(R, m, \mathcal{P}, \text{PK}_\mathcal{P})$ is used to build onions. It takes the recipients address $R \in \mathcal{X}_R$, the ordered set of message fragments $m \in \mathcal{X}_m^f$ (where f is the number of fragments), the corresponding ordered set of paths $\mathcal{P} \in (\mathcal{X}_\text{P}^n)^f$, and public keys $\text{PK}_\mathcal{P} \in \mathcal{X}_\text{PK}^{f \times n}$ for all nodes in the paths. It returns an onion for each fragment and each node on the fragment's path ($\mathbb{O} \in \mathcal{X}_O^{f \times n}$).
- $o \leftarrow \text{ProcOnion}(\text{SK}_H, \mathbb{O}, P_H)$ processes a set $\mathbb{O} \in \mathcal{X}_O^*$ of onions with the given secret key $\text{SK}_H \in \mathcal{X}_\text{SK}$ and the given router name $P_H \in \mathcal{X}_\text{P}$. It returns either the resulting onions, the final message, or $\perp$ ($o \in \mathcal{X}_O^* \cup (\mathcal{X}_R \times \mathcal{X}_m) \cup \{\perp\}$).

We make no assumption on the structure of $\mathcal{X}_O$ or its elements, but we expect a scheme to be "correct" in the sense that properly formed onions by FormOnion will result in the original message and recipient being revealed when processed by a series of ProcOnion calls using the correct secret keys.

We now define the security game:

Definition 3 (Fragmented layer unlinkability)**.** *FLU* is defined as

(1) The adversary receives the honest router name P_H and challenge public key PK_H, chosen by the challenger by letting $(\text{PK}_H, \text{SK}_H) \leftarrow \text{Gen}(1^\kappa, p, P_H)$.
(2) The adversary may submit any number of **Proc** requests for P_H to the challenger. When asked for $\textbf{Proc}(P_H, \mathbb{O} = \{(\eta_1, \delta_1), \ldots\})$, the challenger checks whether any header η_i is on the η_H list. If it is not on the list, it sends the output of $\text{ProcOnion}(\text{SK}_H, \mathbb{O}, P_H)$ to the adversary, extends η_H with all η_i and O_H with O_i.
(3) The adversary submits message fragments $m = m_1, \ldots, m_f$, paths $\mathcal{P} = \mathcal{P}_1, \ldots, \mathcal{P}_f$ with $\mathcal{P}_i = (P_{i,1}, \ldots, P_{i,n})$ and $P_{i,n} = P_H$ for all $1 \le i \le f$, a receiver R, and public keys $\text{PK}_{i,j}$ for all nodes except P_H.
(4) The challenger picks $b \in \{0, 1\}$ at random.
(5) The challenger creates the onions with the adversary's input choice:

$$\begin{bmatrix} O_{1,1} & \ldots & O_{1,n} \\ \ldots & & \\ O_{f,1} & \ldots & O_{f,n} \end{bmatrix} \leftarrow \text{FormOnion}(R, m, \mathcal{P}, \text{PK}_\mathcal{P}),$$

and replacement onions with a random receiver $\bar{R}$, a random message $\bar{m}$ with the same number of fragments, and the same paths $\mathcal{P} = \mathcal{P}_1, \ldots, \mathcal{P}_f$:

$$\begin{bmatrix} \bar{O}_{1,1} & \ldots & \bar{O}_{1,n} \\ \ldots & & \\ \bar{O}_{f,1} & \ldots & \bar{O}_{f,n} \end{bmatrix} \leftarrow \text{FormOnion}(\bar{R}, \bar{m}, \mathcal{P}, \text{PK}_\mathcal{P}).$$

(6) If $b = 0$, the challenger outputs $O_{1,1}, \ldots, O_{f,1}$ to the adversary. Otherwise, the challenger outputs $\bar{O}_{1,1}, \ldots, \bar{O}_{f,1}$.
(7) The adversary may submit any number of $\textbf{Proc}(P_H, \mathbb{O})$ requests to the challenger. If $b = 0$, the challenger proceeds with the requests as in Step 2. If $b = 1$, the challenger answers the oracle queries as in Step 2, except for any $O = (\eta, \delta)$ where $\eta \in \{\bar{\eta}_{1,n}, \ldots, \bar{\eta}_{f,n}\}$. For such onions, processing works as follows:
   - If $\eta = \bar{\eta}_{i,n}$ and $\delta = \bar{\delta}_{i,n}$, treat O as if it was $O_{i,n}$ and continue processing.
   - If $\eta = \bar{\eta}_{i,n}$ and $\delta \neq \bar{\delta}_{i,n}$, treat O as if it was $O = (\eta_{i,n}, \delta')$, where $\delta' \leftarrow^\text{R} \{0, 1\}^l$.
(8) The adversary outputs guess b'.

FLU is achieved if any PPT adversary cannot guess $b' = b$ with a probability non-negligibly better than 1/2.

Fragment hiding. Our second property represents the idea of "hiding", similar to hiding in commitment schemes. This means that as long as not all fragments have arrived, the final hop must not be able to learn anything about the recipient or the message.

Definition 4 (Fragment hiding)**.** *FH* is defined as

(1) The challenger chooses a bit $b \in \{0, 1\}$ at random.
(2) The adversary chooses two recipients R, $\bar{R}$ and two sets of fragments $\{m_1, \ldots, m_f\}$, $\{\bar{m_1}, \ldots, \bar{m_f}\}$. She also selects a non-empty subset $D \subset \mathbb{N}_f$ of fragments that will be dropped, as well as the name and public key of the final hop PK_n.
(3) If $b = 0$, the challenger creates onions with recipient R, fragments $m_1, \ldots, m_f$, and paths that consist of P_n ($\mathcal{P} = \{\{P_n\}, \ldots, \{P_n\}\}$):

$$O_1, \ldots, O_f = \text{FormOnion}(R, \{m_1, \ldots, m_f\}, \mathcal{P}, \text{PK}_n).$$

Otherwise, the challenger creates onions with the choices $\bar{R}$ and $\bar{m_1}, \ldots, \bar{m_f}$:

$$O_1, \ldots, O_f = \text{FormOnion}(\bar{R}, \{\bar{m_1}, \ldots, \bar{m_f}\}, \mathcal{P}, \text{PK}_n).$$

The challenger passes $\{O_i, i \in 1 \ldots f, i \notin D\}$ to the adversary.
(4) The adversary outputs guess b'.

FH is achieved if any PPT adversary cannot guess $b' = b$ with a probability non-negligibly better than 1/2.

6.3 Proving Scylla secure

In this section, we argue that Scylla fulfills the properties formalized in the previous section.

Scylla achieves fragmented layer unlinkability. Intuitively, FLU is achieved because the recipient and message are part of the onion payloads, which are processed with a pseudorandom permutation at each step. Therefore, even if the adversary knows the message (or the fragments that make up the message), it cannot re-identify them without knowing the shared secret at the final hop.

The full proof is given in Section C.

Scylla achieves fragment hiding. Intuitively, Fragment Hiding is achieved because of the All-or-Nothing-Transform: As soon as one fragment is missing, no information about the input can be learned.

Before we give a formal reduction, we recall the security definition of AONTs by Boyko [5]. In particular, we consider *adaptive indistinguishability*:

Definition 5 (AONT-AIND [5])**.** Let AONT be a randomized transform mapping n-bit messages to n'-bit outputs and using random oracle Γ. Let λ be between 1 and n'. An adversary $\mathcal{A}$ is said to succeed in $(T, q_\Gamma, \varepsilon)$-adaptively-distinguishing AONT with λ missing bits if she wins the following game with a non-negligible probability:

(1) The adversary is given λ and access to Γ. She selects λ bit positions and outputs $L \subseteq \{1, \dots, n\}$ with $|L| = \lambda$, and $c_s \in \{0,1\}^*$.
(2) The adversary is given c_s and access to Γ. She outputs $x_0 \in \{0,1\}^n$, $x_1 \in \{0,1\}^n$ and $c_f \in \{0,1\}^*$.
(3) The adversary is given c_f and for a random bit $b \leftarrow^{\mathrm{R}} \{0,1\}$ $\mathrm{AONT}^\Gamma(x_b)$ with bit positions L missing. The adversary has access to Γ. She has to guess b.

We now give a proof that Scylla fulfills Fragment Hiding in two steps: First, we give a reduction of Fragment Hiding to AONT-AIND that is generic over the AONT. Then, we apply Boyko's bounds for OAEP as the specific AONT in our construction to show that Scylla indeed fulfills Fragment Hiding.

Theorem 1. *Given an AONT that is $(T, q_\Gamma, \varepsilon)$-AIND with T and q_Γ polynomial in κ, and ε negligible in κ, then Scylla achieves Fragment Hiding.*

Proof. We prove this theorem by reduction: We assume that we have an adversary $\mathcal{A}_{\mathrm{FH}}$ against Fragment Hiding, and construct an adversary $\mathcal{A}_{\mathrm{AONT}}$ that breaks AONT-AIND with it.

We begin by constructing $\mathcal{A}_{\mathrm{AONT}}$ using $\mathcal{A}_{\mathrm{FH}}$:

- $\mathcal{A}_{\mathrm{AONT}}$ first queries $\mathcal{A}_{\mathrm{FH}}$ to get the recipients $R, \bar{R}$, the message fragments $m, \bar{m}$, the indices of the fragments to drop D and the name and public key of the final hop PK_n.
- $\mathcal{A}_{\mathrm{AONT}}$ translates the specified fragment indices in D to the bit positions in the output, and passes this as L to the AONT-AIND challenger: $L = \{li, li+1, \dots, li+l \; \forall \; i \in D\}$.
- $\mathcal{A}_{\mathrm{AONT}}$ constructs the payloads $x_0 = 0^\kappa || R || m$ and $x_1 = 0^\kappa || \bar{R} || \bar{m}$.
- $\mathcal{A}_{\mathrm{AONT}}$ passes x_0 and x_1 as challenge to the AONT-AIND challenger and receives the challenge x_b.
- The challenge x_b will miss exactly the bits corresponding to the fragments specified in D. $\mathcal{A}_{\mathrm{AONT}}$ uses the remaining bits in x_b as payload and wraps them in an onion using PK_n, a random fragment set identifier, and the correct fragment indices. It passes those onions to $\mathcal{A}_{\mathrm{FH}}$.
- $\mathcal{A}_{\mathrm{FH}}$ produces a guess b'. $\mathcal{A}_{\mathrm{AONT}}$ forwards this guess to the AONT-AIND challenger.

If a PPT adversary $\mathcal{A}_{\mathrm{FH}}$ wins the Fragment Hiding game with probability $\frac{1}{2} + \varepsilon$, then $\mathcal{A}_{\mathrm{AONT}}$ wins the AONT-AIND game with probability $\frac{1}{2} + \varepsilon$. Per assumption, no such $\mathcal{A}_{\mathrm{AONT}}$ that wins with non-negligible ε can exist. Therefore, no $\mathcal{A}_{\mathrm{FH}}$ that wins with non-negligible ε can exist, and Scylla achieves Fragment Hiding. □

Theorem 2. *If $\kappa > 14$ and $l > \kappa$, then OAEP is $(T, q_\Gamma, \varepsilon)$-AIND against a PPT adversary with T and q_Γ polynomial in κ, and ε negligible.*

Proof. We cannot apply Boyko's theorem directly, as $\lambda > \kappa$. However, we can construct a second adversary $\mathcal{A}'_{\mathrm{AONT}}$ that internally uses $\mathcal{A}_{\mathrm{AONT}}$ and only deletes $\lambda = \kappa$ bits: $\mathcal{A}'_{\mathrm{AONT}}$ receives the bit positions to delete from $\mathcal{A}_{\mathrm{AONT}}$, but only passes κ of them to the challenger. Then, when forwarding the challenge to $\mathcal{A}_{\mathrm{AONT}}$, $\mathcal{A}'_{\mathrm{AONT}}$ removes the remaining bits "manually." The winning chance of $\mathcal{A}'_{\mathrm{AONT}}$ is therefore the same as $\mathcal{A}_{\mathrm{AONT}}$.

Now we can apply Boyko's bounds from Theorem 3. In particular, we obtain a winning advantage of

$$\varepsilon \leq 8(q_G + q_H)\frac{\kappa}{\log_2 \kappa} 2^{-\kappa},$$

where q_G and q_H are the number of queries to the hash functions G and H, respectively.

We cannot know the exact values of q_G or q_H as we do not know how often an arbitrary adversary evaluates G or H, but we know them to be polynomially bounded. We therefore know that

$$\varepsilon \leq \mathrm{poly}(\kappa) 2^{-\kappa},$$

which results in a negligible winning chance in our security parameter, and polynomial values for T and q_Γ. □

Theorem 2 states that OAEP meets the requirements of Theorem 1. Together, those theorems prove that Scylla achieves Fragment Hiding.

Finally, since SURBs are constructed and processed in the same way as forward onions in Scylla, the same privacy guarantees hold for them.

7 Evaluation

We evaluate the performance of Scylla via benchmarks and compare it to Sphinx as a state-of-the-art format. To do so, we have implemented a prototype of Scylla in Rust, and we take the Sphinx implementation of the Nym project as comparison.[9] We have published our code open-source at https://github.com/kit-ps/fragging.

We want to understand the overhead that Scylla's additional security transformations have, both in computation time as well as in packet size. We therefore use our prototype to measure four different timings:

- Onion creation time: The time it takes for a client to form onions
- Onion processing time: The time it takes for a relay to process an onion,
- Onion reassembly time: The time it takes for the final relay to reassemble all fragments, and
- SURB creation time.

We instantiate Scylla using the same primitives as Sphinx: Curve-25519 as the Diffie-Hellman group $\mathbb{G}$, LIONESS [1] as the wide-block cipher π, ChaCha20 as the pseudorandom generator ρ, salted SHA256 hashes for the hash functions h_*, and Shake128 for the function G in OAEP.

[9] https://github.com/nymtech/sphinx

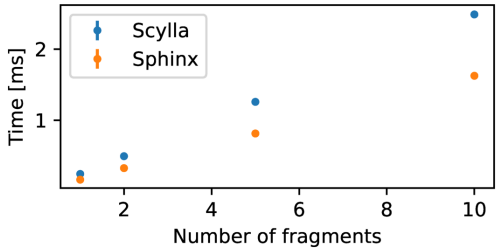

Figure 5: Onion creation time depending on the number of fragments. Here, the path length is fixed to 3, and the payload size is fixed to 128 bytes. Error bars showing the 95% confidence interval are hidden behind the markers.

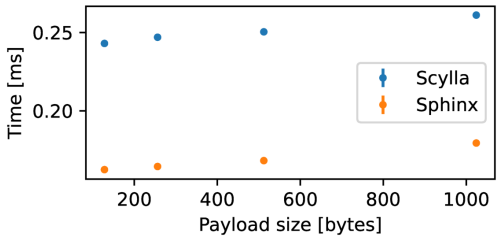

Figure 6: Onion creation time depending on the payload size (the size of a single fragment). Here, the fragment count is fixed to 1, and the path length is fixed to 3. Error bars showing the 95% confidence interval are hidden behind the markers.

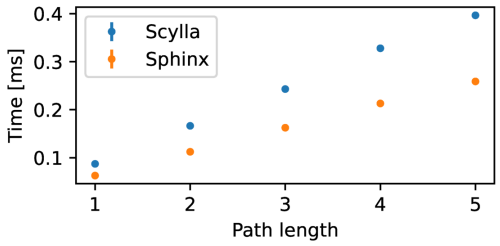

Figure 7: Onion creation time depending on the path length. Here, the fragment count is fixed to 1, and the payload size is fixed to 128 bytes. Error bars showing the 95% confidence interval are hidden behind the markers.

Each experiment was executed on a laptop with an AMD Ryzen 5 5625U and 16 GiB of RAM. The recipient address, mix node keys and payload were generated randomly. The values were collected using the `criterion`[10] framework, which provides 100 measurement samples. We report the average time.

Additionally, we use the implementation to report the sizes of onions under different parameters.

7.1 Onion creation time

We want to understand how large the overhead for clients is to create fragmented messages using Scylla. We therefore measure the time it takes for a client to create Scylla onions. We expect this part to be the most expensive one, as the client has to pre-compute all shared secrets for all nodes along the path.

We expect the path length, the payload size (of a single onion) and the fragment count to have the most impact on the onion creation time, and therefore we vary those parameters in our experiments. These are also the parameters likely to vary in a deployed network, and as such, our results help to determine sensible parameter choices.

As Sphinx does not include the concept of fragmentation, we scale the Sphinx result by creating as many onions as given by the fragment count. This represents the "trivial" approach, where each fragment is simply wrapped in a Sphinx onion.

We expect the fragment count to have a linear influence on the creation time, as each fragment has to be wrapped in a separate onion – therefore, sending n fragments requires the creation of n onions. Our results (shown in Figure 5) confirm this relation. Each fragment takes around 243 µs to create.

Further, we expect the payload size to have only a small influence on the creation time, as the payload is processed using fast symmetric primitives. We can see in Figure 6 that there is a small linear increase, but the difference between 128 B of payload and 1024 B is less than 5 µs. Compared to Sphinx, Scylla is slightly slower, but the difference is only around 50 µs.

Finally, like Sphinx, we expect the path length to have a linear influence on the creation time, as each additional hop requires the client to compute and blind an additional shared secret. Our results (shown in Figure 7) confirm this relation: Each hop adds around 90 µs to the onion creation time.

Overall, we can see that Scylla scales as expected, and is only slightly slower than Sphinx. The biggest difference is that the payload size has a larger impact on the onion creation time in Scylla than in Sphinx. However, we believe that Scylla is still fast enough in practice: By default, Nym adds an expected delay of 15 ms per hop, not counting network latency. An overhead of 243 µs per fragment is therefore negligible.

7.2 Onion processing and reassembly time

We want to understand how large the overhead is for mix nodes that process onions, both for nodes in the middle of the path as well as for the exit node. We therefore measure the time it takes for a mix node to process incoming Scylla onions. We split this evaluation into three parts, representing the three different scenarios for a mix node: A position in the middle of a path, as an exit node for (forward) messages, and as the exit node for replies.

For mix nodes in the middle of a path, we expect that Scylla performs like Sphinx: The majority of the work lies in deriving and re-blinding the Sphinx group element, while the payload is processed with a fast symmetric cipher. We therefore expect that the payload size has a negligible impact on the processing time.

As the middle node does not see further nodes, nor does it see other fragments, the parameters of path length or fragment count are not applicable here.

[10] https://crates.io/crates/criterion

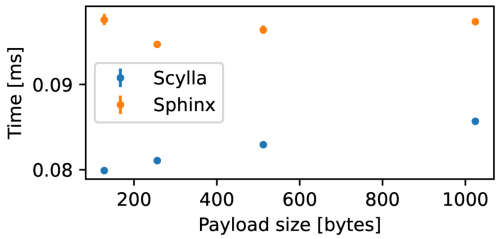

Figure 8: Onion processing time for mix nodes in the middle of a path. Error bars showing the 95% confidence interval are hidden behind the markers.

Our results in Figure 8 show that there is a small linear increase in processing time when increasing the payload size. Both Scylla and Sphinx need approximately the same time of 0.09 ms. Given the expected delay of 15 ms at a mix node, the processing time is only a small part of the overall latency.

For mix nodes at the end of a path, we expect the work for them to be lower, as they do not need to re-blind the group element. We measure a time of 47 µs for single forward fragments and replies alike, both with a payload size of 1024 B.

For fragmented messages, in addition to the time to "unwrap" the onion, the final hop must also re-assemble the message from its fragments. This is the inverse of the client's transformation. As such, we also expect the fragment count to have a small linear impact on this re-assembly time.

Our results in Figure 9 confirm this expectation. We leave out the comparison with Sphinx here, as Sphinx does not have this step. Overall we can see that message re-assembly only adds a few microseconds to the overall processing time.

From Section 7.1 and Section 7.2 we can estimate the total extra time that Scylla adds to a communication: For a path length of 3, and f fragments, we get

$$f \cdot 243\,\mu s + 2 \cdot 86\,\mu s + 47\,\mu s + f \cdot 4\,\mu s = 239\,\mu s + f \cdot 247\,\mu s.$$

This time consists of the onion creation time for the client, onion processing times for two relays and one final hop, and the onion re-assembly time. Compared to the total expected 45 ms of mixing delay, the processing overhead is negligible.

7.3 SURB creation time

We want to know the overhead that Scylla clients have when creating SURBs. Clients have to create a SURB for every reply fragment they intend to receive, so understanding the cost of this feature is important. We therefore measure the time it takes for a client to create a Scylla SURB using our prototype.

We expect the path length to have a linear impact on the SURB creation time, similar to the onion creation time: The client has to derive an additional shared secret for each hop along the path, which is an expensive operation.

Our results in Figure 10 confirm this expectation: The time it takes to create a SURB is similar to the time it takes to create a single-fragment onion for the same path length, indicating that the majority of the work lies in header creation.

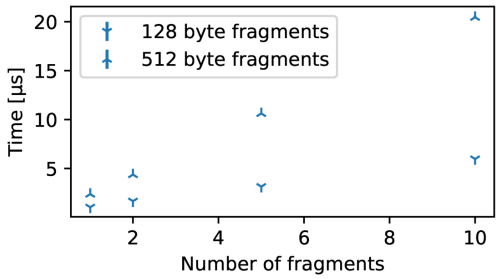

Figure 9: Time it takes for a mix node to re-assemble the message from fragments, depending on the number of fragments. Error bars showing the 95% confidence interval are hidden behind the markers.

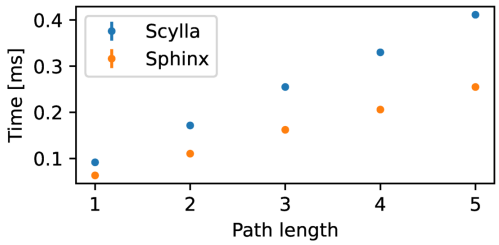

Figure 10: SURB creation time depending on the path length. Error bars showing the 95% confidence interval are hidden behind the markers.

Overall, Scylla SURBs can be generated fast in practice and with only a small overhead compared to Sphinx SURBs.

7.4 Additional communication cost

Finally, we want to know how much size overhead the added security of Scylla adds to the packets. Therefore, we compare the size of Scylla packets to that of Sphinx packets. Since the payload in both Sphinx and Scylla does not introduce any overhead, we only focus on the header size.

The main influence on the header size is the maximum path length, as the headers are padded to accommodate for the longest path. Each hop then requires space for the address, MAC, and flag byte, as well as any additional per-hop information like mixing delays. Added to that are the initial group element α and the initial MAC.

We use our implementation to output various onion sizes, and use the Sphinx implementation to output the corresponding sizes for Sphinx. To provide comparable numbers, we ensure that we use the same lengths for addresses and per-hop information as Sphinx.

From our results in Figure 11 we can see that Scylla adds a small 39 B overhead compared to Sphinx. The per-hop increase is 60 B, the same as in Sphinx. This corresponds to the size of an address (32 B), a MAC (16 B), a mixing delay (8 B), a version identifier (3 B) and a flag byte.

Overall we conclude that Scylla has low overhead, staying in line with Sphinx's compactness.

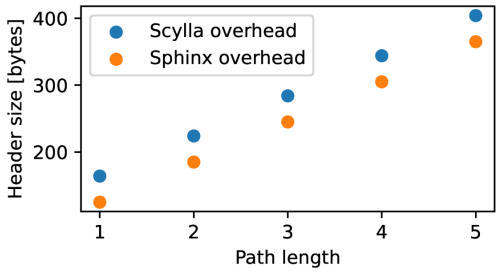

Figure 11: Header size depending on the maximum path length.

8 Limitations and future work

In this section, we will discuss the limitations of our mitigation approach.

First, our mitigation is tailored as a replacement for Sphinx and only works in networks that operate in the service model. It is important that there is a hop before the actual recipient, such that when the tag is discovered (via the missing fragment), the information about the recipient is lost. In the integrated system model, the sender-recipient link can be confirmed by missing fragments even if the content of the fragments is lost.

Second, our mitigation only works against adversaries that drop messages. Adversaries that delay messages can still tag a fragment set by inducing large latencies between the first and last packets of a fragment set. We see this as an orthogonal problem, as it also affects single-fragment onions, and point to work like *Bruisable Onions* by Ando et al. [3] for mitigation techniques.

Finally, we note that our mitigation relies on the whole message being known in advance, and as such, it cannot be used to protect streams of data. In a data stream, once the first packet has reached its recipient, an adversary can always drop all future packets to confirm the relationship.

9 Discussion

Scope. In the design of ACSs, we commonly ignore the message content, under the assumption that the sender will not reveal themselves "voluntarily". As such, it could be argued that fragging is outside of the scope of an ACS's consideration: A similar attack would be possible if the user "manually" fragments their message.

However, fragmentation is done by the ACS, transparent to the application. It is therefore an inherent part of the protocol: Even if users take care not to reveal their identity, the Fragging attack is possible. Therefore, message fragmentation in mix networks must be seen as a fundamental feature of the network.

Approach. In our design, we use an explicit AONT to transform the fragments. With wide-block ciphers like AEZ, this step might be unnecessary: The cipher could also be used as an AONT by concatenating all fragments and treating them as one wide block. This removes the need for an extra AONT step, as it would be combined with the encryption. However, we keep the explicit AONT step to make the security requirements clearer, as usual security notions for block ciphers do not cover our use-case.

Additionally, in our design, the mix node checks whether the set is complete each time after receiving a fragment. While this avoids leaking the exact number of fragments to the mix node in case of incomplete fragment sets, it also increases the work for the final hop. Scylla can be altered slightly such that the index also contains the total number of fragments in a set, which allows the final hop to avoid looking at incomplete sets – at the cost of potentially allowing correlation by fragment set sizes.

Extension. In Loopix, so-called "loop messages" are used to alert senders to active attacks on their messages. Nym has announced that it will implement this feature, but has not yet done so. Since we cannot know the exact specifications of loop messages in Nym, we analyzed the Fragging attack for Nym without loop messages.

We argue here how the addition of loop messages impacts Fragging in Nym: At the sender, the adversary cannot distinguish packets containing loop messages from real ones. We distinguish between three cases:

- In the first case, the adversary *only* drops packets containing loop messages. The sender can detect that an active attack on their traffic likely occurred, but the adversary gains no information, since all fragments reach the receiver.
- In the second case, the adversary drops packets containing both loop messages and real messages. The sender can detect that an active attack on their traffic likely occurred. However, since the real packets have already been sent, the adversary can execute the Fragging attack with the same success rate as described in Section 4.1.
- In the third case, the adversary *only* drops packets containing real messages. Since all loop messages remain unharmed, the sender cannot detect the attack. The adversary can execute the Fragging attack with the same success rate as described in Section 4.1.

The likelihood of each case occurring depends on the frequency with which loop messages are sent compared to real messages. Further, benign packet loss will cause false-positives, which the client cannot distinguish from active attacks.

Adaptation in Nym. In the following paragraphs, we briefly want to discuss how Scylla could be integrated into Nym as it is currently deployed, and what the impact of such an integration would be:

First, Nym would have to change its mix format from Sphinx to Scylla. As Sphinx and Scylla have a similar structure, there will be no issues with regards to the embedding of delays or addresses. The downside of this change is the slight drop in performance, as evaluated in Section 7.

Further, message reassembly has to be moved from the recipient to the gateway. Only then will Scylla prevent the gateway from learning the recipient if fragments are missing. This will increase memory consumption for those gateways, as they have to retain fragments of non-completed messages. To mitigate this, old fragments have to be evicted regularly (e.g., based on timeouts).

Then, the retransmission logic must be changed to work on complete messages instead of single fragments. This – in combination with Scylla – will prevent the attack described in Section 4.1; however, it will increase the message latencies as shown in Section A.

Finally, in order to ensure that messages are not split into too many fragments (see Section A), the message size must be adjusted to match expected message sizes.

10 Related work

Already in Mixminion [11], support for fragmented messages has been mentioned. The authors recognize the danger of sending multiple messages and the ability of an adversary to embed a pattern via timings or message drops. In their protocol specification,[11] they expand on this approach by implementing fragmentation using a k-of-n threshold mechanism, where the message can be recovered once k out of n fragments have arrived. This approach does not prevent Fragging, as the adversary can embed a signal by dropping some of the $n - k$ remaining packets.

Later, Ando, Lysyanskaya and Upfal consider onion routing over unreliable networks [2]. While they do not focus on a Fragging scenario, they recognize that an adversary can drop onions and use this to confirm sender-recipient pairs (a result that has later been called the *dropping bound* [19]). They propose a solution to make message delivery fault-tolerant, but the client has to send a large number of onions carrying the same payload, therefore increasing the overhead by a large factor.

In Tor, there are multiple ways to "tag" a circuit [22, 28], for example by dropping all future cells on it, or by injecting cells that are quietly dropped by a client [24]. This allows an adversary to confirm a sender/recipient pair (e.g., by destroying the circuit and observing this), or to select possible candidates (e.g., by recognizing the injected cells based on their timing). The Tor project continuously works on improvements to mitigate such attacks.[12]

Danezis and Käsper analyze successors of Crowds and Horde [13]. While unrelated to mix networks, they find that composing anonymous channels usually doesn't provide the same guarantees as the underlying channels: Information leakage from the different layers can help in deanonymizing users, similar to how information leakage from message fragmentation can break a mix network's security guarantees.

11 Conclusion

Mix networks were conceived as a mechanism to deliver electronic mail. However, recent systems have used them as a general purpose anonymous transport layer, building other protocols on top. In this paper, we show that such an approach can introduce new attack vectors, diminishing the anonymity that mix networks guarantee.

We identify the fact that larger payloads are commonly fragmented for delivery in ACS packets as a possible vulnerability that can be exploited by a colluding access network or first mix with the last mix of a cascade in a common mix network, like Nym. We verify empirically that the Fragging attack works in deployed systems, and analyze under which conditions a system is susceptible to Fragging. We show that Fragging is a follow-up to previous results in mix networks [2, 19], and it matches similar attacks in low-latency ACS like Tor [28].

To mitigate the attack, we present a new practical and secure mix format that does allow for fragmentation, and we provide a formal security definition as well as security proofs for it. Further, we provide a prototype implementation and measure its performance, showing that there is a negligible cost to the improved security it provides.

Finally, we note that mechanisms such as retransmissions are not enough to solve this issue: The adversary still sees a change in time it took to complete the fragment set. Instead, such mechanisms need to be adjusted to consider whole messages as well, retransmitting all fragments under a fresh identifier.

Acknowledgments

We thank Philip Höbler for pioneering this work in his Master's thesis. We thank Kristina Prpoš for her work on the Nym testbed. We thank Saskia Bayreuther and Christiane Weis for their help in the UC modeling part. We thank the anonymous reviewers for their helpful feedback and their engagement with our paper.

This work was funded by the Topic Engineering Secure Systems of the Helmholtz Association (HGF) and supported by the KASTEL Security Research Labs, the Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (EXC 2050/2 CeTI – DFG Project ID 390696704).

The authors used DeepL to revise the text and to correct any typos, grammatical errors, and awkward phrasing.

Open science

Our prototype implementation, as well as our testbed configuration, are open-source and publicly accessible at https://github.com/kit-ps/fragging.

Ethical considerations

We have carefully considered the ethical implications of our work.

Human subjects and informed consent. No data from human subjects has been collected or processed by us. All data that we have processed is synthetic and has been generated in isolated processes.

Interference with live systems. Our Nym testbed runs completely isolated from Nym's main network and other Nym components. The testbed only communicates internally, without contacting Nym services or publicly advertising itself as part of the network. No user traffic from outside the testbed is routed through the testbed.

The benchmark suite runs offline and builds packets with fixed, synthetic input.

The only time when third party-services are contacted is during the build process, as dependencies are downloaded using Rust's package manager `cargo`, as well as when downloading the Nym repository from GitHub. These requests constitute a normal use of the provided services.

Responsible disclosure. We acknowledge that the attack presented in this paper may harm users of Nym who rely on its anonymization capabilities. However, we also believe that a proper understanding of the limitations of such networks is important, and potential vulnerabilities should not be obscured but rather understood and fixed. To minimize the harm to Nym's users, on August 14, 2025, we have informed the Nym team about the results of this paper

[11] https://www.mixminion.net/E2E-spec.txt

[12] See https://spec.torproject.org/proposals/344-protocol-info-leaks.html and Mathewson et al. [20]

according to their security policy.[13] At the time of submission, we have not received an answer yet.

Area of research. The research of anonymous communication methods is a double-edged sword: On one hand, strong anonymity can encourage nefarious actions and protect offenders, for example in establishing drug markets or distributing illicit material. On the other hand, strong anonymity helps democratic causes such as whistleblowing or access to information and news, especially for people living under repressive governments.

We believe that, from an utilitarian perspective, the benefits of anonymous communication outweigh the harms. There have been many cases where tools like Tor have helped whistleblowers (e.g., Chelsea Manning, Edward Snowden), journalists, or dissidents (e.g., in Hong-Kong or Iran) to protect themselves during their work. Further, and more abstractly, privacy is regarded as a fundamental right in the United Nations Declaration of Human Rights, the International Covenant on Civil and Political Rights, and the European Convention on Human Rights. While this right to privacy is not absolute, we see the construction of tools that aid in protecting this right as ethically correct.

Accordance with the law. During our research, we did not break the law. In particular, our research does not constitute computer sabotage. Further, the code that we have used in our artifact is licensed under licenses which allow modifications and redistributions (Apache, MIT, GPL).

References

[1] Ross Anderson and Eli Biham. 1996. Two Practical and Provably Secure Block Ciphers: BEAR and LION. In *Fast Software Encryption*.

[2] Megumi Ando, Anna Lysyanskaya, and Eli Upfal. 2021. On the Complexity of Anonymous Communication Through Public Networks. In *ITC*.

[3] Megumi Ando, Anna Lysyanskaya, and Eli Upfal. 2024. Bruisable Onions: Anonymous Communication in the Asynchronous Model. *Theory of Cryptography* (2024).

[4] Mihir Bellare and Phillip Rogaway. 1994. Optimal Asymmetric Encryption. In *EUROCRYPT*.

[5] Victor Boyko. 1999. On the Security Properties of OAEP as an All-or-Nothing Transform. In *CRYPTO*.

[6] Jan Camenisch and Anna Lysyanskaya. 2005. A Formal Treatment of Onion Routing. In *CRYPTO*.

[7] R. Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 136–145. https://doi.org/10.1109/SFCS.2001.959888

[8] Ran Canetti and Marc Fischlin. 2001. Universally Composable Commitments. In *CRYPTO*.

[9] David L. Chaum. 1981. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM* (1981).

[10] Ivan Damgård. 2000. Efficient Concurrent Zero-Knowledge in the Auxiliary String Model. In *EUROCRYPT*.

[11] G. Danezis, R. Dingledine, and N. Mathewson. 2003. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *IEEE SP*.

[12] George Danezis and Ian Goldberg. 2009. Sphinx: A Compact and Provably Secure Mix Format. In *IEEE SP*.

[13] George Danezis and Emilia Käsper. 2013. The Dangers of Composing Anonymous Channels. In *Information Hiding*.

[14] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security*.

[15] Daniel Hugenroth, Martin Kleppmann, and Alastair R. Beresford. 2021. Rollercoaster: An Efficient Group-Multicast Scheme for Mix Networks. In *USENIX Security*.

[16] Rob Jansen and Nicholas Hopper. 2012. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *NDSS*.

[17] Ishan Karunanayake, Nadeem Ahmed, Robert Malaney, Rafiqul Islam, and Sanjay K Jha. 2021. De-anonymisation attacks on tor: A survey. *IEEE Communications Surveys & Tutorials* (2021).

[18] Christiane Kuhn, Martin Beck, and Thorsten Strufe. 2020. Breaking and (Partially) Fixing Provably Secure Onion Routing. In *IEEE SP*.

[19] Christiane Kuhn, Friederike Kitzing, and Thorsten Strufe. 2020. SoK on Performance Bounds in Anonymous Communication. In *WPES*.

[20] Nick Mathewson and Mike Perry. 2018. Towards Side Channel Analysis of Datagram Tor vs Current Tor. https://research.torproject.org/techreports/side-channel-analysis-2018-11-27.pdf

[21] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. 2017. The Loopix Anonymity System. In *USENIX Security*.

[22] R. Pries, W. Yu, X. Fu, and W. Zhao. 2008. A New Replay Attack Against Anonymous Communication Networks. In *IEEE ICC*.

[23] Ronald L. Rivest. 1997. All-or-Nothing Encryption and the Package Transform. In *FSE*.

[24] Florentin Rochet and Olivier Pereira. 2018. Dropping on the Edge: Flexibility and Traffic Confirmation in Onion Routing Protocols. *PETS* (2018).

[25] Daniel Schadt, Christoph Coijanovic, Christiane Weis, and Thorsten Strufe. 2024. PolySphinx: Extending the Sphinx Mix Format With Better Multicast Support. In *IEEE SP*.

[26] Philip Scherer, Christiane Weis, and Thorsten Strufe. 2024. A Framework for Provably Secure Onion Routing against a Global Adversary. *PETS* (2024).

[27] Philip Scherer, Christiane Weis, and Thorsten Strufe. 2024. Provable Security for the Onion Routing and Mix Network Packet Format Sphinx. *PETS* (2024).

[28] Xinwen Fu and Zhen Ling. 2009. One Cell Is Enough to Break Tor's Anonymity. In *Black Hat Technical Security Conference*.

A Performance of whole-message acknowledgments

In this section, we evaluate how a change from a "per-fragment" retransmission mechanism to a "per-message" retransmission mechanism affects the latency of messages. To do so, we simulate message transmission over 3 hops: A client generates fragmented messages, and each fragment has a chance to get dropped. If not all fragments of a message arrive, the client waits for the retransmission timer, and depending on the strategy, re-transmits either the single fragment, or the whole message.

The simulation does not model network latencies or congestion, but it does model mixing delays with an avarage delay of 50 ms per hop. We choose a retransmission timeout of 1500 ms (parameters chosen to match Nym), and send 1000 messages consisting of 8 fragments each.

The simulation tool records the timestamps of message transmission and arrival, as well as the total number of messages sent.

We expect to see two effects when changing from per-fragment to per-message retransmissions: On one hand, a per-message retransmission strategy will eliminate the variations in time between receiving the first and the last fragment of a message, as all fragments either arrive in time, or all fragments will be retransmitted. On the other hand, the delay between sending a message and it being received will increase, as all fragments of a single transmission attempt must arrive. Under packet loss, this leads to a larger number of retransmissions being required before all fragments "survive."

We show our results in Figure 12. We see that a per-message acknowledgment strategy indeed eliminates the variation in time from receiving the first fragment of a message to receiving the last fragment of a message, even under high packet loss (left graph). This confirms that an adversary who induces packet loss in a client will no longer be able to recognize the tagged message. However, a per-message acknowledgment strategy also increases the time needed to deliver messages significantly, especially for packet loss larger than 10% (middle graph). At 50% packet loss, the median

[13] https://github.com/nymtech/nym/security

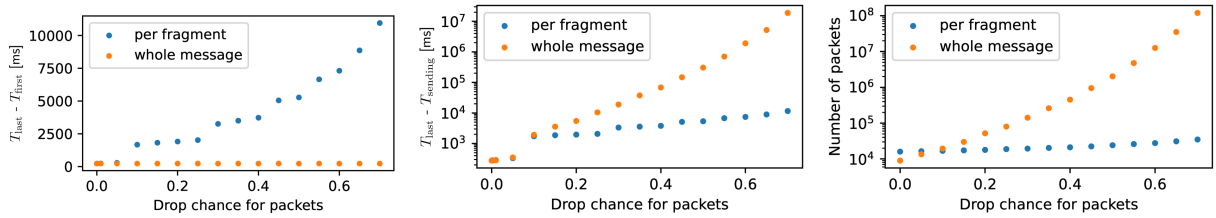

Figure 12: Left: Median time between receiving the first and last fragment of a message. Middle: Median time between sending and receiving a message. Right: Number of packets needed (including acknowledgments) to transmit the message. All settings for a messages with 8 fragments. Note that the middle and right graph have a logarithmic scale on the y-axis.

difference is between 5 s and 305 s. For 70% dropped packets, it is between 12 s and 5 h. For larger packet loss, the simulation did not finish receiving all messages.

As a secondary effect, we can see an impact on the total number of packets needed, and therefore the bandwidth use. For low packet loss, where all message fragments survive in the first transmission, acknowledging them all with a single packet reduces the packet count. However, once messages need to be retransmitted, the packet count increases as *all* fragments of the message will be retransmitted. Therefore, acknowledging complete messages reduces packet counts in scenarios with few dropped packets, but increases them when the packet loss is high. This tradeoff is shown in the right graph in Figure 12.

We note that under normal circumstances, the expected packet loss in the network is low. Systems like Nym use TCP to ensure reliability on the transport layer, and nodes rarely drop messages. Therefore, we repeat our experiment with a fixed packet loss of 1% and a varying fragment count to understand how the size of fragmented messages impacts the network's performance.

As the packet loss is low, we expect both retransmission strategies to perform similarly well. In both cases, a single dropped fragment will lead to a transmission delay of one retransmission cycle. Only once a fragment is dropped multiple times will a second retransmission cycle be needed.

Our results in Figure 13 show this behavior: We see that the per-fragment and per-message strategies have the same median delivery time (middle graph), up to approximately 69 fragments. This is the point where the expected number of needed transmission attempts (which follows a geometric distribution with $p = 0.99^n$, where n is the number of fragments) reaches 2, leading to a jump in median delivery time.

For such a low packet loss, we can further see (Figure 13, right graph) that messages with up to 69 fragments lead to fewer packets being sent. In those cases, the message is expected to arrive on the first transmission attempt, therefore benefiting from the per-message acknowledgment.

In isolation, given the parameters of our simulation, we do not expect that a per-message retransmission strategy will cause large latencies if the packet loss is small. In a real deployment, network latencies, congestion or targeted denial-of-service attacks might affect this behavior negatively.

B UC security

In this section, we define our ideal functionality $\mathcal{F}_{\text{FOR}}$ and describe the security guarantees that it provides. Afterwards, we argue why a protocol that achieves UC security in the standard model is not attainable.

B.1 Ideal functionality

We base our ideal functionality on previous work on UC formalizations for onion routing, namely on Scherer *et al.*'s framework for provably secure onion routing [26] and their related formalization of Sphinx's security [27]. In their framework, we resemble the S$\bar{\text{T}}\bar{\text{I}}\bar{\text{R}}$ variation:

- We assume a *service model*, as we require the reassembly of the fragments to be done by the final mix node.
- We assume no trust in the recipient, as a trusted recipient would not cooperate with the adversary to notice a missing fragment.
- We assume no hop-by-hop integrity, as the fragging provides the same information disclosure.
- We do not model replies.

We extend their ideal functionality with the notion of message fragments: Instead of a single message, `ProcNewOnion` now takes in multiple fragments belonging to a single message, and a packet is created for each message fragment. Additionally, `ProcNextStep` will not directly deliver a message to its recipient, but rather wait for all fragments to arrive.

The complete ideal functionality is shown in Algorithm 1 and Algorithm 2.

B.2 Impossibility of a realization in the standard model

While the functionality we describe in the previous section models how onion routing with fragmentation should work, we note that this functionality cannot be realized in standard UC: An implementation of $\mathcal{F}_{\text{FOR}}$ could be used to build a two-party commitment scheme, which is not possible in standard UC [8]. In particular, the prover will act as the sender of a message m, and the verifier will act as the final mix node. By splitting m into two fragments, the prover generates a commitment (the first message), which it delivers to the final mix node. To open the commitment, the prover then delivers the second fragment, which allows the verifier to reconstruct m.

Algorithm 1: Ideal Functionality $\mathcal{F}_{\text{FOR}}$ I

Data structures:
L_f: List of fragments leaked to the adversary

On message `ProcNewOnion`$(R,\ \mathcal{M},\ \mathfrak{P},\ P_E)$
- `// R: Recipient`
- `// `$\mathcal{M}$`: Set of fragments `m_f
- `// `$\mathfrak{P}$`: Set of paths `$\mathcal{P}_f$
- `// `P_E`: Exit node`
- **if** $|\mathcal{M}| \neq |\mathfrak{P}|$ **or** *any* $|\mathcal{P}_i| > N - 1$ **then**
  - **reject**;
- **else**
  - oid $\leftarrow^{\text{R}}$ onion ID;
  - $k \leftarrow |M|$;
  - **for** f *with* $1 \leq f \leq k$ **do**
    - $O_f \leftarrow (\text{oid}, P_s, R, f, k, m_f, \{\mathcal{P}_f, P_E\}, 0)$;
    - `OutCorSender`(oid, P_s, R, f, k, m_f, $\{\mathcal{P}_f, P_E\}$, `'start'`);
    - `ProcNextStep`(O_f);

On message `ForwardOnion`(tid′)
- **if** (tid, _) $\in B_i$ **then**
  - Pop (tid′, O) from B_i;
  - `ProcNextStep`(O);

On message `DeliverOnion`(tid)
- **if** (tid, _, _) $\in L_O$ **then**
  - (tid, $O = (\text{oid}, P_s, R, f, k, m_f, \mathcal{P}, i), j) \leftarrow L_O$;
  - `// We update the onion with its new position `j
  - $O \leftarrow (\text{oid}, P_s, f, k, m_f, \mathcal{P}, j)$;
  - tid′ $\leftarrow^{R}$ temporary ID;
  - `Send`(P_{o_j}, "tid′ received from $P_{o_{j-1}}$");
  - Store (tid′, O) in B_{o_j};

Procedure `OutCorSender`(oid, P_s, R, f, k, m_f, $\mathcal{P}$, tid)
- **if** $P_s \in$ Bad **then**
  - `Send`($\mathcal{S}$, "tid is from P_s with oid, R, f, k, m_f, $\mathcal{P}$");

Procedure `ProcNextStep`(O)
- $(\text{oid}, P_S, R, f, k, m_f, \mathcal{P}, i) \leftarrow O$;
- **if** $\forall j > i : P_{o_j} \in$ Bad **or** $i = |\mathcal{P}|$ **then**
  - **if** $O \in L_{\text{tag}}$ **then**
    - `OutCorSender`(oid, P_s, R, f, k, m_f, $\mathcal{P}$, `'tagged'`);
    - **if** $i < n$ **then**
      - `Send`($\mathcal{S}$, "P_{o_i} sends tagged via $(P_{o_{i+1}}, \ldots, P_{o_n})$");
      - `Send`($\mathcal{Z}$, "P_{o_i} sends onion to $P_{o_{i+1}}$");
    - **else**
      - `Send`($\mathcal{Z}$, "Onion at P_{o_i} fails integrity check");
  - **else**
    - `LeakFragment`(O);
    - `LeakMessage`(O);
- **else**
  - `ProcToRelay`(O);

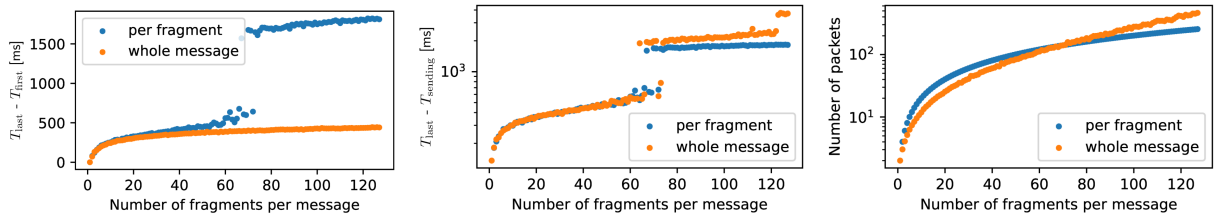

Figure 13: Left: Median time between receiving the first and last fragment of a message. Middle: Median time between sending and receiving a message. Right: Number of packets needed (including acknowledgments) to transmit the message. All settings for a 1% drop chance. Note that the middle and right graph have a logarithmic scale on the y-axis.

Algorithm 2: Ideal Functionality $\mathcal{F}_{\text{FOR}}$ II

Procedure `LeakFragment`(O)
- $(\text{oid}, P_s, R, f, k, m_f, \mathcal{P}, i) \leftarrow O$;
- **if** $\forall j > i : P_{o_j} \in \text{Bad}$ **then**
  - Send$(\mathcal{S}$, "P_{o_i} received fragment f for oid");
- Add O to L_f;

On message `Tag`(tid)
- **if** $(\text{tid}, \_, \_) \in L_o$ **then**
  - Retrieve $(\text{tid}, O, \_)$ from L_o;
  - Store O in L_{tag};

Procedure `ProcToRelay`$(O = (\text{oid}, P_s, R, f, k, m_f, \mathcal{P}, i))$
- $P_{o_j} \leftarrow P_{o_k}$ with smallest $k > i$ such that $P_{o_k} \notin$ Bad;
- tid $\leftarrow^R$ temporary ID;
- Send$(\mathcal{S}$, "P_{o_i} sends tid to P_{o_j} via $(P_{o_{i+1}}, \ldots, P_{o_{j-1}})$");
- Send$(P_{o_j}$, "Sent onion to $P_{o_{i+1}}$");
- `OutCorSender`$(\text{oid}, P_s, R, f, k, m_f, \mathcal{P}, \text{tid})$;
- Add (tid, O, j) to L_o;

Procedure `LeakMessage`$(O = (\text{oid}, P_s, R, f, k, m_f, \mathcal{P}, i))$
- $F \leftarrow \{(\text{oid}', \_, \_, m_f, \_, \_) \in L_f | \text{oid}' = \text{oid}\}$;
- **if** $|F| = k$ **then**
  - Send$(\mathcal{S}$, "P_{o_i} received fragments $m_1, \ldots, m_k$ for R");

Procedure `DeliverMessage`(P_{o_i}, m, R)
- Send$(R$, "Message m received from P_{o_i}");

C Security proofs

Theorem 3. *Scylla satisties FLU under the Gap-Diffie-Hellman Assumption.*

Proof. Our proof is an adoption of Sphinx's security proof [27, G.1], but with a different payload. We recite a (shortened) version of this proof here, and highlight our changes in blue. We only consider the case of $j = n$, as *FLU* only concerns security with an honest final hop.

Hybrid $\mathfrak{H}_0$: This hybrid is the $b = 0$ scenario of FLU.

Hybrid $\mathfrak{H}_1$: In this hybrid, we replace the secrets used at the final hop with randomness. The outputs of the random oracles $h_{\mu,\rho\pi}$ is replaced with random bitstrings $\{0,1\}^\kappa$. When **Proc** is used to process onions, it uses those bitstrings for the challenge onions to stay consistent in behavior.

$\mathfrak{H}_0 \approx \mathfrak{H}_1$: The difference between these hybrids reduces to the security of the Sphinx key encapsulation Sphinx-KEM-IND-CCA.

Hybrid $\mathfrak{H}_2$: In this hybrid, we instruct the **Proc** oracle to ignore any onion with $\alpha_{1,n-1}, \ldots, \alpha_{f,n-1}$, unless the rest of the header also matches exactly.

$\mathfrak{H}_1 \approx \mathfrak{H}_2$: The difference between these hybrids reduces to the security of the MAC, as $\gamma_{i;n-1}$ needs to contain a valid MAC for $\beta_{i;n-1}$. Submitting a valid MAC without re-using $\gamma_{i;n-1}$ and $\beta_{i;n-1}$ only happens in an negligible amount of cases.

Hybrid $\mathfrak{H}_3$: In this hybrid, we swap π and π^{-1} with a random permutation.

$\mathfrak{H}_2 \approx \mathfrak{H}_3$: The difference between these hybrids reduces to the PRP security of π.

Hybrid $\mathfrak{H}_4$: In this hybrid, **Proc** only replies to onions that have exactly the expected payload $\delta_{i;n-1}$.

$\mathfrak{H}_3 \approx \mathfrak{H}_4$: The message fragments are prepended with κ zero-bits and transformed using an AONT. In $\mathfrak{H}_3$, the oracle verifies the correctness of those zero-bits after reversing the AONT transformation.

As π is a random permutation, guessing payloads $\delta'_i \neq \delta_{i;n-1}$ which still produce the correct zero-bits after passing through π^{-1} and having the AONT reversed only happens with a negligible probability. Therefore, the chance to reject an onion that was previously accepted is negligible.

Hybrid $\mathfrak{H}_5$: In this hybrid, we replace the content of the onions with the content as they would be in the case of $b = 1$. The original contents are $A_1, \ldots, A_f = \text{OAEP}(M_1, \ldots, M_f$, where $M_1 = 0^\kappa||R||m_1$ and $M_i = m_i$ for $i > 1$. The replacements are $\bar{A}_1, \ldots, \bar{A}_f = \text{OAEP}(\bar{M}_1, \ldots, \bar{M}_f$, where $\bar{M}_1 = 0^\kappa||\bar{R}||\bar{m}_1$ and $\bar{M}_i = \bar{m}_i$ for $i > 1$, where $\bar{R}$ is a random recipient and $\bar{m}_i$ are random message fragments. The **Proc** oracle uses the original content when processing onions.

$\mathfrak{H}_4 \approx \mathfrak{H}_5$: The oracle behaves consistently between $\mathfrak{H}_4$ and $\mathfrak{H}_5$. The replacement of the payload reduces to indistinguishability security of the underlying random permutation.

We do not consider the case where a relay different than the final one ($j \neq n$) is honest, as this case is covered by previous onion properties. □