Generating trusted sphinx packets

Anonymous

Abstract. We propose a decentralized scheme that prevent mixnets users from sending traffic that does not match the service provider of an anonymous credential. Our scheme is of direct use in the Nym network where users construct an anonymous credential given they receive a certificate of them paying for that service provider. Our scheme prevent users from cheating by using a credential for a free service and send traffic for a paid service. Our solution works even if the majority of the third parties collude. Finally we evaluate the performance of our solution.

Keywords: mixnet · sphinx · malicious users

1 Introduction

Mix network, or mixnet, is an overlay network of servers (called mix nodes) that prevent an adversary from correlating senders with receivers [3,4,6,8,11,14,16,17]. They achieve this by adding delays to messages inside the different mixnodes in order to mitigate timing analysis attacks. Additionally, unlike Tor [10] where all packets sent by a user follow the same path for the entire session (called circuit-based), in mixnets routing is typically packet-based (meaning that each packet follows a different path). These two techniques ensure that mixnets are resilient against a strong adversary who observes the entire input and outputs of the network typically called a Global Passive Adversary (GPA). During the last few decades several systems have been proposed in the literature and few of them have been implemented. Open research problems such as parametrization, authentication and fake dummies have halted the usage on a large scale. Recently, Nym Technologies a mixnet-based company is being commercialized and proposing a mixnet network for services to be integrated with their network for a fee. Their network is based on Loopix [17]. Users of these services are then allowed to use the nym mixnet by using Nym credentials (based on the Coconut credential [18]) that they can construct after getting a certificate of paying for a specific service to use the Nym network. However, nothing prevents users from cheating who might exploit a valid Nym credential for another service they did not pay for. This is a particular difficult problem in anonymous communication networks where the mixnodes do not know the traffic type or the final service a user is communicating with by doing layered encryption where the final IP address is only know by the last node in the path using a packer format such as Sphinx [7].

In this paper we present a scheme that creates the Sphinx header in a decentralized way based on trusted third parties while ensuring they learn nothing about the destination or the path. We aim to provide the following properties:

Aurelien: We could be more concise on the 'properties' in the introduction and reintroduced them in the threat model section

- Cheating users: Users's traffic is only allowed to be routed if the traffic belongs to the same service provider from the credential
- even if the majority of headers issuers are colluding, they do not know which service provider the user is communicating with.
- the spinx headers can not be altered.
- Verifiers can verify that the headers has not been altered without revealing the service provider.
- Unlinkability between sphinx packets, the original sphinx packet that is constructed in a centralized way provide the *unlinkability* property, meaning that an adversary can not know that two packets are connected to the same user. Our scheme that decentralize the headers creation aim at providing this same property.

We highlight related work and motivation in Section 2, then we specify and justify our system model in Section 3, where we also describe the considered threat model. We then present our scheme that decentralize the creation of the Sphinx headers in Section 5 and the evaluation of our proposed solution in Section 6. Finally we conclude and discuss future work in Section 7

2 Motivation and Related Work

Since Chaum's seminal work on untraceable email in 1981 [3], there has been a great amount of research related to mixnets' design [1,3–6,11–13,17]. However most systems that have been deployed and used come with their own system on top of the network meaning that only one type of traffic type is allowed. As beautifully stated by Dingledine et al. in [9], "Anonymity Loves Company" meaning that the more messages there are in the network the more privacy the network provide. This is also shared by Ben Guirat et al. in [2] where the authors show that blending different traffic types on top of a mixnet provide better anonymity, meaning let's imagine an instant messaging system where users do not tolerate latency of more than few seconds and an email app where users tolerate latency of up to 1 minute. The authors show that blending these two types of traffic do actually increase the privacy for both traffic. This only applicable in networks such as Tor or the Nym network that they offer the network for different applications to be integrated on top of the network rather than dictating which application to use the mixnet. However, certain open research problems remain open. For example how can we ensure that certain traffic are not allowed (for whatever reason and we will specify the exact reason for our work) without compromising ? Tor solves the problem with having exit policy that simply drop traffic at the last node. However Tor routing is circuit-based meaning that all traffic from a specefic user session follow one path and hence packets are encrypted using symetric cryptographhy which is cheap, so even if traffic is dropped that is not a problem.

However unlike tor, mixnets can be expensive due to the routing type which is packet based , meaning that every packet takes a different path and hence layered encrypted using public key cryptography. This means that if packets are dropped at the last nodes when not allowed, this waste a huge amount of cryptographic power. Additionally unlike Tor where relay are volunteer based, nodes in mixnets such as Nym are economically incentivized based on the bandwidth they routed so if packets are dropped at the end this is not great.

Aurelien: 'this is not great' feel a bit simple... maybe few extra explanations could be nice

3 System and Threat Model

In this section we describe the system model and the adversary we consider.

Iness: add a simple graph here of a mixnet

3.1 System Model

We abstract the Nym mixnet to the components that are relevant for our work. We consider a mixnet, where each user have a gateway that shows a credential and if accepted their traffic is allowed. To prevent correlation, mixnet relies on fixed-size packet format such as Sphinx packet (section **??**), making it difficult for external observers to link incoming and outgoing messages at any given node. The nym credential is constructed by the user and issued by a third decentralized party after obtainng a certificae that proove payement. For example let's say Signal is integrated with Nym, and Signal users who want their traffic to be anonymous instead of sending traffic directly to the Signal server, traffic will be first routed through the mixnet such that an adversary who observes the signal server and/or the device of the user can not correlate the sender with signal server and evetually the final recepient. Signal (service provider) can add an option for user who want to pay and issue a certified attribute to those users. Users then encode this attributes into a credential and sends it to validators. If the proof is valid, validators return partial signatures. Once the user collects a threshold number of these signatures, they aggregate them to form a valid credential and re-randomize it to ensure unlinkability from previous interactions. The user can then present this credential to a verifier to prove their right to access a service to show that the credential meets all necessary payment and authentication conditions. To prevent double-spending, the verifier checks that the credential has not already been used by consulting the blockchain and then commits the credential's serial number to the blockchain upon acceptance. For example, a user can obtain an certification from the Signal service provider, construct a valid credential and then use it to route traffic to another service provider they didn't pay for or simply not allowed (an illegal website). Such misuse would be detected only at the final node of the mixnet preventing the user from accessing another application. However, prior mixnodes would have already wasted computational resources processing an invalid packet. This vulnerability enables Denial of Service (DoS) attack by exhausting mixnodes computational power with illegitimate packets.

Aurelien: I think we could try to be more concise on this subsection because it feels quite heavy (especially the second half speaking of coconut and blockchain). But for the moment, this part is a bit tricky since we haven't really focus on credentials...

Additionally, each encryption layer includes an integrity tag, which prevents tampering and improves the network's resistance against malicious mixnodes and active adversaries.

3.2 Threat Model

Aurelien: I feel like the bullet points are mixing a bit threat model (GPA, malicious user and honest-but-curious TTP) with desired properties (collusion resistant, integrity, unlinkability, ...)

We consider different types of adversaries:

- A GPA: an adversary who is able to observe all the inputs and outputs of the network. This adversary should not be able to correlate an input with an output based on the packet's appearance. This is achieved by the bitwise unlinkability of the sphinx packets.
- When constructing the headers: By using a Trusted Third Party that constructs the headers we need to make sure that even if the majority of the entities collude, no one should know the final destination of the user.
- An adversary who captures the headers is not able to change headers without intervinging with the integrity check and hence mixes are able to know that the integrity check has been tampered with.
- Malicious users: Users can not cheat and create their own headers and putting the final destination different from the one they have the credential for.

Aurelien: 'Intervinging' ?

4 Sphinx

Iness: This entire subsection explains the original sphinx packet format, I think it's too much and can be reduced to 20%. The rest can be eithger be deleted or used on how you construct the decentralized scheme

Sphinx packets consist of a header and an encrypted payload. The header itself contains a *cryptographic element* α (e.g. g^x or an elliptic curve point), *encrypted routing information* β, and an *integrity tag* γ, as illustrated in Figure 1.

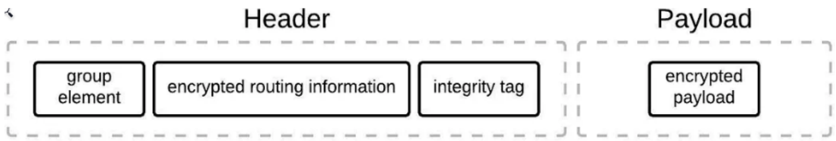

Fig. 1. Structure of sphinx packet. [source]

The *encrypted routing information (β)* is constructed in layers, applied in reverse order along the path. First, the final destination is encrypted, and an integrity tag (γ_i) is computed. The IP address of the last mixnode (n_i) is then prepended. As shown by Figure 2, this process repeats iteratively: each new header is encrypted, an integrity tag (γ_{i-1}) is computed, and the IP address of the preceding mixnode (n_{i-1}) is prepended. This layered encryption ensures that each mixnode can only decrypt its own layer, revealing the next forwarding address while preserving end-to-end confidentiality and protecting against tampering.

To encrypt the routing information, the nym client first chooses a nonce x and compute $\alpha = g^x$ as the *cryptographic element* of the header. Since each mixnode i has a private key x_i and a public key $y_i = g^{x_i}$, the user can create a

previously "user" but JT prefers "nym client"

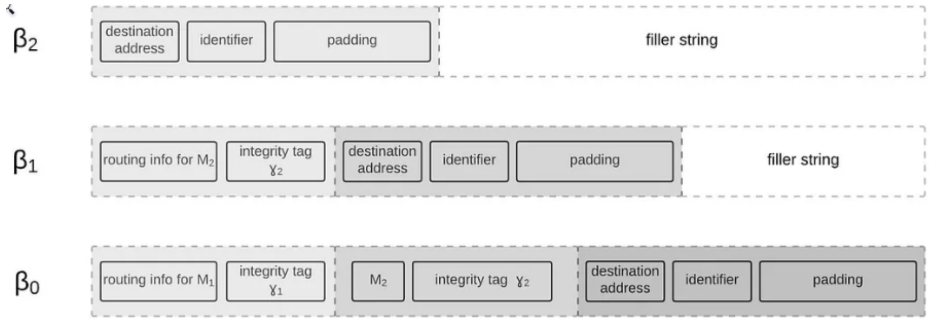

Fig. 2. Sphinx encrypted routing information encapsulation. [source]

shared secret s_i with mixnode i as followed: $s_i = y_i^x = (g^{x_i})^x$. Then the mixnode i receiving the packet will get α allowing him to compute the shared secret as followed: $s_i = \alpha^{x_i} = (g^x)^{x_i}$.

Instead of sending a unique *cryptographic element* α at each node in the path, the sphinx format uses a single *cryptographic element* α, which is progressively modified at each node. Each mixnode updates the cryptographic element using its shared secret as follows:

$$\alpha_{i+1} = \alpha_i^{\text{hash}(\alpha_i, s_i)}$$

Thus, the user iteratively computes the shared secrets in the path's order as:

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

This formulation ensures that each mixnode can independently derive the necessary cryptographic elements without requiring the full path's information, preserving privacy and unlinkability.

The *encrypted routing information (β)* is computed, as illustrated in Figure 2, by processing the path in reverse order. This involves XORing the routing information (β_{i-1}) from the previous layer (with the node's address and integrity tag) with a value derived from the shared secret s_i. Then prepending this new encrypted routing information (β_i) with an integrity tag (γ_i) and the previous mixnode address (remember we build it in reverse order). We repeat the same process for each layer (i.e. each mixnode in the path).

~\ref{}

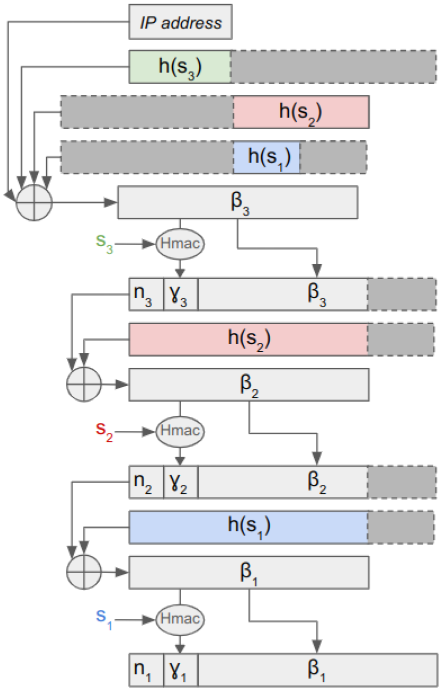

Fig. 3. Construction of the Sphinx header (modified from [7]) [TO FIX: $h(s_1)$ at first XOR]

JT: Explain what the modifications are

JT: Might help to put some (1) (2) (3) labels into the figure and refer to these labels in the text.

The first round of XOR operations differs from the others because it requires combining parts of all shared secrets. Specifically, the destination address is XORed with the last node's shared secret, truncated to match the address size. Next, the result is concatenated with the XORed values of the ending parts of the shared secrets from the other nodes in the path. This ensures that when the entire header is XORed with the full shared secret, these appended values cancel out, allowing the header to be processed in reverse order by the mixnodes. This design choice guarantees fixed-size headers, enabling fixed-size packets which is a crucial property in mixnets for maintaining unlinkability.

JT: Now maybe follow up with an example of the attack from the introduction? We should also compare computational overheads from attack vs. the new protocol design.

5 Our Solution - Multi-Party Computation (MPC)

[NOTE] - To do or improve
- Carefully select variable's notation, e.g. h, h(), H() and hash();
    - Don't like α notation since it should be capital letter (point)
    - Should use $\mathit{\Gamma}$ instead of γ notation (point)
    - And what about β, b, x, ...
- Do or improve schemas;
- Clarify that we work with a path of length 3 (could be generalized)

Our approach to ensure trust in the Sphinx header is to prevent user manipulation by decentralizing the header construction to Trusted Third Parties (TTP) through the use of Multi-Party Computaftion (MPC).

We consider TTPs as *honest-but-curious*. This means that they follow the protocol correctly but may attempt to infer additional information from the data they process. Our design ensures that TTPs cannot infer any information about the shared secrets s_i or the involved mixnodes, even when TTPs collude (except one).

JT: In the Nym ecosystem, who are the TTPs, who operates them, what exactly are they trusted for?

5.1 Development Path / Thinking path

Before introducing our decentralized schema, we briefly outline the key decisions that shaped our implementation.

We considered three main approaches to decentralize the Sphinx header computation:

1. **Distributed:** Each TTP computes a distinct layer (sequential approach);
2. **Partially decentralized:** All TTPs compute a layer based on partial information, aggregate their results to compute the integrity hash, redistribute the outcome, and repeat the process with the next layer;
3. **Fully decentralized:** All TTPs compute the full Sphinx header using partial information, and their results are aggregated at the end.

The first approach compromises privacy, as each TTP learns two consecutive nodes in the path and one of the associated shared secrets, raising serious security concerns. The second approach increases communication overhead and allows the aggregating TTP to observe the intermediate Sphinx header at specific stages, which could enable packet tracking if the aggregator colludes. Finally, the third approach was selected for its stronger privacy guarantees, despite its added complexity.

The main challenge in the fully decentralized setting lies in the integrity tag which relies on a hash function. Although a homomorphic hash would be ideal to support decentralized hash, such construction remains poorly studied and potentially weaken its security due to the homomorphic additive or multiplicative nature (easier for collisions). As a practical workaround, we substitute the hash with a one-way homomorphic encryption scheme (only encryption, not decryption). More specifically, a homomorphic encryption with mutually commutative operators (like RSA and ElGamal but not Paillier) due to the nested structure of the integrity tag, which contains previous tags.

Should I develop why ?

To improve efficiency, we switch from RSA-like primitives to elliptic curve cryptography (ECC), which offers more compact representations while keeping these homomorphic properties. By abstracting away encryption layers, the final header is structured as follows:

[node 1, integrity 1, node 2, integrity 2, node 3, integrity 3, destination]

adding a figure to illustrate ?

Since deriving all these information from just one or two EC points seems impractical, we represent each piece of information by an EC point. To achieve this, we split the encoded string into chunks as represented by Figure 4, each corresponding to a single EC point.

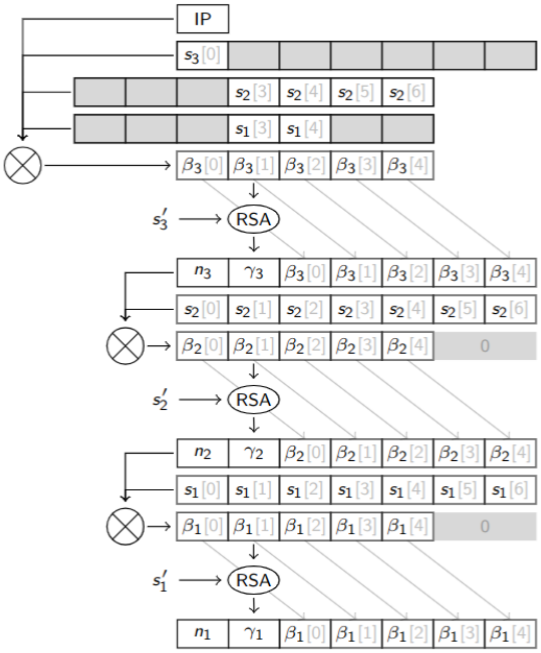
IP
$s_3[0]$
$s_2[3]$ $s_2[4]$ $s_2[5]$ $s_2[6]$
$s_1[3]$ $s_1[4]$
$\beta_3[0]$ $\beta_3[1]$ $\beta_3[2]$ $\beta_3[3]$ $\beta_3[4]$
s'_3
RSA
n_3 γ_3 $\beta_3[0]$ $\beta_3[1]$ $\beta_3[2]$ $\beta_3[3]$ $\beta_3[4]$
$s_2[0]$ $s_2[1]$ $s_2[2]$ $s_2[3]$ $s_2[4]$ $s_2[5]$ $s_2[6]$
$\beta_2[0]$ $\beta_2[1]$ $\beta_2[2]$ $\beta_2[3]$ $\beta_2[4]$ 0
s'_2
RSA
n_2 γ_2 $\beta_2[0]$ $\beta_2[1]$ $\beta_2[2]$ $\beta_2[3]$ $\beta_2[4]$
$s_1[0]$ $s_1[1]$ $s_1[2]$ $s_1[3]$ $s_1[4]$ $s_1[5]$ $s_1[6]$
$\beta_1[0]$ $\beta_1[1]$ $\beta_1[2]$ $\beta_1[3]$ $\beta_1[4]$ 0
s'_1
RSA
n_1 γ_1 $\beta_1[0]$ $\beta_1[1]$ $\beta_1[2]$ $\beta_1[3]$ $\beta_1[4]$

Fig. 4. Chunked representation of the schema

Nice guidelines ref: RFC9380 (Hashing to Elliptic Curves)

One remaining challenge when encoding information into elliptic curve (EC) points is the decoding process. Some type of data such as IP addresses (destination or mixnodes) requires both an encoding method that maps an integer to a point and a corresponding decoding method that retrieves the integer from the point. In this article, we refer to these two processes respectively as *integer-to-point* noted $H()$, and *point-to-integer* noted $h()$.

A classical approach to achieve such a mapping involves the x-coordinate of the point. The point-to-integer function simply returns the x-coordinate of the point. The integer-to-point function computes the y-coordinate where x is the integer. However, a randomly chosen x-coordinate will lie on the curve only about 50% of the time. A simple workaround is to increment the x-coordinate until a valid point is found. Yet, this technique introduces a problem in our context since the x-coordinates correspond to IP addresses. Nearby IPs could map to the same point, potentially leading to collisions that compromise correctness of the mixnet. This risk could be mitigated by padding the IP with a random fixed-length binary string.

From our knowledge, a second alternative exists: the Elligator 2 algorithm. Elligator provides a method to map any integer to a pseudorandom point on a Montgomery curve in a uniform and reversible way. More precisely, the reverse mapping (point-to-integer) requires that the point satisfies specific mathematical conditions, which occur with roughly 50% probability for a random point. Again, a simple workaround is to increment the point by adding the curve's generator until the point fulfills these conditions.

Note that all points derived from integers using Elligator are guaranteed to be reversible. Thus, a point representing an IP address (which originally comes from an integers) can always be correctly decoded. Only random points used as shared secrets may require few extra point additions to meet these conditions which represents a negligeable computational overhead.

To summarize, we need a mapping that is as bijective as possible but we face a trade-off between two imperfect scenarios:

- A complete and easily reversible point-to-integer mapping (i.e. using x-coordinates), but only partial coverage for integer-to-point.
- A complete integer-to-point mapping (i.e. using Elligator), but only partial coverage for point-to-integer.

Elligator seems a better solution since it guarantees stronger security/privacy by providing a more uniformly distributed output. Moreover the complete integer-to-point is more important (to prevent IP collision) and we can deal with partial point-to-integer mapping in our use case.

It feels a bit heavy with too much details and remarks. The whole previous page could be reduced to this shorter alternative. Which one is better ?

[Alternative shorter version]:
One remaining challenge lies in encoding and decoding information to and from elliptic curve points. Specifically, we require a pseudorandom mapping between integers field and curve points group.
Traditional methods achieve this by directly mapping integers to the x-coordinates of points on the curve. However, this approach could leaks information by introducing bias since nearby integers result in nearby points.
To address this, we adopt Elligator, which offers stronger privacy guarantees. Unlike the traditional approach, Elligator produces a uniformly distributed output which is computationally indistinguishable from truly random curve points. This uniformity is critical in our context, where preserving anonymity and avoiding linkability are core goals.

5.2 Protocol description

The overall decentralized scheme is illustrated in Figure 5. The client first computes a sequence of shared secrets and then splits these secrets, along with the IP addresses of the mixnodes in the path and the final destination, into m shares. Each set of shares is sent to a different TTP, along with the necessary cryptographic element (α). Each TTP independently computes a *partial* Sphinx header using the received shares. The client then aggregates these partial headers to reconstruct the final header, ready for transmission through the mixnet.

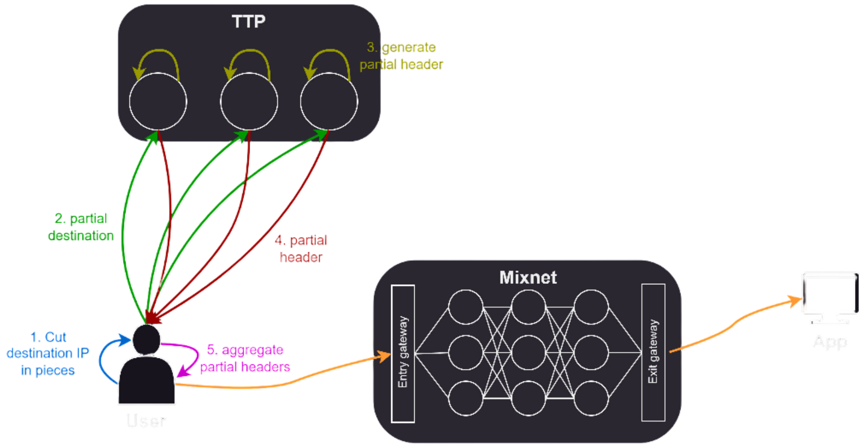
TTP
3. generate partial header
2. partial destination
4. partial header
Mixnet
Entry gateway
Exit gateway
App
1. Cut destination IP in pieces
5. aggregate partial headers
User

Fig. 5. Overview of the decentralized scheme

In summary, the protocol is divided into four main steps:

- **Setup**: fixes random points as *generators* (once but should be refreshed);
- **Client**: *splits and shares* the necessary information to TTPs;
- **TTP**: *encrypts* the routing information into a partial header;
- **Mixnode**: *decrypts* and forwards the header.

Setup In our protocol, routing information is divided into seven chunks (see Figure 4), each one encoded into an elliptic curve point. These points will later be encrypted by adding a masking point of the form $P = sG$, where s is a shared secret and G is the base generator of the curve. However, using the same masking point P for all the seven chunks compromises the *unlinkability* property (see security note in the TTP step 5.2).

To mitigate this, we use a set of seven *independent* generators G_i, one for each chunk. By *independent*, we mean that the scalar relationship between any two generators is unknown. Therefore no one knows the scalars $x_i \in \mathbb{Z}_N$ such that $G_i = x_i G$.

In principle, this setup phase can be executed only once. However, to preserve unlinkability, the set of generators should be refreshed periodically. Every entity (clients, mixnodes, and TTPs) must run the setup phase to ensure they use the same set of generators.

To achieve synchronized and deterministic generator values across the system, we derive them from a common seed. This seed is computed as the hash of the current timestamp, truncated to the chosen refresh interval. To produce distinct seeds for each generator we rehash the seed, or simply increment it since Elligator already provides a uniform (i.e. pseudorandom) integers to points mapping.

In the current implementation, instead of incrementing the seed, we rehash it for each i. However, since Elligator produces uniformly distributed outputs, incrementing may be equally secure and more efficient.

A subtlety with Elligator is that it can map integers to points outside of the base generator's subgroup. Thus, an extra step is required to *clear the cofactor* by multiplying the resulting point by the cofactor h (for Curve25519, $h = 8$). The final formula for generating the i^{th} chunk's generator is:

$$G_i = h \cdot H(\text{hash}(\text{timestamp}) + i), \quad \text{for } i = 1, \dots, 7$$

Client To send a message to a specific destination, the client first randomly chooses a path through the mixnet and generates a nonce. Using this nonce and the public keys of each mixnode along the path, the client derives a chain of shared secret integers (s_1, s_2, s_3) as follows:

Does it seems strange to transform the shared secret point into integer ? Should I explain why we need integer form ? It might be a bit heavy and add too much details...

$$\begin{aligned}
\alpha_i &= x_i\, G \\
S_i &= x_i\, \text{PK}_i \\
s_i &= h(S_i) \\
b_i &= \text{hash}(\alpha_{iy} \parallel S_{iy}) \\
x_{i+1} &= x_i\, b_i \pmod N
\end{aligned}$$

Order of equation lines ? and names of variables (espacially x_i and b_i) ?

where x_1 is the client's nonce, PK_i is the public key of the i^{th} mixnode in the path, N is the prime order of the elliptic curve group, $\|$ is the concatenation operator and subscript y refers to the y-coordinate of the point. The scalar b_i acts as a pseudo-randomizer to update the secret x_i for the next hop, ensuring that each cryptographic element α_i remains unlinkable across mixnodes.

Test code: to verify if the padding is still necessary or optionnal and if limited (max size N)

The IP addresses of the mixnodes and the destination are padded with random bits then mapped to elliptic curve points using Elligator, allowing later recovery of these addresses. Both the resulting IP points and the shared secrets are split into shares and distributed to m different TTPs.

Should I mention how to split point (or integer) into m shares ? It is just taking $m-1$ random points (or integers), sum them and compute the difference with the original point (or integer).

Each TTP then independently computes a partial header from the received shares and returns it to the client. The client aggregates these partial headers by performing chunkwise point additions to reconstruct the complete encrypted header. Finally, the client sends the header along with α_1 to the first mixnode from the path.

Do we need to remind that the user need to encrypt his message using these shared secrets (as in original article) or is uninteresting/obvious enough ?

TTP Each TTP receives from the client:

- a share of the destination address (as an elliptic curve point),
- a share of each mixnode address in the path (as elliptic curve points),
- a share of each shared secrets (as integers).

The TTP constructs a partial encrypted header by layering routing information in reverse path order, starting from the last mixnode.

At each layer, two new chunks are introduced: the next mixnode's address (as an EC point) and an integrity tag. To preserve a fixed-length header while allowing mixnodes to reverse the operation (as in the original design), four additional *filler chunks* are appended. These filler chunks are carefully computed such that at each layer, after applying the corresponding masking points ($s_i\ G_j$), the last two chunks cancel out and become identity point (i.e. point at infinity). This mechanism allows these two chunks to be safely truncated at each layer, ensuring a consistent header size while preserving reversibility for mixnodes.

To initialize the header, the TTP encrypts the destination point using the shared secret of the last mixnode and append the required filler chunks. Let Δ be the partial destination point, s_i the shared secret corresponding to the i^{th} mixnode in the path, and G_j the j^{th} chunk's generator. The chunks for the last layer (layer 3, assuming a 3-hop path) are computed as follows:

$$\begin{aligned}
\beta_{31} &= \Delta + s_3\ G_1 \\
\beta_{32} &= -(s_2\ G_4 + s_1\ G_6) \\
\beta_{33} &= -(s_2\ G_5 + s_1\ G_7) \\
\beta_{34} &= -s_2\ G_6 \\
\beta_{35} &= -s_2\ G_7
\end{aligned}$$

The integrity tag for this layer is computed as:

$$\gamma_3 = s_3\ G + \sum_{j=1}^{5} \beta_{3_j}$$

Subsequent layers (for $i = 2$ and $i = 1$) are computed as:

$$\begin{aligned}
\beta_{i_1} &= N_{i+1} + s_i\ G_1 \\
\beta_{i_2} &= \gamma_{i+1} + s_i\ G_2 \\
\beta_{i_3} &= \beta_{i+1_1} + s_i\ G_3 \\
\beta_{i_4} &= \beta_{i+1_2} + s_i\ G_4 \\
\beta_{i_5} &= \beta_{i+1_3} + s_i\ G_5
\end{aligned}$$

with the corresponding integrity tag:

$$\gamma_i = s_i\ G + \sum_{j=1}^{5} \beta_{i_j}$$

Here, N_{i+1} represents the elliptic curve point corresponding to the $(i+1)^{\text{th}}$ mixnode in the path.

When the first layer is finally computed, it is appended with the first mixnode's point N_1 and the integrity tag γ_1. Finally, the TTP returns the constructed partial header to the client.

Security Note. To preserve unlinkability, it is essential that the generators $(G_1, \ldots, G_7)$ are *independent*, meaning that their scalar relationships are unknown and cannot be derived. If the same generator G' were used across chunks (or their scalar relationships known), an adversary could compute the chunkwise difference between consecutive layers: $\beta_{i_j} - \beta_{i+1_j} = s_i\ G'$. Since the shared secret s_i remains consistent within a layer, the resulting differences would reveal a predictable pattern (uniform or preserved scalar relationships). This consistency could allow an adversary to correlate incoming and outgoing packets at a mixnode, thereby breaking the unlinkability property. Using independent generators for each chunk prevents such correlations and is therefore critical to the protocol's security.

Mixnode When a packet arrives at a mixnode, it begins by extracting the relevant header fields, namely the cryptographic element α and the integrity tag γ. The mixnode then recomputes the shared secret point S using its private key sk and the cryptographic element α:

$$S = \text{sk}\ \alpha$$

This point is subsequently mapped to a shared secret integer s using Elligator. This shared secret integer is then used to verify the integrity tag:

$$\gamma \stackrel{?}{=} s\ G + \sum_{j=1}^{5} \beta_j$$

The integrity tag is constructed as the sum of the chunks, offset by a secret point derived from the shared secret integer. This ensures that only the client and intended mixnode can create, modify, or verify it.
If the integrity check passes, the mixnode pads the header by appending two *identity* chunks (i.e. point at infinity) and then updates it as:

$$\beta'_j = \beta_j - s\ G_j \qquad \forall j = 1, \ldots, 7$$

The first chunk (β'_1), encodes the next mixnode's IP address. This point is mapped back to an integer using Elligator and the random padding is removed by keeping the last 128 bits (i.e. size of IP address).
The second chunk (β'_2) is the new integrity tag (γ') for the remaining five chunks that form the new encrypted routing information (β').

To maintain unlinkability, the cryptographic element α must be updated for the next hop. As in the client's step, it is computed using the y-coordinates of both α and S:

$$\alpha' = \text{hash}(\alpha_y \parallel S_y)\ \alpha$$

Finally, the mixnode forwards the updated header $(\alpha', \gamma', \beta')$ to the next node.

6 Evaluation

Our prototype implementation is available on GitHub[1]. The implementation is written in Python 3.13 using the ECPy library for elliptic curve operations. ECPy is well-suited for prototyping due to its simplicity, but it is significantly slower than many EC python libraries (approximately one order of magnitude slower). Nevertheless, it remains one of the few Python libraries supporting Montgomery Curve25519, which is essential for our implementation.

We evaluate our system on two main axes: computational complexity and unlinkability (i.e. resistance to correlation between incoming and outgoing packets).

6.1 Computational Complexity

Instead of measuring raw execution time, which can vary significantly based on hardware and software environments, we focus on the number of expensive cryptographic operations. Table 6.1 count these operations per party for m TTPs and path of length p. To compute the total cost for sending one message, the TTP cost should be multiplied by m and the mixnode cost by p.

Since long paths are typically unnecessary, we can approximate $\mathcal{O}(p)$ as constant (i.e. $\mathcal{O}(1)$). A path of length $p = 3$ is generally sufficient to ensure strong anonymity. Consequently, the overall computational burden is more sensitive to the number of TTPs m rather than the path length. Because our design remains secure as long as a single TTP is honest (i.e. does not collude), only a small

Aurelien: Iness do you have some source to support this ? If yes, could you cite here, thanks.

[1] https://github.com/AurelienCha/Decentralized-Sphinx

	Client	**TTP**	**Mixnode**
EC Multiplication	$\mathcal{O}(p\,m)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p)$
EC Addition	$\mathcal{O}(p\,m)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p)$
Point → Integer	$\mathcal{O}(p)$	0	2
Integer → Point	$\mathcal{O}(p)$	0	0

Table 1. Number of expensive cryptographic operations per party, where p is the path length (e.g. $p = 3$), and m is the number of TTPs.

number of TTPs may be required. A deeper analysis of how many TTPs are needed to ensure a desired level of trust remains an open question for future work.

6.2 Unlinkability assessment

Quantifying unlinkability is inherently challenging. However, under the assumption that uniformly random packet headers prevent adversaries from correlating incoming and outgoing packets via cryptanalysis, unlinkability can be approximated by assessing the statistical randomness of the headers.

We rely on the NIST SP 800-22 statistical test suite [15], which includes 15 tests designed to assess the quality of random number generators. Each test returns a p-value representing how likely the data could be produced by a uniform source. If the outputs are truly random, the distribution of p-values across many samples should itself be uniform.

Add a brief description of these tests (i.e. what is evaluating) -> no, too long (or for appendix)

For our prototype, we simulate a network of 20 mixnodes and 3 TTPs. At each iteration:

TODO: For the moment, only the first 4 of the 15 NIST tests have been implemented.

- A random destination IP and path of 3 mixnodes are selected;
- The client generates shares and distributes them to the three TTPs;
- TTPs independently compute partial headers, which are then aggregated and forwarded through the simulated mixnet.

Each run produces 4 headers (one per hop). We perform 100 000 runs, resulting in a 400 000 x 1792 matrix (400 000 headers of 1792 bits). Then we apply the NIST tests to each:

- header independently (i.e. row-wise) to obtain p-value distributions across headers (Figure 6).
- bit independently (i.e. column-wise) to ensure no bias exists in specific bit positions (Figure 7).

For comparison, we conduct the same evaluation on the original Sphinx implementation by Danezis[2], which produces 400 000 headers of 1634 bits. Figures

[2] https://github.com/UCL-InfoSec/sphinx

6 and 7 compare header randomness of our implementation (in orange) with the original implementation (in blue).

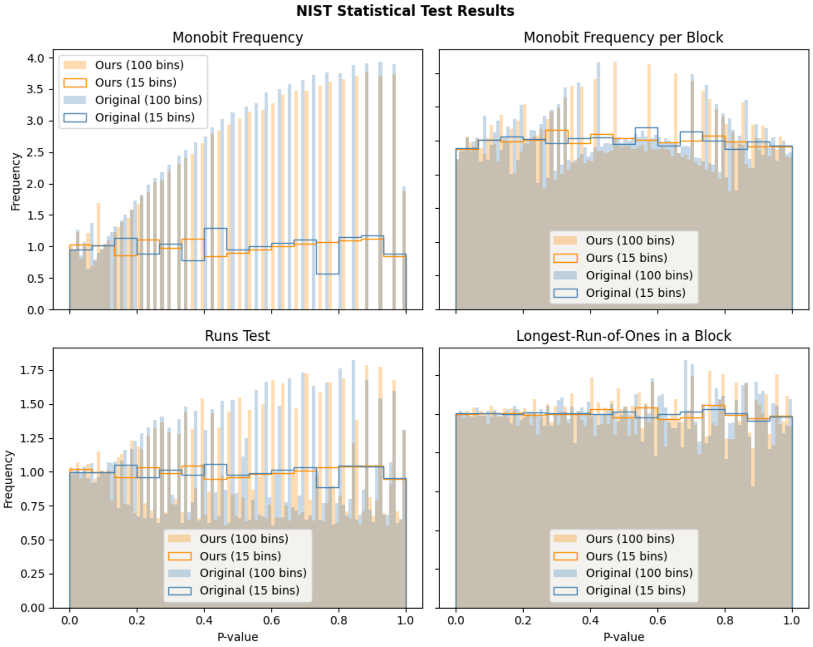

Fig. 6. Distribution of NIST SP 800-22 p-values across 400 000 headers (headerwise evaluation).

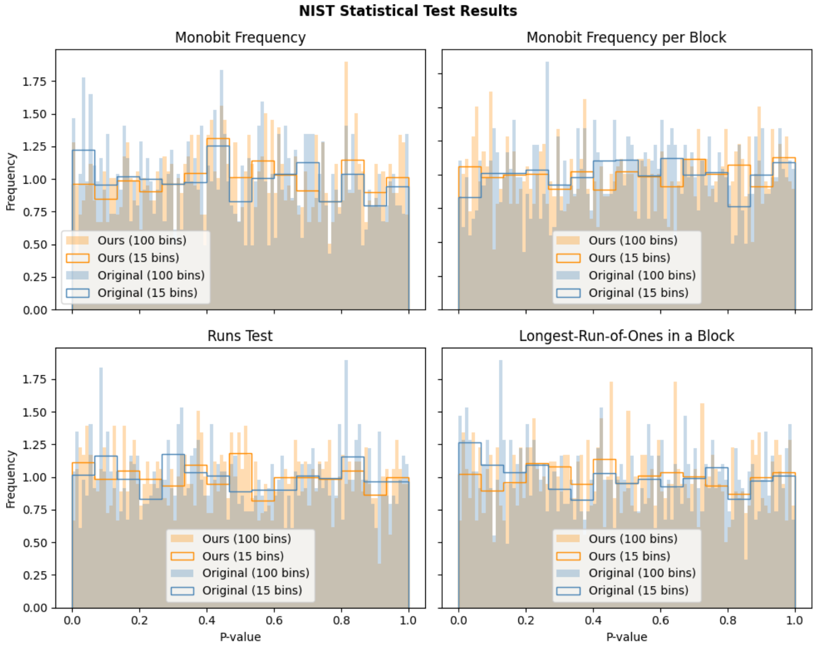

Fig. 7. Distribution of NIST SP 800-22 p-values per bit across 400 000 headers (bitwise evaluation).

Some deviations in the histograms arise from the limited size of the bitstrings (1634 or 1792 bits), which introduces quantization effects. In particular, the slight distortion in the first test of Figure 6 reflects the granularity of possible p-values for short inputs. Variations in Figure 7 are due to the small amount of p-values (1634 or 1792). Increasing the histogram bin size mitigates these artifacts and confirms that the underlying distributions remain approximately uniform.

In both evaluations, the distributions of p-values are approximately uniform, which suggests that headers are statistically indistinguishable from random data, and therefore indicates strong linkability resistance, comparable to the original Sphinx implementation.

7 Conclusion

References

1. Alexopoulos, N., Kiayias, A., Talviste, R., Zacharias, T.: Mcmix: Anonymous messaging via secure multiparty computation. In: 26th {USENIX} Security Symposium ({USENIX} Security 17). pp. 1217–1234 (2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/alexopoulos

2. BEN GUIRAT, I., Das, D., Diaz, C.: Blending different latency traffic with beta mixing. Proceedings on Privacy Enhancing Technologies (2023)
3. Chaum, D.: Untraceable electronic mail, return addresses, and digital pseudonyms. Commun. ACM **24**(2), 84–88 (1981). `https://doi.org/10.1145/358549.358563`, `http://doi.acm.org/10.1145/358549.358563`
4. Chaum, D., Javani, F., Kate, A., Krasnova, A., Ruiter, J., Sherman, A.T., Das, D.: cmix: Anonymization by high-performance scalable mixing. Tech. rep., Technical report (2016), `http://www.cs.bham.ac.uk/~deruitej/papers/cmix.pdf`
5. Cottrell, L.: Mixmaster and remailer attacks (1995), `https://www.obscura.com/~loki/remailer-essay.html`
6. Danezis, G., Dingledine, R., Mathewson, N.: Mixminion: Design of a Type III Anonymous Remailer Protocol. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy. pp. 2–15. IEEE (May 2003)
7. Danezis, G., Goldberg, I.: Sphinx: A compact and provably secure mix format. In: 2009 30th IEEE Symposium on Security and Privacy. pp. 269–282 (2009), `http://research.microsoft.com/en-us/um/people/gdane/papers/sphinx-eprint.pdf`
8. Diaz, C., Halpin, H., Kiayias, A.: The Nym Network. `https://nymtech.net/nym-whitepaper.pdf` (February 2021)
9. Dingledine, R., Mathewson, N.: Anonymity loves company: Usability and the network effect. In: WEIS (2006)
10. Goldschlag, D.M., Reed, M.G., Syverson, P.F.: Hiding routing information. In: Proceedings of the First International Workshop on Information Hiding. p. 137–150. Springer-Verlag, Berlin, Heidelberg (1996)
11. Hooff, J.V.D., Lazar, D., Zaharia, M., Zeldovich, N.: Vuvuzela: Scalable private messaging resistant to traffic analysis. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 137–152 (2015), `https://people.csail.mit.edu/nickolai/papers/vandenhooff-vuvuzela.pdf`
12. Kwon, A., Lu, D., Devadas, S.: {XRD}: Scalable messaging system with cryptographic privacy. In: 17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20). pp. 759–776 (2020)
13. Lazar, D., Gilad, Y., Zeldovich, N.: Karaoke: Distributed private messaging immune to passive traffic analysis. In: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). pp. 711–725 (2018)
14. Möller, U., Cottrell, L., Palfrader, P., Sassaman, L.: Mixmaster Protocol — Version 2. IETF Internet Draft (July 2003)
15. NIST: A statistical test suite for random and pseudorandom number generators for cryptographic applications. Special Publication SP 800-22, National Institute of Standards and Technology (2010). `https://doi.org/10.6028/NIST.SP.800-22r1a`
16. Parekh, S.: Prospects for remailers. First Monday **1** (August 1996)
17. Piotrowska, A.M., Hayes, J., Elahi, T., Meiser, S., Danezis, G.: The loopix anonymity system. In: 26th {USENIX} Security Symposium ({USENIX} Security 17). pp. 1199–1216 (2017)
18. Sonnino, A., Al-Bassam, M., Bano, S., Meiklejohn, S., Danezis, G.: Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019. The Internet Society (February 2019)