Provable Security for the Onion Routing and Mix Network Packet Format Sphinx

Philip Scherer
KIT Karlsruhe
firstname.lastname@student.kit.edu

Christiane Weis
NEC Laboratories Europe
firstname.lastname@neclab.eu

Thorsten Strufe
KIT Karlsruhe
firstname.lastname@kit.edu

ABSTRACT

Onion routing and mix networks are fundamental concepts to provide users with anonymous access to the Internet. Various corresponding solutions rely on the Sphinx packet format. However, flaws in Sphinx's underlying proof strategy were found recently. It is thus currently unclear which guarantees Sphinx actually provides, and, even worse, there is no suitable proof strategy available.

In this paper, we restore the security foundation for all these works by building an analytical framework for Sphinx. We discover that the previously-used Decisional Diffie-Hellman (DDH) assumption is insufficient for a security proof and show that the Gap Diffie-Hellman (GDH) assumption is required instead. We apply it to prove that a slightly adapted version of the Sphinx packet format is secure under the GDH assumption. We are thus, to the best of our knowledge, the first to provide a detailed, in-depth security proof for Sphinx that holds. Our adaptations to Sphinx are necessary, as we demonstrate with an attack on sender privacy that would otherwise be possible in Sphinx's adversary model.

KEYWORDS

Privacy, Anonymity, Provable Security, Onion Routing, Mix Networks, Sphinx

1 INTRODUCTION

The majority of today's Internet traffic discloses private information about its users since the exposed IP addresses serve as identifiers. Onion routing (OR) [16] and mix networks [7] are techniques that address this issue by hiding the users' IP addresses. OR and mix networks are similar in the sense that both use relays together with multiple layers of encryption to hide the link between the sender and the message and receiver. The sender wraps the message in multiple encryption layers, thus creating an "onion" packet. The layers are peeled off one by one while traversing the relays on the onion's path. As a critical distinction, mix networks protect against a global adversary by changing the order of packets at each relay (hence *mixing* the communications). This comes at the cost of introducing additional delays. OR networks avoid these delays, but become vulnerable to global passive adversaries. Proposed works for both OR and mix networks exist in two network models: In the integrated-system model, the receiver acts as the last relay. On the contrary, in the service model, the receiver is unaware of the OR or mix network. The last relay, which is called the exit relay, retrieves and forwards the message to the receiver in plaintext.

To hide the relationship between senders and their messages and receivers, protocol designers aim to make all incoming and outgoing packets at honest relays unlinkable for the adversary. This requires special care when designing the packet formats, as any part of the packet could include linkable information. Tor [14] is an OR scheme that is being broadly applied against local adversaries, but mix networks against global adversaries are being increasingly developed. The most efficient and commonly used mix network packet format is Sphinx [12]. In fact, Sphinx is not only used as a foundation for mix networks [26], but also for OR protocols [8, 9], and even inspired a recent improvement of Tor [20].

Sphinx [12] is built for systems in the service model and assumes an additional party, the nymserver.[1] Senders send onions with reply information to the nymserver. Exit relays send anonymous reply messages via the nymserver, which uses the reply information to build a reply packet. Sphinx packets consist of a header and a payload. The header contains encrypted routing information and keys. The payload contains the encrypted message. Using only group exponentiations and well-known symmetric cryptography, Sphinx is highly efficient [12].

Sphinx's privacy has been proven using a proof strategy by Camenisch and Lysyanskaya [5].[2] Camenisch and Lysyanskaya's proof strategy first proposes an idealized version of OR in the form of an ideal functionality in the Universal Composability framework [6]. This ideal functionality is effectively an abstract version of an OR protocol, from which privacy guarantees can be derived more readily. As proving that a protocol securely realizes this functionality directly is cumbersome, they also create a set of game-based properties which they claim imply realization of the ideal functionality [5]. As it turns out, those properties, which were used in Sphinx's privacy proof, are insuffiecent to realize the ideal functionality [21].

While Kuhn et al. propose new properties that indeed imply the ideal functionality [21], Sphinx is not able to achieve them for two reasons. First, Sphinx, along with many real-world applications, works in the service model, while Camenisch and Lysyanskaya's ideal functionality is in the integrated-system model. Secondly, Sphinx does not protect the integrity of the payload at each hop and thus allows for a malleability attack on the payload: If the adversary *tags* (i.e., flips bits of the payload) an onion leaving its sender, the exit relay processing the Sphinx packet will notice that the payload has been modified and drop the message. If Sphinx is used

[1]While the name is the same, this server is unrelated to the anonymization network "Nym" [13].
[2]In order to comply with [5] and to honor Sphinx's applicability for OR protocols, we (ab-)use the onion terminology in the rest of this paper, while always meaning OR *and* mix networks and considering a global adversary.

Proceedings on Privacy Enhancing Technologies 2024(4), 755–783
© 2024 Copyright held by the owner/author(s).
https://doi.org/10.56553/popets-2024-0140

in the integrated-system model, this attack allows an adversarial receiver colluding with the first relay to learn which user was contacting it. As this violates the desired privacy goal, it follows that Sphinx does not achieve the integrated-system properties proposed by Kuhn et al. [21]. In the service model, this attack only allows an adversary to link the sender to the exit relay and completely destroys the message in the process. Hence, there is hope that the highly efficient Sphinx packet format is still secure to use as long as it is in its intended service model. Indeed, this question is highly relevant since all known protocols that prevent this attack while supporting anonymous replies incur extremely high overhead due to heavy, relatively new cryptographic primitives [22]. As it stands, the Sphinx packet format is used in multiple OR and mixnet protocols, which use it in different settings [8, 9, 13, 19, 26, 27]. Some of these instantiations are known to be insecure due to the tagging attack on Sphinx [21].

In addition to the problems noted above, we discover that the Decisional Diffie-Hellman (DDH) assumption used by Danezis and Goldberg to prove that Sphinx satisfies Camenisch and Lysyanskaya's game-based properties [12] is insufficient for Sphinx's security proof.

In this paper, we hence set out to perform a thorough analysis and provide the missing privacy proofs for Sphinx. We first provide the necessary framework for the service model: A reusable game-based proof strategy which is of independent interest for future work on packet formats as well as as ideal functionality in the UC framework for use in analyzing the privacy guarantees of service-model OR protocols. We first define this new ideal functionality, which incorporates both the relaxed privacy accounting for payload malleability as well as the changes required for the service model. We then construct our game-based "onion properties" and prove that a protocol satisfying them implies that that protocol also realizes the ideal functionality. During the work on this proof, we also discover and fix mistakes and details in the proof for the related work in the integrated-system model [22].

Secondly, we turn to an analysis of Sphinx and realize that an adaptation must be made to the packet format and its operation in order to achieve provable security. As originally defined, Sphinx uses a nymserver to enable its reply functionality. However, the use of such third parties allows for an additional tagging attack based on payload malleability. For secure operation of Sphinx, we hence propose an adaptation of the Sphinx protocol that works without a nymserver, but still supports anonymous replies. This works by simply embedding the reply information in the packet's payload instead of sending it separately. Lastly, we discuss the effect of our privacy relaxation and detail criteria for the secure usage of Sphinx.

In summary, our main contributions are:

- the definition of repliable service OR schemes,
- the construction of an ideal functionality tailored to Sphinx as well as corresponding game-based properties,
- minor fixes in the proof for the related integrated-system model work,
- the discovery that the GDH assumption is required to prove Sphinx secure instead of the DDH assumption,
- the first detailed security proof for (a slightly adapted) Sphinx under the GDH assumption, and
- a discussion of criteria for secure usage of Sphinx.

Outline. Section 2 introduces the required background. Section 3 constructs the formal foundations for repliable OR in the service model, which Section 4 uses to analyze and adapt Sphinx. Section 5 discusses the privacy achieved by our adapted Sphinx as well as relevant criteria under which the adapted Sphinx is considered secure. Finally, Section 6 concludes this paper.

2 BACKGROUND AND RELATED WORK

We first introduce our privacy requirements, onion routing and mix networks and the general network model before providing background on the formal analysis in this work and the Sphinx packet format itself.

2.1 Onion Routing and Mix Network Packet Formats

Onion routing and mix networks aim to hide the sender of a packet in a set of users called the *anonymity set.* They thereby prevent linking the sender both to the sent message and the receiver. The networks employ multiple relays between the sender and receiver that process packets and forward them to the next relay or the receiver. As the message in the packet is typically wrapped into multiple layers of encryption, it is also called an *onion* and every processing result on an onion's relay path is an *onion layer* [16].

Relay services are typically run by volunteers that want to help the sender increase its privacy against adversarial receivers, but also internet providers and possibly even nation-state adversaries. Due to this open nature of the network, it is assumed that a fraction of the relays is controlled by the adversary as well [14]. Mix networks aim to protect against a global adversary and therefore not only change the representation of the onion packet, but also reorder incoming packets before forwarding them. Onion routing networks, however, traditionally protect only against local adversaries and prioritize low-latency service over stronger protection [14]. In terms of the packet format, however, the networks are similar — they share the underlying idea of layered encryption. For the sake of compatibility with related work, we use OR as a representative for both OR and mix network packet formats. We stress that we nonetheless target a global adversary.

2.2 Network Models and Functionalities

We distinguish between two network models for OR protocols: The integrated-system model assumes that the receiver is aware of and runs the OR protocol, while the service model assumes that the receiver can be unaware of the OR protocol. The last relay before the receiver, the *exit relay*, translates the packets accordingly.

We further distinguish between two functionalities: *Non-repliable* OR only sends messages from senders to receivers in one direction and *repliable* OR acts as two-way communication. Repliable OR schemes involve sending requests as *forward* messages and receiving responses as *reply* messages.

By design, the anonymity of the sender is required to hold even against a malicious replying receiver. Thus, the receiver must not know to whom it sends its reply. To achieve this, the sender can prepare an anonymized "return envelope" and include it in its onion. Some repliable OR protocols (including Sphinx) additionally require forward and reply packets to be indistinguishable from each other while they are moving through the network in order to increase the anonymity set of each packet [12].

2.3 Formally Analyzing Mix Network Packet Formats

The state of the art uses the Universal Composability (UC) framework [6] to formalize OR security for packet formats. In UC, the desired security is defined by an ideal functionality $\mathcal{F}$, which is an optimal, abstract version of the protocol. $\mathcal{F}$ performs all computations on a trusted third party and clearly specifies how the environment (which controls the honest parties) and the adversary are allowed to interact with that third party. The goal is then to show that a real protocol is as secure as $\mathcal{F}$ and thus *securely realizes* $\mathcal{F}$. This means all attacks against the real protocol must also work in $\mathcal{F}$ (which is secure by definition). In this model, the environment controls all of the honest parties and the adversary controls the corrupted parties. To show that a protocol securely realizes $\mathcal{F}$, one constructs a simulator that translates the actions of the real-world adversary into the ideal-world $\mathcal{F}$'s adversarial entity's behavior and the ideal world's honest parties' actions into real-world protocol messages. The environment and the adversary are allowed to collaborate and the environment's view after the protocol ends is given to a distinguisher. Only if the view of the environment when it is interacting with the simulator and $\mathcal{F}$ is indistinguishable from the view created from an interaction of the environment with the real-world adversary and the real protocol does the protocol securely realize $\mathcal{F}$ [6].

2.3.1 Overview of Analyses. The first approach to consider OR in the UC framework was proposed by Camenisch and Lysyanskaya. They create an ideal functionality and (as they claim) corresponding game-based security properties for the integrated-system model without replies [5]. The reason for the creation of the game-based properties is that proving that a protocol securely realizes an ideal functionality is complex and involves multiple steps: One must construct a simulator for every adversary and then prove that the simulator simulates the adversary indistinguishably. Since many OR protocols operate in a similar way, Camenisch and Lysyanskaya reduce this overhead by defining their game-based properties. Their secure realization proof works for any OR protocol in their model that satisfies the game-based properties, taking advantage of the similarities between protocols. For authors of packet formats, this means that they only need to prove that their format satisfies the game-based properties in order to securely realize the ideal functionality [5].

A series of protocols have based their security proofs on Camenisch and Lysyanskaya's game-based properties [8, 9, 12]. Kuhn et al. find flaws in the proposed properties as well as in the corresponding protocols [21]. They fix the proof strategy by proposing new OR properties that imply the ideal functionality as originally proposed by Camenisch and Lysyanskaya.

Ando and Lysyanskaya extend the ideal functionality and OR properties to include replies in the integrated-system model and propose a new OR scheme [2] for this model.

Kuhn et al. [22] improve on Ando and Lysyanskaya's work by improving the privacy in the schemes, ideal functionality, and the onion properties for repliable integrated-system-model OR.

2.3.2 Summary of Integrated-System Framework. In the following, we give a high-level summary of the integrated-system framework [22], which we will adapt for our repliable service-model formalization.

The ideal functionality $\mathcal{F}_R$ for repliable OR in the integrated-system model offers an interface to the environment and adversary that allows relays to send and forward onions. The honest and corrupted parties send messages to the trusted third party to trigger these actions. $\mathcal{F}_R$ then provides the appropriate outputs to the parties — these correspond to the ideal outputs one would expect from a secure OR protocol. For example, the adversary is notified when an onion is forwarded by an honest relay and receives a temporary onion identifier that it can send back to $\mathcal{F}_R$ when it decides to deliver that onion to the next honest relay. However, $\mathcal{F}_R$ never constructs or outputs any "real" onions. Instead, the parties only receive information about the onions associated with the aforementioned temporary onion identifiers, These identifiers include no information about the sender, the receiver, the path, or the message of the respective onion. The identifiers are also replaced with a new random temporary identifier at each honest relay. This is done because any honest relay in an OR network breaks the link between the onion layers before and after itself.

Proving UC-realization for each OR protocol individually involves a lot of redundant work. In line with earlier work, Kuhn et al. thus define onion properties that are sufficient to prove that an integrated-system packet format securely realizes $\mathcal{F}_R$. These are *Correctness*, *Forwards Layer Unlinkability* ($LU^{\rightarrow}$), *Backwards Layer Unlinkability* ($LU^{\leftarrow}$), and *Repliable Tail Indistinguishability* ($TI^{\leftrightarrow}$) [22].

Correctness requires that onions follow their set paths and decrypt correctly if they are honestly processed.

In the *Forwards Layer Unlinkability* ($LU^{\rightarrow}$) game, onion layers between honest relays on the forward path of the onion are replaced with random onion layers taking the same path. The adversary is required to distinguish the replacement from the original. If the adversary cannot do so, the packet format ensures that onion layers on the forward path of an onion cannot be linked across an honest relay.

In the *Backwards Layer Unlinkability* ($LU^{\leftarrow}$) game, onion layers on the *reply path* (instead of the forward path) are replaced with random onion layers. Notably, it replaces them with random *forward* onion layers. Again, the adversary must distinguish the replacement from the original. If it cannot do so, the packet format ensures unlinkability of reply onion layers. This property also implies indistinguishability of forward and reply onions.

In the *Repliable Tail Indistinguishability* ($TI^{\leftrightarrow}$) game, onion layers going to a corrupted receiver relay are replaced with random onion layers with the same path and message contents. The adversary must once again tell the difference between the two. If it cannot, the packet format guarantees that different onions with the same message going to a corrupted receiver are indistinguishable.

Note that, if all four properties are satisfied, every layer of an onion can be replaced with a random layer such that only the respective subpaths between honest relays stay the same (and the message, if the layer goes to a corrupted receiver). Effectively, this means that an adversary learns no more from any given sequence of onion layers than it would if given a temporary onion identifier and the corresponding subpath like in $\mathcal{F}_R$.

These onion properties combine to allow us to prove secure realization: When constructing the simulator that simulates the adversary, but interacts only with $\mathcal{F}_R$, the simulator does not learn the full path or message of onions from honest senders, but only the subpaths between two honest relays. This is by construction of $\mathcal{F}_R$. In order to translate the ideal-world onion into a real-world onion, it must thus replace the real onion layers that the adversary and environment would expect to see with random (forward) onion layers that only match the original onion in their subpath (and their message if an adversarial receiver gets the onion). The properties ensure that the replacement cannot be detected, allowing the proof to be completed in this manner.

2.4 Sphinx

Sphinx [12] is a compact repliable mix packet format in the service model following the adversary model and privacy goals described above.

2.4.1 The Sphinx Packet. A Sphinx packet consists of a header η and a payload δ. The header of the packet contains all of the routing information for the packet while the payload contains the message and the receiver address. The sender of the packet builds both components layer by layer, starting at the final innermost layer and adding one additional layer of encryption for each relay on the packet's "path" as chosen by the sender. If the path is shorter than the configured maximum path length, padding is added to the final layer of the header. The relays each remove one layer of that encryption when they process the packet. The final relay removes the last layer of encryption from the packet and discovers the message and receiver address. It then delivers the message to the receiver [12]. The packet format is described in more detail in Section 4.1.

The separation of the header and payload allows for Sphinx's replies: A sender forms a repliable packet by creating a header for the reply as well as a key for the reply payload before sending the "forward" packet. The reply header and the key are sent to a third-party *nymserver* using another forward packet and stored under a pseudonym there. After receiving a reply from the receiver, the exit relay of the packet sends the nymserver the reply message and the pseudonym it learned from the forward packet. The nymserver encrypts the reply payload using the key and attaches it to the reply header before sending it [12].

2.4.2 Sphinx Security. Sphinx (as defined in [12]) has two known flaws. The first is due to the padding in the final layer of the header. In the original Sphinx definition, this padding consists of only 0 bits and depends on the chosen path length [15, 21]. As this pattern is recognizable, a corrupted exit relay learns information about the path length. The Sphinx implementation [15] fixes this issue by using random bits instead. In the following, we consider the version of Sphinx that includes this fix.

Payload Tagging. Sphinx's second flaw concerns the integrity of the payload. Due to how Sphinx handles replies, it cannot use a standard integrity check like a MAC of the payload δ at each relay: In order to calculate a MAC of the payload, the payload contents must be known. This is trivial for forward packets, but impossible for replies since the sender does not know what the receiver's reply will be ahead of time. Since the exit relay is not trusted, it cannot be used to calculate the payload MACs either: A global adversary could re-identify MACs it calculated at exit relays elsewhere in the network. As it stands, providing integrity for reply payloads has yet to be done efficiently. Some OR schemes provide integrity using complex cryptographic primitives like SNARGS and updatable encryption, but these are too computationally expensive to be viable in practice [22].

A different option is to provide integrity only for forward payloads, but not reply payloads. However, this comes at the cost of the adversary being able to distinguish forward and reply packets, which Sphinx is deliberately designed to avoid. We consider the indistinguishability of forward and reply packets an important feature of Sphinx since it doubles the size of the potential anonymity set of a packet. As a result, we choose to preserve this property of Sphinx at the expense of payload malleability. Adversarial modifications of Sphinx payloads thus go unnoticed until the packet reaches its exit relay, which notices (and drops) the modified packet during the integrity check on the payload. Crucially, this allows the adversary to link the packet it "tagged" (by, e.g., flipping bits in its payload) to the packet that was dropped by the exit relay. The payload is completely destroyed in the process. If Sphinx is used in the integrated-system model, (as, e.g., in [8]), this attack links sender and receivers and thereby completely breaks the scheme's security. However, if Sphinx is used in its intended service model, the attack only links the sender to its corresponding exit relay. This can be an acceptable leak in some settings, as we discuss later.[3]

2.4.3 Current Relevance of Sphinx. As of the writing of this paper, the Sphinx packet format is used in several different mix networks and OR protocols. However, many of these instantiations of Sphinx are insecure due to the tagging attack allowing adversaries to link senders to their chosen final relay, which is not independent of the sender's choice of receiver.

Sphinx was originally designed in 2009 by Danezis and Goldberg. In 2015, HORNET was developed, which uses Sphinx in its setup phase to negotiate circuits before switching to a different format for data transmission [8]. Beato et al. created a modified version of Sphinx without replies, but with payload integrity in 2016 [3]. Loopix, a mix network built in 2017, uses Sphinx in its original form but without its reply functionality [26]. TARANET (2018) uses Sphinx in a similar way as HORNET does [9]. In 2020, Kuhn et al. discovered that Sphinx is vulnerable to tagging attacks especially in the integrated-system model, demonstrating that HORNET and TARANET's setup phases are insecure [21]. The Nym mix network is in active deployment since 2021 and is based on the Loopix design and Sphinx packet format with replies [13]. Hugenroth et al. also published their Loopix- and Sphinx-based multicast mix scheme Rollercoaster in 2021 [18]. Pudding, a user discovery protocol again based on Loopix with replies, was created in 2023 [19]. Designed as recently as 2024, PolySphinx is a modified version of Sphinx

[3]Note that tagging the Sphinx header in this manner is not possible: The header's integrity is protected by a MAC that is checked at every relay for both forward and reply onions. The reason for this being easier to achieve than payload integrity is that the reply header can be built by the sender ahead of time, while the reply payload cannot (as the sender does not know the reply message).

that allows users to efficiently send a message to multiple receivers at once [27]. Sphinx has also been used in other works including measurement studies and payment channel networks such as the Lightning Network [1, 17, 25].

3 REPLIABLE SERVICE ONION ROUTING

We formalize repliable OR in the service model, hereafter referred to as repliable service OR (RSOR), and construct an ideal functionality and corresponding onion properties that allow for the payload malleability attack on Sphinx. We base our formalization on Kuhn et al.'s work [22].

3.1 Defining Repliable Service Onion Routing

3.1.1 Notation. In general, we follow the notation of the corresponding related work ([12, 22]) as much as possible. We represent messages with m, headers with η, and payloads, which include m along with other metadata, with δ. A packet is the combination of η and δ. We also refer to packets as onion layers and shorten the term *onion layer* to *onion* where appropriate. Onion paths $\mathcal{P} = (P_1, \ldots, P_n)$ consist of a sequence of relays with P_i being the i-th relay's name and P_n being the exit relay. R, the receiver, is not part of the onion's path. PK_i and SK_i are the public and secret key of relay P_i. O_i is the i-th onion layer, i.e., the processing result that P_{i-1} produces and sends to P_i. For reply information, we use the same notation with an additional superscript arrow: $x^{\leftarrow}$ indicates the reply counterpart to the forward component x. A notation table is provided in Table 1.

3.1.2 Assumptions. We make use of the following standard assumptions regarding the OR protocol, which we inherit from related work [21, 22]. Note that none of them require additional trust, but just limit the packet schemes our model applies to. To the best of our knowledge, every previously-proposed OR scheme and protocol adheres to these assumptions.

Assumption 1. *A maximum path length (number of relays on the path) of N is used (inclusive upper bound).*

Assumption 2. *Honest senders choose acyclic paths (to increase the chance of picking at least one honest relay).*

Assumption 3. *Replay protection at honest relays drops onions whose headers are bit-for-bit identical to ones that have already been seen at that relay. A non-duplicate onion is only dropped with a negligible probability.*

Assumption 4. *A sender always knows the public keys PK_i of any relays P_i it uses for its onions' paths.*

Assumption 5. *An onion O consists of a header η and a payload δ such that $O = (\eta, \delta)$.*

Next, we introduce new assumptions related to the service model.

Assumption 6. *Receivers drop any onions sent to them. Similarly, relays drop any onions they get from links to receivers.*

Since we assume that receivers are unaware of the OR network, they cannot process onions. Traffic between receivers is not part of our OR model.

Notation	Definition
$\mathcal{F}$	Ideal Functionality
$LU^{\rightarrow}$	Forwards Layer Unlinkability
$TLU^{\rightarrow}$	Tagging-Forward Layer Unlinkability
$LU^{\leftarrow}$	Backwards Layer Unlinkability
$SLU^{\leftarrow}$	RSOR-Backwards Layer Unlinkability
$TI^{\leftrightarrow}$	Tail Indistinguishability
$STI^{\leftrightarrow}$	RSOR-Tail Indistinguishability
η	Onion header
δ	Onion payload
m	Plaintext message
$\mathcal{P}$	Onion path
P	Onion relay
E	Exit relay
P_s	Onion sender
R	Message receiver
PK, SK	Public and secret keys
O	Onion
O_i	Onion layer
λ	Security parameter
p	Public parameters
$\mathcal{N}$	Set of onion relays
$\mathcal{R}$	Randomness
rid	Reply ID
$\mathcal{Z}$	Environment
$\mathcal{S}$	UC Simulator
$\mathcal{A}$	Adversary

Table 1: Notation used throughout the paper.

Assumption 7. *An onion in an RSOR packet format never has an empty forward path. If it is repliable, it does not have an empty backward path.*

An onion with an empty forward path is effectively not an onion at all since the sender is also the exit relay. The packet format should thus not allow a valid onion to have an empty path. We use an empty backward path as a sentinel value for a non-repliable onion.

Assumption 8. *An honest relay will always drop an unsolicited reply (i.e., a reply with a header the relay does not recognize as belonging to the final reply layer of an onion it sent).*

This is not a surprising limitation. Honest relays will only process replies that they expect. Finally, we add one cryptographic assumption that we require for our game-based properties and UC realization proof:

Assumption 9. *Onion payloads are encrypted with a pseudorandom permutation (PRP).*

With this assumption, any modification of a payload (i.e., a tagging attack) will completely randomize and thus destroy the payload contents, which we require for security against these attacks.

3.1.3 Formal RSOR Schemes. We build an RSOR scheme using the following algorithms (following [22]):

- Key generation algorithm G for $P \in \mathcal{N}$:
  $$(PK, SK) \leftarrow G(1^{\lambda}, p, P),$$
  where p are the public parameters of the protocol and $\mathcal{N}$ is the set of participating onion relays.
- Onion sending algorithm FormOnion ($n, n^{\leftarrow} \leq N$):
  $$O_i \leftarrow \text{FormOnion}(i, \mathcal{R}, m, \boldsymbol{R}, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}),$$
  $$\mathcal{P}^{(\leftarrow)} = (P_1^{(\leftarrow)}, \ldots, P_{n^{(\leftarrow)}}^{(\leftarrow)}) \in \mathcal{N}^{n^{(\leftarrow)}},$$
  $$PK_{\mathcal{P}^{(\leftarrow)}} = (PK_1^{(\leftarrow)}, \ldots, PK_{n^{(\leftarrow)}}^{(\leftarrow)}),$$
  where i is the index of the onion layer to output (in practice, $i = 1$, but our proofs require $i > 1$ as well) and $\mathcal{R}$ is the randomness to be used. If $i > n$, R is ignored and the onion O_i is formed like a reply with the message m. $\mathcal{P}^{\leftarrow}$ may be empty if no reply is desired. Otherwise, $\mathcal{P}^{\leftarrow}$ contains the "reply receiver" (which is the sender P_s) as its final relay. FormOnion is deterministic in its inputs.
- Onion processing algorithm ProcOnion at P:
  $$(O', P') \leftarrow \text{ProcOnion}(SK, O, P).$$
  P processing O with its secret key SK results in an onion O' to send to the next relay P'. In case of an error, $(O', P') = (\bot, \bot)$. ProcOnion may use internal randomness.
  $(m, R) = \text{ProcOnion}(SK_E, O_n, E)$ if E is the exit relay of the onion. m is the message for receiver R.
  $(m^{\leftarrow}, \bot) = \text{ProcOnion}(SK_s, O_{n+n^{\leftarrow}}, P_s)$ if sender P_s (with private key SK_s) received the reply onion $O_{n+n^{\leftarrow}}$ containing reply message $m^{\leftarrow}$.
- Reply sending algorithm FormReply:
  $$(O^{\leftarrow}, P^{\leftarrow}) \leftarrow \text{FormReply}(m^{\leftarrow}, O, E, SK).$$
  The reply onion $O^{\leftarrow}$ to be sent to $P^{\leftarrow}$ is created from the reply message $m^{\leftarrow}$ and the original onion O as processed by O's exit relay E with the secret key SK. If an error occurs, $(O^{\leftarrow}, P^{\leftarrow}) = (\bot, \bot)$. FormReply may use internal randomness.

Definition 1
An *RSOR scheme* is a tuple of the polynomial-time algorithms (G, FormOnion, ProcOnion, FormReply), as defined above.

Using an RSOR scheme, we can construct a corresponding *RSOR protocol* as follows:

(1) A sender selects the parameters for its onion, builds it using FormOnion, and sends O_1 to P_1.
(2) Each relay P_i processes the onion in turn using ProcOnion and sends O_{i+1} to P_{i+1}.
(3) The onion's exit relay P_n gets $(m, R) \leftarrow \text{ProcOnion}(SK_n, O_n, P_n)$. It generates a random "reply ID" rid, remembers O_n in a map as (rid, O_n), and sends (m, rid) to R.
(4) R receives (m, rid) and decides to reply to the message. It sends its reply $(m^{\leftarrow}, rid)$ back to P_n.
(5) P_n gets the reply message and finds (rid, O_n) in its map. It calculates $(O^{\leftarrow}, P^{\leftarrow}) \leftarrow \text{FormReply}(m^{\leftarrow}, O_n, P_n, SK_n)$ and sends $O^{\leftarrow}$ to $P^{\leftarrow}$.
(6) The reply onion follows its reply path like in step four until the sender receives $O_{n^{\leftarrow}}^{\leftarrow}$ and processes it to receive $(m^{\leftarrow}, \bot)$.

We introduce the concept of "reply IDs" (rids) in the protocol description. These IDs are intended to be abstractions of some stateful delivery and reply mechanism used by the receivers (e.g., TCP connections in the case of web servers). To avoid the complexity these mechanisms would bring into our definitions, we represent them as a simple device that allows a receiver to send a reply directly to the relay it received the rid from. Our protocols do not protect rids, so they can be manipulated by adversaries.[4] This corresponds to the adversaries' capability to control links and the delivery of packets on those (unauthenticated) links, which cannot be mitigated in the service model.

In addition to the algorithms defined above, we also require the algorithm $\text{RecognizeOnion}(i, O, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$ to be defined for RSOR analogously to its original definition by Kuhn et al. [22]: The algorithm compares the header of O to the i-th layer of the onion originally created from the given inputs including the secret randomness $\mathcal{R}$. For our RSOR schemes, this means that the message is not checked since it is in the payload. Note that this algorithm is required even though FormOnion is deterministic in its inputs because ProcOnion and FormReply may introduce internal randomness.

3.2 RSOR Ideal Functionalities

In this section, we provide an overview of our ideal functionality $\mathcal{F}_{RSOR}$ for RSOR, focusing on the core ideas and issues in its construction. We base it on Kuhn et al.'s $\mathcal{F}_R$ [22]. We also explain the differences between $\mathcal{F}_{RSOR}$ and $\mathcal{F}_R$. $\mathcal{F}_{RSOR}$ is given in pseudocode in Appendix A.

3.2.1 Ideal Functionality. Recall from Section 2.3 that, in an UC ideal functionality $\mathcal{F}$, all processing happens on a trusted party with a set of interfaces and procedures that the environment $\mathcal{Z}$ and the adversary $\mathcal{S}$ interact with and receive information from.

Fundamental concept. Following the related work, onion layers are abstracted into a series of random temporary identifiers, the temp IDs (tids). When an onion is forwarded through an honest relay, it receives a new tid, thus rendering the layers before the relay and after the relay unlinkable. Corrupted relays do not cause the tid to be replaced. As an onion is forwarded between honest relays, $\mathcal{S}$ receives the tids for the layers along with information on the layers' paths. This corresponds to the intuition that the adversary learns nothing about the content of the onion and cannot link the layers before and after an honest relay to each other. $\mathcal{F}_{RSOR}$ models all of the interactions between relays and between relays and receivers in the network including all possible adversarial capabilities in our adversary model. This includes corner cases and unusual adversary behavior.

Detailed Interaction. To send a new onion in $\mathcal{F}_{RSOR}$, $\mathcal{Z}$ (for honest parties) or $\mathcal{S}$ (for adversarial parties) notify $\mathcal{F}_{RSOR}$ with the desired receiver, message, and forward and reply paths. The resulting temp ID tid is given to $\mathcal{S}$, which controls the links and may choose to deliver the tid (i.e., the onion) to the next honest relay. If the onion was sent by a corrupted sender, $\mathcal{S}$ receives all of the information

[4]Note that these rids must not be linkable to the onion they are mapped to, or an adversary could use them to link the onion's receiver to the onion layers before an honest exit relay.

on the onion for every path segment.[5] $\mathcal{Z}$ can decide when honest relays are done processing an onion and forward it to the next relay.

While the onion is being forwarded through the network, $\mathcal{S}$ can also choose to *tag* it. This feature behaves like the tagging of a Sphinx packet (which is explained in Section 2.4.2). When the onion is forwarded by its last honest relay, it is either discarded (if it was tagged) or the message and receiver are leaked to $\mathcal{S}$.

Reply handling comes in two variants: If the last honest relay is the exit relay of the onion, $\mathcal{F}_{RSOR}$ generates a reply ID *rid* (an abstraction of a connection that could be established by a protocol like TCP) and gives it to $\mathcal{S}$. $\mathcal{S}$ can use the *rid* to provide the exit relay with a message for the reply onion. On the other hand, if the exit relay of the onion is corrupted, $\mathcal{S}$ receives a *tid* it can use to send the reply onion with a reply message from any corrupted relay.[6] Since the channels between the relays and receivers are not secure, we have to assume that $\mathcal{S}$ has complete control over message and reply delivery. Honest receivers can also initiate sending a reply to a message with an *rid*.

3.2.2 Comparison with Repliable Integrated-System OR. While we are able to reuse large parts of Kuhn et al.'s $\mathcal{F}_R$ [22] to build $\mathcal{F}_{RSOR}$, switching to the service model requires several additions to the existing $\mathcal{F}_R$.

First, the new tagging feature marks onions to be discarded when they reach their last honest relay.[7] Notably, tagging behaves asymmetrically: when a forward onion is tagged, the last honest relay discards it. On the reply path, a tagged onion is noticed by the reply receiver. If the reply receiver is honest, $\mathcal{S}$ is not informed about the tag.

Second, $\mathcal{F}_{RSOR}$ adds handling of message and reply delivery on the final link. Since these links are not secure, a real-life adversary is capable of manipulating the traffic on them in many ways. These include delivering messages and *rid*s to the wrong receivers or exit relays, swapping *rid* and message pairs, impersonating exit relays to receivers and vice versa, and sending reply onions from other corrupted relays if the exit relay is corrupted. $\mathcal{F}_{RSOR}$ permits all of these adversarial behaviors in its interface to $\mathcal{S}$.

In particular, the first of these two changes relaxes $\mathcal{F}_{RSOR}$'s security requirements, which allows RSOR schemes to have malleable payloads. Accordingly, using RSOR protocols requires additional care, as detailed in Section 5.1.

3.3 RSOR Onion Properties

Our properties are based on the properties defined by Kuhn et al. [22] with appropriate adjustments for the new algorithms and functionality. We detail one property and sketch the others here. For a formal definition of the other properties see Appendix B.

3.3.1 RSOR-Correctness. This property requires that the scheme works as intended if no adversarial actions take place. Precisely, this means that, as the onion is processed using ProcOnion, each relay decrypts the correct address of the next relay and the next onion layer and the final layer decrypts to the message chosen by the sender. The same applies to the corresponding reply onion.

[5]This matches the behavior of real packet formats like Sphinx, where the sender calculates every layer itself and can thus recognize them.

[6]A packet format cannot prevent the adversary from sending messages using any of the relays under its control.

[7]Since the ideal functionality itself does not model adversarial processing, this is the case even when the last honest relay is not the exit relay. In that case, $\mathcal{S}$ is provided with the remainder of the onion's path.

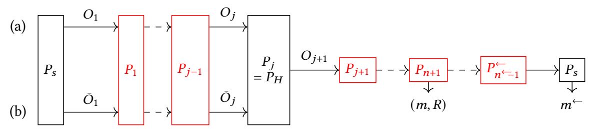


Figure 1: The $TLU^{\rightarrow}$ onion property. The adversary is given either its chosen onion O_1 (a) or the random onion $\bar{O}_1$ (b) and must distinguish the two cases. Relays marked in this style are adversarial, while those in the normal style are honest. Omitted adversary-chosen paths are shown with dashed lines. The secondary output O_{j+1} after P_j is the same in both cases.

3.3.2 Tagging-Forwards Layer Unlinkability (See Figure 1). $TLU^{\rightarrow}$ is the RSOR equivalent of $LU^{\rightarrow}$. It replaces onion layers between the honest sender and the first honest relay of an onion with random layers using that path segment. The introduction of tagging requires us to adjust this property. The original definition of $LU^{\rightarrow}$ provides the adversary with the challenge onion O_1 or its replacement onion $\bar{O}_1$ as produced by the sender and recognizes the processed layer O_j or $\bar{O}_j$ when the adversary submits it to the processing oracle of the first honest relay. This recognition is based on the onion's header, so a tagged onion is still recognized. In the $b = 0$ case, the tagged O_j is processed normally and a tagged O_{j+1} is output to the adversary. However, in the $b = 1$ case, the tagged $\bar{O}_j$ is recognized by the oracle and the original O_{j+1} is output without being tagged. The adversary can simply finish the onion's processing to tell the difference.

To alleviate this issue, $TLU^{\rightarrow}$'s oracle recognizes when the payload of the challenge onion has been tagged and "forwards" the tag by tagging O_{j+1} before sending it. If the honest relay is the exit relay, the oracle outputs nothing in this case. Since we assume that the payload is encrypted using a PRP, tagging the onion simply involves replacing the payload with randomness.

The following full definition of $TLU^{\rightarrow}$ is derived from Kuhn et al. [22]. Our modifications are given in this style. We also abbreviate RecognizeOnion as ROnion for the sake of formatting.

Definition 2
Tagging-Forwards Layer Unlinkability is defined as:

(1) The adversary receives the router names P_H, P_s and challenge public keys PK_S, PK_H, chosen by the challenger as $(PK_H, SK_H) \leftarrow G(1^{\lambda}, p, P_H)$ and $(PK_S, SK_S) \leftarrow G(1^{\lambda}, p, P_s)$.
(2) Oracle access: The adversary may submit any number of **Proc** and **Reply** requests for P_H or P_s to the challenger. For any **Proc**(P_H, O), the challenger checks whether η is on the η_H-list. If it is not on the list, it sends the output of ProcOnion(SK_H, O, P_H), stores η on the η_H-list and O on the O_H-list. For any **Reply**(P_H, O, m), the challenger checks if O is on the O_H-list and if so, the challenger sends FormReply(m, O, P_H, SK_H) to the adversary. (Similar for requests on P_s with the η_S-list).
(3) The adversary submits a message m, a receiver R, a position j with $1 \leq j \leq n$, a path $\mathcal{P} = (P_1, \ldots, P_j, \ldots, P_n)$ with

$P_j = P_H$, a path $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{n^{\leftarrow}}^{\leftarrow} = P_s)$ and public keys for all relays PK_i ($1 \leq i \leq n$ for the relays on the path and $n < i$ for the other relays).

(4) The challenger checks that the chosen paths are acyclic, the router names and public keys are valid and that the same key is chosen if the router names are equal, and if so, sets $PK_j = PK_H$ and $PK_{n^{\leftarrow}}^{\leftarrow} = PK_S$ and picks $b \in 0, 1$ at random.

(5) The challenger creates the onion O_1 with the adversary's input choice and honestly chosen randomness $\mathcal{R}$:

$$\text{FormOnion}(1, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$$

and a replacement onion $\bar{O}_1$ with the first part of the forward path $\bar{\mathcal{P}} = (P_1, \ldots, P_j)$, a random message $\bar{m} \in M$, another honestly chosen randomness $\bar{\mathcal{R}}$, an honestly chosen random receiver $\bar{R}$, and an empty backward path $\bar{\mathcal{P}}^{\leftarrow} = ()$:

$$\text{FormOnion}(1, \bar{\mathcal{R}}, \bar{m}, \bar{R}, \bar{\mathcal{P}}, \bar{\mathcal{P}}^{\leftarrow}, PK_{\bar{\mathcal{P}}}, PK_{\bar{\mathcal{P}}^{\leftarrow}})$$

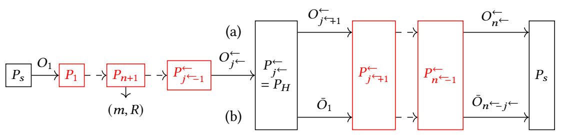
(6) If $b = 0$, the challenger gives O_1 to the adversary. Otherwise, the challenger gives $\bar{O}_1$ to the adversary.

(7) Oracle access: If $b = 0$ the challenger processes all oracle requests as in step 2).
Otherwise, the challenger processes all requests as in step 2) except for:

- If $j < n$:
  - **Proc**(P_H, O) with
    $\text{ROnion}(j, O, \bar{\mathcal{R}}, \bar{m}, \bar{R}, \bar{\mathcal{P}}, \bar{\mathcal{P}}^{\leftarrow}, PK_{\bar{\mathcal{P}}}, PK_{\bar{\mathcal{P}}^{\leftarrow}})$,
    and the expected payload δ_j, η is not on the η_H-list and
    $\text{ProcOnion}(SK_H, O, P_H) \neq (\bot, \bot)$:
    The challenger outputs (P_{j+1}, O_c) with
    $O_c \leftarrow \text{FormOnion}(j+1, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$
    and adds η to the η_H-list and O to the O_H-list.
  - **Proc**(P_H, O) with
    $\text{ROnion}(j, O, \bar{\mathcal{R}}, \bar{m}, \bar{R}, \bar{\mathcal{P}}, \bar{\mathcal{P}}^{\leftarrow}, PK_{\bar{\mathcal{P}}}, PK_{\bar{\mathcal{P}}^{\leftarrow}})$
    but the incorrect payload δ', η is not on the η_H-list and
    $\text{ProcOnion}(SK_H, O, P_H) \neq (\bot, \bot)$:
    The challenger outputs $(P_{j+1}, \tilde{O}_c)$ with
    $O_c \leftarrow \text{FormOnion}(j+1, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$
    and $\tilde{O}_c$ being O_c with a tagged payload and adds η to the η_H-list and O to the O_H-list.
- If $j = n$:
  - **Proc**(P_H, O) with
    $\text{ROnion}(j, O, \bar{\mathcal{R}}, \bar{m}, \bar{R}, \bar{\mathcal{P}}, \bar{\mathcal{P}}^{\leftarrow}, PK_{\bar{\mathcal{P}}}, PK_{\bar{\mathcal{P}}^{\leftarrow}})$,
    η is not in the η_H-list and
    $\text{ProcOnion}(SK_H, O, P_H) \neq (\bot, \bot)$:
    The challenger outputs (m, R) and adds η to the η_H-list and O to the O_H-list.
  - **Reply**$(P_H, O, m^{\leftarrow})$ with
    $\text{ROnion}(j, O, \bar{\mathcal{R}}, \bar{m}, \bar{R}, \bar{\mathcal{P}}, \bar{\mathcal{P}}^{\leftarrow}, PK_{\bar{\mathcal{P}}}, PK_{\bar{\mathcal{P}}^{\leftarrow}})$
    O is on the O_H-list and has not been replied before and
    $\text{FormReply}(m^{\leftarrow}, O, P_H, SK_H) \neq (\bot, \bot)$:
    The challenger outputs $(O_c, P_1^{\leftarrow})$ with
    $O_c \leftarrow \text{FormOnion}(j+1, \mathcal{R}, m^{\leftarrow}, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$.

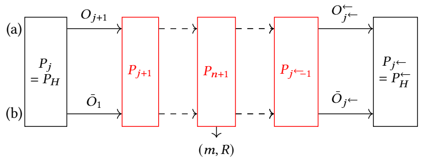
(8) The adversary produces guess b'.

$TLU^{\rightarrow}$ is achieved if any PPT adversary $\mathcal{A}$ cannot guess $b' = b$ with a probability non-negligibly better than $\frac{1}{2}$.



(a)
(b)
P_s
O_1
P_1
P_{n+1}
(m, R)
$P_{j^{\leftarrow}-1}^{\leftarrow}$
$O_{j^{\leftarrow}}^{\leftarrow}$
$P_{j^{\leftarrow}}^{\leftarrow} = P_H$
$O_{j^{\leftarrow}+1}^{\leftarrow}$
$\bar{O}_1$
$P_{j^{\leftarrow}+1}^{\leftarrow}$
$P_{n^{\leftarrow}-1}^{\leftarrow}$
$O_{n^{\leftarrow}}^{\leftarrow}$
$\bar{O}_{n^{\leftarrow}-j^{\leftarrow}}$
P_s

Figure 2: The $SLU^{\leftarrow}$ onion property. The adversary is initially given its chosen onion O_1. The oracle at the relay $P_{j^{\leftarrow}}^{\leftarrow}$ will then return either its chosen onion $O_{j^{\leftarrow}+1}^{\leftarrow}$ (a) or the random onion $\bar{O}_1$ (b) and must distinguish the two scenarios. Relays marked in this style are adversarial, while those in the normal style are honest. Omitted adversary-chosen paths are shown with dashed lines.



(a)
(b)
$P_j = P_H$
O_{j+1}
$\bar{O}_1$
P_{j+1}
P_{n+1}
(m, R)
$P_{j^{\leftarrow}-1}$
$O_{j^{\leftarrow}}^{\leftarrow}$
$\bar{O}_{j^{\leftarrow}}$
$P_{j^{\leftarrow}} = P_H^{\leftarrow}$

Figure 3: The $STI^{\leftrightarrow}$ onion property. The adversary is given either its chosen onion O_{j+1} (a) or the random onion $\bar{O}_1$ (b) and must distinguish the two cases. Relays marked in this style are adversarial, while those in the normal style are honest. Omitted adversary-chosen paths are shown with dashed lines.

3.3.3 RSOR-Backwards Layer Unlinkability (See Figure 2). The definition of the $SLU^{\leftarrow}$ property is analogous to the $TLU^{\rightarrow}$ property, but replaces the challenge onion on a path segment of the reply path. Additionally, the replacement onion is a forward onion, not a reply onion. This ensures that a scheme satisfying $SLU^{\leftarrow}$ will have indistinguishable forward and reply onions in the OR network. We do not need to adapt $SLU^{\leftarrow}$ to account for the tagging attack since the reply receiver processing oracle never produces output for onions with the challenge headers, so it cannot produce a wrong (untagged) output.

3.3.4 RSOR-Tail Indistinguishability (See Figure 3). The $STI^{\leftrightarrow}$ property is RSOR's counterpart for the $TI^{\leftrightarrow}$ property in the integrated-system model. It replaces the challenge onion with a random onion using the same path segment between an honest relay on the forward path and a second honest relay on the reply path. In $STI^{\leftrightarrow}$, we do not allow the exit relay to be the honest relay since that situation is already covered by $SLU^{\leftarrow}$.

3.3.5 Comparison with Integrated-System Properties. Besides the adaptions necessary for the different representation of the receiver and path, the main difference between the integrated-system properties and RSOR properties is the introduction of tagging into $TLU^{\rightarrow}$. The other two properties do not require provisions for tagging mitigation: $SLU^{\leftarrow}$ remains unaffected because of the previously-described asymmetry of the tagging attack on the reply path, while $STI^{\leftrightarrow}$ never outputs a challenge onion from its oracles.

3.4 Properties imply $\mathcal{F}_{RSOR}$

With the properties defined above, we can now define *secure* RSOR schemes and protocols as those that satisfy the properties and behave in the way described in Section 3.1.3.

Definition 3
An RSOR scheme (G, FormOnion, ProcOnion, FormReply) that satisfies the four onion properties *RSOR-Correctness*, $TLU^{\rightarrow}$, $SLU^{\leftarrow}$, and $STI^{\leftrightarrow}$ is a *secure* RSOR scheme.

Definition 4
A *secure RSOR protocol* is based on a secure RSOR scheme and behaves as defined in Section 3.1.3. A full definition of the protocol behavior is given in Appendix C.

Theorem 1. *A secure RSOR protocol securely realizes $\mathcal{F}_{RSOR}$.*

The proof of this theorem is analogous to Kuhn et al.'s original proof for integrated-system protocols and $\mathcal{F}_R$ [22]. We provide a brief sketch of the proof and the required changes here. The full proof is given in Appendix D. First, we construct a simulator $\mathcal{S}$ that interacts with the adversary $\mathcal{A}$ and $\mathcal{F}_{RSOR}$, replicating the adversary's real-world actions in $\mathcal{F}_{RSOR}$ and vice versa. After that, we use a hybrid proof to show that any RSOR protocol that satisfies the RSOR properties is indistinguishable from our simulator interacting with $\mathcal{F}_{RSOR}$.

When interacting with $\mathcal{F}_{RSOR}$ and $\mathcal{A}$, $\mathcal{S}$ must correctly translate the onions and messages sent by honest relays in $\mathcal{F}_{RSOR}$ into the real world without complete information on the contents or paths of the onions. The RSOR properties ensure that the simulator can create replacement onions with randomized contents and truncated paths whenever it needs to do so without $\mathcal{A}$ or $\mathcal{Z}$ noticing the replacement.

Differences to $\mathcal{F}_R$. Introducing tagging requires additional logic in the simulator. When the adversary tags an onion from an honest sender, the simulator can tell that the payload has been manipulated and tag the onion in $\mathcal{F}_{RSOR}$. If the onion is from a corrupted sender, $\mathcal{S}$ notices the manipulation when it processes the onion at its honest exit relay and can tag it in $\mathcal{F}_{RSOR}$ at that point. This behavior allows the simulator to handle tagging attacks correctly.

In addition, $\mathcal{S}$ now also needs to handle communication with external receivers involving messages and reply IDs. Translating these between $\mathcal{F}_{RSOR}$ and the real world involves forwarding the appropriate communications while potentially impersonating exit relays or receivers.

In our hybrid machine construction, we fix an error in Kuhn et al.'s proof: They apply the $LU^{\rightarrow}$ (and $LU^{\leftarrow}$) properties to path segments that do not start at the honest sender (or end at the honest reply receiver) [22]. However, the properties do not apply to these situations. We fix this problem by applying the $STI^{\leftrightarrow}$ property once in these situations to truncate the paths of the corresponding onions. Afterwards, we can apply the $TLU^{\rightarrow}$ or $SLU^{\leftarrow}$ properties as before. The same change (using $TI^{\leftrightarrow}$) can be used to repair the proof in [22].

4 ANALYZING SPHINX'S SECURITY

We now aim to show that the Sphinx securely realizes $\mathcal{F}_{RSOR}$. Therefore, we use our RSOR properties and show that Sphinx satisfies each of them. However, to be able to do so, we have to make two further adjustments to the Sphinx protocol: Firstly, we remove the nymserver, as it allows for attacks (in Section 4.2). Doing so is possible by including reply headers in payloads instead of forwarding them to the nymserver separately. Secondly, we need to change the underlying cryptographic assumption to enable our proof (in Section 4.3.2).

4.1 The Sphinx Packet

A Sphinx packet consists of a header $\eta = (\alpha, \beta, \gamma)$ and a payload δ (see Section 2.4.1 for an overview). α contains the shared secret keys encrypted for each relay, β holds the routing information padded by the sender, and γ is a MAC for β. Note that we adopt the notation of the original Sphinx paper [12].

Each relay P_i has an asymmetric key pair of the form $(x_i, y_i := g^{x_i})$ with $x_i \in \mathbb{Z}_q^*$ and g as a generator of a cyclic group $\mathcal{G}$ with the prime order q. q should be approximately $2^{2\kappa}$, where κ is the security parameter [12]. The public keys are used along with random oracles in order to encapsulate the shared secret keys used for the symmetric cryptographic primitives in each layer of the Sphinx packet. We refer to this part of the packet format as Sphinx's random oracle-key encapsulation mechanism (RO-KEM). The RO-KEM's ciphertext is α, the first component of the Sphinx packet header [12].

α is formed using the public keys of the relays on the sender's chosen path. First, the sender chooses a secret $x \in \mathbb{Z}_q^*$, where $\mathbb{Z}_q^*$ is the multiplicative group of whole numbers modulo q without 0. x is the only source of randomness in the Sphinx packet. Using x, the sender generates the α-key encapsulation for the first onion layer (which is layer 0):[8] $\alpha_0 = g^x$. The sender then derives the shared secret for the first relay: $s_0 = y_0^x$, where y_0 is the public key of the first relay.[9] When the first layer of the onion reaches the first relay, it can derive s_0 using its secret key: $s_0 = \alpha_0^{x_0}$. The keys and random values used in the cryptographic primitives in the Sphinx packet are all derived from s_0 using the random oracles h_b, h_ρ, h_μ, and h_π [12].

In particular, the random oracle h_b is used to build the key encapsulations for the following onion layers: To calculate α_1, the sender lets $b_0 = h_b(\alpha_0, s_0)$ and $\alpha_1 = g^{xb_0}$. The corresponding shared secret is similarly calculated as $s_1 = y_1^{xb_0}$, using the public key of the second relay on the onion's path. The b_i are referred to as *blinding factors.* When the first relay processes the first onion layer, it can calculate $\alpha_1 = \alpha_0^{b_0}$ with the blinding factor b_0 that it gets from the same random oracle h_b. With this, the second relay can derive its shared secret using its private key. This process is repeated for each onion layer. Note that only α_0 is included in the "final first onion layer" — the later α_i are calculated by the relays themselves and replace the respective α_{i-1} in the packet during processing [12].

[8]In this section, onion layer indices start at 0 for consistency with Sphinx's original definition in [12].
[9]Note that our indices i for relay names are relative to a single packet's path for clarity.

Since we focus on Sphinx's RO-KEM in the following sections, we only give a short overview of the remaining packet structure here. For a more detailed and technical description, see Appendix E.

The other two components of the Sphinx header are β and γ. In each onion layer i, γ_i is simply a MAC μ of β_i keyed with $h_\mu(s_i)$. Here, h_μ is one of the random oracles keyed with that layer's shared secret. β_i contains the address of the next relay P_{i+1}, the next MAC γ_{i+1}, and a prefix of the next β_{i+1} in that order. This information is XORed with the output of a PRG ρ keyed with $h_\rho(s_i)$. When processing an onion layer, the relay P_i creates β_{i+1} from β_i by appending a string of zero bits to β_i, XORing the PRG output onto the result, and cutting off the relay address and MAC at the start of the end result. The construction of Sphinx's padding scheme means that the XORing the zero-bit extension with the PRG output results in exactly the missing suffix of β_{i+1}.

Finally, the Sphinx packet payload δ is simply constructed thorugh multiple layers of encryption with a pseudorandom permutation (PRP) π, which is keyed with $h_\pi(s_i)$ in the i-th layer.[10] The payload is constructed by the sender in reverse order, layering encryption from the final onion layer to the first. The innermost layer contains a zero padding of length equal to the security parameter, the receiver address, and the message in that order. When a relay processes an onion, it simply removes one layer of encryption from the payload. The final relay performs an integrity check by checking that the zero padding at the beginning of the payload is intact.

4.2 Nymserver

The original Sphinx definition by Danezis and Goldberg [12] uses a *nymserver* to hold the reply headers for the onions in the network. To create a repliable onion, the sender first sends a non-repliable onion containing the reply header and a symmetric key for the payload to the nymserver under a pseudonym. The sender is responsible for embedding that pseudonym in the forward onion for the exit relay to find. If the receiver decides to reply, it sends its reply message back to the exit relay. The exit relay sends the reply message and the pseudonym to the nymserver. The nymserver finds the reply header associated with the pseudonym in its database, encrypts the payload, attaches it to the reply header, and sends the reply onion [12].

We observe that this nymserver construction is insecure in the presence of an attacker that can tag or drop onions and controls the nymserver or observes its traffic: A sender that wants to send a repliable onion actually sends two onions, one going to the nymserver. The attacker tags (or drops) one of these onions, hoping that it picked the nymserver onion. If it is successful, the reply header is not stored in the nymserver. When the exit relay (on behalf of the receiver, which is trivally linkable to the exit relay) sends the message and pseudonym to the nymserver, the attacker can observe that no onion is produced by the nymserver,[11] thus learning the connection between the sender and the receiver.

To fix this problem and create a version of Sphinx that can be proven secure, we adapt Sphinx to include the reply header and payload symmetric key in the forward onion payload directly.[12] In the original definition of Sphinx, a forward payload has the contents $0_\kappa \| R \| m$ and a reply payload has the contents $0_\kappa \| m^{\leftarrow}$ [12].[13]In our adaptation, a forward payload is now formed as $0_\kappa \| R \| \eta_0 \| \tilde{k} \| m$ with R as the receiver address, η_0 as the reply header, and $\tilde{k}$ as the symmetric key. Non-repliable forward payloads and reply payloads contain zero paddings $\text{pad}^{\rightarrow}_{\kappa,N}$ and $\text{pad}^{\leftarrow}_{\kappa,N}$ of the appropriate lengths instead of $\eta_0 \| \tilde{k}$ and $R \| \eta_0 \| \tilde{k}$ respectively. This means a non-repliable forward payload contains $0_\kappa \| R \| \text{pad}^{\rightarrow}_{\kappa,N} \| m$ and a reply payload contains $0_\kappa \| \text{pad}^{\leftarrow}_{\kappa,N} \| m^{\leftarrow}$. This change fixes the issue while removing a third party from the protocol and simplifying Sphinx. With this change, the exit relay is now responsible for taking the reply message from the receiver, embedding it in the reply payload, and sending the completed reply onion.

Efficiency. Including the reply header in the Sphinx payload leads to an increase in the overhead of a Sphinx packet. The size of a Sphinx header in bytes is given by Danezis and Goldberg as $p + (2N + 1)s$, where p is the size of a group element, N is the fixed maximum hop count, and s is the security parameter in bytes. The total overhead of an original Sphinx packet thus comes out to $p + (2N + 2)s$ bytes when accounting for the additional zero padding in the payload [12]. Our modification adds another header and one key to the payload, exactly doubling the overhead: $2p + (4N + 4)s$ bytes of overhead go into a nymserverless Sphinx packet. Using Danezis and Goldberg's calculations for a Sphinx instantiation on an EC group with $p = 32$, $s = 16$, and $N = 5$, the total overhead is now 448 bytes. This is less overhead than incurred by Mixminion, which needs 848 bytes [11]. Möller's scheme has less overhead than our modified Sphinx at only 400 bytes using the same parameters, but at the cost of not supporting replies [23].

In addition, our modified Sphinx does not send a packet to the nymserver, which would otherwise incur its own overhead of $p + (2N + 2)s$ bytes in addition to the reply header and symmetric key in the payload. Accounting for this, the total traffic caused by a single repliable Sphinx packet is reduced by our modification. If the reduction in the number of packets is a concern for practical instantiations, cover traffic can be added to replace the packets to the nymserver.

4.3 Sphinx Key Encapsulation Mechanism (KEM)

Before we move on to the RSOR properties, we discuss the security of Sphinx's random oracle (RO-)KEM, which is used to form the α in each header. To simplify its analysis in isolation, we define the KEM separately from the rest of the packet format and prove that it satisfies a modified version of the IND-CCA property for KEMs (KEM-IND-CCA) as defined by Cramer and Shoup [10][14] where

[10]Note that using a PRP here means that every added layer of encryption is indistinguishable from a random permutation to an attacker. In particular, tagging one of the layers will completely destroy the payload after decryption.

[11]If the attacker controls the nymserver itself, it can simply see that the requested pseudonym does not exist.

[12]Note that there are other feasible mitigations for this attack, e.g., sending multiple reply headers ahead of time to make linking the missing reply header to the original sender more difficult for an attacker. However, our adaptation both completely prevents the attack and simplifies the Sphinx protocol while being very simple itself. We thus consider it the most appropriate fix for the problem.

[13]Messages are always padded to the full length of the PRP's message space.

[14]The property we use here is defined in Section 7.1.2.

the challenger outputs additional information that we require in our later onion property proofs. In those proofs, we will make use of our Sphinx-KEM-IND-CCA property in order to randomize the blinding factors and symmetric keys used in the challenge onion. In this section, we abbreviate the concatenation of the three random oracles h_ρ, h_μ, and h_π as h_* for legibility since they operate identically with regards to the KEM.

Definition 5 (Sphinx RO-KEM)
The Sphinx RO-KEM is a tuple of polynomial-time algorithms (KeyGen, Encap, Decap) with:

- Key generation:
  $$\textsc{KeyGen}(1^\kappa) := (PK = g^x, SK = x)$$
  with $x \xleftarrow{R} \mathbb{Z}_q^*$ and g as the public generator of the group $\mathcal{G}$.
- Encapsulation:
  $\textsc{Encap}(1^\kappa, PK = g^x) := ((h_*(PK^{x'}), h_b(g^{x'}, PK^{x'})), g^{x'})$
  for a random $x' \in \mathbb{Z}_q^*$, with h_* and h_b being the random oracles used to key the components of the Sphinx header.
- Decapsulation:
  $\textsc{Decap}(1^\kappa, SK = x, \alpha = g^{x'}) := (h_*(\alpha^{SK}), h_b(\alpha, \alpha^{SK}))$,
  where α is an encapsulation produced by Encap [12].

4.3.1 Sphinx-KEM-IND-CCA. The basic KEM-IND-CCA game as defined by Cramer and Shoup [10] is unfortunately insufficient for our later proofs, where we require information on the RO outputs used in other (non-challenge) layers. We thus define a modified Sphinx-KEM-IND-CCA that outputs all of the information required to build a Sphinx packet while embedding the KEM challenge at an adversary-chosen index.

Definition 6 (Sphinx-KEM-IND-CCA)

(1) The challenger chooses $(PK, SK) \leftarrow \textsc{KeyGen}(1^\kappa)$ and sends PK to the adversary.
(2) Oracle access: The adversary can submit requests to the decapsulation oracle O and the random oracles h_* and h_b.
(3) The adversary submits
   - $n-1$ public keys $y_0, \dots, y_{j-1}, y_{j+1}, \dots, y_{n-1}$ with $n < N$. These are the public keys for the non-honest relays on the "KEM's path",
   - and a position j with $0 \le j < n$.
(4) The challenger checks that the y_i are all distinct and valid public keys.
(5) The challenger creates the KEM challenge for the adversary by choosing a random $x' \in \mathbb{Z}_q^*$ and generating the first j encapsulations α_0 through α_{j-1} and secrets s_i like for a Sphinx header: $\alpha_i \leftarrow g^{x' b_0 \cdots b_{i-1}}$, $s_i \leftarrow y_i^{x' b_0 \cdots b_{i-1}}$, $b_i \leftarrow h_b(\alpha_i, s_i)$.
(6) The challenger sends the adversary its "auxiliary information":[15]
   - The first encapulation α_0,
   - the h_* outputs $h_*(s_0), \dots, h_*(s_{j-1})$,
   - and the blinding factors $b_0, \dots, b_{j-1}$.
(7) The challenger provides the adversary with the KEM challenge: It picks $b \in \{0,1\}$ at random. If $b = 0$, the challenger lets $b_j = h_b(\alpha_j, s_j)$ and gives the adversary $(\alpha_j, h_*(s_j), b_j)$. Otherwise, the adversary gets (α_j, r_1, b_j) for $r_1 \xleftarrow{R} \{0,1\}^{3\kappa}$ and $b_j \xleftarrow{R} \mathbb{Z}_q^*$. Finally, the challenger generates the rest of the KEM layers $\alpha_{j+1}, \dots, \alpha_n$ with $s_{j+1}, \dots, s_n$ and $b_{j+1}, \dots, b_n$ the same way as the previous layers (using the corresponding b_j) and gives the adversary $h_*(s_{j+1}), \dots, h_*(s_n)$ and $b_{j+1}, \dots, b_n$.
(8) Oracle access: The adversary gets access to the same O, h_*, and h_b oracles.
(9) The adversary submits its guess b' to the challenger.

Sphinx-KEM-IND-CCA is achieved if any PPT adversary $\mathcal{A}$ cannot guess $b' = b$ with a probability non-negligibly better than $\frac{1}{2}$.

4.3.2 Security. In order to show that the Sphinx RO-KEM satisfies our security property, we can perform a reduction proof to a Diffie-Hellman assumption. The original Sphinx definition uses the *Decisional Diffie-Hellman* (DDH) assumption for the Sphinx RO-KEM in order to show that the blinding factors and symmetric keys are indistinguishable from randomness for the adversary [12]. However, a more detailed analysis reveals that the DDH assumption is insufficient to prove the Sphinx RO-KEM secure: In a reduction from Sphinx-KEM-IND-CCA to the DDH assumption, the DDH attacker must simulate both the decapsulation oracle O as well as the random oracles h_* and h_b consistently for the Sphinx-KEM-IND-CCA attacker. Doing so correctly for adversary-chosen inputs involves identifying which encapsulations α and secrets s belong together. Since being able to do so efficiently would already break DDH, the reduction is not possible in this form. We use the *Gap Diffie-Hellman* assumption instead. It states that the CDH problem is hard even given an oracle that solves the DDH problem. It is generally assumed that the GDH assumption holds in the standard elliptic curve groups, which Sphinx already uses [24]. For our reduction, this means that we are reducing Sphinx-KEM-IND-CCA to the CDH problem, but our CDH attacker additionally receives a DDH oracle O_G. Using O_G, the CDH attacker can correctly identify matching secrets and encapsulations. It follows that:

Theorem 2. *The Sphinx RO-KEM satisfies Sphinx-KEM-IND-CCA under the GDH assumption.*

Proof Sketch. We use a Sphinx-KEM-IND-CCA attacker $\mathcal{A}$ on the Sphinx RO-KEM to construct a GDH attacker $\mathcal{B}^{O_G}$ with the DDH oracle O_G. $\mathcal{B}$ uses its CDH challenge (g, g^{x_1}, g^{x_2}) as a public key $PK = g^{x_1}$ and a challenge $\alpha = g^{x_2}$. After getting the challenge index j and the other public keys from $\mathcal{A}$, $\mathcal{B}$ sets $\alpha_j = \alpha$ and constructs the rest of the KEM's path by choosing random blinding factors b_i and h_* outputs for every layer, programming its choices into the ROs. $\mathcal{B}$ also randomly chooses a bit $b \in \{0,1\}$ to determine which scenario it simulates. The only difference between the two is whether the random oracle outputs on layer j are programmed into the ROs. $\mathcal{B}$ then simulates the oracles, keeping the outputs consistent using its DDH oracle O_G. In order for $\mathcal{A}$ to tell the scenarios apart, it must request h_* or h_b with $g^{x_1 x_2}$, allowing $\mathcal{B}$ to win the GDH game. For the full proof see Appendix F.

4.4 Sphinx Security Analysis

In order to prove that Sphinx securely realizes $\mathcal{F}_{RSOR}$, we show that it satisfies our new properties. For the sake of brevity, we only

[15]Note that this is all of the information required to build any layer of the Sphinx packet preceding the challenge layer j.

sketch the proof for $TLU^{\rightarrow}$ here along with short proof outlines for $SLU^{\leftarrow}$ and $STI^{\leftrightarrow}$. The full proofs for all three properties can be found in Appendix G.1, Appendix G.2 and Appendix G.3. RSOR-Correctness follows from inspection of the Sphinx scheme.

Theorem 3. *Sphinx satisfies $TLU^{\rightarrow}$ under the GDH assumption.*

Proof Sketch. To show that Sphinx satisfies $TLU^{\rightarrow}$, we prove that an adversary cannot distinguish the $b = 0$ scenario of the $TLU^{\rightarrow}$ game from the $b = 1$ scenario through a hybrid argument starting at $b = 0$ and ending at $b = 1$. We gradually move from the $b = 0$ scenario to the $b = 1$ scenario by constructing one hybrid after another. The first hybrid game is simply the $b = 0$ scenario. Each hybrid changes the challenge onion or the challenger's behavior in an indistinguishable way until the final hybrid is identical to the $b = 1$ scenario. We summarize the hybrids' construction here, see Appendix G.1 for details. For clarity, we also separate the proof into two cases: One where $j = n$ and the other where $j < n$. In the $TLU^{\rightarrow}$ game, j determines the index of the honest relay on the challenge onion's path.

Case 1 ($j = n$): In this case, the honest relay on the forward path is also the exit relay of the onion and thus also the sender of the reply onion. This case demands that the entire forward onion is replaced with an onion using the same forward path, but containing a random message and receiver as well as having an empty reply path.

To perform this replacement, we take advantage of the innermost layer of the PRP protecting the payload's contents: Since the adversary cannot tell what the contents of the payload are, we can replace the message, receiver, and reply header with a random message, a random receiver, and padding (i.e., an empty reply path) respectively.

The original reply header is still used to form the actual reply onion at the honest exit relay. We use the PRP's security and the implicit integrity check in the forward payload (via the zero padding) to show that the adversary does not notice this change and cannot manipulate the payload in order to to affect the reply header. With this, the forward onion given to the adversary is completely independent of the challenge onion and contains a random message, a random receiver, and an empty reply path. The reply header of the challenge onion is still used on the reply path. This is identical to the $b = 1$ scenario.

Case 2 ($j < n$): Now, we consider the case where the honest relay on the forward path is not the exit relay. This case is more complex since the forward path of our replacement onion is not the same as the forward path of the challenge onion. Our hybrids thus need to truncate the forward path of the challenge onion at the honest relay in order to move to the $b = 1$ scenario. In addition, our hybrids must also handle a potential tagging attack by the adversary correctly. Proceed as follows:

(1) First, we need to handle tagging attacks. If the adversary tags the challenge onion in the $b = 0$ scenario, the payload will be mangled by the PRP decryption step in the honest relay's processing. The key aspect of this mangling is that the PRP's output in that case is indistinguishable from a random bitstring of the same length. We take advantage of this fact in the first hybrids: Instead of processing the challenge payload normally at the honest relay, the hybrid checks whether the payload has been modified. If not, the next layer of the challenge onion with the correct next payload layer is output. If the payload has been modified (i.e., tagged), then the payload output is replaced with a random bitstring. This change reduces to the PRP's security at the honest relay and the reply receiver.
With the new payload handling, we have effectively decoupled the onion layers before and after the honest relay: Any adversary modification to the layers before the honest relay either results in a failure in processing (due to the MAC if the header is modified) or in the payload being replaced with randomness (if the payload is modified). We can thus change the contents of both the header and payload layers before the honest relay and replace the changed onion with the original challenge onion at the honest relay oracle without the adversary being able to "sneak" information through the honest relay. The contents of the payload layers before the honest relay are replaced with a random message, a random receiver, and an empty reply path like in the $j = n$ case.

(2) Next, we "detach" the layers of the challenge onion's header before the honest relay (which we will refer to as A) from the layers after it (referred to as B).
As a first step, the final layer of A is no longer processed at the honest relay. The first layer of B is always output as the next layer's header instead. We detach A from B in multiple steps: First, replace the innermost contents of A's header layers with the contents of a final Sphinx header layer such that the A layers are now formed as if the onion's path ended at the honest relay. We can do so since the PRG protects that header information until the honest relay. Second, the KEM keys used to build B are replaced with a new instance of the KEM that starts at the honest relay. We use the randomness of the blinding factor multiplied onto the exponent at the honest relay to secure this step. Finally, the padding in B, which still contains A's padding, is changed: The bits corresponding to A's padding are replaced with random bits in the last header layer of B. This is possible due to Sphinx's padding construction and the PRG's security. After these steps, the A header layers are completely independent of the B header layers.

(3) In a last step, we adjust B's padding and KEM construction so that the B layers are built like part of the complete challenge onion again. Now, A is an independent forward onion with a truncated path, an empty reply path, and a random message and receiver, while B is built like the original challenge onion. This corresponds to the $b = 1$ scenario of $TLU^{\rightarrow}$.

$SLU^{\leftarrow}$: The $SLU^{\leftarrow}$ proof works similarly to the $TLU^{\rightarrow}$ proof. First, the part of the challenge reply onion after the honest relay is "detached" from the first part. Then, the second part's header and payload contents are adapted into those of a forward onion.
$STI^{\leftrightarrow}$: For the $STI^{\leftrightarrow}$ proof, we have to truncate the forward and reply paths of the challenge onion to move from one scenario to the other. Truncating the forward path is like performing hybrid

$\mathcal{H}_9$ from the $TLU^{\rightarrow}$ proof and adjusting the padding accordingly. Truncating the reply path is analogous to hybrid $\mathcal{H}_7$.

Given that Sphinx satisfies each of the RSOR properties, it follows that

Theorem 4. *Nymserverless Sphinx securely realizes $\mathcal{F}_{RSOR}$ under the GDH assumption.*

5 DISCUSSION

In this section, we argue that the relaxation used in our ideal functionality is acceptable in practice under certain specific conditions.

5.1 Relaxed Security Requirements of $\mathcal{F}_{RSOR}$

We stress that $\mathcal{F}_{RSOR}$ still prevents all tagging attacks except for the malleability attack on the payload. Thus, if an adversary is able to link layers of an honest sender's onion that do not involve the exit relay, both $\mathcal{F}_{RSOR}$ and our properties are not achieved.

However, we also emphasize that the reduction in security due to allowing the malleability attack can be critical and RSOR protocols should only be used under two conditions:

(1) On the forward path, the link between a sender and its chosen exit relay must not leak any critical information about the sender's communication. Since the tagging attack lets the adversary learn sender-exit relay links, these must not contain information that would help the adversary break the protocol's privacy goals.
For example, the sender's choice of exit relay cannot depend on the receiver[16] or the message of the onion. Instead, (as in many protocols) the exit relays must be chosen uniformly at random or randomly according to their capacities. In particular, this means an RSOR protocol cannot be used in the integrated-system model, where the exit relay is identical to the receiver.
As another example, RSOR protocols also cannot use sessions visible to the exit relay. If they did, the adversary could tag one of the onions in the session to learn the sender-exit relay link and observe another onion from the same session at the exit relay to discover the receiver and message.
Several existing instantiations of Sphinx violate this condition. These include HORNET [8] and TARANET [9], as discovered by Kuhn et al. [21], but also Loopix [26], where the exit relays are long-term service providers chosen by receivers. Since Pudding [19] is built upon Loopix, it suffers from the same issue. PolySphinx [27], on the other hand, modifies Loopix to use random exit relays instead of fixed service providers, mitigating the vulnerability.

(2) Similarly, on the reply path, tagging a reply payload must not leak any information about the sender of the forward message (who is the reply receiver). Note that a tagging attack on the reply path will only be discovered by the honest reply receiver. Hence, it is crucial that whether the reply receiver received a tagged reply payload or a well-formed payload must not be visible to the adversary. If it was visible, a corrupted exit relay could tag the reply payload to link the sender and the receiver (which is known to the exit relay). As an example, if an RSOR packet format is used as part of a larger protocol, an honest sender receiving a reply message must not trigger any output to the adversary.

The examples listed for each condition are intended to provide an intuition for the aspects of OR protocol design that must be carefully considered when using $\mathcal{F}_{RSOR}$. We stress that these examples are by no means exhaustive. An in-depth analysis of when protocols satisfy our two generic conditions is interesting future work.

5.2 Using Sphinx in a Network

We want to give practical advice on the cases in which we consider the usage of Sphinx in its intended RSOR model a secure choice. First of all, Sphinx should only be used with the changes we apply in this paper. Precisely, one must include the fix for path padding (random bits instead of zero bits) as described in Section 2.4.2 and the Sphinx reply header has to be included in the forward payload to avoid attacks based on the nymserver (see Section 4.2). Finally, it is important to ensure that all of the conditions mentioned for security in Section 5.1 are met.

6 CONCLUSION

The widely-used Sphinx packet format has thus far lacked a suitable analytical framework as well as a security proof. With this paper, we aim to rectify this. We provide the privacy formalization for repliable service-model OR protocols with our ideal functionality $\mathcal{F}_{RSOR}$ and the four new onion properties RSOR-Correctness, Tagging-Forward Layer Unlinkability, RSOR-Backwards Layer Unlinkability, and RSOR-Tail Indistinguishability, which we prove imply $\mathcal{F}_{RSOR}$. Our formalization pays close attention to consider all the new edge cases of the service model and to relax the security in an acceptable way to allow for payload malleability.

To prove Sphinx's security, we change the cryptographic group assumption for the Sphinx scheme from DDH to GDH. Additionally, we realize that a security proof is not possible in the presence of the nymserver. We propose to include the reply header in the forward payload instead. With our formal groundwork, we are then able to prove this adapted version of Sphinx secure according to $\mathcal{F}_{RSOR}$. To our knowledge, we are the first to provide a security proof for Sphinx at our level of detail. We thereby ensure that the OR and mix networks that base their protocols on the Sphinx packet format can rely on a thoroughly-analyzed foundation again. Considering that Sphinx is currently actively used in real-world mix networks like Nym, this is a very important step.

Of course, there is still progress in OR, mix networks, and packet formats to be expected in future works. Authors of new OR and mix network protocols benefit from our investigation of the criteria for using Sphinx in a secure way to decide whether or not to base their protocols on Sphinx. In addition, future works on OR and mix network packet formats profit from our formalization in the service model, especially by using our new onion properties to build their algorithms and prove their privacy.

[16]This might be considered in order to have an exit relay that is topologically close to the receiver.

ACKNOWLEDGMENTS

We thank Dennis Hofheinz for pointing us towards the GDH assumption. This work has been funded by the Helmholtz Association through the KASTEL Security Research Labs (HGF Topic 46.23), and by funding of the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy — EXC 2050/1 — Project ID 390696704 — Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden.

REFERENCES

[1] 2024. Lightning Network Specifications BOLT #4. Git version control. 04-onion-routing.md in the Lightning Bolts repository - available at https://github.com/lightning/bolts/blob/master/04-onion-routing.md.

[2] Megumi Ando and Anna Lysyanskaya. 2021. Cryptographic Shallots: A Formal Treatment of Repliable Onion Encryption. In *Theory of Cryptography*, Kobbi Nissim and Brent Waters (Eds.). Springer International Publishing, Cham, 188–221.

[3] Filipe Beato, Kimmo Halunen, and Bart Mennink. 2016. Improving the Sphinx Mix Network. In *Cryptology and Network Security*, Sara Foresti and Giuseppe Persiano (Eds.). Springer International Publishing, Cham, 681–691.

[4] Mihir Bellare and Phillip Rogaway. 2005. The Birthday Problem. In *Introduction to Modern Cryptography*. 273–274. https://web.cs.ucdavis.edu/~rogaway/classes/227/spring05/book/main.pdf

[5] Jan Camenisch and Anna Lysyanskaya. 2005. A Formal Treatment of Onion Routing. In *Advances in Cryptology – CRYPTO 2005*, Victor Shoup (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–187.

[6] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, Newport Beach, CA, USA, 136–145. https://doi.org/10.1109/SFCS.2001.959888

[7] David L Chaum. 1981. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM* 24, 2 (1981), 84–90.

[8] Chen Chen, Daniele E. Asoni, David Barrera, George Danezis, and Adrain Perrig. 2015. HORNET: High-Speed Onion Routing at the Network Layer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1441–1454. https://doi.org/10.1145/2810103.2813628

[9] Chen Chen, Daniele E. Asoni, Adrian Perrig, David Barrera, George Danezis, and Carmela Troncoso. 2018. TARANET: Traffic-Analysis Resistant Anonymity at the Network Layer. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, Newport Beach, CA, USA, 137–152. https://doi.org/10.1109/EuroSP.2018.00018

[10] Ronald Cramer and Victor Shoup. 2004. Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack. *SIAM J. Comput.* 33, 1 (Jan 2004), 167–226. https://doi.org/10.1137/S0097539702403773

[11] George Danezis, Roger Dingledine, and Nick Mathewson. 2003. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *2003 Symposium on Security and Privacy, 2003.* IEEE Computer Society, Newport Beach, CA, USA, 2–15. https://doi.org/10.1109/SECPRI.2003.1199323

[12] George Danezis and Ian Goldberg. 2009. Sphinx: A Compact and Provably Secure Mix Format. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE Computer Society, Newport Beach, CA, USA, 269–282. https://doi.org/10.1109/SP.2009.15

[13] Claudia Diaz, Harry Halpin, and Aggelos Kiayias. 2021. The Nym Network. (2021). https://nymtech.net/nym-whitepaper.pdf

[14] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router

[15] Ian Goldberg. 2016. SphinxClient.py - Padding Fix. Git version control system. Python source file SphinxClient.py in the UCL-InfoSec Sphinx repository - available at https://github.com/UCL-InfoSec/sphinx/blob/c05b7034eaffd8f98454e0619b0b1548a9fa0f42/SphinxClient.py#L67.

[16] David Goldschlag, Michael Reed, and Paul Syverson. 1996. Hiding Routing Information. In *Proceedings of the First International Workshop on Information Hiding*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/3-540-61996-8_37

[17] Daniel Hugenroth and Alastair R. Beresford. 2023. Powering Privacy: On the Energy Demand and Feasibility of Anonymity Networks on Smartphones. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5431–5448. https://www.usenix.org/conference/usenixsecurity23/presentation/hugenroth

[18] Daniel Hugenroth, Martin Kleppmann, and Alastair R. Beresford. 2021. Rollercoaster: An Efficient Group-Multicast Scheme for Mix Networks. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, San Diego, CA, 3433–3450. https://www.usenix.org/conference/usenixsecurity21/presentation/hugenroth

[19] Ceren Kocaoğullar, Daniel Hugenroth, Martin Kleppmann, and Alastair R. Beresford. 2023. Pudding: Private User Discovery in Anonymity Networks. (2023). arXiv:2311.10825 [cs.CR]

[20] Chelsea H Komlo, Nick Mathewson, and Ian Goldberg. 2020. Walking Onions: Scaling Anonymity Networks while Protecting Users. In *Proceedings of the 29th USENIX Conference on Security Symposium*. USENIX Association, San Diego, CA, USA, 1003–1020.

[21] Christiane Kuhn, Martin Beck, and Thorsten Strufe. 2020. Breaking and (Partially) Fixing Provably Secure Onion Routing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Newport Beach, CA, USA, 168–185. https://doi.org/10.1109/SP40000.2020.00039

[22] Christiane Kuhn, Dennis Hofheinz, Andy Rupp, and Thorsten Strufe. 2021. Onion Routing with Replies. In *Advances in Cryptology – ASIACRYPT 2021*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 573–604.

[23] Bodo Möller. 2003. Provably Secure Public-Key Encryption for Length-Preserving Chaumian Mixes. In *Topics in Cryptology — CT-RSA 2003*, Marc Joye (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 244–262.

[24] Tatsuaki Okamoto and David Pointcheval. 2001. The Gap-Problems: A New Class of Problems for the Security of Cryptographic Schemes. In *Public Key Cryptography*, Kwangjo Kim (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 104–118.

[25] Lennart Oldenburg, Marc Juarez, Enrique Argones Rúa, and Claudia Diaz. 2024. MixMatch: Flow Matching for Mixnet Traffic. *Proceedings on Privacy Enhancing Technologies* 2024, 2 (2024). https://lirias.kuleuven.be/4133083

[26] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. 2017. The Loopix Anonymity System. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, San Diego, CA, 1199–1216.

[27] Daniel Schadt, Christoph Coijanovic, Christiane Weis, and Thorsten Strufe. 2024. PolySphinx: Extending the Sphinx Mix Format With Better Multicast Support. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/SP54263.2024.00044

A IDEAL FUNCTIONALITY

Modifications to Kuhn et al.'s $\mathcal{F}_R$ are highlighted in this style in the pseudocode.

Algorithm 1 Ideal Functionality $\mathcal{F}_{RSOR}$ (1)

```
▷ Data structures:
Bad: Set of corrupted relays and receivers
$L_o$: List of onions processed by adversarial relays
$B_i$: List of onions held by relay $P_i$
$B_i^r$: List of receiver replies held by relay $P_i$
$L_{tag}$: List of messages tagged by the adversary
$Back$: Map from $tid$s to reply paths and forward IDs
$ID_{fwd}$: Map from a reply onion ID to a forward onion ID
$Rep_i$: Map of reply identifiers to $tid$s at exit relay $P_i$

On message ProcessNewOnion($R, m, \mathcal{P}, \mathcal{P}^{\leftarrow}$) from $\mathcal{Z}$ or $\mathcal{S}$ via $P_s$
    if $|\mathcal{P}| > N$ or $|\mathcal{P}^{\leftarrow}| > N$ then reject
    else
        $sid \leftarrow^R$ session ID
        $O \leftarrow (sid, P_s, R, m, \mathcal{P}, 0, f)$
        Out.Cor.Sender($P_s, sid, R, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, start, f$)
        Proc.NextStep($O$)

procedure ProcessNewReply($m, tid$)
    if $(tid, \dots) \notin Back$ then reject
    else
        $(tid, P_s, \mathcal{P}, \mathcal{P}^{\leftarrow}, sid', \_) \leftarrow Back$
        $sid \leftarrow^R$ session ID
        Store $(sid, sid')$ in $ID_{fwd}$
        $O \leftarrow (sid, P_i, P_s, m, \mathcal{P}^{\leftarrow}, (), 0, b)$
        Out.Cor.Sender($P_i, sid, P_s, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, start, b$)
        Proc.NextStep($O$)

On message DeliverOnion($tid$) from $\mathcal{S}$
    if $(tid, \_, \_) \in L_o$ then
        $(tid, O = (sid, P_s, R/P_r, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d), j) \leftarrow L_o$
        $O \leftarrow (sid, P_s, R/P_r, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, j, d)$
        if $d = b$ and $j = |\mathcal{P}|$ then
            if $m \neq \perp$ and $O \notin L_{tag}$ then
                Send($P_r$, "Message $m$ received as reply")
                ▷ Not forwarded to $\mathcal{Z}$
        else
            $tid' \leftarrow^R$ temporary ID
            Send($P_{o_j}$, "$tid'$ received")
            Store $(tid', O)$ in $B_{o_j}$

On message ForwardOnion($tid'$) from $\mathcal{Z}$ or $\mathcal{S}$ via $P_i$
    if $(tid', \_) \in B_i$ then
        Pop $(tid', O)$ from $B_i$
        Proc.NextStep($O$)
    else if $(tid', \_) \in B_i^r$ then
        Pop $(tid', m, tid)$ from $B_i^r$
        ProcessNewReply($m, tid$)

On message Tag($tid$) from $\mathcal{S}$
    if $(tid, \_, \_) \in L_o$ then
        Retrieve $(tid, O, \_)$ from $L_o$
        Store $O$ in $L_{tag}$

procedure Out.Cor.Sender($P_s, sid, R/P_r, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, tid, d$)
    if $d = f$ and $P_s \in$ Bad then
        Send($\mathcal{S}$, "$tid$ is from $P_s$ with $sid, R, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, d$")
    else if $d = b$ and $P_r \in$ Bad then
        Send($\mathcal{S}$, "$tid$ is reply from $P_s$ with $sid, P_r, m, \mathcal{P}, \mathcal{P}^{\leftarrow}$,
                $d$, replying to onion from $P_r$ with $ID_{fwd}(sid)$")
```

Algorithm 2 Ideal Functionality $\mathcal{F}_{RSOR}$ (2)

```
procedure Proc.ToRelay($O = (sid, P_s, R/P_r, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d)$)
    $P_{o_j} \leftarrow P_{o_k}$ 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 $(P_{o_{i+1}}, \dots, P_{o_{j-1}})$")
    Send($P_{o_i}$, "Sent onion to $P_{o_{i+1}}$")
    Out.Cor.Sender($P_s, sid, R, m, n, \mathcal{P}, tid, d$)
    if $d = b$ and $i = 0$ then
        Send($\mathcal{S}$, "$tid$ belongs to $sid$")
    Add $(tid, O, j)$ to $L_o$

procedure SetupReply($O = (sid, P_s, R, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d), rid$)
    $tid \leftarrow^R$ temporary ID
    Store $(tid, P_s, \mathcal{P}, \mathcal{P}^{\leftarrow}, P_{o_i}, sid)$ in $Back$
    if $i = |\mathcal{P}|$ then
        Store $(rid, tid)$ in $Rep_{o_i}$
        Send($\mathcal{S}$, "Reply with reply ID $rid$")
    else
        $\mathcal{P}_1^{\leftarrow} \leftarrow$ prefix of $\mathcal{P}^{\leftarrow}$ up to (including) the first honest relay
        Send($\mathcal{S}$, "Reply with $tid$, reply path begins with $\mathcal{P}_1^{\leftarrow}$")

procedure LeakMessage($O = (sid, P_s, R, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d)$)
    if $m = \perp$ then return
    Out.Cor.Sender($P_s, sid, R, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, end, d$)
    if $\mathcal{P} \neq ()$ then
        $rid \leftarrow^R$ temporary ID
        SetupReply($O, rid$)
    if $i = |\mathcal{P}|$ then Send($P_{o_i}$, "Sent message to $R$")
    else Send($P_{o_i}$, "Sent onion to $P_{o_{i+1}}$")
    Send($\mathcal{S}$, "$P_{o_i}$ sends onion with message $m$ to $R$
            via $(P_{o_{i+1}}, \dots, P_{o_n})$")

On message DeliverMessage($m, rid, R$) from $\mathcal{S}$
    Send($R$, "Message $m$ received")
    if $rid \neq \perp$ then
        Send($R$, "Message is repliable with $rid$")

On message InitiateReply($m, rid$) from $\mathcal{Z}$ or $\mathcal{S}$ via $R$
    Send($\mathcal{S}$, "$R$ replies to $rid$ with message $m$")

▷ $P_i$ creates an onion from $R$'s reply request
On message DeliverReply($P_i, m, rid$) from $\mathcal{S}$
    Send($P_i$, "Reply $(m, rid)$ received")
    if $(rid, \_) \in Rep_i$ then
        $(rid, tid) \leftarrow Rep_i$
        $tid' \leftarrow^R$ temporary ID
        Store $(tid', m, tid)$ in $B_i^r$
        Send($P_i$, "Send reply onion with $tid'$")

▷ $\mathcal{S}$ uses a $tid$ ID to bypass replying via an exit relay
On message BypassReply($m, tid$) from $\mathcal{S}$ via $P_i$
    if $(tid, \dots) \in Back$ then
        ProcessNewReply($m, tid$)

procedure LeakReply($O = (sid, P_s, P_r, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d)$)
    Send($\mathcal{S}$, "$P_{o_i}$ sends reply $tid$ with message $m$ to $P_r$
            via $(P_{o_{i+1}}, \dots, P_{o_{n-1}})$")
    Send($P_{o_i}$, "Sent onion to $P_{o_{i+1}}$")
    Out.Cor.Sender($P_s, sid, P_r, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, tid, b$)

procedure Proc.NextStep($O = (sid, P_s, R/P_r, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d)$)
    if $\forall j > i : P_{o_j} \in$ Bad or $i = |\mathcal{P}|$ then
        if $O \in L_{tag}$ then
            Out.Cor.Sender($P_s, sid, R/P_r, m, \mathcal{P}, tagged, d$)
            if $i < n$ then
                Send($\mathcal{S}$, "$P_{o_i}$ sends tagged via $(P_{o_{i+1}}, \dots, P_{o_n})$")
                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
            if $d = f$ then LeakMessage($O$)
            else LeakReply($O$)
    else ProcessToRelay($O$)
```

B RSOR ONION PROPERTIES

The following definitions are copied from Kuhn et al. [22]. Our modifications are given in this style. We also abbreviate RecognizeOnion as ROnion for the sake of formatting.

B.1 RSOR-Correctness

RSOR-Correctness is defined as:[17]

Let $(G, \text{FormOnion}, \text{ProcOnion}, \text{FormReply})$ be a RSOR scheme with maximal path length N and polynomial $|\mathcal{N}|$ and $|D|$. Then for all $n, n^{\leftarrow} < N, \lambda \in \mathbb{N}$, all choices of the public parameter p, all choices of randomness $\mathcal{R}$, all choices of receiver R, all choices of the paths $\mathcal{P} = (P_1, \dots, P_n)$ and $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \dots, P_n^{\leftarrow})$, all keypairs $(PK_i^{(\leftarrow)}, SK_i^{(\leftarrow)})$ generated by $G(1^\lambda, p, P_i^{(\leftarrow)})$, all messages $m, m^{\leftarrow}$, all possible choices of internal randomness used by ProcOnion and FormReply, the following needs to hold:

Correctness of forward path.

$$Q_i = P_i, \text{ for } 1 \le i \le n \text{ and } Q_1 := P_1,$$

$$O_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}),$$

$$(O_{i+1}, Q_{i+1}) \leftarrow \text{ProcOnion}(SK_i, O_i, Q_i).$$

Correctness of request reception.

$$(m, R) = \text{ProcOnion}(SK_n, O_n, P_n).$$

Correctness of backward path.

$$Q_i^{\leftarrow} = P_i^{\leftarrow} \text{ for } 1 \le i \le n$$

$$\text{and } (O_1^{\leftarrow}, Q_1^{\leftarrow}) \leftarrow \text{FormReply}(m^{\leftarrow}, O_n, P_n, SK_n),$$

$$(O_{i+1}^{\leftarrow}, Q_{i+1}^{\leftarrow}) \leftarrow \text{ProcOnion}(SK_i^{\leftarrow}, O_i^{\leftarrow}, Q_i^{\leftarrow}).$$

Correctness of reply reception.

$$(m^{\leftarrow}, \perp) = \text{ProcOnion}(SK_{n^{\leftarrow}}^{\leftarrow}, O_{n^{\leftarrow}}^{\leftarrow}, P_{n^{\leftarrow}}^{\leftarrow}).$$

B.2 RSOR-Backw. Layer Unlinkability ($SLU^{\leftarrow}$)

RSOR-Backward Layer Unlinkability is defined as:

(1) The adversary receives the router names P_H, P_s and challenge public keys PK_S, PK_H, chosen by the challenger as $(PK_H, SK_H) \leftarrow G(1^\lambda, p, P_H)$ and $(PK_S, SK_S) \leftarrow G(1^\lambda, p, P_s)$.

(2) Oracle access: The adversary may submit any number of **Proc** and **Reply** requests for P_H or P_s to the challenger. For any $\textbf{Proc}(P_H, O)$, the challenger checks whether η is on the η_H-list. If it is not on the list, it sends the output of $\text{ProcOnion}(SK_H, O, P_H)$, stores η on the η_H-list and O on the O_H-list. For any $\textbf{Reply}(P_H, O, m)$, the challenger checks if O is on the O_H-list and if so, the challenger sends $\text{FormReply}(m, O, P_H, SK_H)$ to the adversary. (Similar for requests on P_s with the η_S-list).

(3) The adversary submits a message m, a receiver R, a position $j^{\leftarrow}$ with $0 \le j^{\leftarrow} \le n^{\leftarrow}$, a path $\mathcal{P} = (P_1, \dots, P_n)$ where $P_n = P_H$ if $j^{\leftarrow} = 0$, a path $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \dots, P_{j^{\leftarrow}}^{\leftarrow}, \dots, P_{n^{\leftarrow}}^{\leftarrow} = P_s)$ with the honest relay P_H at backward position $j^{\leftarrow}$ if $1 \le j^{\leftarrow} \le n^{\leftarrow}$, and the second honest relay P_s at position $n^{\leftarrow}$, and public keys for all relays PK_i ($1 \le i \le n$ for the relays on the path and $n < i$ for the other relays).

(4) The challenger checks that the chosen paths are acyclic, the router names and public keys are valid and that the same key is chosen if the router names are equal, and if so, sets $PK_{j^{\leftarrow}}^{\leftarrow} = PK_H$ (resp. PK_n if $j^{\leftarrow} = 0$), $PK_{n^{\leftarrow}}^{\leftarrow} = PK_S$ and sets bit b at random.

(5) The challenger creates the onion O_1 with the adversary's input choice and honestly chosen randomness $\mathcal{R}$:
$\text{FormOnion}(1, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$
and sends O_1 to the adversary.

(6) The adversary gets oracle access as in step 2) except if:
 (a) The request is...
  - for $j^{\leftarrow} > 0$: $\textbf{Proc}(P_H, O)$ with $\text{ROnion}(n + j^{\leftarrow}, O, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}) = True$, η is not on the η_H-list and $\text{ProcOnion}(SK_H, O, P_H) \neq (\perp, \perp)$: stores η on the η_H and O on the O_H-list and ...
  - for $j^{\leftarrow} = 0$: $\textbf{Reply}(P_H, O, m^{\leftarrow})$ with $\text{ROnion}(n, O, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}) = True$, O is on the O_H-list and no onion with this η has been replied to before and $\text{FormReply}(m^{\leftarrow}, O, P_H, SK_H) \neq (\perp, \perp)$ ...

 ...then: The challenger picks the rest of the return path $\bar{\mathcal{P}} = (P_{j^{\leftarrow}+1}^{\leftarrow}, \dots, P_{n^{\leftarrow}}^{\leftarrow})$, an empty backward path $\bar{\mathcal{P}}^{\leftarrow} = ()$, and a random message $\bar{m}$, another honestly chosen randomness $\bar{\mathcal{R}}$, an honestly chosen random receiver $\bar{R}$, and generates an onion $\bar{O}_1$:
 $\text{FormOnion}(1, \bar{\mathcal{R}}, \bar{m}, \bar{R}, \bar{\mathcal{P}}, \bar{\mathcal{P}}^{\leftarrow}, PK_{\bar{\mathcal{P}}}, PK_{\bar{\mathcal{P}}^{\leftarrow}})$
  - If $b = 0$, the challenger calculates
$$(O_{j^{\leftarrow}+1}, P_{j^{\leftarrow}+1}^{\leftarrow}) = \begin{cases} \text{ProcOnion}(SK_H, O, P_{j^{\leftarrow}}^{\leftarrow}) & , j^{\leftarrow} > 0, \\ \text{FormReply}(m^{\leftarrow}, O, P_{j^{\leftarrow}}^{\leftarrow}, SK_H) & , j^{\leftarrow} = 0 \end{cases}$$
  and gives $O_{j^{\leftarrow}+1}$ for $P_{j^{\leftarrow}+1}^{\leftarrow}$ to the adversary.
  - Otherwise, the challenger gives $\bar{O}_1$ for $P_{j^{\leftarrow}+1}^{\leftarrow}$ to the adversary.

 (b) $\textbf{Proc}(P_s, O)$ with O being the challenge onion as processed for the final receiver on the backward path, i.e.:
  - for $b = 0$: $\text{ROnion}(n + n^{\leftarrow}, O, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}) = True$
  - for $b = 1$: $\text{ROnion}(n^{\leftarrow} - j^{\leftarrow}, O, \bar{\mathcal{R}}, \bar{m}, \bar{R}, \bar{\mathcal{P}}, \bar{\mathcal{P}}^{\leftarrow}, PK_{\bar{\mathcal{P}}}, PK_{\bar{\mathcal{P}}^{\leftarrow}}) = True$

 ...then the challenger outputs nothing.

(7) The adversary produces guess b'.

$SLU^{\leftarrow}$ is achieved if any PPT adversary $\mathcal{A}$ cannot guess $b' = b$ with a probability non-negligibly better than $\frac{1}{2}$.

B.3 RSOR-Tail Indistinguishability ($STI^{\leftrightarrow}$)

RSOR-Tail Indistinguishability is defined as:

(1) The adversary receives the router names P_H, $P_H^{\leftarrow}$, P_s, and challenge public keys PK_S, PK_H, $PK_H^{\leftarrow}$, which are chosen by the challenger by letting $(PK_H, SK_H) \leftarrow G(1^\lambda, p, P_H)$, $(PK_H^{\leftarrow}, SK_H^{\leftarrow}) \leftarrow G(1^\lambda, p, P_H^{\leftarrow})$, $(PK_S, SK_S) \leftarrow G(1^\lambda, p, P_s)$.

(2) Oracle access: The adversary may submit any number of **Proc** and **Reply** requests for P_H, $P_H^{\leftarrow}$, or P_s to the challenger. For any $\textbf{Proc}(P_H, O)$, the challenger checks whether η is on the η_H-list. If it is not on the list, it sends the output of $\text{ProcOnion}(SK_H, O, P_H)$, stores η on the η_H-list and O on the O_H-list. For any $\textbf{Reply}(P_H, O, m)$, the challenger checks if O is on the O_H-list and if so, the challenger sends

[17]This definition was originally proposed by Camenisch and Lysyanskaya [5] in a slightly different format.

FormReply(m, O, P_H, SK_H) to the adversary. (Similar for requests on $P_H^{\leftarrow}, P_s$).

(3) The adversary submits a message m, a receiver R, a path $\mathcal{P} = (P_1, \ldots, P_j, \ldots, P_n)$ with the honest relay P_H or $P_H^{\leftarrow}$ at position $j, 0 \leq j < n$, a path $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_n^{\leftarrow})$ with the honest relay $P_H^{\leftarrow}$ at position $1 \leq j^{\leftarrow} \leq n^{\leftarrow}$ and public keys for all relays PK_i $(1 \leq i \leq n^{\leftarrow}$ for the relays on the path and $n < i$ for the other relays).

(4) The challenger checks that the given paths are acyclic, the router names and public keys are valid and that the same key is chosen if the router names are equal, and if so, sets $PK_j = PK_H$ (or $PK_j = PK_H^{\leftarrow}$, if the adversary chose $P_H^{\leftarrow}$ at this position as well), $PK_{j^{\leftarrow}}^{\leftarrow} = PK_H^{\leftarrow}$, $PK_{n^{\leftarrow}}^{\leftarrow} = PK_S$ and sets bit b at random.

(5) The challenger creates the onion O_{j+1} with the adversary's input choice and honestly chosen randomness $\mathcal{R}$:
$$\text{FormOnion}(j+1, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$$
and a replacement onion $\bar{O}_1$ with the path from the honest relay P_H to the corrupted exit relay $\bar{\mathcal{P}} = (P_{j+1}, \ldots, P_n)$ and the backward path ending at $j^{\leftarrow}$: $\bar{\mathcal{P}}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{j^{\leftarrow}}^{\leftarrow})$; and another honestly chosen randomness $\bar{\mathcal{R}}$:
$$\text{FormOnion}(1, \bar{\mathcal{R}}, m, R, \bar{\mathcal{P}}, \bar{\mathcal{P}}^{\leftarrow}, PK_{\bar{\mathcal{P}}}, PK_{\bar{\mathcal{P}}^{\leftarrow}})$$

(6) If $b = 0$: The challenger sends O_{j+1} to the adversary. Otherwise: The challenger sends $\bar{O}_1$ to the adversary.

(7) Oracle access: the challenger processes all requests as in step 2) except for…
…**Proc**$(P_H^{\leftarrow}, O)$ with O being the challenge onion as processed for the honest relay on the backward path, i.e.:
- for $b = 0$:
$$\text{ROnion}(n + j^{\leftarrow}, O, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}) = True$$
- for $b = 1$:
$$\text{ROnion}((n-j) + j^{\leftarrow}, O, \bar{\mathcal{R}}, m, R, \bar{\mathcal{P}}, \bar{\mathcal{P}}^{\leftarrow}, PK_{\bar{\mathcal{P}}}, PK_{\bar{\mathcal{P}}^{\leftarrow}}) = True$$

…then the challenger outputs nothing.

(8) The adversary produces guess b'.

$STI^{\leftrightarrow}$ is achieved if any PPT adversary $\mathcal{A}$ cannot guess $b' = b$ with a probability non-negligibly better than $\frac{1}{2}$.

C SECURE RSOR DEFINITIONS

The following definition is adapted from Kuhn et al.'s definition for repliable integrated-system-model OR (as shown in Section 2.3). Our changes compared to their definitions are shown in this style, and [. . .] indicates a deletion.

Definition 7

An RSOR protocol is secure in the $\mathcal{F}_{PKI}$-$\mathcal{F}_{SC}$-hybrid model if and only if it is built on a secure RSOR scheme according to Definition 3 and operates the following way:

- *Setup:* Each relay P_i generates a keypair $(SK_i, PK_i) \leftarrow G(1^{\lambda})$ and publishes PK_i by using $\mathcal{F}_{PKI}$.
- *Sending a message:* If P_s wants to send $m \in \mathcal{M}$ to R over the path $\mathcal{P} = (P_1, \ldots, P_n)$ with $n \leq N$ and wants to allow a reply over the path $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{n^{\leftarrow}}^{\leftarrow})$ with $n^{\leftarrow} \leq N$ and $P_{n^{\leftarrow}}^{\leftarrow} = P_s$, it chooses a randomness $\mathcal{R}$ and calculates
$$O_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}^{\leftarrow})$$
and sends O_1 to P_1 using $\mathcal{F}_{SC}$.
- *Processing an onion:* P_i receives O_i and runs
$$(O_j, P_j) \leftarrow \text{ProcOnion}(SK_i, O_i, P_i).$$
If $P_j = \perp$, P_i outputs "Received $m = O_j$ as a reply" if $O_j \neq \perp$ [. . .]. If $P_j \neq \perp$, P_j is a valid relay name or receiver. and P_i generates a random tid and stores $(tid, (O_j, P_j))$ in its outgoing buffer and notifies the environment about tid.
- *Sending an onion:* When the environment instructs P_i to forward tid, P_i looks up tid in its buffer. If P_i does not find such an entry, it aborts. Otherwise, it finds $(tid, (O_j, P_j))$. If P_j is a relay name, it sends O_j to P_j using $\mathcal{F}_{SC}$. If $P_j = R$ for a receiver R, P_i checks whether FormReply$(m^{\leftarrow}, O_i, P_i, SK_i) \neq \perp$ for an arbitrary $m^{\leftarrow}$. If so, P_i chooses a random rid, stores (rid, O_i) in its reply buffer, and sends (O_j, rid) to P_j without $\mathcal{F}_{SC}$. If not, P_i sends $(O_j, \perp)$ to P_j without $\mathcal{F}_{SC}$.
- *Receiving a message:* When a receiver gets a message (m, rid) from a relay P_i, it outputs "Received message m from P_i" to the environment. If $rid \neq \perp$, it additonally outputs "It is repliable with rid".
- *Sending a reply message:* When the environment instructs R to reply to P_i with rid and $m^{\leftarrow}$, R sends $(m^{\leftarrow}, rid)$ to P_i without $\mathcal{F}_{SC}$.
- *Creating a reply onion:* When P_r receives $(m^{\leftarrow}, rid)$ from a receiver R, P_r looks up rid in its reply buffer. If there is no entry with rid, P_r stops. If it finds (rid, O) in its buffer, it calculates
$$(O_1^{\leftarrow}, P_1^{\leftarrow}) \leftarrow \text{FormReply}(m^{\leftarrow}, O, P_r, SK_r)$$
and sends $O_1^{\leftarrow}$ to $P_1^{\leftarrow}$ using $\mathcal{F}_{SC}$.

D $\mathcal{F}_{RSOR}$ UC REALIZATION PROOF

Theorem 1. *An RSOR protocol according to Section 3.1.3 with a PRP-encrypted payload that satisfies RSOR-Correctness, Tagging-Forward Layer Unlinkability, RSOR-Backward Layer Unlinkability, and RSOR-Tail Indistinguishability securely realizes $\mathcal{F}_{RSOR}$.*

This proof is taken from Kuhn et al. [22] and modified to fit RSOR. Our modifications are given in this style. Some parts of the proof also make use of elements from Kuhn et al.'s proof of UC-realization in [21] — these parts are explicitly marked with citations.

Proof. For UC-realization, we show that every attack on the real world protocol Π can be simulated by an ideal world attack without the environment being able to distinguish those. We first describe the simulator S. Then we show indistinguishability of the environment's view in the real and ideal world.

Constructing Simulator $\mathcal{S}$. $\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}_{RSOR}$ as the ideal world adversary, and simulates the real-world honest parties for the real world adversary $\mathcal{A}$. All outputs $\mathcal{A}$ does are forwarded to the environment by $\mathcal{S}$. First, $\mathcal{S}$ carries out the trusted set-up stage: It generates public and private key pairs for all the real-world honest parties. $\mathcal{S}$ then sends the respective public keys to $\mathcal{A}$ and receives the real world corrupted parties' public keys from $\mathcal{A}$. The simulator $\mathcal{S}$ maintains four internal data structures:

- The r-list consisting of tuples of the form (*onion, prevRelay, nextRelay, tid, a*). Each entry in this list corresponds to a stage in processing an onion that belongs to a communication of an honest sender or an onion that was injected into $\mathcal{F}_{RSOR}$ by $\mathcal{S}$. By "stage", we mean that the next action to this onion is adversarial (i.e., it is sent over a link or processed by an adversarial router).

- The O-list containing onions sent by corrupted senders together with the information about the communication (*onionlist, path, currentPosition, information).*
- The *Reply*-list containing reply information together with the forward id for communications with a corrupted sender (sid_{fwd}, reply information).
- The C-list containing reply information with the tid for communications with an honest sender ($P_i, reply, tid$).

$\mathcal{S}$'s behavior on a message from $\mathcal{F}_{RSOR}$: *In case the received output belongs to an adversarial sender's communication:*

Case I: "*start* belongs to reply from P_s with $sid, P_r, m, n, \mathcal{P}, \mathcal{P}^{\leftarrow}, d$, replying to onion from P_r with sid"; an honest relay is replying to an onion of a corrupted sender. $\mathcal{S}$ knows that the next output "Onion tid in direction d from ..." includes the first part of this backward path, that he chose to consist of the correct sequence of honest relays potentially followed by one adversarial relay [...]. To construct the right real world reply onion, $\mathcal{S}$ looks up the reply information (sid, *replyinfo*) for this sid in the *Reply*-list and uses the information to construct the reply onion

$$(O_1, P_1) \leftarrow \text{FormReply}(m, replyinfo, P_s, SK_s)$$

followed by the next onion layers as far as $\mathcal{S}$ can process them with the secret keys of the honest relays. Since the sender of the forward onion is corrupted, there must be at least one adversarial relay on the reply path of the onion, so $\mathcal{S}$ will be able to process the onion up to the pair (O', P') with an adversarial P'. This results in a list of onions $O = (O_1, \ldots, O_{last})$ and a list of relays $\mathcal{P} = (P_1, \ldots, P_{last})$. $\mathcal{P}$ is identical to the reply path of the onion that was already created in the ideal world. $\mathcal{S}$ sends O_1 to P_1, if P_1 is adversarial, or to $\mathcal{A}$'s party representing the link between P_s and P_1, if P_1 is honest. (Note that P_s cannot be adversarial for this output as then both sender and receiver would be corrupted, which only activates cases **VIIIb** and **II** (as it works without including any reply onion from the view of the ideal world). If the next relay P_1 on the reply onion's path is honest, then $\mathcal{S}$ needs to be able to associate that onion layer and the ones following it in the real world with the ideal-world onion as the layers are processed and sent along the honest relays. Conversely, if P_1 is adversarial, then the onion leaves $\mathcal{S}$'s control after this case.[18]

(1) If the first relay P_1 is an honest relay, $\mathcal{S}$ adds the tuple $(O, \mathcal{P}, 0, (P_s, sid, P', m, \mathcal{P}, ()))$ onto the O-list and the tuple (O_1, P_s, P_1, tid, a) to the r-list, where tid is the ID that $\mathcal{S}$ received along with the output from $\mathcal{F}_{RSOR}$ and a is the index of the O-list entry.
(2) If P_1 is adversarial, $\mathcal{S}$ does no additional work.

Case II: "*start* belongs to onion from P_s with $sid, R, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, d$". This is just the result of $\mathcal{S}$'s reaction to an onion from $\mathcal{A}$ that was not the protocol-conform processing of an honest sender's communication (Case **VIII**). $\mathcal{S}$ does nothing.

Case IIIa: any output together with "tid belongs to onion/reply from P_s with $sid, R/P_r, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, d$" for $tid \notin \{start, end\}$. This means an honest relay is done processing an onion received from $\mathcal{A}$ that was not the protocol-conform processing of an honest sender's communication (processing that follows Case **VII**). $\mathcal{S}$ finds (*onionlist, path, c := currentPosition, information*) with these inputs as information and $\mathcal{P}[c] = P_{o_i}$ where P_{o_i} is the relay that sent the onion tid in $\mathcal{F}_{RSOR}$ [21] in the O-list (notice that there has to be such an entry). Let a be the index of the entry in the O-list [21].

$\mathcal{S}$ must now send the correct onion from the list in the O-list entry over the next link while keeping track of it so that it can reassociate it with the O-list entry when it next receives it as the following honest relay. To this end, $\mathcal{S}$ stores $(O[c], \mathcal{P}[c], \mathcal{P}[c+1], tid, a)$ to the r-list and sends $O[c]$ to the link to $\mathcal{P}[c+1]$ from $\mathcal{P}[c]$ [21].

Case IIIb: any output together with "*end* belongs to onion/reply from P_s with $sid, R/P_r, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, d$". This case occurs whenever one of the onions $\mathcal{S}$ creates in $\mathcal{F}_{RSOR}$ in case **VIII** reaches the end of its path and either an adversarial relay or the receiver of the onion comes next. $\mathcal{S}$ can tell the difference by examining whether another relay $\mathcal{P}[c+1]$ and onion $O[c]$ remain in the lists of the O-list entry (*onionlist, path, c := currentPosition, information*) corresponding to this onion ($\mathcal{S}$ finds the entry like in case **IIIa**).

(1) If another relay follows, $\mathcal{S}$ sends $O[c]$ to the link to $\mathcal{P}[c+1]$ from $\mathcal{P}[c]$ [21].
(2) If there is no next relay, then $\mathcal{S}$ must send the message contained in the onion to the receiver along with the correct reply ID if the onion is repliable (if it is, $\mathcal{S}$ also receives "Reply with reply ID rid" in the output). $\mathcal{S}$ sends (m, rid) (or $(m, \perp)$ if the onion is not repliable) to the link to R from P_{o_i} in the real world, where P_{o_i} is the relay that sent the message in $\mathcal{F}_{RSOR}$, triggering this case.

Case IIIc: Any output together with the new output "*tagged* belongs to onion from P_s with $sid, R, m, n, \mathcal{P}$". If $\mathcal{S}$ receives this message, the final honest relay on the forward path of a corrupted sender's onion just processed a tagged onion over either the final path segment consisting of only corrupted relays or the final link to the receiver itself. Depending on which is the case, $\mathcal{S}$ performs different actions:

(1) If $\mathcal{P}[c+1]$ is set: The onion is not at the exit relay yet and the tagging will not be discovered by an honest router. $\mathcal{S}$ behaves like in case **IIIa**.
(2) If $\mathcal{P}[c+1]$ is not set: The honest exit relay has noticed the tagging. The protocol would discard such an onion, so no action is required from $\mathcal{S}$.

In case the received output belongs to an honest sender's communication:

Case IV: "P_{o_i} sends onion tid to $P_{o_{i+1}}$ via ()". In this case, $\mathcal{S}$ needs to make it look as though an onion was passed from the honest party P_{o_i} to the honest party $P_{o_{i+1}}$: $\mathcal{S}$ picks the path $\mathcal{P} = (P_{o_i}, P_{o_{i+1}})$ and random message m_{rdm}. $\mathcal{S}$ honestly picks a randomness $\mathcal{R}$ and a random receiver R and calculates

$$O_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m_{rdm}, R, \mathcal{P}, (), PK_{\mathcal{P}}, ())$$

and sends the onion O_1 to $\mathcal{A}$'s party representing the link between the honest relays as if it was sent from P_{o_i} to $P_{o_{i+1}}$. $\mathcal{S}$ stores $(O_1, P_{o_i}, P_{o_{i+1}}, tid, \perp)$ on the r-list.

Case V: "P_{o_i} sends onion tid to P_{o_j} via $(P_{o_{i+1}}, \ldots, P_{o_{j-1}})$". To handle this case, $\mathcal{S}$ picks the path $\mathcal{P} = (P_{o_{i+1}}, \ldots, P_{o_{j-1}})$, a randomness $\mathcal{R}$ and a random receiver R and a message m_{rdm} and calculates

$$O_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m_{rdm}, R, \mathcal{P}, (), PK_{\mathcal{P}}, ())$$

[18]If P_1 is adversarial, the reply path of the ideal-world onion $\mathcal{S}$ created for this reply in case **VIII** will also end at P_1.

and sends the onion O_1 to $P_{o_{i+1}}$, as if it came from P_{o_i}. $\mathcal{S}$ stores $(O_{j-i-1}, P_{o_{j-1}}, P_{o_j}, tid, \perp)$ on the r-list.

Case VIa: $\mathcal{S}$ receives the message "P_{o_i} sends onion with message m to R via $(P_{o_{i+1}}, \ldots, P_{o_{j-1}})$". The behavior in this case depends on whether the onion is repliable and whether P_{o_i} is the onion's exit relay or not:

(1) The onion is not repliable and P_{o_i} is its exit relay. In this case, $\mathcal{S}$ sends $(m, \perp)$ to the adversary's link to R as P_{o_i} in the real world.

(2) The onion is not repliable and P_{o_i} is not its exit relay. Here, $\mathcal{S}$ needs to build an onion that will carry the message across the remaining adversarial relays to the receiver. This happens just like in the original case: $\mathcal{S}$ picks the path $\mathcal{P} = (P_{o_{i+1}}, \ldots, P_{o_n})$, randomness $\mathcal{R}$, calculates

$$O_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m, R, \mathcal{P}, (), PK_{\mathcal{P}}, ())$$

and sends the onion O_1 to $P_{o_{i+1}}$, as if it came from P_{o_i}.

(3) The onion is repliable and P_{o_i} is the exit relay. $\mathcal{S}$ receives the additional output "Reply with reply ID rid" and sends (m, rid) to the adversary's link to R as P_{o_i} in the real world.

(4) The onion is repliable and P_{o_i} is not the exit relay. Now, $\mathcal{S}$ needs to use the extra output "Reply with tid. Its reply path begins with $\mathcal{P}^{\leftarrow}$" to construct an onion that will carry the message to the receiver and allow it to reply such that the reply onion will follow the beginning of the reply path to the first honest relay on it, where $\mathcal{S}$ will expect it. $\mathcal{S}$ picks the path $\mathcal{P} = (P_{o_{i+1}}, \ldots, P_{o_n})$, randomness $\mathcal{R}$, calculates

$$O_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$$

and sends the onion O_1 to the relay $P_{o_{i+1}}$, as if it was sent from the relay P_{o_i}. Further, $\mathcal{S}$ stores $(\mathcal{P}^{\leftarrow}.last, info, tid)$ with $info = (n + \mathcal{P}^{\leftarrow}.lastPos, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$ on the C-list. (Note that, as this is an honest communication, $\mathcal{P}^{\leftarrow}.last$ is honest.)

Case VIb: $\mathcal{S}$ receives the message "P_{o_i} sends tagged onion via $(P_{o_{i+1}}, \ldots, P_{o_n})$". This means that an honest relay has processed a tagged onion from an honest sender on the forward path and is delivering it to the exit relay. $(P_{o_{i+1}}, \ldots, P_{o_n})$ is never empty when $\mathcal{S}$ receives this input since $\mathcal{F}_{RSOR}$ guards against that case with the condition $i < n$. To translate this onion into the real world, $\mathcal{S}$ behaves like in case **VIa2**, but doesn't learn the message or the receiver of the onion and must additionally tag the new onion before sending it. To this end, $\mathcal{S}$ chooses a random $m_{rdm} \in M$ and $R_{rdm} \in D$ and builds the onion using those. Before sending it, $\mathcal{S}$ tags the onion.

Case IX: "R replies to rid with message m ". $\mathcal{S}$ needs to recreate this reply in the real world by sending (m, rid) to the adversary from the honest receiver R.[19]

$\mathcal{S}$'s behavior on a message from $\mathcal{A}$: $\mathcal{S}$, as real world honest party P_i, received an onion $O = (\tilde{\eta}, \tilde{\delta})$ or a message-reply ID-pair (m, rid) from $\mathcal{A}$ as adversarial player P_a.

Case VIIa: $((\tilde{\eta}, \tilde{\delta}), P_{i-1}, P_i, tid, a)$ is on the r-list for some tid. In this case, O is the protocol-conform processing of an onion from an honest sender's communication. $\mathcal{S}$ calculates $\text{ProcOnion}(SK_{P_i}, O, P_i)$. If it returns a fail ($O$ is a replay or modification that is detected and dropped by P_i), $\mathcal{S}$ does nothing. Otherwise, $\mathcal{S}$ sends the message

[19] $\mathcal{S}$ can keep track of which relay to send (m, rid) to by remembering which relay originally sent a forward message with rid.

$\text{DeliverOnion}(tid)$ to $\mathcal{F}_{RSOR}$ and increments $currentPosition$ for the a-th entry in the O-list if $a \neq \perp$ [21].

Case VIIb: $((\tilde{\eta}, \delta'), P_{i-1}, P_i, tid, a)$ is on the r-list for some tid and a $\delta' \neq \tilde{\delta}$. $\mathcal{A}$ has tagged the onion in flight. $\mathcal{S}$ calculates $\text{ProcOnion}(SK_{P_i}, O, P_i)$. If it returns a fail (e.g., O is a replay that is detected and dropped by the protocol), $\mathcal{S}$ does nothing. If ProcOnion does not return a fail, $\mathcal{S}$ calls $\text{Tag}(tid)$ to tag the onion in the ideal world as well before calling $\text{DeliverOnion}(tid)$. If $a \neq \perp$, $\mathcal{S}$ increases the position of the a-th entry in the O-list. To forward the tag in the real world, $\mathcal{S}$ also replaces the onion list $O = (O_1, \ldots, O, \ldots, O_k)$ in the O-list entry with the new onion list $O' = (O_1, \ldots, O', \ldots, O'_k)$, where O'_i is the result of processing O' repeatedly like in case **VIII**.

Case VIII: $(\tilde{\eta}, P_{i-1}, P_i, tid, a)$ is not on the r-list for any tid. This onion must have been sent by the P_a relay itself since the links between relays are secure channels due to $\mathcal{F}_{SC}$. In order to replicate this onion in the ideal world, $\mathcal{S}$ must first process it until it cannot be processed any further because either: 1) the next relay is a receiver, in which case the reply path must also be processed, 2) there is no next relay because the onion is a reply, 3) the next relay is adversarial, so $\mathcal{S}$ does not know the necessary keys, 4) or processing the onion fails [21].

In any case, $\mathcal{S}$ processes O repeatedly until it has the final result (O', P') along with the list of onions $O = (O_1, \ldots, O_{last})$ and relays $\mathcal{P} = (P_i, \ldots, P_{last})$ encountered along the way. The following behavior depends on what form (O', P') takes [21]:

(1) $(O', P') = (m, R)$: First, $\mathcal{S}$ checks whether O_{last} is repliable. If so, then $\mathcal{S}$ forms a reply

$$(O_1^{\leftarrow}, P_1^{\leftarrow}) \leftarrow \text{FormReply}(m', O_{last}, P_i, SK_i)$$

to it with an arbitrary message m' and processes it until it cannot proceed, resulting in a list of reply onions $O^{\leftarrow} = (O_1^{\leftarrow}, \ldots, O_k^{\leftarrow})$ and relays $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_k^{\leftarrow})$. If the onion is not repliable, let $O^{\leftarrow} = \mathcal{P}^{\leftarrow} = ()$.

Now, $\mathcal{S}$ creates the ideal-world onion by sending the message $\text{ProcessNewOnion}(R, m, \mathcal{P}, \mathcal{P}^{\leftarrow})$ to $\mathcal{F}_{RSOR}$ in the role of P_a. After doing so, $\mathcal{S}$ immediately delivers the first onion by calling $\text{DeliverOnion}(tid)$ with the tid ID it received from $\mathcal{F}_{RSOR}$ without running case **IIIa**. It does so because $\mathcal{A}$ has already delivered the onion from P_a to P_i. $\mathcal{S}$ also stores $(O, \mathcal{P}, 0, (P_a, sid, R, m, \mathcal{P}, \mathcal{P}^{\leftarrow}))$ on the O-list [21] and (sid, O_{last}) on the $Reply$-list.

(2) $(O', P') = (m, \perp)$: [. . .] P_i is the recipient and O' is a message. This means the adversary possibly replied to an honest senders forward onion with a corrupted exit relay. $\mathcal{S}$ checks for all $(P_i, reply, tid)$ tuples in the C-list to see if $\tilde{\eta}$ matches any $reply$-info on this list. If so (it was a reply to tid), $\mathcal{S}$ sends the message $\text{BypassReply}(m, tid)$ to $\mathcal{F}_{RSOR}$ on P_a's behalf and, as $\mathcal{A}$ already delivered this message to the honest party, sends $\text{DeliverOnion}(tid')$ for the belonging tid'. Otherwise, this onion is an unsolicited reply to an honest sender and will be ignored. $\mathcal{S}$ (creating this onion in the $\mathcal{F}_{RSOR}$) sends the invalid onion on behalf of P_a: $\text{ProcessNewOnion}(\perp, \perp, \mathcal{P}, ())$ and $\text{DeliverOnion}(tid)$ for the corresponding tid without running case **IIIa**. (Notice

that $\mathcal{S}$ knows which tid and sid belongs to this communication as it is started at an adversarial party P_a). $\mathcal{S}$ adds $(O, \mathcal{P}, 0, (P_a, sid, \perp, \perp, \mathcal{P}, ()))$ to the O-list.

(3) $(O', P' \neq \perp)$: P' is the next adversarial relay and O' is the onion it should receive. $\mathcal{S}$ picks a message $m \in \mathcal{M}$ and a new random receiver R. $\mathcal{S}$ sends on P_a's behalf the message, ProcessNewOnion$(R, m, \mathcal{P}, ())$ (notice that this onion cannot be replied to) [...] and DeliverOnion(tid) for the belonging tid to $\mathcal{F}_{RSOR}$ without running case **IIIa** (notice that $\mathcal{S}$ knows the tid as in case (a)). As the last step, $\mathcal{S}$ adds the entry $(O \| (O'), \mathcal{P} \| (P'), 0, (P_a, sid, R, m, \mathcal{P}, ()))$ to the O-list.

(4) $(O', P') = (\perp, \perp)$: This onion failed to be processed at P_{last}, so $\mathcal{S}$ must also send an invalid onion that takes this path in $\mathcal{F}_{RSOR}$: $\mathcal{S}$ sends ProcessNewOnion$(\perp, \perp, \mathcal{P}, ())$ in the role of P_a. If the header of the onion processes correctly as a last layer header, but the payload does not, $\mathcal{S}$ calls Tag(tid) on the corresponding tid ID. Then, $\mathcal{S}$ follows it with DeliverOnion(tid) without running case **IIIa**. $\mathcal{S}$ adds $(O, \mathcal{P}, 0, (P_a, sid, \perp, \perp, \mathcal{P}, ()))$ to the O-list.

Case X: $\mathcal{S}$ receives (m, rid) as the honest receiver R. $\mathcal{S}$ sends the message DeliverMessage(m, rid, R) to $\mathcal{F}_{RSOR}$.

Case XI: $\mathcal{S}$ receives (m, rid) as the honest relay P_i $\mathcal{S}$ sends the message DeliverReply(P_i, m, rid) to $\mathcal{F}_{RSOR}$.

Indistinguishability:

Notation: $\mathcal{H}_i$ describes the first hybrid that replaces a certain part of any communication for the first communication. In $\mathcal{H}_i^{<x}$ this part of the communication is replaced for the first $x-1$ communications. Finally in $\mathcal{H}_i^*$ this part of the communication is replaced in all communications.

Hybrid $\mathcal{H}_0$: This machine sets up the keys for the honest parties (so it has their secret keys). Then it interacts with the environment and $\mathcal{A}$ on behalf of the honest parties. It invokes the real protocol for the honest parties in interacting with $\mathcal{A}$.

Replacing between honest - Forward Onion: We replace the onion layers in the way they appear in the communication. So the first onion layers (close to the sender) are replaced first.

Hybrid $\mathcal{H}_1$: In this hybrid, for the first one forward communication the onion layers from its honest sender to the next honest relay on the forward path (relay or receiver) are replaced with random onion layers embedding the same path. More precisely, this machine acts like $\mathcal{H}_0$ except that the consecutive onion layers $O_1, O_2, \ldots, O_j$ from an honest sender P_0 to the next honest relay P_j are replaced with $\bar{O}_1$ and its following processings by calculating (with honestly chosen randomness $\mathcal{R}'$ and a random receiver R_{rdm})

$$\bar{O}_1 \leftarrow \text{FormOnion}(1, \mathcal{R}', m_{rdm}, R_{rdm}, \mathcal{P}', (), PK_{\mathcal{P}'}, ())$$

where m_{rdm} is a random message, $\mathcal{P}' = (P_1, \ldots, P_j)$. $\mathcal{H}_1$ now maintains a new $\bar{O}$-list, which it uses to recognize replacement onions. It stores $(info = (\mathcal{R}', m_{rdm}, R_{rdm}, \mathcal{P}', (), PK_{\mathcal{P}}, ()), \delta_j, P_j, (O_1^R, P_{j+1}))$ there, where $info$ are the randomness and parameters used for the replacement onion's creation, δ_j is the payload of the j-the onion layer, and O_1^R is calculated as

$$O_1^R \leftarrow \text{FormOnion}(j+1, \mathcal{R}, m, R, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}),$$

where the randomness, receiver, paths and message are chosen as in the original sender's call in $\mathcal{H}_0$. If an onion $\tilde{O}$ is sent to P_j, the machine tests if processing results in a fail (replay/modification detected and dropped). If it does not, $\mathcal{H}_1$ uses

$$\text{ROnion}(j, \tilde{O}, \mathcal{R}', m_{rdm}, R_{rdm}, \mathcal{P}', \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$$

for every recognize-information stored in the $\bar{O}$-list where the third entry is P_j. If it finds a match, it compares $\tilde{O}$'s payload to the δ_j in the entry. If those also match, the belonging O_1^R is sent to P_{j+1} as the processing result of P_j. If only the payload comparison fails, the onion has been tagged. $\mathcal{H}_1$ adds the original onion's input parameters to a *Tag*-list for later recognition. If P_j is the onion's exit relay ($P_{j+1} = \perp$), $\mathcal{H}_1$ produces no output. Otherwise, $\mathcal{H}_1$ recreates the tag on O_1^R and sends it to P_{j+1}. If no onion with a matching header is found, ProcOnion$(SK_{P_j}, \tilde{O}, P_j)$ is used.

$\underline{\mathcal{H}_0 \approx_I \mathcal{H}_1}$: The environment gets notified when an honest party receives an onion layer (and about their repliability) and inputs when this party is done. As we just exchange onion layers with others (with the same repliability), the behavior to the environment is indistinguishable for both machines. $\mathcal{A}$ observes the onion layers after P_0 and, if it sends an onion to P_j, the result of the processing after the honest relay. Depending on the behavior of $\mathcal{A}$, three cases occur: $\mathcal{A}$ drops the onion belonging to this communication before P_j, $\mathcal{A}$ behaves protocol-conform and sends the expected onion to P_j or $\mathcal{A}$ modifies the expected onion before sending it to P_j. Notice that dropping the onion leaves the adversary with no further output. Thus, we can focus on the other cases:

We assume there exists a distinguisher $\mathcal{D}$ between $\mathcal{H}_0$ and $\mathcal{H}_1$ and construct a successful attack on $TLU^{\rightarrow}$.

The attack receives key and name of the honest relay and uses the input of the replaced communication as choice for the challenge, where it replaces the name of the first honest relay with the one that it got from the challenger. For the other relays, the attack decides on the keys as $\mathcal{A}$ (for corrupted) and the protocol (for honest) do. It receives $\tilde{O}$ from the challenger. The attack uses $\mathcal{D}$. For $\mathcal{D}$, it simulates all communications except the one chosen for the challenge, with the oracles and knowledge of the protocol and keys. For simulating the challenge communication, the attack hands $\tilde{O}$ to $\mathcal{A}$ as soon as $\mathcal{D}$ instructs to do so. To simulate further for $\mathcal{D}$ it uses $\tilde{O}$ to calculate the later layers and does any actions $\mathcal{A}$ does on the onion.

$\mathcal{A}$ either sends the honest processing of $\tilde{O}$ to the challenge router or $\mathcal{A}$ modifies it. The attack uses the oracle to simulate the further processing of $\tilde{O}$ or its modification. If $\mathcal{A}$ chooses to tag $\tilde{O}$, then the challenger will output an onion with a random payload in both $\mathcal{H}_0$ and $\mathcal{H}_1$ since the tag completely randomizes the payload contents.

Thus, either the challenger chose $b = 0$ and the attack behaves like $\mathcal{H}_0$ under $\mathcal{D}$; or the challenger chose $b = 1$ and the attack behaves like $\mathcal{H}_1$ under $\mathcal{D}$. The attack outputs the same bit as $\mathcal{D}$ does for its simulation to win with the same advantage as $\mathcal{D}$ can distinguish the hybrids.

Hybrid $\mathcal{H}_1^{<x}$: In this hybrid, for the first $x-1$ forward communications, onion layers from an honest sender to the next honest relay on the forward path are replaced with a random onion sharing this path. [Note that $\mathcal{H}_1 = \mathcal{H}_1^{<2}$ and let $\mathcal{H}_1^*$ be the hybrid where the replacement happened for all communications.]

$\underline{\mathcal{H}_1^{<x-1} \approx_I \mathcal{H}_1^{<x}}$: Analogous to above. Apply argumentation of indistinguishability ($\mathcal{H}_0 \approx_I \mathcal{H}_1$) for every replaced subpath.

Hybrid $\mathcal{H}_{2a}$: In the two hybrids $\mathcal{H}_{2a}$ and $\mathcal{H}_{2b}$, for the first forward communication for which, in the adversarial processing, no recognition-falsifying modification (i.e. on η) occurred and other modification does not result in a fail, onion layers between two consecutive honest relays on the forward path (the second might be the exit relay) are replaced with random onion layers embedding the same path. We do so in two hybrid steps because we require both the $STI^{\leftrightarrow}$ and $TLU^{\rightarrow}$ properties to truncate the forward path of the onion before the first of the two honest relays and then replace the onion layers between the honest relays. Additionally, for all forward communications, replacements between the sender and the first relay happen as in $\mathcal{H}_1^*$. More precisely, $\mathcal{H}_{2a}$ acts like $\mathcal{H}_1^*$ except for the processing of O_j. The onion layers $O_{j+1}, \dots, O_n$, $O_1^{\leftarrow}, \dots, O_{n^{\leftarrow}}^{\leftarrow}$ are replaced with $\bar{O}_1, \dots, \bar{O}_{n-j}$, $\bar{O}_1^{\leftarrow}, \dots, \bar{O}_{n^{\leftarrow}}^{\leftarrow}$; the hybrid sends $\bar{O}_1$ instead of O_{j+1}. The replacement is formed as

$$\bar{O}_1 \leftarrow \textsc{FormOnion}(1, \mathcal{R}', m, R, \mathcal{P}', \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}'}, PK_{\mathcal{P}^{\leftarrow}})$$

with an honestly chosen randomness $\mathcal{R}'$ and $\mathcal{P}' = (P_{j+1}, \dots, P_n)$. If the onion's information is on the *Tag*-list, the tag is recreated on $\bar{O}_1$ before it is sent.

$\underline{\mathcal{H}_1^* \approx_I \mathcal{H}_{2a}}$: $\mathcal{H}_{2a}$ replaces the onion layers on the path after an honest relay and does so for an onion that has already had all of its subpaths between honest relays prior to this honest relay replaced before. The original onion layers before this honest relay are thus never output to the adversary while the layers used as replacements are chosen independently at random. We can reduce the new replacement to $STI^{\leftrightarrow}$ with $j_{STI^{\leftrightarrow}} = j$ and $j^{\leftarrow}_{STI^{\leftrightarrow}} = n^{\leftarrow}$ as the positions of the honest relays in the challenge thanks to this. Since the layers before P_j are independent of the challenge, the attack can recognize tagged payloads on those layers and recreate the tag on the challenge onion from the $STI^{\leftrightarrow}$ challenger if necessary.

Hybrid $\mathcal{H}_{2b}$: In this hybrid, we perform the actual replacement of the onion layers between the honest relays for the onion whose forward path was truncated in $\mathcal{H}_{2a}$. In essence, the consecutive onion layers $\bar{O}_1, \dots, \bar{O}_{j'-j}$ from a communication of an honest sender, starting at the next honest relay P_j to the next following honest relay $P_{j'}$, are replaced with $\hat{O}_1, \dots, \hat{O}_{j'-j}$ by sending $\hat{O}_1$. Thereby, for honestly chosen randomness $\mathcal{R}'$ and a random receiver R_{rdm}:

$$\hat{O}_1 \leftarrow \textsc{FormOnion}(1, \mathcal{R}', m_{rdm}, R_{rdm}, \mathcal{P}'', (), PK_{\mathcal{P}''}, ()),$$

where m_{rdm} is a random message, $\mathcal{P}'' = (P_{j+1}, \dots, P_{j'})$ is the path between the honest relays. $\mathcal{H}_{2b}$ stores

$$((\mathcal{R}'', m_{rdm}, R_{rdm}, \mathcal{P}'', (), PK_{\mathcal{P}''}, ()),$$
$$\delta_{j'-j}, P_{j'}, (O_1^R, P_{j'+1})),$$

where $\bar{O}_1^R$ is calculated with

$$\textsc{FormOnion}(j'-j+1, \mathcal{R}', m, R, \mathcal{P}', \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}'}, PK_{\mathcal{P}^{\leftarrow}}),$$

where the randomness, receiver, paths and message are chosen as $\mathcal{H}_{2a}$ chose them, on the $\bar{O}$-list. Like in $\mathcal{H}_1^*$, if an onion $\tilde{O}$ is sent to $P_{j'}$, processing is first checked for a fail. If it does not fail, $\mathcal{H}_{2b}$ checks

$$\textsc{ROnion}(j'-j, \tilde{O}, \mathcal{R}'', m_{rdm}, R_{rdm}, \mathcal{P}'', (), PK_{\mathcal{P}''}, ())$$

for any info on the $\bar{O}$-list where the second entry is $P_{j'}$. If it finds a match, it checks whether the original onion's information is on the *Tag*-list and compares $\tilde{O}$'s payload to $\delta_{j'-j}$. If the information is on the list or the payloads do not match, one of the previous replacements were tagged or the current replacement was tagged. If the original onion's information is not on the *Tag*-list yet, it is added. If P_j is the onion's exit relay, $\mathcal{H}_{2b}$ does nothing. Otherwise, $\mathcal{H}_{2b}$ recreates the tag on O_1^R and sends it to $P_{j'+1}$. On the other hand, if the onion was not tagged until now, the belonging O_1^R is used as the processing result of $P_{j'}$. If no onion with a matching header is found, $\textsc{ProcOnion}(SK_{P_{j'}}, \tilde{O}, P_{j'})$ is used.

$\underline{\mathcal{H}_{2a} \approx_I \mathcal{H}_{2b}}$: $\mathcal{H}_{2b}$ replaces, for one communication (and all its replays), the first subpath between two consecutive honest relays after an honest sender. The output to $\mathcal{A}$ includes the earlier (by $\mathcal{H}_1^*$) replaced onion layers $\bar{O}_{earlier}$ before the first honest relay (these layers are identical in $\mathcal{H}_{2a}$ and $\mathcal{H}_{2b}$) that take the original subpath but are otherwise chosen randomly; the original onion layers after the first honest relay for all communications not considered by $\mathcal{H}_{2b}$ (output by $\mathcal{H}_1^*$) or in case of the communication considered by $\mathcal{H}_{2b}$, the newly drawn random replacement (generated by $\mathcal{H}_{2b}$); and the processing after $P_{j'}$.

Similarly to our argument for $\mathcal{H}_{2a}$, the $\bar{O}_{earlier}$ layers are random and independent of the replaced layers, so they can be built without needing the $TLU^{\rightarrow}$ challenger.

Thus, all that is left are the original/replaced onion layer after the first honest relay and the processing afterwards. This is the same output as in $\mathcal{H}_0 \approx_I \mathcal{H}_1$. Hence, if there exists a distinguisher between $\mathcal{H}_{2a}$ and $\mathcal{H}_{2b}$ there exists an attack on $TLU^{\rightarrow}$.

Counting explanation for $\mathcal{H}_2^{<x}$: From here, we refer to the combination of $\mathcal{H}_{2a}$ and $\mathcal{H}_{2b}$ as $\mathcal{H}_2$ for convenience. Communication paths consist of multiple possible honest subpaths (paths from an honest relay to the next honest relay). We count (and replace) all these subpaths from the subpath closest to the sender until the one closest to the receiver. We first replace all such subpaths for the first communication, then for the second and so on. Below we use $< x$ to signal how many such subpaths will be replaced in the current hybrid. [Note that $\mathcal{H}_2 = \mathcal{H}_2^{<2}$ and let $\mathcal{H}_2^*$ be the hybrid where the replacement happened for all such subpaths.]

Hybrid $\mathcal{H}_2^{<x}$: In this hybrid, the first $x - 1$ honest subpaths (honest relay to next honest relay) of honest senders' forward communications is replaced with a random onion sharing the path. Additionally, for all forward communications, replacements between the sender and the first relay happen as in $\mathcal{H}_1^*$. If $\mathcal{A}$ previously (i.e., in onion layers up to the honest relay starting the selected subpath) modified η of an onion layer in this communication or modifies other parts such that processing fails, the communication is skipped.

$\underline{\mathcal{H}_2^{<x-1} \approx_I \mathcal{H}_2^{<x}}$: Analogous to above.

Replacing between Honest - Backward Onion

On the backward path, we replace the last onion layers first, then the second last and so on. Each machine only starts replacing at a certain point and if a message does not come that far (it is modified or dropped), they simply do not use any replacement. For all following hybrids, the replacements on the forward path are done as in $\mathcal{H}_2^*$.

Hybrid $\mathcal{H}_1^{\leftarrow}$: Similar to $\mathcal{H}_1$, but this time one backward communication between the last honest relay (which could be the exit relay) until the honest (forwards) sender is replaced. More precisely, this machine acts like $\mathcal{H}_2^*$ except that the consecutive onion layers $O_{j+1}^{\leftarrow}, \dots, O_{n^{\leftarrow}}^{\leftarrow}$ from a reply to an honest (forward) sender from the last honest relay $P_j^{\leftarrow}$ to the (forward) sender $P_{n^{\leftarrow}}^{\leftarrow} = P_0$ are replaced with $\bar{O}_1, \dots, \bar{O}_{n^{\leftarrow}-j}$ with (for an honestly chosen $\mathcal{R}'$ and a random receiver R_{rdm}):

$\bar{O}_1 \leftarrow$ FormOnion$(1, \mathcal{R}', m_{rdm}, R_{rdm}, \mathcal{P}', (), PK_{\mathcal{P}'}, ())$

where m_{rdm} is a random message, $\mathcal{P}' = (P^{\leftarrow}_{j+1}, \dots, P^{\leftarrow}_{n^{\leftarrow}})$ is the path from $P^{\leftarrow}_{j+1}$ to $P^{\leftarrow}_{n^{\leftarrow}}$. $\mathcal{H}^{\leftarrow}_1$ stores $(info, P^{\leftarrow}_{n^{\leftarrow}} = P_0, m_{rdm})$ on the $\bar{O}$-list. When looking up entries (with ROnion) on the $\bar{O}$-list, $\mathcal{H}^{\leftarrow}_1$ checks the belonging last entry to be an onion before sending it to the next relay.

$\underline{\mathcal{H}^*_2 \approx_I \mathcal{H}^{\leftarrow}_1}$: The environment gets notified when an honest party receives an onion layer and inputs when this party is done. As we just exchange onion layers with others (with the same repliability), the behavior to the environment is indistinguishable for both machines.

$\mathcal{A}$ observes the onion layers before $P^{\leftarrow}_j$ and, if it sends an onion to $P^{\leftarrow}_{n^{\leftarrow}}$, the result of the processing after the honest relay. Depending on the behavior of $\mathcal{A}$, three cases occur: $\mathcal{A}$ drops the onion belonging to this communication before $P^{\leftarrow}_{n^{\leftarrow}}$, $\mathcal{A}$ behaves protocol-conform and sends the expected onion to $P^{\leftarrow}_{n^{\leftarrow}}$ or $\mathcal{A}$ modifies the expected onion before sending it to $P^{\leftarrow}_{n^{\leftarrow}}$. Notice that dropping the onion leaves the adversary with no further output. Thus, we can focus on the other cases.

We assume there exists a distinguisher $\mathcal{D}$ between $\mathcal{H}^*_2$ and $\mathcal{H}^{\leftarrow}_1$ and construct a successful attack on $SLU^{\leftarrow}$:

The attack receives key and name of the honest relay and uses the input of the replaced communication as the choice for the challenge, where it replaces the name of the honest relay with the one that it got from the challenger. For the other relays, the attack decides on the keys as $\mathcal{A}$ (for corrupted) and the protocol (for honest) do. It receives O_1 from the challenger and forwards it to $\mathcal{A}$ for the corrupted first relay (on the forward path). The attack simulates all other communications with oracles (or their replacements as in the games before) and at some point, as $\mathcal{A}$ replies to O_1 (after receiving its processing O_{n+1}), so does our attack. The reply is processed (with the knowledge of the keys) until the honest relay, where the replaced onion layers start and this processed reply is forwarded to the oracle of the challenger as O to process it. The challenger returns $\tilde{O}$. The attack sends $\tilde{O}$, as the processing of the answer, to $\mathcal{A}$ as soon as $\mathcal{D}$ instructs to do so. To simulate further for $\mathcal{D}$ it uses $\tilde{O}$ to calculate the later layers and does any actions $\mathcal{A}$ does on the onion. Further, the attack simulates all other communications with the oracles and knowledge of the protocol and keys (or the random replacement onions, if replaced before).

Thus, either the challenger chose $b = 0$ and the attack behaves like $\mathcal{H}^*_2$ under $\mathcal{D}$; or the challenger chose $b = 1$ and the attack behaves like $\mathcal{H}^{\leftarrow}_1$ under $\mathcal{D}$. The attack outputs the same bit as $\mathcal{D}$ does for its simulation to win with the same advantage as $\mathcal{D}$ can distinguish the hybrids.

Hybrid $\mathcal{H}^{<x\leftarrow}_1$: In this hybrid, for the first $x-1$ backward communications, onion layers from the last honest relay to the honest sender (=backwards receiver) are replaced with a random onion sharing this path. The replacement is again stored on the $\bar{O}$-list as before.

$\underline{\mathcal{H}^{<x-1\leftarrow}_1 \approx_I \mathcal{H}^{<x\leftarrow}_1}$: Analogous to above. Apply argumentation of indistinguishability ($\mathcal{H}^*_2 \approx_I \mathcal{H}^{\leftarrow}_1$) for every replaced subpath.

Hybrid $\mathcal{H}^{\leftarrow}_{2a}$: In the hybrids $\mathcal{H}^{\leftarrow}_{2a}$ and $\mathcal{H}^{\leftarrow}_{2b}$, for the first backward communication (and all its replays) for which, in the adversarial processing, no recognition-falsifying modification occurred and other modification did not lead to failed processing, onion layers between the two last consecutive honest relays (the first might be the forward receiver (=backward sender)) are replaced with random onion layers embedding the same path. We separate this hybrid into two just like $\mathcal{H}_2$ for the same reason. Let j be the index of the first of the two honest relays in question and j' the index of the second. $\mathcal{H}^{\leftarrow}_{2a}$ acts like $\mathcal{H}^{*\leftarrow}_1$ except for the processing of $O_{j''}$, where j'' is the index of the last honest relay on the forward path of the onion. Due to $\mathcal{H}_2$, the onion's path starts at the second-to-last honest relay on the original onion's forward path. The onion layers $O_{j''+1}, \dots, O_n, O_1, \dots, O_{j'}$ are replaced with $\bar{O}_{j''+1}, \dots, \bar{O}_n, \bar{O}_1, \dots, \bar{O}_{j'}$, where $\bar{O}_{j''+1}$ is formed as

FormOnion$(j''+1, \mathcal{R}', m, R, \mathcal{P}, \mathcal{P}^{\leftarrow\prime}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow\prime}})$

with honestly chosen randomness $\mathcal{R}'$ and $\mathcal{P}^{\leftarrow\prime} = (P^{\leftarrow}_1, \dots, P^{\leftarrow}_{j'})$, effectively cutting off the end of the onion's reply path. If the onion's information is on the *Tag*-list, the tag is recreated on $\bar{O}_{j''+1}$ before it is sent.

$\underline{\mathcal{H}^{*\leftarrow}_1 \approx_I \mathcal{H}^{\leftarrow}_{2a}}$: We construct an attacker on $STI^{\leftrightarrow}$ with $j_{STI^{\leftrightarrow}} = 0$ and $j^{\leftarrow}_{STI^{\leftrightarrow}} = j'$ from any distinguisher of the two hybrids the same way as for $\mathcal{H}_{2a}$. Due to the previous hybrid, the original onion layers after relay $P_{j'}$ are never output to the adversary since they have been replaced with random layers.

Hybrid $\mathcal{H}^{\leftarrow}_{2b}$: In this hybrid, we perform the actual replacement of the onion layers between the honest relays for the onion whose backward path was truncated in $\mathcal{H}^{\leftarrow}_{2a}$. In essence, the consecutive onion layers $\bar{O}^{\leftarrow}_{j+1}, \dots, \bar{O}^{\leftarrow}_{j'}$ from a backward communication of an honest (forward) sender, starting at the second last honest relay $P^{\leftarrow}_j$ to the next following honest relay $P^{\leftarrow}_{j'}$ (on the backward path), are replaced with $\hat{O}_1, \dots, \hat{O}_{j'-j}$. Thereby for an honestly chosen $\mathcal{R}''$ and a random receiver R_{rdm}:

$\hat{O}_1 \leftarrow$ FormOnion$(1, \mathcal{R}'', m_{rdm}, R_{rdm}, \mathcal{P}', (), PK_{\mathcal{P}'}, ())$

where m_{rdm} is a random message, $\mathcal{P}' = (P^{\leftarrow}_{j+1}, \dots, P^{\leftarrow}_{j'})$ is the path from $P^{\leftarrow}_j$ to $P^{\leftarrow}_{j'}$.

Further, the hybrid calculates (and stores) another replacement for the next part after the current replacement $(\tilde{P}^{\leftarrow}_{j'+1}, \tilde{O}_k)$ (by exploiting the fact that the sender knows the backward path and can infer the message from any layer) as in the hybrid $\mathcal{H}^{*\leftarrow}_1$ before. Then it also stores $(info, P^{\leftarrow}_{j'}, (\tilde{O}_k, \tilde{P}^{\leftarrow}_{j'+1}))$ to the $\bar{O}$-list (to ensure the replacement of the later path is used as well). As before, the $\bar{O}$-list will be checked to pick the right processing of an onion.

$\underline{\mathcal{H}^{\leftarrow}_{2a} \approx_I \mathcal{H}^{\leftarrow}_{2b}}$: $\mathcal{H}^{\leftarrow}_{2b}$ replaces for one backward communication, the last subpath between two consecutive honest relay before an honest (forward) sender. The output to $\mathcal{A}$ includes the later (by $\mathcal{H}^{*\leftarrow}_1$) replaced onion layers $\bar{O}_{later}$ after the second honest relay (these layers are identically generated in $\mathcal{H}^{\leftarrow}_{2a}$ and $\mathcal{H}^{\leftarrow}_{2b}$) that take the original subpath but are otherwise chosen randomly; the original onion layers after the first of the honest relays for all communications not considered by $\mathcal{H}^{\leftarrow}_{2b}$ (output by $\mathcal{H}^{\leftarrow}_{2a}$) or, in case of the communication considered by $\mathcal{H}^{\leftarrow}_{2b}$, the newly drawn random replacement (generated by $\mathcal{H}^{\leftarrow}_{2b}$); and the processing before the first honest relay $P^{\leftarrow}_j$.

Similarly to our argument for $\mathcal{H}^{\leftarrow}_{2a}$, the $\bar{O}_{later}$ layers are random and independent of the replaced layers, so they can be built without needing the $SLU^{\leftarrow}$ challenger.

Thus, all that is left are the original/replaced onion layer after the honest relay and the original layers before. This is the same output as in $\mathcal{H}_2^* \approx_I \mathcal{H}_1^{\leftarrow}$. Hence, if there exists a distinguisher between $\mathcal{H}_{2a}^{\leftarrow}$ and $\mathcal{H}_{2b}^{\leftarrow}$, there exists an attack on $SLU^{\leftarrow}$.

Hybrid $\mathcal{H}_2^{<x\leftarrow}$: From here, we refer to the combination of $\mathcal{H}_{2a}{\leftarrow}$ and $\mathcal{H}_{2b}^{\leftarrow}$ as $\mathcal{H}_2^{\leftarrow}$ for convenience. In this hybrid, for the first $x-1$ honest subpaths on backwards communications are replaced with a random onion sharing the path and the other replacements calculated as before and all are stored on the $\bar{O}$-list. If $\mathcal{A}$ previously (i.e., in onion layers up to the honest relay starting the selected subpath) modified η of an onion layer in this communication or modified another part such that processing fails, the communication is skipped.

$\underline{\mathcal{H}_2^{<x-1\leftarrow} \approx_I \mathcal{H}_2^{<x\leftarrow}}$: Analogous to above.

Onion replacement for corrupted receivers

We replace the missing part between the onion layers already replaced on the forward path and the onion layers already replaced on the backward path. Due to the structure of RSOR, this part exists for every onion and includes the link between the exit relay and the receiver. Note that if the exit relay is the last honest relay on the onion's path, then no replacement can take place because the message is simply sent to the receiver in plaintext. We thus only consider onions that still have at least one adversarial relay left on their path (which also implies that their exit relay is adversarial).

Hybrid $\mathcal{H}_3$: In this hybrid, for the first forward communication for which, in the adversarial processing, no recognition-falsifying modification (i.e., a modification on η) occurred (and no other modification caused the processing to fail) so far, forward onion layers from its last honest relay to the corrupted exit relay are replaced with random onions sharing this path, receiver, and message and the first part of the reply-path. More precisely, this machine acts like $\mathcal{H}_2^{*\leftarrow}$ except for the processing of O_j; in essence, the consecutive onion layers $O_{j+1}, \ldots, O_n$ from a communication of an honest sender, starting at the last honest relay P_j to the corrupted exit relay P_n are replaced with $\bar{O}_1, \ldots, \bar{O}_{n-j}$; Thereby, for an honestly chosen $\mathcal{R}'$:

$$\bar{O}_1 \leftarrow \text{FormOnion}(1, \mathcal{R}', m, R, \mathcal{P}', \mathcal{P}'^{\leftarrow}, PK_{\mathcal{P}'}, PK_{\mathcal{P}'^{\leftarrow}}),$$

where m is the message of this communication, R is the receiver, $\mathcal{P}' = (P_j, \ldots, P_n)$ is the path from P_j to P_n and $\mathcal{P}'^{\leftarrow}$ is the first part of the reply-path (until the first honest relay), that a reply to the original onion would have taken. If the onion's information is on the Tag-list, then the tag is recreated on $\bar{O}_1$ before it is sent.

$\mathcal{H}_3$ further checks for every onion (ending at) $\mathcal{P}^{\leftarrow}.last$, if it was a reply to these replaced onion layers (by using the information *info* stored and ROnion). If so, it uses its knowledge about the original forward onion (before replacement) and the sender to construct the belonging original reply. With it, it computes the replacement of the later onion layers for this communication as in hybrid $\mathcal{H}_2^{*\leftarrow}$ and stores the corresponding information on the $\bar{O}$-list. As before, the $\bar{O}$-list will be checked to pick the right processing of an onion.

$\underline{\mathcal{H}_2^{*\leftarrow} \approx_I \mathcal{H}_3}$: Similar to $\mathcal{H}_1^* \approx_I \mathcal{H}_2$, the forward onion layers before P_j are independent and hence can be simulated for the distinguisher by an attack on $STI^{\leftrightarrow}$. If the adversary tags one of those layers, the attack can recognize the tag since it knows the expected payload it should be receiving at each honest relay. The attack can then recreate the tag after getting the challenge onion from the $STI^{\leftrightarrow}$ challenger. Similarly to $\mathcal{H}_1^{*\leftarrow} \approx_I \mathcal{H}_2^{\leftarrow}$, the backward onion layers after $\mathcal{P}^{\leftarrow}.last$ are independent and hence can be simulated for the distinguisher by an attack on $STI^{\leftrightarrow}$. The remaining outputs suffice to construct an attack on $STI^{\leftrightarrow}$ similar to the one on $TLU^{\rightarrow}$ in $\mathcal{H}_1^*$ and $\mathcal{H}_2$.

Hybrid $\mathcal{H}_3^{<x}$: In this hybrid, for the first $x-1$ forward communications for which, in the adversarial processing, no recognition-falsifying modification (and no other modification that results in failed processing) occurred so far, the onion layers between its last honest relay to corrupted exit relay are replaced with random onion layers sharing the path, message, receiver, and first part of the reply path.

$\mathcal{H}_3^{<x-1} \approx_I \mathcal{H}_3^{<x}$: Analogous to above.

Hybrid $\mathcal{H}_4$: This machine acts the way that $\mathcal{S}$ acts in combination with $\mathcal{F}_{RSOR}$. Note that $\mathcal{H}_3^*$ only behaves differently from $\mathcal{S}$ in (a) routing onions through the honest parties and (b) where it gets its information needed for choosing the replacement onion layers: (a) $\mathcal{H}_3^*$ actually routes them through the real honest parties that do all the computation. $\mathcal{H}_4$ instead runs the way that $\mathcal{F}_{RSOR}$ and $\mathcal{S}$ operate: there are no real honest parties, and the ideal honest parties do not do any crypto work. (b) $\mathcal{H}_3^*$ gets inputs directly from the environment and gives output to it. In $\mathcal{H}_4$, the environment instead gives inputs to $\mathcal{F}_{RSOR}$ and $\mathcal{S}$ gets the needed information (i.e., parts of path and the included message, if the receiver is corrupted) from outputs of $\mathcal{F}_{RSOR}$ as the ideal world adversary. $\mathcal{F}_{RSOR}$ gives the outputs to the environment as needed.

$\underline{\mathcal{H}_3^* \approx_I \mathcal{H}_4}$: For the interaction with the environment from the protocol/ideal functionality, it is easy to see that the simulator directly gets the information it needs from the outputs of the ideal functionality to the adversary: Whenever an honest relay is done processing, it needs the path from it to the next honest relay or path from it to the corrupted receiver and in this case also the message and beginning of the backward path. This information is given to $\mathcal{S}$ by $\mathcal{F}_{RSOR}$.

Further, in the real protocol, the environment is notified by honest relays when they receive an onion together with some random ID that the environment sends back to signal that the honest relay is done processing the onion. The same is done in the ideal functionality. Notice that the simulator ensures that every communication is simulated in $\mathcal{F}_{RSOR}$ such that those notifications arrive at the environment without any difference (this includes them having the same repliability).

For the interaction with the real world adversary, we distinguish the outputs in communications from honest and corrupted senders.

(0) Corrupted (forward) senders: In the case of a corrupted sender, both $\mathcal{H}_3^*$ and $\mathcal{H}_4$ (i.e., $\mathcal{S} + \mathcal{F}_{RSOR}$) do not replace any onion layers except that with negligible probability a collision on the $\bar{O}$-list resp. O-list occurs. (Notice that even for honest receivers (and thus backward senders) layers following the protocol can be and are created.)

(1) Honest senders:

1.1) No recognition-falsifying modification of the onion by the adversary happens (and if modification happens at all, the processing does not fail [note that a failing processing is the same as dropping; see 1.3)]): All parts of the path are replaced with randomly drawn onion layers $\bar{O}_i$. The way those layers are chosen is identical for $\mathcal{H}_3^*$ and $\mathcal{H}_4$ (i.e., $\mathcal{S} + \mathcal{F}_{RSOR}$).

1.2) The onion is tagged by the adversary: If the tagging occurs on the forwards path between two honest relays, the onion's information is added to the *Tag*-list at the next honest relay. This list exactly corresponds to the L_{tag} list in $\mathcal{F}_{RSOR}$ for forward onions and the onion is treated accordingly at the last honest relay: If that relay is the exit relay, no output is produced. If it is not the exit relay, the steps in $\mathcal{H}_3^*$ ensure that the corrupted exit relay receives a tagged onion. If the tagging occurs on the backwards path, the onion's information is not added to the *Tag*-list and the final onion layer delivered to the reply receiver is not tagged. This does not change the output to $\mathcal{Z}$ because received replies are never output to the environment by honest relays.

1.3) Some recognition-falsifying modification of the onion or a drop or insert happens: As soon as a recognition-falsifying modification happens, both $\mathcal{H}_3^*$ and $\mathcal{H}_4$ continue to use the bit-identical onion for the further processing except when, with negligible probability, a collision on the $\bar{O}$-list resp. O-list occurs. In case of a dropped onion, it is simply not processed further in any of the two machines.

Note that the view of the environment in the real protocol is the same as its view in interacting with $\mathcal{H}_0$. Similarly, its view in the ideal protocol with the simulator is the same as its view in interacting with $\mathcal{H}_4$. As we have shown indistinguishability in every step, we have indistinguishability in their views. □

E SPHINX: PACKET FORMAT DESCRIPTION

This section is meant as a compact reference for the structure of the Sphinx packet, which is used in the following appendices. For a complete introduction to the Sphinx packet format, see [12]. A Sphinx packet is made of a header $\eta = (\alpha, \beta, \gamma)$ and a payload δ. α is built using the public keys $y_i := g^{x_i}$ for each relay P_i on the onion's path. The sender chooses a secret $x \in \mathbb{Z}_q^*$ and lets $\alpha_i = g^{xb_0 \cdots b_{i-i}}$ and $s_i = y_i^{xb_0 \cdots b_{i-1}}$, where $b_i = h_b(\alpha_{i-1}, s_{i-1})$. α_i is the group element contained in the i-th layer of the header, and s_i is the corresponding secret it shares with P_i.[20] The b_i are blinding factors that transform α at each relay. They are calculated with a random oracle h_b. The remainder of the header is built using the shared secrets s_i after passing them through more random oracles: h_ρ, h_μ, and h_π are each used to key a different primitive. The β_i are built starting at the final layer:

$$\beta_{\nu-1} := \{*/P_s \| I \| rand_{(2(N-n)+2)\kappa-|R|}\} \oplus \{\rho(h_\rho(s_{n-1}))_{[\ldots(2(N-n)+3)\kappa-1]}\} \| \Phi_{n-1}.^{21}$$

In this definition, $*/P_s$ is either a sentinel value indicating that this is a forward packet or the address of the original sender in a reply packet. I is an identifier used by P_s to recognize replies. N is the global maximum path length in this Sphinx instance. ρ is a PRG keyed with $h_\rho(s_i)$. Φ_i is padding formed via the repeated application of the ρ PRG: Φ_0 is empty, while

$$\Phi_i = \{\Phi_{i-1} \| 0_{2\kappa}\} \oplus \rho(h_\rho(s_i))_{[(2(N-i)+3)\kappa \ldots (2N+3)\kappa-1]}.$$

The remaining β_i are built as

$$\beta_i = \{P_{i+1} \| \gamma_{i+1} \| \beta_{i+1_{[\ldots(2N-1)\kappa-1]}}\} \oplus \rho(h_\rho(s_i))_{[\ldots(2N+1)\kappa-1]}.$$

Effectively, the construction of the padding is designed such that Φ_i is a suffix of β_i. γ_i is the MAC $\mu(h_\mu(s_i), \beta_i)$ of β_i. Finally, a forward payload δ is formed as $\delta_{n-1} = \pi(h_\pi(s_{n-1}), 0_\kappa \| R \| m)$ and $\delta_i = \pi(h_\pi(s_i), \delta_{i+1})$, where π is a PRP keyed with $h_\pi(s_i)$.

The packet sent by the sender is $((\alpha_0, \beta_0, \gamma_0), \delta_0)$. Each relay P_i then processes the packet it gets into $((\alpha_{i+1}, \beta_{i+1}, \gamma_{i+1}), \delta_{i+1})$ [12].

F SPHINX: ADAPTED KEM-IND-CCA

Proof. We will use a PPT attacker $\mathcal{A}$ on the Sphinx-KEM-IND-CCA property for the Sphinx RO-KEM to construct an attacker $\mathcal{B}^{O_G}$ on the GDH assumption using the DDH oracle O_G.

The GDH attacker $\mathcal{B}$ maintains five lists:

- L: List of group elements g^z with their associated oracle outputs $(h_*(g^z), h_*(g^{x_1 * z}), h_b(g^z, g^{x_1 * z}))$.
- L_y: List of up to N tuples, one for each adversarial relay on the KEM path: Each holds $(\alpha_i, h_*(s_i), h_b(\alpha_i, s_i))$.
- L_b: List of (g^a, g^z) element pairs with their corresponding $h_b(g^a, g^z)$ values.
- L_O: List of α's that have been requested from O.
- L_h: List of group elements that have been requested from h_*.

$\mathcal{B}$ receives a CDH challenge (g, g^{x_1}, g^{x_2}) from the GDH challenger C. $\mathcal{B}$ uses g^{x_1} as the public key PK of the "honest relay" and g^{x_2} as the challenge α_j. The attacker sends PK to $\mathcal{A}$ and gives $\mathcal{A}$ access to the programmed random oracles h_* and h_b and the decapsulation oracle O (which are described below). Next, $\mathcal{B}$ receives j and y_i for $i \neq j$ from $\mathcal{A}$ and checks that the public keys are valid. $\mathcal{B}$ now chooses $b_0, \ldots, b_n$ randomly from $\mathbb{Z}_p^*$. To calculate α_i for $i < j$, $\mathcal{B}$ calculates the inverses $b_0^{-1}, \ldots, b_{j-1}^{-1}$ and uses them to form $\alpha_i = \alpha_j^{b_i^{-1} \ldots b_{j-1}^{-1}}$. To make α_i for $i > j$, let $\alpha_i = \alpha_j^{b_j \ldots b_{i-1}}$. Next, $\mathcal{B}$ chooses n random $r_0, \ldots, r_n$ values as outputs for $h_*(s_i)$. To remember these choices in the programmed random oracles, $\mathcal{B}$ stores (α_i, r_i, b_i) in L_y for $i \neq j$. Now, it flips a bit b to determine whether it will simulate the KEM game for $b = 0$ or $b = 1$. If $b = 0$, $\mathcal{B}$ sets $L[\alpha][1] := r_j$, and $L[\alpha][2] := b_j$. Finally, $\alpha_0, r_0, \ldots, r_{j-1}, r_{j+1}, \ldots, r_n$ and $b_0, \ldots, b_{j-1}, b_{j+1}, \ldots, b_n$ are sent to $\mathcal{A}$ along with (α_j, r_j, b_j).

To simulate the decapsulation oracle O and the random oracles h_* and h_b, $\mathcal{B}$ behaves as described in Algorithm 3.

Using a bad-flag analysis, we can see that any attacker $\mathcal{A}$ with a non-negligible advantage in the KEM game must trigger the *bad* flag non-negligibly often, so $\mathcal{B}$ can also win the GDH game non-negligibly often. The random oracles h_* and h_b are set up to behave correctly in combination with O except if a "collision" is created in the challenge creation phase (steps 3–7). Here, a collision refers to $\mathcal{B}$ assigning two different random values to the same random oracle inputs on accident. This can occur in two ways:

(1) The attacker already requested $O(\alpha)$ in step two.
(2) The attacker already requested $h_*(s_i)$ or $h_b(\alpha_i, s_i)$ for $0 \leq i \leq n$ in step two or $\mathcal{B}$ generates its own collision on accident when $s_{i_1} = s_{i_2}$ for $i_1 \neq i_2$.

[20] Note that our indices i for relay names are relative to a single packet's path for clarity.
[21] Note that we use the randomized padding described in Section 2.4.2 here.

Algorithm 3 KEM attacker $\mathcal{B}$'s oracles

```
function $O(\alpha')$
    if $\alpha$ generated and $\alpha' = \alpha$ then abort
    if $L[\alpha'][1]$ is not set then
        Add $\alpha'$ to $L_O$
        for $g^z \in L_h$ do
            if $O_G(g, PK, \alpha', g^z)$ then
                $L[\alpha'][1] \leftarrow L[g^z][0]$
        if $L[\alpha'][1]$ is not set then
            $L[\alpha'][1] \leftarrow^R \{0,1\}^{3\kappa}$
        for $(g^a, g^z) \in L_b$ where $g^a = \alpha'$ do
            if $O_G(g, PK, \alpha', g^z)$ then
                $L[\alpha'][2] \leftarrow L_b[(\alpha', g^z)]$
        if $L[\alpha'][2]$ is not set then
            $L[\alpha'][2] \leftarrow^R \mathbb{Z}_q^*$
    return $(L[\alpha'][1], L[\alpha'][2])$
function $h_*(g^z)$
    if $L[g^z][0]$ is not set then
        Add $g^z$ to $L_h$
        if $\alpha$ generated and $O_G(g, PK, \alpha, g^z)$ then
            $bad \leftarrow 1$
        for $(\alpha_i, r_i, \_) \in L_y$ do
            if $O_G(g, y_i, \alpha_i, g^z)$ then
                $L[g^z][0] \leftarrow r_i$
        for $\alpha' \in L_O$ do
            if $O_G(g, PK, \alpha', g^z)$ then
                $L[g^z][0] \leftarrow L[\alpha'][1]$
        if $L[g^z][0]$ is not set then
            $L[g^z][0] \leftarrow^R \{0,1\}^{3\kappa}$
    return $L[g^z][0]$
function $h_b(g^a, g^z)$
    if $L_b[(g^a, g^z)]$ is not set then
        if $O_G(g, PK, g^a, g^z)$ then
            if $\alpha$ generated and $g^a = \alpha$ then
                $bad \leftarrow 1$
            if $L[g^a][2]$ is set then
                $L_b[(g^a, g^z)] \leftarrow L[g^a][2]$
        if $(g^a, \_, \_) \in L_y$ at index $i$
          and $O_G(g, y_i, g^a, g^z)$ then
            Retrieve $(g^a, \_, b_i)$ from $L_y$
            $L_b[(g^a, g^z)] \leftarrow b_i$
        if $L_b[(g^a, g^z)]$ is not set then
            $L_b[(g^a, g^z)] \leftarrow^R \mathbb{Z}_q^*$
    return $L[(g^a, g^z)]$
```

In any of the above cases, $\mathcal{B}$ simulates the oracles incorrectly.[22] Let the number of requests $\mathcal{A}$ makes to each oracle O, h_*, and h_b be bounded by the polynomial $p(\kappa)$. The probability of each case is bounded by $1/q$ and $\lessapprox \frac{(p(\kappa)+N)^2}{2q}$ each for a collision on h_* and h_b respectively.[23] According to Sphinx's definition, the order q of the group G is on the order of $2^{2\kappa}$, meaning that both of these probabilities are negligible.

[22]Unless $\mathcal{B}$ happened to choose the same randomness in both cases, which only happens negligibly often.
[23]This corresponds to an upper bound for the likelihood for a successful birthday attack given $p(\kappa) + N$ requests and a pre-image set with q members [4].

If neither of these events occur, $\mathcal{B}$ simulates the oracles correctly and wins the GDH game whenever $\mathcal{A}$ wins the Sphinx-KEM-IND-CCA game. □

G SPHINX ONION PROPERTY PROOFS

G.1 RSOR-Tagging Layer Unlinkability ($TLU^{\rightarrow}$)

THEOREM 3. *Sphinx satisfies $TLU^{\rightarrow}$ under the GDH assumption.*

PROOF. We prove that an adversary cannot distinguish the $b = 0$ scenario from the $b = 1$ scenario through a hybrid argument starting at $b = 0$ and ending at $b = 1$. For clarity, we separate the proof into two cases: One where $j = n$ and the other where $j < n$. Initially, this may seem problematic since the adversary chooses j adaptively after the first round of oracle accesses, so we cannot predict which it will choose beforehand for our reductions. However, every step in our proofs is either common to both cases (so it does not require predicting j) or only applies after the adversary has made its choice. Note that Danezis and Goldberg index Sphinx packet layers starting at 0 [12]. Here, this means that $P_0(= P_s)$ would send the Sphinx layer with α_0 in its header.

Case 1 ($j = n$)**:** In this case, the honest relay on the forward path is also the exit relay of the onion and thus the sender of the reply onion.

Hybrid $\mathcal{H}_0$: This hybrid is just the $b = 0$ scenario of $TLU^{\rightarrow}$ with $j = n$.

Hybrid $\mathcal{H}_1$: As a first step, we replace the secrets used at the honest relay (the exit relay) with randomness. $\mathcal{H}_1$ replaces the random oracle outputs $h_*(s_{n-1})$ with random $\{0,1\}^\kappa$ bitstrings when building the reply header.[24] **Proc** requests for onions with the challenge α_{n-1} are also served using these random keys to keep the oracle's behavior consistent.

$\underline{\mathcal{H}_0 \approx_I \mathcal{H}_1}$: The difference between these hybrids reduces to $\mathrm{Exp}^{\text{RO-KEM-IND-CCA}}_{\text{RO-KEM},\mathcal{A}}(\kappa)$: The Sphinx-KEM-IND-CCA attacker $\mathcal{A}$ uses its inputs from the challenger and the assumed hybrid distinguisher $\mathcal{D}$ to build the challenge onion and serves requests to the **Proc** and **Reply** oracles using its decapsulation oracle. **Proc** requests with the challenge α_{n-1} are served using the keys provided by the challenger.

Hybrid $\mathcal{H}_2$: To ensure that only the challenge onion is recognized for "challenge processing" by the exit relay, the **Proc** oracle at the relay now returns $(\perp, \perp)$ on every request with α_{n-1} in its header except if the rest of the header also matches the expected header of the challenge onion.

$\underline{\mathcal{H}_1 \approx_I \mathcal{H}_2}$: This reduces to $\mathrm{Exp}^{\text{sEUF-CMA-vq}}_{\mu,\mathcal{A}}(\kappa)$ using the fact that the PRF μ can be viewed as a MAC with the randomized key $h_\mu(s_{n-1})$. Any request to the **Proc** oracle at the exit relay with α_{n-1} must contain a valid γ MAC for the β in the header to be processed. To notice a difference between the two hybrids, a distinguisher must submit such a request with a modified β or γ. Since μ is sEUF-CMA-vq-secure, this request constitutes a MAC forgery.

Hybrid $\mathcal{H}_3$: This hybrid swaps $\pi(h_\pi(s_{n-1}), \cdot)$ with a random permutation (RP). Note that $h_\pi(s_{n-1})$ is already a random key before this hybrid. Since Sphinx requires π to be a strong PRP [12], both π and π^{-1} can be used as RPs after the replacement.

[24]It also randomizes b_{n-1}, but we do not need that for this proof.

$\underline{\mathcal{H}_2 \approx_I \mathcal{H}_3}$: A distinguisher $\mathcal{D}$ for these hybrids is easily converted into an attacker $\mathcal{A}$ on $\text{Exp}^{\text{prp}}_{\pi,\mathcal{A}}(\kappa)$.

Hybrid $\mathcal{H}_4$: In this hybrid, the honest exit relay P_H only sends the challenge reply in response to a **Reply** request if the onion received in the corresponding **Proc** request has a payload that matches the expected payload δ_{n-1} of the forward onion exactly.

$\underline{\mathcal{H}_3 \approx_I \mathcal{H}_4}$: Since $\pi^{-1}(h_\pi(s_{n-1}), \cdot)$ is now an RP, every input is mapped to a random output. In order for a distinguisher to notice a difference between the two hybrids, it must submit a modified payload attached to the challenge header that is accepted by the oracle. After decrypting the payload, the oracle checks that the first κ bits of the contents are all zero and discards the onion it that is not the case. A manipulated payload only starts with 0_κ negligibly often, so the distinguisher only has a negligible chance of success.

Hybrid $\mathcal{H}_5$: This hybrid replaces the contents of the forward payload with the contents as they would be in the $b = 1$ scenario. The original contents are $0_\kappa \| R \| \eta_0 \| \tilde{k} \| m$, while the replacements are $0_\kappa \| \bar{R} \| pad^{\rightarrow}_{\kappa,N} \| \bar{m}$ with a random receiver $\bar{R}$, a random message $\bar{m}$, and padding instead of a reply header. When building the challenge reply onion, P_H still uses the original η_0 reply header. The **Proc** oracle also outputs the original message and receiver.

$\underline{\mathcal{H}_4 \approx_I \mathcal{H}_5}$: The **Proc** and **Reply** oracles at the exit relay behave the same way in $\mathcal{H}_4$ and $\mathcal{H}_5$. P_H consistently uses η_0 as the challenge reply header in both hybrids. The replacement of the payload contents reduces to $\text{Exp}^{\text{1-LR-CPA}}_{\pi,\mathcal{A}}(\kappa)$.

Hybrid $\mathcal{H}_6$: In this hybrid, we rewind the temporary changes made in the hybrids $\mathcal{H}_4$, $\mathcal{H}_3$, $\mathcal{H}_2$, and $\mathcal{H}_1$.

$\underline{\mathcal{H}_5 \approx_I \mathcal{H}_6}$: Apply the previous arguments in reverse. This concludes the $j = n$ case of the $TLU^{\rightarrow}$ proof.

Case 2 ($j < n$): For the second part of this proof, the honest relay on the forward path is not the exit relay, i.e., $j < n$.

Hybrid $\mathcal{H}_0$: The $b = 0$ case of $TLU^{\rightarrow}$ with $j < n$.

Hybrid $\mathcal{H}_1$: This hybrid performs the same steps as the hybrids $\mathcal{H}_1$ and $\mathcal{H}_2$ in the $j = n$ case.

$\underline{\mathcal{H}_0 \approx_I \mathcal{H}_1}$: See the corresponding hybrids in the $j = n$ case.

Hybrid $\mathcal{H}_2$: In this hybrid, we replace $\pi(h_\pi(s_{j-1}), \cdot)$ with an RP.

$\underline{\mathcal{H}_1 \approx_I \mathcal{H}_2}$: Analogous to Case 1's $\mathcal{H}_2 \approx_I \mathcal{H}_3$.

Hybrid $\mathcal{H}_3$: Previously (in $\mathcal{H}_2$), if δ'_{j-1} matches the original δ_{j-1}, the resulting output is the correct δ_j. Otherwise, the RP causes the output to be a uniformly random $\{0,1\}^{l_\pi(\kappa)}$ string.

In $\mathcal{H}_3$, instead of actually processing δ'_{j-1}, the honest relay only checks whether $\delta'_{j-1} = \delta_{j-1}$. If so, δ_j is used as the output onion's payload. Otherwise, a random $\{0,1\}^{l_\pi(\kappa)}$ string is output instead.

$\underline{\mathcal{H}_2 \approx_I \mathcal{H}_3}$: If a distinguisher $\mathcal{D}$ chooses to submit the correct challenge payload, the output in both hybrids is identical. If $\mathcal{D}$ sends a manipulated payload, $\mathcal{H}_2$ outputs a new RP output while $\mathcal{H}_3$ produces a completely random string. These two distributions are only distinguishable if $\mathcal{H}_3$ happens to choose δ_j as its random output, which $\mathcal{H}_2$ would never do. The probability of that happening is negligible.

Hybrid $\mathcal{H}_4$: This hybrid is analogous to Case 1's $\mathcal{H}_5$. It replaces the contents of δ_{j-1} (which were originally δ_j) with the new contents $0_\kappa \| \bar{R} \| pad^{\rightarrow}_{\kappa,N} \| \bar{m}$, $\bar{R}$ and $\bar{m}$ being a random receiver and message.

$\underline{\mathcal{H}_3 \approx_I \mathcal{H}_4}$: Analogous to $\mathcal{H}_5$ in Case 1.

Hybrid $\mathcal{H}_5$: In the honest relay's challenge processing, $\mathcal{H}_5$ always produces the same challenge header (the one belonging to the challenge onion's layer O_j) without actually processing the header input the relay is given. The challenge output of P_H now only depends on whether the payload was manipulated (i.e., tagged).

$\underline{\mathcal{H}_4 \approx_I \mathcal{H}_5}$: Due to $\mathcal{H}_1$, the honest relay only performs the challenge processing steps on headers that match the challenge header exactly. Thus, both hybrids always output the same challenge header for the challenge onion.

Hybrid $\mathcal{H}_6$: This hybrid replaces the PRG output $\rho(h_\rho(s_{j-1}))$ with a random string.

$\underline{\mathcal{H}_5 \approx_I \mathcal{H}_6}$: Given a distinguisher $\mathcal{D}$ for the two hybrids, construct an attacker $\mathcal{A}$ on $\text{Exp}^{prg}_{\rho,\mathcal{A}}(\kappa)$.

Hybrid $\mathcal{H}_7$: In this hybrid, we replace the first $(2(N-j)+3)\kappa$ bits of the contents of β_{j-1}. These bits correspond to the address of the relay P_{j+1}, the MAC γ_j, and $\beta_{j[\ldots(2N-1)\kappa-1]}$. The rest of β_j constitutes padding that we leave unchanged. The replacement is $\{* \| 0_\kappa \| rand_{[(2(N-j)+2)\kappa-|\delta|]}\}$.

$\underline{\mathcal{H}_6 \approx_I \mathcal{H}_7}$: Since $\rho(h_\rho(s_{j-1}))$ is a random string, this change can be reduced to $\text{Exp}^{\text{1-LR-CPA}}_{\text{OTP},\mathcal{A}}$. The new contents are exactly what β_{j-1} would contain if $P^{\leftarrow}_H$ were the last relay on the path. The suffix

$$[(2(N-j)+3)\kappa \ldots (2N+1)\kappa - 1]$$

of β_{j-1} is Φ_{j-1} by construction.[25] With this change, the layers β_0, …, β_{j-1} are now independent of the later layers $\beta_j, \ldots, \beta_{n-1}$.

Hybrid $\mathcal{H}_8$: The second part of the challenge onion still contains the padding $\Phi_0, \ldots, \Phi_{j-1}$ nested in β_j's Φ_j padding. To alleviate this, $\mathcal{H}_8$ replaces Φ_j with a random string of length $2j\kappa$.

$\underline{\mathcal{H}_7 \approx_I \mathcal{H}_8}$: In $\mathcal{H}_7$, Φ_j is calculated from Φ_{j-1} as

$$\Phi_j \leftarrow \rho(h_\rho(s_{j-1}))_{[(2(N-j)+3)\kappa\ldots]} \oplus \{\Phi_{j-1} \| 0_{2\kappa}\}.$$

Since $\rho(h_\rho(s_{j-1}))$ is a random string, the replacement

$$\Phi_j \leftarrow \rho(h_\rho(s_{j-1}))_{[(2(N-j)+3)\kappa\ldots]}$$

is indistinguishable from the original. As a result, this change reduces to $\text{Exp}^{\text{1-LR-CPA}}_{\text{OTP},\mathcal{A}}$.

Hybrid $\mathcal{H}_9$: This hybrid replaces the KEM instance used for the second part of the challenge onion after the honest relay. Previously, $\alpha_j = \alpha_{j-1}^{b_{j-1}}$ and $s_j = y_j^{xb_0\cdots b_{j-1}}$. Now, $\mathcal{H}_9$ picks a new $x' \xleftarrow{R} \mathbb{Z}_q^*$, setting $\alpha_j = g^{x'}$ and $s_j = y_j^{x'}$ and adjusting the later α_i and s_i following them accordingly.

$\underline{\mathcal{H}_8 \approx_I \mathcal{H}_9}$: $\mathcal{H}_1$ randomizes b_{j-1} into a unformly distributed element of $\mathbb{Z}_q^*$. It follows that $\alpha_{j-1}^{b_{j-1}}$ and $g^{x'}$ are identically distributed. The same argument holds for the later αs and secrets.

Hybrid $\mathcal{H}_{10}$: This hybrid "fixes" the second part of the onion so that it becomes a complete onion starting at P_S again. To that end, $\mathcal{H}_{10}$ starts building new $O'_0, \ldots, O'_{j-1}$ onion layers that follow the same path as the original $O_0, \ldots, O_{j-1}$. These new layers are built as a prefix to O_j, so the payload content of δ'_{j-1} is δ_j and β'_{j-1} is formed with β_j in its contents. The random oracle outputs $h_*(s'_{j-1})$ and $h_b(\alpha'_{j-1}, s'_{j-1})$ are randomized.

[25] Technically, Sphinx uses the output of ρ twice when building a header: Once to generate the padding and a second time to encrypt β. However, these use different substrings of the PRG output. $\mathcal{A}$ can thus submit the two different β contents with an appropriate zero padding to the 1-LR-CPA challenger to extract the random string required for the padding calculation.

Most importantly, α_j is now formed as $\alpha'^{\,b'_{j-1}}_{j-1}$, with the secrets being built analogously.

$\underline{\mathcal{H}_9 \approx_I \mathcal{H}_{10}}$: The new layers $O'_0, \ldots, O'_{j-1}$ are never actually given to the adversary. Their construction is thus entirely invisible to the attacker except for the change in how α_j is formed. Using $\mathcal{H}_8 \approx_I \mathcal{H}_9$'s argument, α_j still has the same distribution in both hybrids.

Hybrid $\mathcal{H}_{11}$: This hybrid replaces $\rho(h_\rho(s'_{j-1}))$ with a random string.

$\underline{\mathcal{H}_{10} \approx_I \mathcal{H}_{11}}$: See $\mathcal{H}_5 \approx_I \mathcal{H}_6$.

Hybrid $\mathcal{H}_{12}$: In $\mathcal{H}_8$, Φ_j is replaced with a random string instead of containing the previous Φ_{j-1}. Now, we replace that random string with

$$\rho(h_\rho(s'_{j-1}))_{[(2(N-j)+3)\kappa\ldots]} \oplus \{\Phi_{j-1}\|0_\kappa\},$$

so that Φ_j is formed as the j-th layer of padding in the $O'_0, \ldots, O_j, \ldots, O_{n-1}$ onion.

$\underline{\mathcal{H}_{11} \approx_I \mathcal{H}_{12}}$: Analogous to $\mathcal{H}_8$.

Hybrid $\mathcal{H}_{13}$: This hybrid replaces the randomized oracle outputs for s'_{j-1} with the actual outputs $h_*(s'_{j-1})$ and $h_b(\alpha'_{j-1}, s'_{j-1})$.

$\underline{\mathcal{H}_{12} \approx_I \mathcal{H}_{13}}$: See $\mathcal{H}_1 \approx_I \mathcal{H}_0$.

Hybrid $\mathcal{H}_{14}$: In this hybrid, we rewind all of the temporary changes made in the previous hybrids in the reverse order: $\mathcal{H}_{11}$, $\mathcal{H}_6$, $\mathcal{H}_2$, and $\mathcal{H}_1$.

$\underline{\mathcal{H}_{13} \approx_I \mathcal{H}_{14}}$: Apply the previous arguments in reverse. This concludes the $j < n$ case of the $\mathit{TLU}^{\rightarrow}$ proof.

We have proven that Sphinx satisfies $\mathit{TLU}^{\rightarrow}$. □

G.2 RSOR-Backw. Layer Unlinkability ($\mathit{SLU}^{\leftarrow}$)

Theorem 5. *Sphinx satisfies $\mathit{SLU}^{\leftarrow}$ under the GDH assumption.*

Proof. We prove that an adversary cannot distinguish the $b = 0$ scenario from the $b = 1$ scenario through a hybrid argument starting at $b = 0$ and ending at $b = 1$. For clarity, we separate the proof into two cases: One where $j^{\leftarrow} = 0$ and the other where $j^{\leftarrow} > 0$.

Case 1 ($j^{\leftarrow} = 0$)**:** In this case, the honest relay on the return path is identical to the exit relay on the forward path and is thus also the sender of the reply onion.

Hybrid $\mathcal{H}_0$: This hybrid is just the $b = 0$ case of $\mathit{SLU}^{\leftarrow}$ with $j^{\leftarrow} = 0$.

Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ replaces the random oracle outputs $h_*(s^{\leftarrow}_{n^{\leftarrow}-1})$ with random $\{0,1\}^\kappa$ bitstrings when building the reply header. **Proc** requests for onions with the challenge $\alpha^{\leftarrow}_{n^{\leftarrow}-1}$ are also served using these random keys.

$\underline{\mathcal{H}_0 \approx_I \mathcal{H}_1}$: This difference reduces to Sphinx-KEM-IND-CCA. See hybrid $\mathcal{H}_1$ in the $\mathit{TLU}^{\rightarrow}$ proof for details.

Hybrid $\mathcal{H}_2$: To ensure that only the challenge reply onion is "absorbed" by the reply receiver, the **Proc** oracle now returns $(\bot, \bot)$ on every request with the challenge $\alpha^{\leftarrow}_{n^{\leftarrow}-1}$ except if the rest of the header also matches the challenge reply (in that case, no output is produced at all).

$\underline{\mathcal{H}_1 \approx_I \mathcal{H}_2}$: See hybrid $\mathcal{H}_2$ in the $\mathit{TLU}^{\rightarrow}$ proof.

Hybrid $\mathcal{H}_3$: We repeat the changes in hybrids $\mathcal{H}_1$ and $\mathcal{H}_2$ for the honest relay P_H to randomize $h_*(s_{n-1})$ and reject challenge onions with modified headers in **Proc** at P_H.

$\underline{\mathcal{H}_2 \approx_I \mathcal{H}_3}$: Analogous to $\mathcal{H}_0 \approx_I \mathcal{H}_1$ and $\mathcal{H}_1 \approx_I \mathcal{H}_2$.

Hybrid $\mathcal{H}_4$: This hybrid exchanges $\pi(h_\pi(s_{n-1}), \cdot)$ with an RP.

$\underline{\mathcal{H}_3 \approx_I \mathcal{H}_4}$: A distinguisher $\mathcal{D}$ for these hybrids is easily converted into an attacker $\mathcal{A}$ on $\mathrm{Exp}^{\mathrm{prp}}_{\pi,\mathcal{A}}(\kappa)$.

Hybrid $\mathcal{H}_5$: In this hybrid, the honest exit relay P_H only sends the challenge reply in response to a **Reply** request if the onion received in the corresponding **Proc** request has a payload that matches the expected payload δ_{n-1} of the forward onion exactly.

$\underline{\mathcal{H}_4 \approx_I \mathcal{H}_5}$: See hybrid $\mathcal{H}_4$ in $\mathit{TLU}^{\rightarrow}$'s Case 1.

Hybrid $\mathcal{H}_6$: This hybrid replaces the reply header η_0 and symmetric key $\tilde{k}$ in the contents of δ_{n-1} with a new reply header that uses the same path, but different randomness and a random $\tilde{k}'$ when building the forward onion. The challenge reply onion's header is no longer read from the payload of the forward onion. Instead, the actual reply header and symmetric key are built "at" P_H when the challenge **Reply** is requested.

$\underline{\mathcal{H}_5 \approx_I \mathcal{H}_6}$: The behavior of the **Proc** and **Reply** oracles is indistinguishable between the two hybrids, since both process the onion and reply to it using the same header. The only other change is to the contents of δ_{n-1}. We can construct an attacker $\mathcal{A}$ on $\mathrm{Exp}^{\text{1-LR-CPA}}_{\pi,\mathcal{A}}(\kappa)$ using any distinguisher $\mathcal{D}$.

Hybrid $\mathcal{H}_7$: In this hybrid, replace $\pi(h_\pi(s^{\leftarrow}_{n^{\leftarrow}-1}), \cdot)$ and $\pi(\tilde{k}, \cdot)$ with RPs when building the reply onion.

$\underline{\mathcal{H}_6 \approx_I \mathcal{H}_7}$: Analogous to $\mathcal{H}_3 \approx_I \mathcal{H}_4$.

Hybrid $\mathcal{H}_8$: Previously, the first layer $O^{\leftarrow}_1$ of the reply onion had a payload encrypted with the RP $\pi(\tilde{k}, \cdot)$. We now replace this permutation with $\pi(h_\pi(s^{\leftarrow}_0), \pi(h_\pi(s^{\leftarrow}_1), \cdots \pi(h_\pi(s^{\leftarrow}_{n^{\leftarrow}-1}), \cdot) \cdots))$ while encrypting the same contents. The new permutation corresponds to how a forward onion payload is encrypted at the sender.

$\underline{\mathcal{H}_7 \approx_I \mathcal{H}_8}$: Since chaining the permutations $\pi(h_\pi(s^{\leftarrow}_i), \cdot)$ after the RP $\pi(h_\pi(s^{\leftarrow}_{n^{\leftarrow}-1}), \cdot)$ results in a new RP, we have simply replaced one RP with another.

Hybrid $\mathcal{H}_9$: Until now, the contents of the reply onion payload were $0_\kappa\|\mathrm{pad}^{\leftarrow}_{\kappa,N}\|m^{\leftarrow}$, where $m^{\leftarrow}$ is the adversary-chosen message. We replace them with $0_\kappa\|\bar{R}\|\mathrm{pad}^{\rightarrow}_{\kappa,N}\|\bar{m}$, where $\bar{R}$ and $\bar{m}$ are randomly chosen receivers and messages.

$\underline{\mathcal{H}_8 \approx_I \mathcal{H}_9}$: Analogous to $\mathcal{H}_5 \approx_I \mathcal{H}_6$.

Hybrid $\mathcal{H}_{10}$: This hybrid replaces $\rho(h_\rho(s^{\leftarrow}_{n^{\leftarrow}-1}))$ with a random string when building the "reply" onion header.

$\underline{\mathcal{H}_9 \approx_I \mathcal{H}_{10}}$: Reduce to $\mathrm{Exp}^{\mathrm{prg}}_{\rho,\mathcal{A}}(\kappa)$.

Hybrid $\mathcal{H}_{11}$: When building the "reply" onion header, $\mathcal{H}_{11}$ uses $*\|0_\kappa$ instead of $P_s\|I$ in the contents of $\beta^{\leftarrow}_{n^{\leftarrow}-1}$.

$\underline{\mathcal{H}_{10} \approx_I \mathcal{H}_{11}}$: Since the change in $\beta^{\leftarrow}_{n^{\leftarrow}-1}$'s contents does not affect the padding, we can reduce this change to $\mathrm{Exp}^{\text{1-LR-CPA}}_{\mathrm{OTP},\mathcal{A}}$ without further provisions. At this stage, $O^{\leftarrow}_1$ is built just like $\bar{O}_1$ in the $b = 1$ case of $\mathit{SLU}^{\leftarrow}$.

Hybrid $\mathcal{H}_{12}$: This hybrid rewinds all of the temporary changes made in the previous hybrids $\mathcal{H}_{10}$, $\mathcal{H}_7$, $\mathcal{H}_5$, $\mathcal{H}_4$, $\mathcal{H}_3$, $\mathcal{H}_2$, and $\mathcal{H}_1$ in that order.

$\underline{\mathcal{H}_{11} \approx_I \mathcal{H}_{12}}$: Apply the previous arguments in reverse. This concludes the $j^{\leftarrow} = 0$ case of the $\mathit{SLU}^{\leftarrow}$ proof.

Case 2 ($j^{\leftarrow} > 0$)**:** Now, we consider the case where the honest relay is on the reply path of the challenge onion.

Hybrid $\mathcal{H}_0$: The $b = 0$ scenario of $\mathit{SLU}^{\leftarrow}$ with $j^{\leftarrow} > 0$.

Hybrid $\mathcal{H}_1$: This hybrid performs the same steps as the hybrids $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$ in Case 1.

$\mathcal{H}_0 \approx_I \mathcal{H}_1$: See the corresponding hybrids in Case 1.

Hybrid $\mathcal{H}_2$: In this hybrid, we exchange the two permutations $\pi^{-1}(h_\pi(s^{\leftarrow}_{j^{\leftarrow}-1}), \cdot)$ and $\pi(h_\pi(s^{\leftarrow}_{n^{\leftarrow}-1}), \cdot)$ for RPs.

$\mathcal{H}_1 \approx_I \mathcal{H}_2$: A distinguisher $\mathcal{D}$ for this hybrid is easily converted into attackers $\mathcal{A}$ and $\mathcal{B}$ on $\mathrm{Exp}^{\mathrm{prp}}_{\pi^{-1},\mathcal{A}}(\kappa)$. and $\mathrm{Exp}^{\mathrm{prp}}_{\pi,\mathcal{B}}(\kappa)$.

Hybrid $\mathcal{H}_3$: In this hybrid, we change how the challenge onion's payload $\delta^{\leftarrow}_{j^{\leftarrow}-1}$ is processed at P_H. Normally, Sphinx calculates $\delta^{\leftarrow}_{j^{\leftarrow}}$ by applying the π^{-1} (P)RP to the payload. We replace π^{-1} with the RP $\pi(h_\pi(s^{\leftarrow}_{j^{\leftarrow}}), \pi(h_\pi(s^{\leftarrow}_{j^{\leftarrow}+1}), \cdots \pi(h_\pi(s^{\leftarrow}_{n^{\leftarrow}-1}), \cdot) \cdots))$.

$\mathcal{H}_2 \approx_I \mathcal{H}_3$: Analogous to Case 1's $\mathcal{H}_8$.

Hybrid $\mathcal{H}_4$: Now, instead of running the RP on $\delta^{\leftarrow}_{j^{\leftarrow}-1}$ during the challenge processing, $\mathcal{H}_4$ runs it on $0_\kappa \| \bar{R} \| \mathrm{pad}^{\rightarrow}_{\kappa,N} \| \bar{m}$ for a random receiver $\bar{R}$ and message $\bar{m}$. This completely replaces the original, adversary-chosen payload with a forward payload.

$\mathcal{H}_3 \approx_I \mathcal{H}_4$: See $\mathcal{H}_5 \approx_I \mathcal{H}_6$ in Case 1.

Hybrid $\mathcal{H}_5$: In this hybrid, P_H does not process the challenge onion. Instead, it outputs the header of $O^{\leftarrow}_{j^{\leftarrow}}$ and the payload as defined in $\mathcal{H}_3$ and $\mathcal{H}_4$.

$\mathcal{H}_4 \approx_I \mathcal{H}_5$: The processing output to the adversary is identical in both $\mathcal{H}_4$ and $\mathcal{H}_5$.

Hybrid $\mathcal{H}_6$: $\mathcal{H}_6$ replaces $\rho(h_\rho(s^{\leftarrow}_{j^{\leftarrow}-1}))$ with a random string of the same length when building $O^{\leftarrow}_1$.

$\mathcal{H}_5 \approx_I \mathcal{H}_6$: See Case 1's $\mathcal{H}_{10}$.

Hybrid $\mathcal{H}_7$: In this hybrid, we replace the first $(2(N - j^{\leftarrow}) + 3)\kappa$ bits of the contents of $\beta^{\leftarrow}_{j^{\leftarrow}-1}$ with randomness when building $O^{\leftarrow}_1$. This corresponds to the next relay address and the next MAC as well as the $(2(N - (j^{\leftarrow} - 1)) + 3)\kappa$-bit prefix of $\beta^{\leftarrow}_{j^{\leftarrow}}$ that does not contain padding. $\beta^{\leftarrow}_{j^{\leftarrow}}$ itself is still used for the $O^{\leftarrow}_{j^{\leftarrow}}$ reinserted at P_H.

$\mathcal{H}_6 \approx_I \mathcal{H}_7$: Reduce this change to $\mathrm{Exp}^{\text{1-LR-CPA}}_{\mathrm{OTP},\mathcal{A}}(\kappa)$ like in Case 1's $\mathcal{H}_{11}$. After this change, $\beta^{\leftarrow}_{j^{\leftarrow}-1}$ can be built without using any of the secrets $s^{\leftarrow}_{j^{\leftarrow}-1}, \ldots, s^{\leftarrow}_{n^{\leftarrow}-1}$ or the random oracle outputs derived from them.

Hybrid $\mathcal{H}_8$: This hybrid replaces the keys used to generate $O^{\leftarrow}_{j^{\leftarrow}}$. Instead of choosing $\alpha^{\leftarrow}_{j^{\leftarrow}} = \alpha_{j^{\leftarrow}-1}^{\leftarrow b^{\leftarrow}_{j^{\leftarrow}-1}}$ and $s^{\leftarrow}_{j^{\leftarrow}} = y_{j^{\leftarrow}}^{x^{\leftarrow} b^{\leftarrow}_0 \cdots b^{\leftarrow}_{j^{\leftarrow}-1}}$, $\mathcal{H}_8$ picks a new $x' \leftarrow^R \mathbb{Z}^*_q$ and lets $\alpha^{\leftarrow}_{j^{\leftarrow}} = g^{x'}$ and $s^{\leftarrow}_{j^{\leftarrow}} = y^{x'}_{j^{\leftarrow}}$. The later $\alpha^{\leftarrow}_i, s^{\leftarrow}_i, i > j^{\leftarrow}$ are calculated accordingly. [26]

$\mathcal{H}_7 \approx_I \mathcal{H}_8$: See hybrid $\mathcal{H}_9$ in $\mathit{TLU}^{\rightarrow}$'s Case 2.

Hybrid $\mathcal{H}_9$: In $\mathcal{H}_9$, we move the first $2j^{\leftarrow}\kappa$ bits of $\Phi_{n^{\leftarrow}-1}$ (corresponding to $\Phi_{j^{\leftarrow}}$) into the *rand*-padding inside $\beta^{\leftarrow}_{n^{\leftarrow}-1}$. To do so, the *rand*-padding is extended by $2j^{\leftarrow}\kappa$ bits and $\Phi_{n^{\leftarrow}-1}$ is truncated to $\Phi_{n^{\leftarrow}-1[2j^{\leftarrow}\kappa\ldots]}$.

$\mathcal{H}_8 \approx_I \mathcal{H}_9$: For this step, assume a distinguisher $\mathcal{D}$ for the two hybrids. We will construct an attacker $\mathcal{A}$ on $\mathrm{Exp}^{\text{1-LR-CPA}}_{\mathrm{OTP},\mathcal{A}}(2j\kappa)$. $\mathcal{A}$ submits

$$\rho(h_\rho(s^{\leftarrow}_{j^{\leftarrow}}))_{[(2(N-(j^{\leftarrow}+1))+3)\kappa\ldots(2(N-1)+3)\kappa-1]}$$
$$\oplus \cdots$$
$$\oplus \rho(h_\rho(s^{\leftarrow}_{n-2}))_{[(2(N-(n-1))+3)\kappa\ldots(2(N-(n-1-j^{\leftarrow}))+3)\kappa-1]}$$

[26] In $\mathcal{H}_1$, the random oracle outputs for $s^{\leftarrow}_{n^{\leftarrow}-1}$ at the reply receiver P_S are randomized. $\mathcal{H}_8$ remains consistent with this behavior by also randomizing the random oracle outputs for the "new" $s^{\leftarrow}_{n^{\leftarrow}-1}$. Since the "original" $s^{\leftarrow}_{n^{\leftarrow}-1}$ is not in use, there is still exactly one set of randomized oracle outputs at P_S.

and

$$\rho(h_\rho(s^{\leftarrow}_{n-1}))_{[(2(N-n)+3)\kappa\ldots(2(N-(n-j^{\leftarrow}))+3)\kappa-1]}$$

to the challenger and uses the challenge ciphertext as the substring

$$[(2(N - (j^{\leftarrow} + 1)) + 3)\kappa \ldots (2(N - 1) + 3)\kappa - 1]$$

of $\beta^{\leftarrow}_{n-1}$. $\Phi^{\leftarrow}_{j^{\leftarrow}}$ is already a random string due to it being the result of an XOR operation with the random string $\rho(h_\rho(s^{\leftarrow}_{j^{\leftarrow}-1}))$, so the first scenario simulates $\mathcal{H}_3$ and the second simulates $\mathcal{H}_4$.

Hybrid $\mathcal{H}_{10}$: This hybrid replaces $\rho(h_\rho(s^{\leftarrow}_{n^{\leftarrow}-1}))$ with a random string when building $O^{\leftarrow}_{j^{\leftarrow}}$.

$\mathcal{H}_9 \approx_I \mathcal{H}_{10}$: Analogous to this case's $\mathcal{H}_5 \approx_I \mathcal{H}_6$.

Hybrid $\mathcal{H}_{11}$: Just like in Case 1's $\mathcal{H}_{11}$, we use $*\|0_\kappa$ instead of $P_s\|I$ in the contents of $\beta^{\leftarrow}_{n^{\leftarrow}-1}$.

$\mathcal{H}_{10} \approx_I \mathcal{H}_{11}$: Analogous to Case 1's $\mathcal{H}_{10} \approx_I \mathcal{H}_{11}$.

Hybrid $\mathcal{H}_{12}$: This hybrid rewinds the temporary changes made in the previous hybrids: $\mathcal{H}_{10}$, $\mathcal{H}_7$, $\mathcal{H}_6$, $\mathcal{H}_2$, and $\mathcal{H}_1$ are unwound in that order. Note that unwinding $\mathcal{H}_7$ means replacing the random contents in $\beta^{\leftarrow}_{j^{\leftarrow}-1}$ with the "legitimate" rest of the reply header, not the βs that were transformed into the $\bar{O}_1$ header.

$\mathcal{H}_{11} \approx_I \mathcal{H}_{12}$: Apply the previous arguments in reverse. This concludes the $j^{\leftarrow} > 0$ case of the $\mathit{SLU}^{\leftarrow}$ proof.

We have now shown that $\mathcal{H}_0 \approx_I \mathcal{H}_{12}$ with $\mathcal{H}_0 = \mathit{SLU}^{\leftarrow}_{b=0}$ and $\mathcal{H}_{12} = \mathit{SLU}^{\leftarrow}_{b=1}$ for both the $j^{\leftarrow} = 0$ and $j^{\leftarrow} > 0$ cases, proving that Sphinx satisfies $\mathit{SLU}^{\leftarrow}$. □

G.3 RSOR-Tail Indistinguishability ($\mathit{STI}^{\leftrightarrow}$)

Theorem 6. *Sphinx satisfies $\mathit{STI}^{\leftrightarrow}$ under the GDH assumption.*

Proof. We prove that an adversary cannot distinguish the $b = 0$ scenario from the $b = 1$ scenario through a hybrid argument starting at $b = 0$ and ending at $b = 1$. We gradually transform the O_j onion into the $\bar{O}_0$ onion in successive hybrids.

Hybrid $\mathcal{H}_0$: This hybrid is just the $b = 0$ case of $\mathit{STI}^{\leftrightarrow}$.

Hybrid $\mathcal{H}_1$: If $j = 0$, the following hybrids do nothing. Skip to hybrid $\mathcal{H}_6$ in that case. In this hybrid, we begin truncating the forward path. When building O_j, choose $\alpha_i := g^{x'}$ and $s_i := y_i^{x'}$ with a random $x' \in \mathbb{Z}^*_q$.

$\mathcal{H}_0 \approx_I \mathcal{H}^1_1$: See hybrid $\mathcal{H}_9$ in the $\mathit{TLU}^{\rightarrow}$ proof's Case 2. Note that b_{j-1} is a random oracle output that is never used elsewhere.

Hybrid $\mathcal{H}_2$: When building the Sphinx packet, replace $h_\rho(s_{j-1})$ with a random $\{0,1\}^\kappa$-bitstring. $h_\rho(s_{j-1})$ is only required to calculate Φ_j.

$\mathcal{H}^j_1 \approx_I \mathcal{H}_2$: s_{j-1} behaves like a uniformly random group element. By definition of a random oracle, these hybrids are indistinguishable.

Hybrid $\mathcal{H}_3$: Replace $\rho(h_\rho(s_{j-1}))$ with a random string. Since Φ_j is calculated from an XOR operation with $\rho(h_\rho(s_{j-1}))$, it is now also a random string.

$\mathcal{H}_2 \approx_I \mathcal{H}_3$: See hybrid $\mathcal{H}_6$ in the $\mathit{TLU}^{\rightarrow}$'s Case 2.

Hybrid $\mathcal{H}_4$: When building β_{n-1}, $\mathcal{H}_3$ extends the random bits in its contents by $2j\kappa$ extra random bits and truncates Φ_{n-1} by the same amount.

$\mathcal{H}_3 \approx_I \mathcal{H}_4$: See hybrid $\mathcal{H}_9$ in $\mathit{SLU}^{\leftarrow}$'s Case 2.

Hybrid $\mathcal{H}_5$: This hybrid does not generate $\alpha_0, \ldots, \alpha_{j-1}, \beta_0, \ldots, \beta_{j-1}, \gamma_0, \ldots, \gamma_{j-1}, \delta_0, \ldots, \delta_{j-1}$, or $\Phi_0, \ldots, \Phi_{j-1}$.

$\mathcal{H}_4 \approx_I \mathcal{H}_5$: The parts of the packet destined for the path prefix are not used in $\mathcal{H}_4$, so not generating them in the first place goes unnoticed by any distinguisher.

Hybrid $\mathcal{H}_6$: If $j^{\leftarrow} = n^{\leftarrow}$, we can skip the following hybrids because the original and truncated paths are identical. We thus assume $j^{\leftarrow} < n$ in the following. This hybrid replaces $h_\rho(s^{\leftarrow}_{j^{\leftarrow}-1})$, $h_\mu(s^{\leftarrow}_{j^{\leftarrow}-1})$, and $h_\pi(s^{\leftarrow}_{j^{\leftarrow}-1})$ with random $\{0,1\}^\kappa$ strings. In order to process **Proc** requests for $P^{\leftarrow}_H$ with αs that are identical to the α of the challenge header, $\mathcal{H}_6$ uses the new random keys.

$\mathcal{H}_5 \approx_I \mathcal{H}_6$: See hybrid $\mathcal{H}_1$ in $TLU^{\rightarrow}$'s Case 1.

Hybrid $\mathcal{H}_7$: This hybrid adjusts the processing of onions at the second honest relay $P^{\leftarrow}_{j^{\leftarrow}}$ by returning $(\perp, \perp)$ for any **Proc** request with the challenge α unless the entire header matches the one of the challenge onion.

$\mathcal{H}_6 \approx_I \mathcal{H}_7$: See hybrid $\mathcal{H}_2$ in $TLU^{\rightarrow}$'s Case 1.

Hybrid $\mathcal{H}_8$: This hybrid replaces $\rho(h_\rho(s^{\leftarrow}_{j^{\leftarrow}-1}))$ with a random string when forming $\beta^{\leftarrow}_{j^{\leftarrow}-1}$ and $\Phi_{j^{\leftarrow}}$.

$\mathcal{H}_7 \approx_I \mathcal{H}_8$: The difference between these two hybrids reduces to $\mathrm{Exp}^{prg}_{\rho,\mathcal{A}}(\kappa)$.

Hybrid $\mathcal{H}_9$: This hybrid replaces the actual contents of $\beta^{\leftarrow}_{j^{\leftarrow}-1}$. In $\mathcal{H}_8$, these were

$$\{n^{\leftarrow}_{j^{\leftarrow}} \| \gamma^{\leftarrow}_{j^{\leftarrow}} \| \beta^{\leftarrow}_{j^{\leftarrow}[\ldots(2N-1)\kappa-1]}\}.$$

$\mathcal{H}_9$ replaces the first $(2(N - j^{\leftarrow}) + 3)\kappa$ bits of that with

$$\{P_S \| I^{\leftarrow} \| rand_{[(2(N-j^{\leftarrow})+2)\kappa-|\Delta|]}\}.$$

The rest of $\beta^{\leftarrow}_{j^{\leftarrow}-1}$ is unchanged padding.

$\mathcal{H}_8 \approx_I \mathcal{H}_9$: See $\mathcal{H}_3 \approx_I \mathcal{H}_4$.

Hybrid $\mathcal{H}_{10}$: This hybrid reverts the temporary changes in the previous hybrids: $\mathcal{H}_8$, $\mathcal{H}_7$, and $\mathcal{H}_6$ are unwound in that order.

$\mathcal{H}_9 \approx_I \mathcal{H}_{10}$: Apply the previous arguments in reverse.

Hybrid $\mathcal{H}_{13}$: This hybrid does not generate $\alpha^{\leftarrow}_{j^{\leftarrow}}, \ldots, \alpha^{\leftarrow}_{n^{\leftarrow}-1}$, $\beta^{\leftarrow}_{j^{\leftarrow}}, \ldots, \beta^{\leftarrow}_{n^{\leftarrow}-1}$, $\gamma^{\leftarrow}_{j^{\leftarrow}}, \ldots, \gamma^{\leftarrow}_{n^{\leftarrow}-1}$, $h_\pi(s^{\leftarrow}_{j^{\leftarrow}}), \ldots, h_\pi(s^{\leftarrow}_{n^{\leftarrow}-1})$, or $\Phi^{\leftarrow}_{j^{\leftarrow}}, \ldots, \Phi^{\leftarrow}_{n^{\leftarrow}-1}$.

$\mathcal{H}_{12} \approx_I \mathcal{H}_{13}$: These components are no longer required to form the challenge packet: The later $\beta^{\leftarrow}$s have been replaced by the new contents of $\beta^{\leftarrow}_{j^{\leftarrow}-1}$ along with the later $\Phi^{\leftarrow}$s.

$\mathcal{H}_{13}$ is the $b = 1$ case of $STI^{\leftrightarrow}$. Since $\mathcal{H}_0 \approx_I \mathcal{H}_{13}$, Sphinx satisfies $STI^{\leftrightarrow}$. □