EROR: Efficient Repliable Onion Routing with Strong Provable Privacy

Michael Klooß
michael.klooss@aalto.fi
Aalto University
Department of Computer Science
Espoo, Finland

Andy Rupp
andy.rupp@uni.lu
University of Luxembourg
Department of Computer Science
Esch-sur-Alzette, Luxembourg

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

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

Christiane Weis
christiane.weis@neclab.eu
NEC Laboratories Europe
Security Group
Heidelberg, Germany

ABSTRACT

To provide users with anonymous access to the Internet, onion routing and mix networks were developed. Assuming a stronger adversary than Tor, Sphinx is a popular packet format choice for such networks due to its efficiency and strong protection. However, it was recently shown that Sphinx is susceptible to a tagging attack on the payload in some settings. The only known packet formats which prevent this attack rely on advanced cryptographic primitives and are highly inefficient, both in terms of packet sizes and computation overhead.

In this paper, we provide the first packet format that protects against the tagging attack with an acceptable overhead. At the cost of doubling the payload size, we are able to build a provably private solution from basic cryptographic primitives. Our implementation demonstrates that our solution is as computationally efficient as Sphinx, beating previous schemes by a large margin. For our security proof, we first strengthen the state-of-the-art proof strategy, before applying it to our solution to demonstrate that not only the tagging attack is prevented, but our scheme is provably private.

CCS CONCEPTS

• **Security and privacy** → *Network security*; • **Networks** → *Network privacy and anonymity*;

KEYWORDS

anonymous communication; onion routing; mix nets; privacy

1 INTRODUCTION

When accessing the Internet, IP addresses can easily serve as personal identifiers. Users however might want to stay anonymous for various reasons, like e.g. whistleblowing, browsing for information related to a disease or simply to protect themselves from surveillance. To prevent the leakage of the users' IP address and thereby to enable anonymous access to the Internet, onion routing and mix networks employ a series of relays between the sender and receiver of a message.

While onion routing [17] and mix networks [9] differ in their adversary model, the idea for the underlying packet format, namely layered encryption, is very similar. Layered encryption uses a series of encryption layers, of which each relay removes one. Thereby, it is ensured that the sender and potentially adversarial receiver (as well as the message) are unlinkable to each other – as long as at least one honest relay was used.

Practically, Tor [14] has been a great success. In recent years, however, new deployed systems, like the Nym network,[1] as well as various academic proposals [11, 10, 23, 25, 13, 12] aim to offer alternatives for stronger adversary models. A large subset [11, 10, 23] of these new onion routing and mix networks rely on the packet format Sphinx [12].

Sphinx packets consist of a header and payload. The header contains the routing information and temporary keys and is protected with a message authentication code (MAC) for every hop on the path. The payload contains the encrypted message for the receiver. Sphinx enables replies from the message receiver back to the still anonymous sender, by providing the receiver with a header for the backward path. To reply, the receiver attaches her reply message to the reply header. For those reply packets, indistinguishability to request packets is required. This ensures that all participants, no matter if they send only requests, replies or both, contribute to the anonymity set of every other participant.

Challenge for onion routing with replies. Several systems use Sphinx to implement an integrated system[2] [11, 10], i.e. the receiver is the last node on the path and discovers the included message after processing the onion packet. In this setting, Sphinx is however susceptible to a tagging attack [19]. As Sphinx does not detect payload modifications at intermediate relays, an adversary can modify the payload and only the final receiver notices this modification. A collaborating first relay and receiver can thus recover information about the sender-receiver relationship, which the network wanted to protect. Consider, e.g., Alice contacting a server dedicated to provide information for cancer patients, clearly hiding the sender-receiver connection is critical in this case.

Achieving both, request-reply indistinguishability and protection against the payload tagging attack at the same time, however, presents a big challenge: To prevent the attack, we need to include

[1] https://nymtech.net/, accessed at 2023-12-19
[2] Sphinx has not originally been designed for this setting.

strong integrity protection for the forward payload, where payload integrity needs to be verifiable at every relay such that an honest relay is able to drop a tampered packet. To achieve request-reply indistinguishability, one might conclude that the same protection mechanism is required for both forward and backward payload. While a classical MAC can be calculated by our trusted sender for forward onions (request), the same is not possible for backward onions (replies) as the sender does not know the backward payload beforehand. The one that knows the reply, the receiver, could compute such MACs. He is, however, not trusted in the adversary model and could, colluding with adversarial relays, use the receiver-added MACs to recognize the onion even after it passed honest relays [20].

Recent work [20] tackled this challenge by using heavyweight cryptography to realize an *implicit* protection of the payload. Employing a version of updatable encryption with plaintext integrity for the payload ensures that once sender/receiver chose a payload, its encryption can be randomized but its content not be modified or replaced without detection by an honest relay on the path. The second and, to the best of our knowledge, only other solution [20] realizes the implicit payload protection via SNARGs. There, at every step on the onion path, the relay proves that its output is a correctly processed onion. Due to their poor efficiency, both protocols introduced in [20] can only be seen as a theoretical contribution as also admitted by the authors.

Our contribution. After a critical review of the payload tagging attack, we realize that it is only dangerous in the forward direction as there the potentially adversarial receiver recognizes the tagging. In an attack on the backward direction, a modification results in a decryption failure at the honest (request) sender. However, the honest (request) sender will not share this observation with an adversary. Hence, the payload tagging attack is just a denial of service attack in the backwards direction, and the adversary could simply drop the onion instead of modifying it. Thus, tagging in the backwards direction does not allow to learn privacy critical information about honest (request) senders.

This shows that, regarding the tagging attack, the backward payload does not need the same strong form of integrity protection as the forward payload. In fact, the security model from [20] actually does not enforce any integrity protection at all for the backward payload. The alleged need to also have strong integrity protection for the backward payload results from the request-reply indistinguishability requirement.

Our most important insight to avoid the inefficient solutions required for an implicit payload protection is that, we can keep the indistinguishability of forward and backward onions without having the same protection for forward and backward payloads. For this, we allocate a specific part of the payload for the forward message and another part for the reply. The forward part of the payload (including the request) is protected at every step of the path with a sender pre-calculated MAC, while we gladly forgo this payload protection for replies. As only one part of the payload (forward or backward) is actually used in any packet, we fill the other part with dummy data. Request-reply indistinguishability is given as long as we can hide which part of the payload includes only dummy data from the intermediate relays.

This insight allows us to avoid inefficient building blocks like SNARGs and updatable encryption. Indeed, the construction of our efficient repliable onion routing scheme, dubbed EROR, is generic and only relies on symmetric-key primitives and public-key encryption. Additionally, we make use of "weaker", but still sufficient security requirements for building blocks to allow for even more efficient primitives in particular. Concretely, in [2, 20], PRP-CCA secure pseudorandom permutations were used for symmetric encryption, whereas we relax this requirement, allowing for stream ciphers.

We implement and benchmark the EROR packet format, and compare it to Sphinx (even though Sphinx does not satisfy out stronger security requirements). Our implementation of EROR has larger onions, but the size stays within a factor of 2 compared to Sphinx. Packet processing for EROR is about twice as fast; our larger header contains more key material, which reduces public-key operations. Onion creation is slightly slower. Compared to the packet formats in [20] which presumably satisfy our stronger security, EROR is at least two orders of magnitude faster.

As an additional contribution, we strengthen the state-of-the-art security requirements and properties for onion routing packet formats (in comparison to [20], see Remark 4.3) to also include end-to-end integrity for the backward direction, simplify the underlying scheme definition, and reduce the number of necessary security properties.

2 PRELIMINARIES

This section introduces our notation and background. Cf. Appendix A.1 for cryptographic notions.

2.1 Basic Notation

We use standard pseudo-code notation throughout. By λ we denote the security parameter, and a function $f\colon \mathbb{N} \to \mathbb{R}$ is negligible if $f \in \lambda^{\omega(1)}$. We write $x \leftarrow_{\mathrm{R}} X$ for uniform random sampling of an element from the set X. To avoid cluttering our pseudocode with bitlength specifications, we use the shorthand $x \leftarrow_{\mathrm{R}} \$$ for sampling uniformly at random from an implicitly specified set, usually, a ciphertext space of some (known) bitlength.

2.1.1 *Pseudocode and Onion-specific Conventions.*

Our notation regarding onions, e.g. in definitions, pseudo-code and proofs, has the following conventions:

- Superscripts denote layer depth. E.g., P^i is the i-th party and K^i is the key of the i-th processing (unwrapping) node.
- Indices distinguish between similar objects, in particular, forward ($\rightarrow$) or backward direction ($\leftarrow$), e.g. parties $P^i_{\rightarrow}$ $P^i_{\leftarrow}$, or derived keys, e.g., K^i_{SKE} and K^i_{MAC}. If a direction is required, but not specified, it will be clear from the context.

We use "object-oriented" notation, e.g., $O.hdr$ to denote the header hdr of an onion O.

2.2 Background on Onion Routing and Mix Networks

Onion routing [14] and mix networks [9] aim to provide sender anonymity by preventing the adversary from linking the sender's IP address to the receiver and sent message. To achieve this, they

rely on relays provided by volunteers. Onion routing classically assumes local adversaries, while mix networks consider global adversaries. Both additionally assume that some receivers and a subset of relays collude with the adversary.

To send a message anonymously, the sender picks a sequence of relays, the *packet's path*, and encrypts the packet layer by layer for each hop on the path. The sender additionally provides information about the next relay in the path to each relay via a header or previously created tunnels. The relays then *process* the packet, i.e. decrypt it and forward it to the next hop. The last relay retrieves the included message. Note that as long as one of the chosen relays is honest and the processing perfectly unlinks incoming from outgoing packets, the adversary can indeed not link the sender of the packet to the included message or final receiver. Additionally, networks often support replies from the receiver back to the anonymous sender. To differentiate we use *forward* communication or *request* for the initial contact from the sender to the receiver and *backward* or *reply* for the communication in the reverse direction.

Onion routing and mix networks can be used in two different system models. In the *service model* the receiver is unaware of the onion routing or mix network and the last relay, the *exit relay*, retrieves both the message for the receiver and receiver address to act as a proxy for the sender's communication. In the *integrated system model* the receiver is the last relay and retrieves the message as processing result of the received packet.

It has proven useful to view the problem of providing unlinkability as two orthogonal subproblems [5, 19, 12]. The first is concerned with providing a secure packet format, i.e. preventing any linking based on the actual bits of the sent and received packets at honest relays. The second subproblem is to provide protection against additional side channels, like timing or traffic patterns of the packets.

2.3 Related Work on Onion Routing and Mix Network Packet Formats

The first ideas for onion routing and mix networks included their own packet formats. Chaum's mixnet [9] adds random delays to the idea of layered encryption described in Section 2.2. The first work on onion routing was proposed by Goldschlag, Reed and Syverson [17] and relies on a clever tunnel setup, while avoiding the additional delays. This work served as foundation for the well-known Tor network [14]. Over the years many works followed those seminal papers, e.g. [11, 10, 23, 25, 13, 12], and a strategy for provable privacy for their packet formats were developed [5]. The provable privacy strategy has been first used for the correction [25] of Minx [13] and in an attempt to prove the security of Sphinx [12]. As Sphinx has been used for many recent academic proposals [11, 10, 23], as well as for the deployed Nym network,[3] we focus on Sphinx and recent follow-up works as related work.

Sphinx packets consist of a header, for routing information and keying material, and a payload, for the message. Employing nearly only symmetric cryptographic primitives, Sphinx is highly efficient. Further, Sphinx supports replies by precalculating the header for the backward path and sending it to the receiver, who can attach the reply message as payload. To guarantee strong privacy, Sphinx requires indistinguishability between request and reply packets at intermediate relays.

Sphinx was originally proposed in the service model, but has been used by other works in the integrated system model. If it is used in the integrated system model, however, a *payload tagging attack* can be used to link the sender to the receiver, as has recently been discovered [19]. To the best of our knowledge only two solutions proposed in [20] exist that prevent the payload tagging attack. They both rely on recent cryptographic primitives (namely updatable encryption (UE) and succinct non-interactive arguments (SNARGs)) and include a high overhead (cf. Section 7).

2.4 Background on Formalization

2.4.1 Universal Composability. In the Universal Composability (UC) framework [6], the required privacy is defined by the *ideal functionality*. The ideal functionality interacts with the adversary (as combination of all adversarial parties) and the environment.

A protocol realizes the ideal functionality if any real world attack can be translated into an attack on the ideal functionality by a simulator. Crucially, all observations that the real world adversary or the environment make in the simulated attack on the ideal functionality are indistinguishable from the real world attack on the protocol. The real world protocol thus indeed cannot reveal more information than the ideal functionality.

2.4.2 Game-based Security Properties. Game-based security properties typically challenge an imagined game adversary to distinguish between two executions of a game. Based on the observed outcome of one randomly chosen option, the adversary has to make a guess as to which option was chosen. If no adversarial algorithm is able to distinguish the options, the property is achieved. As proofs for game-based security properties are usually more convenient than proving realization of an ideal functionality, it is useful to derive a set of game-based properties that implies the realization of an ideal functionality [5].

2.5 Related Work Formalization of Onion Routing and Mix Network Packet Formats

While different works conduct efforts to formalize Onion Routing and Mix Networks [21, 3, 15, 8, 7, 18], the current state of the art for their network packet formats proposes ideal functionalities in the UC-framework [5, 2, 19, 20]. Additionally, game-based properties are given, such that if all properties are satisfied by an OR scheme, it also realizes the ideal functionality. The seminal paper of Camenisch and Lysyanskaya [5] introduces this approach for Onion Routing packet formats, but their proof strategy was later discovered to be flawed [19]. The proposed game-based properties were not sufficient to imply realization of the ideal functionality. Kuhn et al. [19] corrected the necessary properties for this functionality. Later on, this work was extended similtaneously by Kuhn et al. [20] and Ando and Lysyanskaya [2] to additionally allow for replies. The work of Ando and Lysyanskaya [2] however proposes a weaker privacy requirement in the ideal functionality as compared to Kuhn et al. [20], as the latter also require to protect against payload tagging attacks for replies. We adapt this strong requirement and extend it by an end-to-end integrity requirement for the reply path, which is also present in [2].

[3] https://nymtech.net/, accessed at 2023-12-19

3 THREAT MODEL AND TAGGING ATTACK

3.1 Threat Model

In this work, we assume the mix network adversary model. That is, we aim to protect an honest sender against a global adversary colluding with the receiver and all but one relay on the (forward and backward) path. Any non-honest party can maliciously deviate from the protocol, in particular, (try to) inject modified onions.

We limit our scope to solely focus on the subproblem of providing a secure packet format that supports replies in the integrated system model. As this packet format can be used for both onion routing and mix networks and to be in line with closely related work [5, 2, 19, 20], we use the onion routing terminology.

3.2 Tagging Attack

In line with Sphinx and the mix network adversary model, the tagging attack [19] assumes a corrupted first relay and receiver. The adversary modifies the *payload* of the victim's packet at the first relay. As only the header's integrity is protected by a MAC, the honest relays before the receiver do not notice anything suspicious. The final receiver, however, notices that the payload was modified as the integrity check fails. Thereby the adversary learns which adversarial receiver the victim sender was trying to contact.

3.3 Impact of the Tagging Attack

The only prerequisites for the attack are to actively corrupt the first relay connected to the victim sender (to modify the payload) and to passively corrupt the (suspected) receiver (to observe the failed onion processing). Then the attack allows to confirm or disprove the sender-receiver relationship. By corrupting multiple suspected receivers the relationship can be tested for all of them.

There are critical application scenarios where those prerequisites are easily satisfied: For instance, in an authoritarian country, agencies might setup their own onion routing relays and honeypot servers (e.g. for whistleblowing). These relays and servers might also be rented by them in foreign countries. Search warrants might be another means to "corrupt" servers.

In the integrated system model, this attack *always* applies. We note that for the practically used mix network Nym, their usage of Sphinx actually corresponds to the integrated system model: "The last layer of Sphinx encoding is removed by the application itself, not the exit mix (like in Tor)."[4]

In the service model it depends on whether the combination of sender and used exit relay is considered private information. E.g., if each exit relay only serves a small subset of receivers in general, the linking is clearly interesting information as the adversary can use the linking to reduce the set of potential communication partners of the sender (from the set of all receivers to the set of receivers of the linked exit relay). If the exit relay is chosen uniformly random every time, the sender-exit-relay linking might be considered acceptable leakage [24].

[4]https://blog.nymtech.net/sphinx-tl-dr-the-data-packet-that-can-anonymize-bitcoin-and-the-internet-18d152c6e4dc, accessed at 2023-12-19

3.4 Protecting forward direction is sufficient

Notice that the tagging attack works as a change in the onion is introduced early in the onion's path and recognized at the end of its processing. Now, given our adversary model, the response receiver is the request sender, and thus always honest. Therefore, in the backwards direction the adversary cannot recognize a response at the response receiver anymore. The adversary's tagging early in the path however still changes the payload from what the response sender actually wanted to send, resulting in a decryption error. This effectively destroys the communication functionality, but in itself does not reveal private information. Hence, in the backwards direction, modification of the payload becomes a denial of service attack. Note that if the adversary's goal is a denial of service, it could just drop the onion. No packet format can protect against attacks based on denial of service attack and hence we consider denial of service attacks as out of scope.

3.5 Difficulty of securing the payload

To prevent the tagging attack while still providing request-reply indistinguishability is challenging. While payload in the forward direction is to protect with a classical sender pre-calculated MAC, this approach cannot be used for the payload in the backward direction, as the sender does not know the response payload and cannot pre-calculate the MAC, and as according to the adversary model the receiver is not trusted. When one tries to have a single forward/backward payload, the payload must be handled symmetrically in both directions to achieve indistinguishability of forward and backward packets. Thus, the payload protection/integrity provided in the forward direction, must "by symmetry" also be provided in the backward direction. As the receiver could be malicious, implementing such a protection properly and forward/backward-symmetrically is highly non-trivial. It seems to require heavy cryptographic tools as seen in [20].

We instead decide to avoid such heavy cryptography while preserving forward/backward-symmetry of the payload — at the "mere" price of having a separate (hop-by-hop integrity protected) forward and an (only end-to-end integrity protected) backward payload for each packet.

4 OR SECURITY DEFINITION AND PROPERTIES

In this section, we strengthen the formalization of Kuhn et al. [20] to include end-to-end integrity for the backward path, as well as recognition of the request if a reply is received. Additionally, we simplify the model without losing expressibility. In line with this change, we reduce the number of properties that need to be proven for realization of the ideal functionality.

4.1 Scope of the Model

Similar to previous work [19, 20], we build our model with the following common constrictions of OR packet formats.

There exists a fixed *maximum* path length that the packet format supports. Honestly chosen paths are acyclic (adversarial users however might deviate from that). The OR format runs the processing algorithm, which outputs the received message, at the receiver before the receiver can reply to the onion. Onions consist of

a header and a payload part. Duplicate detection is realized on the header hdr, i.e. every onion with the same header as an already processed one results in a processing fail except with negligible probability. Further, there exists a public key infrastructure (or any other means that ensures the sender's knowledge of authentic public keys for the relays).

4.2 Onion Routing Scheme

The following definition slightly simplifies the model of a repliable onion routing scheme from Kuhn et al. [20] by limiting to only form onions at the first layer (as needed for sending). Except for this and changes in the notation, we verbatimly reuse [20]'s definition.

Definition 4.1 (Repliable Onion Routing (OR) Scheme). A Repliable OR Scheme is a tuple of PPT algorithms (G, $\mathsf{FormOnion}$, $\mathsf{ProcOnion}$, $\mathsf{ReplyOnion}$) defined as:

Key generation. $G(1^\lambda, p, P^i)$ outputs a key pair (pk^i, sk^i) on input of the security parameter 1^λ, some public parameters p and a router identity P^i.

Forming an onion. $\mathsf{FormOnion}(\mathit{onionParams})$ returns an onion layer O^1 on input of the onion parameters $\mathit{onionParams} = (m, \mathcal{P}_\rightarrow, \mathcal{P}_\leftarrow, (pk)_{\mathcal{P}_\rightarrow}, (pk)_{\mathcal{P}_\leftarrow}; \mathcal{R})$ with message m, a forward path $\mathcal{P}_\rightarrow = (P^1, \dots, P^n)$, a backward path $\mathcal{P}_\leftarrow = (P^1_\leftarrow, \dots, P^{n_\leftarrow}_\leftarrow)$, public keys $(pk)_{\mathcal{P}_\rightarrow} = (pk^1, \dots, pk^n)$ for forward relays, public keys $(pk)_{\mathcal{P}_\leftarrow} = (pk^1_\leftarrow, \dots, pk^{n_\leftarrow}_\leftarrow)$ for backward relays and randomness $\mathcal{R}$. The backward path can be empty if the onion is not intended to be repliable.

Forwarding an onion. $\mathsf{ProcOnion}(sk, O, P)$ returns the next layer of the onion and identity of the next router (O', P') on input of P's secret key sk, an onion layer O and the router identity P. (O', P') equals $(\bot, \bot)$ in case of an error or $(m, \bot)$ if P is the recipient.

Replying to an onion. $\mathsf{ReplyOnion}(sk, m_\leftarrow, O, P)$ returns a reply onion $O_\leftarrow$ along with the next router $\mathcal{P}_\leftarrow$ on input of P's secret key sk, a reply message $m_\leftarrow$, a received (forward) onion O and the receiver identity P. $O_\leftarrow$ and $\mathcal{P}_\leftarrow$ attains $\bot$ in case of an error.

Remark 4.2 (Forming later layers). Kuhn et al. [20] require that an OR scheme directly specifies how to form later layers O^i with $i > 1$ of the onions, which they used in the technical proofs. We avoid this unnatural forming of partially peeled onions and instead use new complete onions at the corresponding proof steps. This reduces the number of properties that need to be shown for secure OR schemes at the cost of each single property being slightly stronger.

An OR scheme is correct if the repeated processing of an onion results in the chosen (backward) path being used and finally the chosen message delivered to the chosen (backward) receiver. See Appendix A.2 for details.

4.3 Ideal Functionality

On a high-level, the ideal functionality replaces all onions with only temporary identifiers *tempID*. At each honest relay the *tempID* is replaced by a newly drawn random one, ensuring that the onions of honest senders before and after the honest relay are perfectly unlinkable to each other. Corrupt senders can however recognize their own onion layers, thus for these temporary identifiers all information of the onion parameters are output. The correlation between reply (also called "backward") onions and original requests (or "forward" onions) is stored in the functionality as well. In the version of [20] this linking was not output to the environment $\mathcal{Z}$ or adversary for honest senders. Futher in the previous work [20], honest nodes only inform the environment $\mathcal{Z}$ about any further processable or repliable *tempID* they received. $\mathcal{Z}$ does not learn whether a non-repliable forward (or backward) message was delivered or which (repliable or non-repliable) message was received.

We strengthen the ideal functionality to enforce end-to-end integrity for replies, as well as correct matching to request for onions of honest (request) senders. Thus, in our version, honest nodes additionally inform $\mathcal{Z}$ when they successfully received a message (forward or backward) by outputting the message and the information whether it was forward or backward. If the received message was a reply (backward message), they further inform $\mathcal{Z}$ to which request (own forward message) this reply belongs.

We detail the ideal functionality in Algorithm 1 and 2 and highlight the changes (except for notation) as compared to previous work [20] in blue. Notice that while the received message m was sent to the receiving party P^r before, this message was not forwarded to the environment in [20].

We detail the ideal functionality in Algorithm 1 and 2 and highlight the changes (except for notation) as compared to previous work [20] in blue. Notice that while the received message m was sent to the receiving party P^r before, this message was not forwarded to the environment in [20].

4.4 Auxiliary Functions

To express the security properties, we inherit one auxiliary function ($\mathsf{RecognizeOnion}$) from [20] and introduce a new function ($\mathsf{ExtractPayload}$) that is necessary because of the additional message output to the environment. Recall from Section 4.2 that we use *onionParams* as shorthand for all parameters used in $\mathsf{FormOnion}$.

$\mathsf{RecognizeOnion}((i, d), O, \mathit{onionParams})$ outputs true on input of the current onion layer i in direction $d \in \{\rightarrow, \leftarrow\}$, an onion layer O and parameters used for the original onion generation, if the onion O matches the i-th layer of an onion in direction d generated according to the parameters *onionParams*. The correctness of $\mathsf{RecognizeOnion}$ will follow implicitly from our other property definitions.

$\mathsf{ExtractPayload}((i, d), O, \mathit{onionParams})$ returns the message m currently contained in the payload or a fail symbol $\bot$ on input of the onion O for layer i in direction $d \in \{\rightarrow, \leftarrow\}$ and the original onion parameters *onionParams*. The correctness of $\mathsf{ExtractPayload}$ will follow implicitly from our other property definitions.

4.5 OR Properties

Directly proving that a packet format UC-realizes the ideal functionality (Section 4.3) is tedious, as the proof's complexity is significantly increased due to the required bookkeeping for the UC simulator. Thus, we provide equivalent game-based security notions, for which we prove that they imply UC security. This approach

Algorithm 1: Ideal Functionality $\mathcal{F}$ (Part 1)

Data structure:
Bad: Set of corrupted nodes
L: List of onions processed by adversarial nodes
B^i: List of onions held by node P^i
$Back$: Mapping from temps to path and forward id
ID_{fwd}: Mapping from backward id to forward id
// Notation:
// $\mathcal{S}$: Adversary (resp. Simulator)
// $\mathcal{Z}$: Environment
// $\mathcal{P} = (P^{O^1}, \dots, P^{O^{n-1}})$: Onion path, ($\mathcal{P}_{\rightarrow}$ forward, $\mathcal{P}_{\leftarrow}$ backward)
// $O = (id, P^0, P^r, m, \mathcal{P}, \mathcal{P}', i, d)$: Onion = (identifier, sender, receiver, message, path in current direction, path in other direction, traveled distance, direction)
// N: Maximal onion path length

On message Process_New_Onion($P^r, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}$) *from* P^0
  // P^0 creates and sends a new onion (either instructed by $\mathcal{Z}$ if honest or $\mathcal{S}$ if corrupted)
  if $|\mathcal{P}_{\rightarrow}| > N$ *or* $|\mathcal{P}_{\leftarrow}| > N$; // selected path too long
  then
    Reject;
  else
    $id \leftarrow_R$ ID ; // pick random ID
    $O \leftarrow (id, P^0, P^r, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, 0, f)$; // create new onion
    Output_Corrupt_Sender($P^0, id, P^r, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}$, start, f);
    if $P^0 \notin Bad$ **then**
      Send "Request got id id" to $\mathcal{Z}$
    Process_Next_Step(O);

On message Process_New_Backward_Onion($m, temp$) *from* P
  // P creates and sends a backward onion (either instructed by $\mathcal{Z}$ if honest or $\mathcal{S}$ if corrupted)
  if $Back(temp) = \perp$; // no forward onion was sent
  then
    Reject;
  else
    $Back(temp) = (P^0, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, P^r, id')$; // lookup the corresponding path
    $id \leftarrow_R$ ID ; // pick random session ID
    Store id' under $ID_{fwd}(id)$; // add ID linking to mapping
    $O \leftarrow (id, P^r, P^0, m, \mathcal{P}_{\leftarrow}, \mathcal{P}_{\rightarrow}, 0, b)$; // create new onion
    Output_Corrupt_Sender($P^r, id, P^0, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}$, start, b);
    Process_Next_Step(O);

Procedure Output_Corrupt_Sender($P^0, id, P^r, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, temp, d$)
  // Give all information about onion to adversary if sender is corrupt
  if $P^0 \in Bad$ **then**
    Send "$temp$ belongs to onion from P^0 with $id, P^r, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, b$" to $\mathcal{S}$;
    if $d = b$ **then**
      add "as answer to $ID_{fwd}(id)$" to the output for $\mathcal{S}$

Algorithm 2: Ideal Functionality $\mathcal{F}$ (Part 2)

Procedure Process_Next_Step($O = (id, P^0, P^r, m, \mathcal{P}, \mathcal{P}' i, d)$)
  // Router P^{O^i} just processed O that is now passed to router $P^{O^{i+1}}$
  if $P^{O^j} \in Bad$ *for all* $j > i$; // All remaining nodes including receiver are corrupt
  then
    Send "Onion $temp$ in direction d from P^{O^i} with message m for P^r routed through $(P^{O^{i+1}}, \dots, P^{O^n})$" to $\mathcal{S}$;
    if $d = f$ **then**
      Store $(P^0, \mathcal{P}, \mathcal{P}', P^r, id)$ under $Back(temp)$;
      Add "$temp$'s first part of the backward path is $\mathcal{P}'_H$" with $\mathcal{P}'_H$ being $\mathcal{P}'$ until (and including) the first honest node to the message for $\mathcal{S}$;
      Output_Corrupt_Sender($P^0, id, P^r, m, \mathcal{P}, \mathcal{P}', temp, f$);
    else
      Output_Corrupt_Sender($P^r, id, P^0, m, \mathcal{P}', \mathcal{P}, temp, b$);
  else
    // there exists an honest successor P^{O^j}
    $P^{O^j} \leftarrow P^{O^k}$ with smallest k such that $P^{O^k} \notin$ Bad;
    $temp \leftarrow_R$ temporary ID;
    Send "Onion $temp$ from P^{O^i} routed through $(P^{O^{i+1}}, \dots, P^{O^{j-1}})$ to P^{O^j}" to $\mathcal{S}$;
    Add $(temp, O, j)$ to L; // see Deliver_Message($temp$) to continue this routing
    if $d = f$ **then**
      Output_Corrupt_Sender($P^0, id, P^r, m, \mathcal{P}, \mathcal{P}', temp, f$);
    else
      Output_Corrupt_Sender($P^r, id, P^0, m, \mathcal{P}', \mathcal{P}, temp, b$);
      if $P^0 \in Bad$ *and* $i = 0$ **then**
        Send "temp belongs to id" to $\mathcal{S}$

On message Deliver_Message($temp$) *from* $\mathcal{S}$
  // Adversary $\mathcal{S}$ (controlling all links) delivers onion belonging to $temp$ to next node
  if $(temp, _, _) \in L$ **then**
    Retrieve $(temp, O = (id, P^0, P^r, m, \mathcal{P}, \mathcal{P}', i), j)$ from L;
    $O \leftarrow (id, P^0, P^r, m, \mathcal{P}, \mathcal{P}', j)$; // jth router reached
    if $j < |\mathcal{P}| + 1$ **then**
      $temp' \leftarrow_R$ temporary ID;
      Send "$temp'$ received" to P^{O^j};
      Store $(temp', O)$ in B^{O^j}; // See Forward_Onion($temp'$) to continue
    else
      if $m \neq \perp$ **then**
        Send "Message m under $temp$ in direction d received" on behalf of P^r to $\mathcal{Z}$;
        if $d = b$ **then**
          add "that belongs to request with id id" to the message for $\mathcal{Z}$
        if $\mathcal{P}' \neq ()$ *and* $d = f$ **then**
          add "that is repliable" to the message for P^r ;
          Store $(P^0, \mathcal{P}, \mathcal{P}', P^r, id)$ under $Back(temp)$

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

was introduced in [5] and has since been used. As we build on the ideal functionality of [20], we also build on their properties.

On a high level, we need properties which allow us to replace "real" onions with "simulated" onions, which only contain adversarially observable or random information. Our first property, called Strong Forward Layer-Unlinkability, asserts that an adversary cannot distinguish between (1) a game where the onion O^1 is honestly generated and processed on its path $(P^1, \dots, P^n)$, (2) a game where the onion O^1 is *split into two fresh* onions $\bar{O}^1$ and O^c, where $\bar{O}^1$ contains a random message and path $(P^1, \dots, P^j)$ and onion O^c contains the original message and remainder path $(P^{j+1}, \dots, P^n)$, where P^j is an uncorrupted relay which outputs O^c when it should process (an unwrapped version of) $\bar{O}^1$. All relays except P^j are considered corrupted by the adversary.

We achieve two things with this: First, since case (1) and (2) are indistinguishable, we can "split" (the path of) an onion undetectably, replacing it by two fresh onions. In particular, the onion O^1 does not reveal anything about the path taken after P^j, as in case (2) the path of $\bar{O}^1$ stops at P^j. Second, since the message in $\bar{O}^1$ is random (in case (2)), the observation of O^1 and its processing until P^j cannot reveal any information about the original message. All in all, we see that the onion hides its path and its message.

We need a similar property, Strong Backward Layer-Unlinkability, which basically asserts that we can "split" an onion on the backward path, and replace the reply message with randomness. Taken together, these properties will be used to prove that a packet format UC-realizes the ideal functionality, by replacing real onions

with simulated ones on any path between two honest relays (treating specially the case of a corrupted receiver).

Remark 4.3 (Comparison to [20]). Our formalization differs in two ways from [20]. First, we also want to assert message integrity and thus modified the ideal functionality. In the ideal functionality, this is modelled by giving the decrypted message to the environment $\mathcal{Z}$ for honest receivers (resp. senders for replies). We incorporate this in our properties, by making the games do the same in case (1), while in case (2) they always output the original message.

Second, three security properties are used in [20]: Forward Layer-Unlinkability, Backward Layer-Unlinkability, and Tail Indistinguishability. Instead of replacing onions with *fresh* onions, as we do, in [20] some are replaced with "late" onion layers (see also Remark 4.2). By avoiding "late" onion layers, our definitions are arguably simpler and more natural and, more importantly, Tail Indistinguishability is not required anymore. To prevent confusion, our modified properties are called *strong* Forward/Backward Layer-Unlinkability. In our definitions, we highlight the changes from the OR properties from [20] in blue, otherwise we reuse their definitions verbatim (except for changes in the notation).

4.5.1 Strong Forward Layer-Unlinkability. Strong Forward Layer-Unlinkability is used to replace onion layers between the sender and the next honest relay on the forward path. More precisely, the adversary gets oracle access to process any onions at the honest relay and picks onion parameters for the challenge. The adversary is challenged to decide which case the challenger randomly picked. In case 1, the adversary gets the onion for the first relay according to her parameter choice and later (via an oracle request) the actual processing of the challenge onion after the honest relay. In case 2, the adversary instead gets an onion for the first relay that uses the same path between sender and the honest relay, but ends at the honest relay and includes a random message. The later oracle request of the challenge onion at the honest relay returns a newly created forward onion that uses the remainder of the adversarially chosen path and the adversarially chosen message. This is illustrated in Fig. 1.

Notice that this property implies correctness of RecognizeOnion as otherwise the output of the oracle in step 7 would allow to distinguish the cases (real processing for $b = 0$ vs. replacement upon recognition for $b = 1$).

Definition 4.4 (Strong Forward Layer-Unlinkability $LU^{+}_{\rightarrow}$). Strong Forward Layer-Unlinkability is defined as:

(1) The adversary receives the router names P^H, P^0 as well as the challenge public keys pk^0, pk^H, chosen by the challenger by letting $(pk^H, sk^H) \leftarrow G(1^\lambda, p, P^H)$; $(pk^0, sk^0) \leftarrow G(1^\lambda, p, P^0)$.
(2) Oracle access: The adversary may submit any number of Proc and Reply requests for P^H or P^0 to the challenger. For any Proc(P^H, O), the challenger checks whether hdr is on the hdr^H-list. If not, it sends ProcOnion(sk^H, O, P^H), stores hdr on the hdr^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. If so, the challenger sends ReplyOnion(sk^H, m, O, P^H) to the adversary. (Similar for requests on P^0 with the hdr^0-list).
(3) The adversary submits a message m, a position j with $1 \le j \le n$, a path $\mathcal{P}_{\rightarrow} = (P^1, \ldots, P^j, \ldots, P^n)$ with $P^j = P^H$, a path $\mathcal{P}_{\leftarrow} = (P^1_{\leftarrow}, \ldots, P^{n_{\leftarrow}}_{\leftarrow} = P^0)$ and public keys for all nodes pk^i ($1 \le i \le n$ for the nodes on the path and $n < i$ for the other relays).
(4) The challenger checks that the chosen paths are acyclic, the router names are valid and that the same key is chosen if the router names are equal, and if so, sets $pk^j = pk^H$ and $pk^{n_{\leftarrow}}_{\leftarrow} = pk^0$ and picks $b \in \{0, 1\}$ at random.
(5) The challenger creates the onion with the adversary's input choice and honestly chosen randomness $\mathcal{R}$: $O^1 \leftarrow$ FormOnion$(m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}; \mathcal{R})$ and a replacement onion with the first part of the forward path $\bar{\mathcal{P}}_{\rightarrow} = (P^1, \ldots, P^j)$, a random message $\bar{m} \in \mathcal{M}$, another honestly chosen randomness $\bar{\mathcal{R}}$, and an empty backward path $\bar{\mathcal{P}}_{\leftarrow} = ()$: $\bar{O}^1 \leftarrow$ FormOnion$(\bar{m}, \bar{\mathcal{P}}_{\rightarrow}, \bar{\mathcal{P}}_{\leftarrow}, (pk)_{\bar{\mathcal{P}}_{\rightarrow}}, (pk)_{\bar{\mathcal{P}}_{\leftarrow}}; \bar{\mathcal{R}})$
(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:
   - If $j < n$: Proc(P^H, O) with RecognizeOnion$((j, \rightarrow), O, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}, \bar{\mathcal{R}}) = True$, hdr is not on the hdr^H-list and ProcOnion$(sk^H, O, P^H) \neq \perp$:
     The challenger outputs (P^{j+1}, O^c) with $O^c \leftarrow$ FormOnion$(m, \tilde{\mathcal{P}}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\tilde{\mathcal{P}}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}; \tilde{\mathcal{R}})$ with $\tilde{\mathcal{P}}_{\rightarrow} = (P^{j+1}, \ldots, P^n)$ and adds hdr to the hdr^H-list and O to the O^H-list.
   - If $j = n$:
     – Proc(P^H, O) with RecognizeOnion$((j, \rightarrow), O, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}, \bar{\mathcal{R}}) = True$, hdr is not on the hdr^H-list and ProcOnion$(sk^H, O, P^H) \neq \perp$:
       The challenger outputs $(m, \perp)$ and adds hdr to the hdr^H-list and O to the O^H-list.
     – Reply$(P^H, O, m_{\leftarrow})$ with RecognizeOnion$((j, \rightarrow), O, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}, \bar{\mathcal{R}}) = True$, O is on the O^H-list and has not been replied before and ReplyOnion$(sk^H, m_{\leftarrow}, O, P^H) \neq \perp$:
       The challenger outputs a tuple $(P^1_{\leftarrow}, O^c)$ with $O^c \leftarrow$ FormOnion$(m_{\leftarrow}, \mathcal{P}_{\leftarrow}, (), (pk)_{\mathcal{P}_{\leftarrow}}, (); \tilde{\mathcal{R}})$
(8) The adversary produces guess b'.

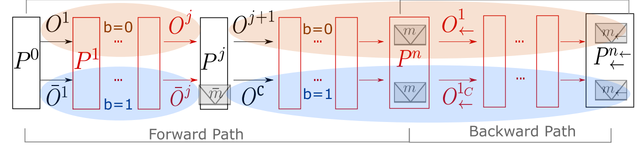

Figure 1: Illustration of Strong Forward Layer-Unlinkability (adapted from [20]): Boxes are relays (black – honest, red –adversarial). Ellipses are observations of the game adversary (orange – $b = 0$, blue – $b = 1$ case). The game adversary has to decide whether the adversarially chosen parameters were used to construct one onion O for all outputs ($b = 0$) or whether the outputs belong to two onions: one ($\bar{O}$) with a random message $\bar{m}$ until the honest relay P^j and one (O^C) using adversaries choices for the remaining path ($b = 1$).

$LU^{+}_{\rightarrow}$ is achieved if no probabilistic polynomial time (PPT) adversary $\mathcal{A}$, can guess $b' = b$ with a probability non-negligibly better than $\frac{1}{2}$.

4.5.2 Strong Backward Layer-Unlinkability. Strong Backward Layer-Unlinkability is used to replace onion layers between honest nodes on the reply path. The game works similar to Strong Forward Layer-Unlinkability, except that we now use the last part of the reply path (instead of the first part of the forward path). Here, however, our new output to the environment requires more elaborate changes than before. We need to output the included reply message as response of the oracle on the challenge onion. If the request receiver is adversarial, the challenger does however not directly learn the reply message. In this case the challenger uses ExtractPayload on the layer it gets from the adversary. The resulting message will be output by the oracle of the honest (request) sender if the challenge onion is recognized and processing does not fail. See Fig. 2.

Notice that this property implies correctness of ExtractPayload, as otherwise the output of the oracle in step 6 would allow to distinguish the cases (real processing for $b = 0$ vs. extracted message for $b = 1$). Notice further that this oracle output forces integrity. If the adversary gets a mofied message accepted at P^0 (i.e. no fail output at P^0), $b = 0$ outputs the modified message to the adversary, while $b = 1$ outputs the message extracted before the modification.

As in earlier work, the replacement of a reply onion with a new forward onion further implies that forward and backward onions are indistinguishable.

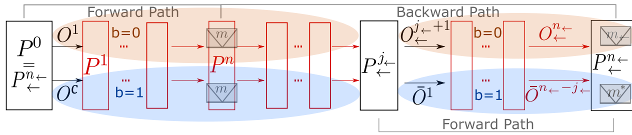

Figure 2: Illustration of Strong Backward Layer-Unlinkability (adapted from [20]): Boxes are relays (black – honest, red –adversarial). Ellipses are observations of the game adversary (orange – $b = 0$, blue – $b = 1$ case). The game adversary has to decide whether the adversarially chosen parameters were used to construct one onion O for all outputs ($b = 0$) or whether the outputs belong to two onions: one (O^C) using the adversaries choices until the honest relay $P^{j_{\leftarrow}}_{\leftarrow}$ and one forward onion ($\bar{O}$) transporting the extracted message m^* for the remaining path ($b = 1$).

Definition 4.5 (Strong Backward Layer-Unlinkability $LU^{+}_{\leftarrow}$). Strong Backward Layer-Unlinkability is defined as:

(1) The adversary receives the router names P^H, P^0 as well as the challenge public keys pk^0, pk^H, chosen by the challenger by letting $(pk^H, sk^H) \leftarrow G(1^\lambda, p, P^H)$; $(pk^0, sk^0) \leftarrow G(1^\lambda, p, P^0)$.

(2) Oracle access: The adversary may submit any number of Proc and Reply requests for P^H or P^0 to the challenger. For any Proc(P^H, O), the challenger checks whether hdr is on the hdr^H-list. If not, it sends ProcOnion(sk^H, O, P^H), stores hdr on the hdr^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. If so, the challenger sends ReplyOnion(sk^H, m, O, P^H) to the adversary. (Similar for requests on P^0 with the hdr^0-list).

(3) The adversary submits message m, a position $j_{\leftarrow}$ with $0 \leq j_{\leftarrow} \leq n_{\leftarrow}$, a path $\mathcal{P}_{\rightarrow} = (P^1, \ldots, P^j, \ldots, P^n)$, where $P^n = P^H$, if $j_{\leftarrow} = 0$, a path $\mathcal{P}_{\leftarrow} = (P^1_{\leftarrow}, \ldots, P^{j_{\leftarrow}}_{\leftarrow}, \ldots, P^{n_{\leftarrow}}_{\leftarrow} = P^0)$ with the honest node P^H at backward position $j_{\leftarrow}$, if $1 \leq j_{\leftarrow} \leq n_{\leftarrow}$, and the second honest node P^0 at position $n_{\leftarrow}$ and public keys for all nodes pk^i ($1 \leq i \leq n$ for the nodes on the path and $n < i$ for the other relays).

(4) The challenger checks that the chosen paths are acyclic, the router names 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 $j_{\leftarrow} = 0$), $pk^{n_{\leftarrow}}_{\leftarrow} = pk^0$ and sets bit b at random.

(5) The challenger creates the onion using the adversary's input and honestly chosen randomness $\mathcal{R}$:
$O^1 \leftarrow$ FormOnion$(m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}; \mathcal{R})$ and a shortened onion with honestly chosen randomness $\tilde{\mathcal{R}}$ and $\tilde{\mathcal{P}}_{\leftarrow} = (P^1_{\leftarrow}, \ldots, P^{j_{\leftarrow}}_{\leftarrow})$:
$O^c \leftarrow$FormOnion$(m, \mathcal{P}_{\rightarrow}, \tilde{\mathcal{P}}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\tilde{\mathcal{P}}_{\leftarrow}}; \tilde{\mathcal{R}})$
If $b = 0$ the challenger sends O^1 to the adversary.
If $b = 1$ let $\mathcal{R} = \tilde{\mathcal{R}}$ and $\mathcal{P}_{\leftarrow} = \tilde{\mathcal{P}}_{\leftarrow}$. The challenger sends $O^1 = O^c$ to the adversary.

(6) The adversary gets oracle access as in step 2) unless:
Exception 1) The request is ...
- for $j_{\leftarrow} > 0$:
Proc(P^H, O) with RecognizeOnion$((j_{\leftarrow}, \leftarrow), O, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}, \mathcal{R}) = True$, hdr is not on the hdr^H-list and ProcOnion$(sk^H, O, P^H) \neq \perp$:
stores hdr on the hdr^H-list, O on the O^H-list, $m^* =$ ExtractPayload$((j_{\leftarrow}, \leftarrow), O, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}, \mathcal{R})$ and ...
- for $j_{\leftarrow} = 0$:
Reply$(P^H, O, m_{\leftarrow})$ with RecognizeOnion$((n, \rightarrow), O, m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}, \mathcal{R}) = True$, O is on the O^H-list and no onion with this hdr has been replied to before and ReplyOnion$(sk^H, m_{\leftarrow}, O, P^H) \neq \perp$:
stores $m^* = m_{\leftarrow}$

.. then: The challenger picks the rest of the return path $\bar{\mathcal{P}}_{\rightarrow} = (P^{j_{\leftarrow}+1}_{\leftarrow}, \ldots, P^{n_{\leftarrow}}_{\leftarrow})$, an empty backward path $\bar{\mathcal{P}}_{\leftarrow} = ()$, a random message $\bar{m}$, another honestly chosen randomness $\bar{\mathcal{R}}$, and sets: $\bar{O}^1 \leftarrow$ FormOnion$(\bar{m}, \bar{\mathcal{P}}_{\rightarrow}, \bar{\mathcal{P}}_{\leftarrow}, (pk)_{\bar{\mathcal{P}}_{\rightarrow}}, (pk)_{\bar{\mathcal{P}}_{\leftarrow}}; \bar{\mathcal{R}})$
- If $b = 0$, the challenger calculates for $j_{\leftarrow} > 0$:
$(O^{j_{\leftarrow}+1}, P^{j_{\leftarrow}+1}_{\leftarrow}) =$ ProcOnion$(sk^H, O, P^{j_{\leftarrow}}_{\leftarrow})$
resp. for $j_{\leftarrow} = 0$:
$(O^{j_{\leftarrow}+1}, P^{j_{\leftarrow}+1}_{\leftarrow}) =$ ReplyOnion$(sk^H, m_{\leftarrow}, O, P^{j_{\leftarrow}}_{\leftarrow})$
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.

Exception 2) in the $b = 1$ case Proc(P^0, O) with O being the challenge onion processed for the first time (hdr not on hdr^H-list) without a fail (ProcOnion$(sk^0, O, P^0) \neq \perp$) for the final backward receiver, i.e.: RecognizeOnion$((n_{\leftarrow} - j_{\leftarrow}, \rightarrow), O, \bar{m}, \bar{\mathcal{P}}_{\rightarrow}, \bar{\mathcal{P}}_{\leftarrow}, (pk)_{\bar{\mathcal{P}}_{\rightarrow}}, (pk)_{\bar{\mathcal{P}}_{\leftarrow}}, \bar{\mathcal{R}}) = True$:
Then the challenger checks that ProcOnion(sk^0, O, P^0) does

not fail and if so outputs $(m^*, \perp)$ and adds hdr to the hdr^0-list and O to the O^0-list.

(7) The adversary produces guess b' .

$LU^+_{\leftarrow}$ is achieved if no PPT adversary $\mathcal{A}$, can guess $b' = b$ with a probability non-negligibly better than $\frac{1}{2}$.

4.6 UC Realization

Following [20] verbatimly, we define a secure OR scheme to fulfill our properties except that we require one property less. Further following [20], we prove that any protocol build from a secure OR scheme realizes our adapted ideal functionality.

Definition 4.6. An OR scheme is a secure, if it achieves Correctness, Strong Forward Layer-Unlinkability, and Strong Backward Layer-Unlinkability.

THEOREM 4.7. *Any secure OR scheme realizes the ideal functionality from Algorithms 1 and 2.*

For the proof of this theorem see Appendix B. In the proof, we construct a simulator that translates events from the ideal to the real world and vice versa. On a high-level, the simulator translates ideal world communications into random real world onions that take the same subpath (or deliver the same message if they reach the final receiver). The simulator gets the according information of the subpath (resp. message) from the ideal functionality. Real world onions to honest parties are translated into new onion requests in the ideal world. To be able to do this translation the simulator keeps lists of the onions from the real world and their ideal world representation and the other way around. Further, information to reply to received onions is stored. The stored information is used to decide on the correct action when an onion is recognized, e.g. to continue an ideal world communication if the processing of the real world replacement of this communication is received. We prove that this translation is indistinguishable for the adversary by reducing the stepwise onion replacement to our onion security properties. Thereby Strong Forward Layer-Unlinkability is used for all replacements on the request and Strong Backward Layer-Unlinkability for all replacements on the reply.

5 CONSTRUCTION OF EROR

In this section, we describe our new onion routing scheme EROR, which satisfies the stronger security properties from Section 4.

The construction of the header follows the usual onion wrapping paradigm. That is, the onion consists of a header and a payload, cf. Fig. 3. The header itself consists of several blocks, with the first block containing a public-key ciphertext c and a MAC. Contrary to the usual design, our payload is split into a *forward payload fwd* and a *backward payload bwd*. An onion is processed by:

1. Decrypt c to obtain the ephemeral key K and next relay P.
2. Use K to verify the MAC-tag over the header *and* the forward payload.
3. Use K to decrypt ("unwrap") all header blocks, the forward and the backward payload (with a permutation cipher).
4. If this node is not the receiver, shift all header blocks to the left (dropping the first block) and fill in a decryption of 0 as a (garbage) last block.

Note that the MAC usually asserts integrity of only the header itself, not the forward payload on every step of the processing (cf. [12, 25, 13, 2]). This is our first important change. Also note that the MAC does *not* protect the backward payload. See Figs. 3 to 5 for sketches of the onion/header structure, and the unwrapping procedure, which we describe in detail in Section 5.2.

Our main idea is to prevent the tagging attack by protecting the forward payload with the MAC in the header for every relay, while not protecting the backward payload in this way. To enable replies, we assume that the forward payload contains the reply onion header $O_{\leftarrow}.hdr$ and a PRG seed for generating the garbage forward payload that needs to be used together with the actual reply that is the backward payload.

While we are not detecting a tampering of the backward payload on every step of the path, the final reply receiver (i.e. request sender) shall be able to detect any modification of the received reply. We thus use an Encrypt-then-MAC construction, with a PRF as MAC, that is only checked by the (request) sender for the backward payload. Every relay masks the MAC by computing the XOR with $\mathsf{PRF}_K(bwd.ctxt)$, where $bwd.ctxt$ is the current backward ciphertext. This ensures that modifying the backward payload results in a decryption failure at the (request) sender, even if the receiver (which knows the keys used to compute the MAC) is corrupted. For decryption, the original sender reverses the random masking, and, intuitively, if there was any change to the ciphertext of bwd, the MAC is completely randomized.

To obtain a stateless construction, all ephemeral keys for intermediate relays are derived from a *onion master secret key* using a PRF as key derivation function KDF. The sender then public-key-encrypts the onion master key for the backward onion to himself, so that he can derive all intermediate keys, unwrap the wrapped bwd ciphertext, and check the MAC.

5.1 Notation and Conventions for Pseudocode

We use the notation from Section 2.1. For our construction, we assume the following:

- PRG is a pseudorandom generator (PRG).
- MAC is a message authentication code (MAC) which protects header and forward payload.
- KDF and PRF are pseudorandom functions (PRF). KDF is used as key derivation function, PRF is used to construct AE (see below).
- SKE is a (nonce-based) permutation cipher. It is used to in encryptions of all payloads (header, forward, backward).
- AE is an (authenticated) encryption scheme built from PRF and SKE (via encrypt then MAC) with additional AE.Wrap and AE.Unwrap procedures, see Fig. 6. This can be simplified if SKE is PRP-CCA secure, cf. Section 5.4.3.
- PKE is a PKE scheme, used to encrypt information for the relays, in particular their ephemeral key K.
- P denotes an address which has an associated public key denoted $\mathsf{PKI}[P].pk$. Therefore, in FormOnion the public keys are omitted from the inputs.

Besides omitting public keys from FormOnion (using PKI instead), we also omit input P to ProcOnion and ReplyOnion, as this is the

calling party's identity — this was convenient for abstract proofs, but the information is unused in EROR and implicitly given by sk.

We used the following conventions for consistent indexing in the pseudocode:

- ProcOnion takes secret key sk^i and onion layer O^i and outputs O^{i+1};
- the MAC with K^i is over O^i;
- index i indicates the i-th *relay* (in some direction); the sender is the 0-th relay (resp. $n_{\leftarrow}$-th relay); the receiver is the $n_{\rightarrow}$-th relay;
- the "final" onion is O^n and when O^n is processed with K^n, this yields the output.

5.2 EROR: Efficient Repliable Onion Routing

We discuss the structure and handling of onions on a high level, explaining the pseudocode in Fig. 6.

An onion. An onion O consists of a 3-tuple (hdr, fwd, bwd) (see Fig. 3), where

- hdr is the onion's header which contains (wrapped, i.e. multiply encrypted) routing information;
- fwd is the forward payload, i.e., a wrapping of $(hdr_{\leftarrow}, m)$, where $hdr_{\leftarrow}$ is the backward onion header and m the forward message.
- bwd is a (wrapped) AE ciphertext containing the backward payload.

In a forward (resp. backward) onion, the backward (resp. forward) payload is chosen (pseudo-)randomly, as it has no meaning in that direction.

Remark 5.1. We stress that within an onion, all fields have *fixed size* so that are not distinguishable by size. We also note that non-repliable onions could reuse the space in $hdr_{\leftarrow}$ (and $P^1_{\leftarrow}$) to send more information, but we have not pursued this here.

An onion header. A header is a tuple $(B_1, \dots, B_N)$, where N is the maximal number of hops a packet takes. The term B_1 is a pair (τ, c) where c is a public-key encryption of triple $(role, K, P)$, where

- $role$ describes the role of the processing party.
- K is always an (ephemeral) symmetric key.
- P depends on the role. If $role \in \{\texttt{HOP}, \texttt{RCVR}\}$, then P is the address of the next relay. If $role = \texttt{SNDR}$, then $P = (n_{\rightarrow}, n_{\leftarrow})$ contains the path lengths instead of a next relay.

The other terms $B_2, \dots, B_N$ are encrypted under the key K. Thus, we have the typical "onion structure" with layers of encryption. The onion structure and onion header are illustrated in Fig. 3 and Fig. 4, respectively.

Unwrapping and Processing of Onions. First, process the header $(B_1, \dots, B_n)$ by decrypting c in $B_1 = (\tau, c)$ to obtain the ephemeral key K. Check that τ (which is the MAC over hdr and fwd) is valid, and abort otherwise. If so, decrypt $B_2, \dots, B_N$, fwd, and bwd. Now, shift the blocks left for the next onion. This shift is why we decrypt the j-th block with nonce $j + 1$ — during encryption, that was the block's position and nonce. Observe that this shift results in a missing N-th block. This block is filled in by a garbage term, a decryption of 0. The procedure is illustrated in Fig. 5. Finally, also decrypt the forward and backward payload. This yields our processed onion. The pseudocode for the above is in Unwrap (Fig. 6).

The ProcOnion routine uses Unwrap, and then handles the specific cases for forward (and backward) receiver. Namely, decrypting and outputting the (backward) message; see the pseudocode from Fig. 6 for details.

Replying to an onion. After unwrapping the onion, the receiver finds a backward header $O_{\leftarrow}.hdr$, as part of its forward payload. The reply onion uses $O_{\leftarrow}.hdr$ as the onion header, and sets $O_{\leftarrow}.fwd$ to a pseudorandom value $\mathsf{PRG}(K^{n_{\rightarrow}}_{\rightarrow,\mathsf{PRG}})$. The backward payload $O_{\leftarrow}.bwd$ is then computed via AE.Enc, completing the onion. (There is no MAC over the backward payload, so it can be set arbitrarily However, the backward payload itself is end-to-end protected with AE, detecting any tampering of intermediate relays.)

Remark 5.2. The (handling of) *backward* payload in forward and backward onions differs: In forward onions, it is just randomness, and integrity cannot be checked upon decryption. Indeed, this is crucial to avoid the tagging attack of [19].

Processing Replies. To process a reply, the sender must re-wrap the backward payload using AE.Wrap and then decrypt it using AE.Dec. To allow this, all ephemeral keys, in particular $(K^i_{\leftarrow})_{i=1}^{n_{\leftarrow}-1}$, are derived from an onion master secret key $omsk$, which the sender sends to itself as $K^{n_{\leftarrow}}_{\leftarrow}$ in $c^{n_{\leftarrow}}$.

Wrap, Onionize, and FormOnion. We first explain how onion layers are wrapped. A header is generated from an initial choice of blocks $(B^n_2, \dots, B^n_N)$ (as explained later) by repeatedly wrapping it as follows: Let i denote the layer of wrapping, going from n down to 1. To wrap for relay P^i, use PKE to encrypt meta-information $meta^i = (role^i, K^i, P^{i+1})$ in ciphertext c^i. Moreover, wrap the previous header blocks B^{i+1}_j to obtain $B^i_{j+1} = \mathsf{Enc}_{K^i_{\mathsf{SKE}}}(j+1, B^{i+1}_j)$ for $j = 1, \dots, N-1$. Note the choice of indices, which effectively shifts the blocks to the right (and drops B^{i+1}_N) and encrypts with the shifted position as the nonce. Similarly, the forward payload fwd^{i+1} is wrapped to obtain fwd^i. Finally, compute the MAC τ^i over $(c, B^i_2, \dots, B^i_{N-1}, fwd^i)$ to complete $B^i_1 = (\tau^i, c^i)$. A single step of this wrapping is defined in Wrap (Fig. 6).

Observe that a dropped block B^i_N cannot be recovered and is replaced by garbage in Unwrap, namely, by a decryption of 0. Thus, to make sure all MACs τ^i are valid, we must predict the garbage terms (via repeated decryptions of 0) and choose $(B^n_2, \dots, B^n_N)$ appropriately. In Onionize (lines 37 to 43), the initial choice of blocks $(B^n_2, \dots, B^n_N)$ is thus set randomly, except for precomputed garbage terms. By repeated use of Wrap, the fully wrapped onion is computed. Lastly, Onionize sets a random backward payload (in line 59). (Wrap does not use bwd, as it is meaningless there.)

Finally, to form a repliable onion, FormOnion generates two headers $hdr_{\rightarrow}$, $hdr_{\leftarrow}$ using Onionize, one for the forward and backward direction, respectively. To generate $hdr_{\leftarrow}$, the pseudorandom forward payload $\mathsf{PRG}(K^{n_{\rightarrow}}_{\rightarrow,\mathsf{PRG}})$ is used. (Recall that MACs cover the forward payload, so we need to fix it during onion generation.) Lastly, note that the ephemeral keys K^i are derived from an "onion master secret key" $omsk$, which the senders sends backs to itself (as $K^{n_{\leftarrow}}_{\leftarrow}$). The senders uses this to decrypt a reply.

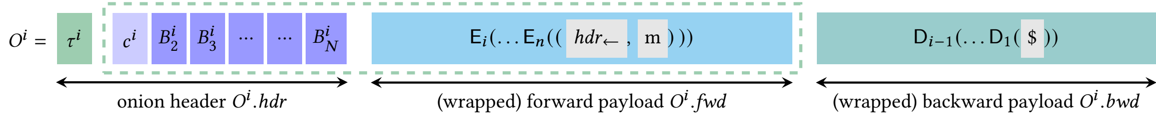

Figure 3: Repliable onion structure in forward direction illustrated. All blocks are fixed size. D_i (resp. E_i) is short-hand for $\mathsf{Dec}_{K^i_{\mathsf{SKE}}}(j+1,_)$ (resp. $\mathsf{Enc}_{K^i_{\mathsf{SKE}}}(j,_)$) with proper choices of nonces $j \in \{1,\ldots,N\}$ (resp. nonces FWD and BWD) in repeated en-/decryptions (for forward/backward payload). (The blocks, hence used nonces, shift after each (un)wrapping.) While *bwd* has special treatment (uses AE.Unwrap, not SKE.Dec), we use D instead of AE.Unwrap in the illustration for simplicity.

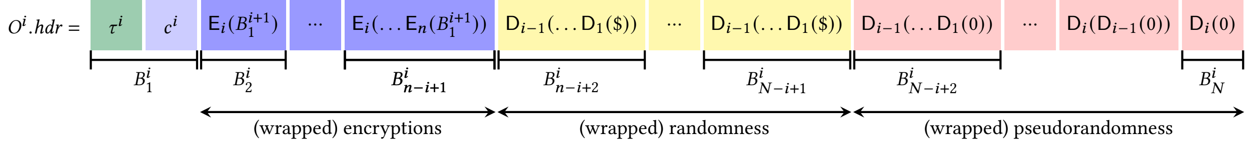

Figure 4: Intermediate onion header at layer i illustrated. If $i = n$, there are no wrapped encryptions. If $i = 1$, there is no wrapped pseudorandomness. If $n = N$, there is no wrapped randomness. D_i (resp. E_i) is used as in Fig. 3.

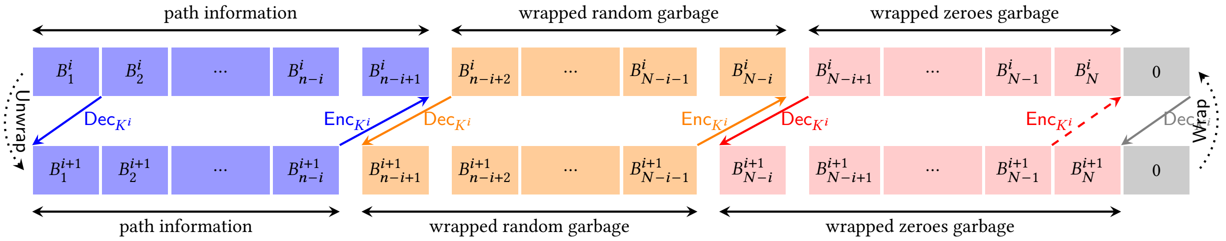

Figure 5: Illustration of the header evolution during (un)wrapping. The "imaginary" zero block extending the header to the right is due to the Unwrap procedure, which fills the missing block B_N^{i+1} with a decryption of 0 by definition.

5.3 Pseudocode and remarks

The pseudocode in Fig. 6 assumes that both $n_\rightarrow \geq 1$ and $n_\leftarrow \geq 1$, in particular, there is a reply path and reply onion. If $n_\leftarrow = 0$, i.e., there is no reply path, then in FormOnion the nonsensical term $P^1_\leftarrow$ is replaced by $\bot$, and $O_\leftarrow.hdr$ is garbage, also denoted $\bot$.[5] In this case, we could reuse the unused space to encrypt a larger forward payload, however, we do not consider this possibility in our security model or proofs.

Remark 5.3. For simplicity of presentation, our pseudocode for ProcOnion and ReplyOnion includes some evidently unnecessary computations. In particular, some variables B^i_j are assigned multiple times in Onionize (Fig. 6), however, their re-assignment coincides with their previous definition.

5.4 Further Discussion on Security, Efficiency, and Trade-offs

5.4.1 Identifying Replies. In its current form, Definition 4.1 does not provide a way to identify if a received message is a reply to a previously sent onion. To allow this, FormOnion and ProcOnion can be modified appropriately to additionally output an identifier for an onion (and the caller must keep track of it). For EROR, one natural choice it to use the ciphertext $O_\leftarrow.c^{n_\leftarrow}$, i.e., the ciphertext which encrypts the onion master secret key. This choice is natural, but many sensible alternatives are possible, e.g., adding an explicit identifier to the schemes header, or using *omsk* as the identifier.

We discussed how to model the identification of replies above, however, if one adds this, then it is natural to also require that "replies cannot be faked", i.e., it should be impossible for an adversary to create an onion which is treated as a reply. Clearly, EROR does not satisfy this, as anyone can generate a reply onion. To obtain this property, secret-key material of the purported original (honest) sender must be used in FormOnion, not just in ProcOnion. One can use simple approaches, e.g., keeping a separate secret key sk_{MAC} (alongside sk_{PKE}) for the MAC in $O^{n_\leftarrow}_\leftarrow$ (instead of deriving a key from $K^{n_\leftarrow}_\leftarrow$). If MAC is EUF-CMA secure,[6] then it is impossible to generate a fake reply-onion (in the stateless setting).

5.4.2 Using KEMs. Our scheme EROR from Section 5.3 makes some choices to simplify the presentation or adhere to prior work. Namely, the use of an "onion master secret key" (to derive the intermediate keys $K^i_\rightarrow, K^j_\leftarrow$) and public-key encryption (PKE) instead of key-encapsulation mechanisms (KEM). Definition 4.1, which follows [5, 2, 20], requires FormOnion and ProcOnion to be *stateless*,

[5] For simplicity, we assume the $\bot$ symbols are of the same size as original terms.

[6] Currently, Theorem 6.1 only needs MAC to be selectively secure (*SUF*-CMA), which is weaker than EUF-CMA. But, if a PRF is used as MAC, it is EUF-CMA secure anyway.

$\mathsf{Wrap}(K, c, (\overline{B}_2, \ldots, \overline{B}_N), \overline{fwd})$

1 : $K_{\mathsf{SKE}} = \mathsf{KDF}(K, \mathtt{SKE}); K_{\mathsf{MAC}} = \mathsf{KDF}(K, \mathtt{MAC});$
2 : **for** $i = 2, \ldots, N$
3 : $\quad B_i \leftarrow \mathsf{SKE.Enc}_{K_{\mathsf{SKE}}}(i, \overline{B}_i)$
4 : $fwd = \mathsf{SKE.Enc}_{K_{\mathsf{SKE}}}(\mathtt{FWD}, \overline{fwd})$
5 : $\tau \leftarrow \mathsf{MAC.Sign}_{K_{\mathsf{MAC}}}(c, B_2, \ldots, B_N, fwd)$
6 : $B_1 = (\tau, c)$
7 : **return** $((B_1, \ldots, B_N), fwd)$

$\mathsf{Unwrap}(sk, (\overline{B}_1, \ldots, \overline{B}_N), \overline{fwd}, \overline{bwd})$

11 : $(\tau, c) \leftarrow B_1$
12 : $(K, meta) \leftarrow \mathsf{PKE.Dec}_{sk}(c)$
13 : **if** $K = \perp$ **then return** $\perp$
14 : $K_{\mathsf{SKE}} = \mathsf{KDF}(K, \mathtt{SKE}); K_{\mathsf{MAC}} = \mathsf{KDF}(K, \mathtt{MAC});$
15 : **if** $\mathsf{MAC.Verify}_{K_{\mathsf{MAC}}}(\tau, (c, \overline{B}_2, \ldots, \overline{B}_N, \overline{fwd})) \neq 1$
16 : $\quad$ **return** $\perp$
17 : **for** $i = 1, \ldots, N-1$
18 : $\quad B_i = \mathsf{SKE.Dec}_{K_{\mathsf{SKE}}}(i+1, \overline{B}_{i+1})$
19 : $B_N = \mathsf{SKE.Dec}_{K_{\mathsf{SKE}}}(N+1, 0)$
20 : $fwd = \mathsf{SKE.Dec}_{K_{\mathsf{SKE}}}(\mathtt{FWD}, \overline{fwd})$
21 : $bwd = \mathsf{AE.Unwrap}(K, \overline{bwd})$
22 : **return** $(K, meta, ((B_1, \ldots, B_N), fwd, bwd))$

$\mathsf{FormOnion}(m, (P^i_{\rightarrow})_{i=1}^{n_{\rightarrow}}, (P^i_{\leftarrow})_{i=1}^{n_{\leftarrow}}; \mathcal{R})$

61 : **if** $n_{\rightarrow} > N$ **or** $n_{\leftarrow} > N$ **then abort**
62 : $K^{n_{\leftarrow}}_{\leftarrow} = omsk \leftarrow \mathcal{K}$ // Onion master secret key
63 : **for** $i = 1, \ldots, n_{\rightarrow}$
64 : $\quad K^i_{\rightarrow} = \mathsf{KDF}(omsk, (\rightarrow, i))$
65 : **for** $i = 1, \ldots, n_{\leftarrow} - 1$
66 : $\quad K^i_{\leftarrow} = \mathsf{KDF}(omsk, (\leftarrow, i))$
67 : $K_{\mathsf{PRG}} = \mathsf{KDF}(K^n_{\rightarrow}, \mathtt{PRG})$
68 : // If $n_{\leftarrow} = 0$, then $P^1_{\leftarrow} := \perp$ and $O_{\leftarrow} := \perp$.
69 : $O_{\leftarrow} = \mathsf{Onionize}((K^i_{\leftarrow})_i, (P^i_{\leftarrow})_i, (\mathtt{SNDR}, (n_{\rightarrow}, n_{\leftarrow})),$
70 : $\qquad dir = \leftarrow, fwd = \mathsf{PRG}(K_{\mathsf{PRG}}))$
71 : $O_{\rightarrow} = \mathsf{Onionize}((K^i_{\rightarrow})_i, (P^i_{\rightarrow})_i, (\mathtt{RCVR}, P^1_{\leftarrow}),$
72 : $\qquad dir = \rightarrow, fwd = (O_{\leftarrow}.header, m))$
73 : **return** $O_{\rightarrow}$

$\mathsf{ReplyOnion}(sk, m, \overline{O})$

101 : // Repeat ProcOnion steps and sanity checks.
102 : $out \leftarrow \mathsf{Unwrap}(sk, \overline{O})$
103 : **if** $out = \perp$ **then return** $\perp$
104 : **parse** $out = (K, meta, O)$
105 : **if** $meta.role \neq \mathtt{RCVR}$
106 : $\quad$ **return** $\perp$ // Not the receiver!
107 : // Construct reply
108 : **parse** $(header, _) = O.fwd$
109 : $O^{\leftarrow}.header = header$
110 : $K_{\mathsf{PRG}} = \mathsf{KDF}(K, \mathtt{PRG})$
111 : $O^{\leftarrow}.fwd = \mathsf{PRG}(K_{\mathsf{PRG}})$
112 : $O^{\leftarrow}.bwd = \mathsf{AE.Enc}_K(m)$
113 : **return** $(O^{\leftarrow}, meta.next)$

$\mathsf{Onionize}((K^i)_{i=1}^n, (P^i_{i=1})^n, (role, next), dir, \overline{fwd})$

31 : **for** $i = 1, \ldots, n$
32 : $\quad K^i_{\mathsf{SKE}} = \mathsf{KDF}(K^i, \mathtt{SKE}); K^i_{\mathsf{MAC}} = \mathsf{KDF}(K^i, \mathtt{MAC});$
33 : // Precompute all PKE ciphertexts of $meta^i$.
34 : **for** $i = 1, \ldots, n-1$
35 : $\quad c^i \leftarrow \mathsf{PKE.Enc}_{\mathsf{PKI}[P^i].pk}((\mathtt{HOP}, K^i, P^{i+1}))$
36 : $c^n \leftarrow \mathsf{PKE.Enc}_{\mathsf{PKI}[P^n].pk}((role, K^n, next))$
37 : // Precompute all (garbage) values
38 : **for** $i = 1, \ldots, n-1$ // Iterated decryptions of 0
39 : $\quad$ **for** $j = N - i + 1, \ldots, N-1$
40 : $\qquad B^{i+1}_j = \mathsf{SKE.Dec}_{K^i_{\mathsf{SKE}}}(j+1, B^i_{j+1})$
41 : $\quad B^{i+1}_N = \mathsf{SKE.Dec}_{K^i_{\mathsf{SKE}}}(N+1, 0)$
42 : **for** $j = 2, \ldots, N - n + 1$ // Random dummy terms
43 : $\quad B^n_j \leftarrow \$$
44 : // Compute O^n, i.e., O^n without bwd
45 : **if** $dir = \rightarrow$
46 : $\quad fwd^n = \mathsf{SKE.Enc}_{K^n_{\mathsf{SKE}}}(\mathtt{FWD}, \overline{fwd})$
47 : **else if** $dir = \leftarrow$
48 : $\quad fwd^1 = \overline{fwd}$
49 : $\quad$ **for** $i = 1, \ldots, n-1$
50 : $\qquad fwd^{i+1} = \mathsf{SKE.Dec}_{K^i_{\mathsf{SKE}}}(\mathtt{FWD}, fwd^i)$
51 : $\tau \leftarrow \mathsf{MAC.Sign}_{K^n_{\mathsf{MAC}}}(c^n, B^n_2, \ldots, B^n_N, fwd^n)$
52 : $B_1 = (\tau, c^n)$
53 : $U^n = ((B_1, B^n_2, \ldots, B^n_N), fwd)$
54 : // Wrap up the onion
55 : **for** $i = n-1, \ldots, 1$
56 : $\quad U^i = (B^i_1, \ldots, B^i_N, fwd^i) \leftarrow$
57 : $\qquad \mathsf{Wrap}(K^i, c^i, (B^{i+1}_1, \ldots, B^{i+1}_{N-1}), fwd^{i+1})$
58 : // Finish by adding garbage bwd
59 : $bwd^1 = (bwd^1.mac, bwd^1.ctxt) \leftarrow \$$
60 : **return** $O^1 = (B^1_1, \ldots, B^1_N, fwd^1, bwd^1)$

$\mathsf{ProcOnion}(sk, \overline{O})$

81 : $out \leftarrow \mathsf{Unwrap}(sk, O)$
82 : **if** $out = \perp$ **then return** $(\perp, \perp)$
83 : **parse** $out = (K, meta, O)$
84 : **if** $meta.role = \mathtt{HOP}$ // Intermediary node
85 : $\quad$ **return** $(O, meta.next)$
86 : **if** $meta.role = \mathtt{RCVR}$ // Receiver node
87 : $\quad$ **parse** $(_, _, m) = O.fwd$
88 : $\quad$ **return** $(m, \perp)$
89 : **if** $meta.role = \mathtt{SNDR}$ // Sender node
90 : $\quad (n_{\rightarrow}, n_{\leftarrow}) = meta.next$
91 : $\quad bwd^{n_{\leftarrow}} = \overline{O}.bwd$
92 : $\quad$ **for** $i = n_{\leftarrow} - 1, \ldots, 1$
93 : $\qquad K^i_{\leftarrow} = \mathsf{KDF}(K, (\leftarrow, i))$
94 : $\qquad bwd^i = \mathsf{AE.Wrap}(K^i_{\leftarrow}, bwd^{i+1})$
95 : $\quad K^{n_{\rightarrow}}_{\rightarrow} = \mathsf{KDF}(K, (\rightarrow, n_{\rightarrow}))$
96 : $\quad m = \mathsf{AE.Dec}_{K^{n_{\rightarrow}}_{\rightarrow}}(bwd^1)$
97 : $\quad$ **return** $(m, \perp)$

$\mathsf{AE.Wrap}(K, \overline{bwd})$

121 : $K_{\mathsf{SKE}} = \mathsf{KDF}(K, \mathtt{SKE}); K_{\mathsf{PRF}} = \mathsf{KDF}(K, \mathtt{PRF});$
122 : $bwd.ctxt = \mathsf{SKE.Enc}_{K_{\mathsf{SKE}}}(\mathtt{BWD}, \overline{bwd}.ctxt)$
123 : $bwd.mac = \overline{bwd}.mac \oplus \mathsf{PRF}_{K_{\mathsf{PRF}}}(\overline{bwd}.ctxt)$
124 : **return** bwd

$\mathsf{AE.Unwrap}(K, \overline{bwd})$

131 : $K_{\mathsf{SKE}} = \mathsf{KDF}(K, \mathtt{SKE}); K_{\mathsf{PRF}} = \mathsf{KDF}(K, \mathtt{PRF});$
132 : $bwd.ctxt = \mathsf{SKE.Dec}_{K_{\mathsf{SKE}}}(\mathtt{BWD}, \overline{bwd}.ctxt)$
133 : $bwd.mac = \overline{bwd}.mac \oplus \mathsf{PRF}_{K_{\mathsf{PRF}}}(bwd.ctxt)$
134 : **return** bwd

$\mathsf{AE.Enc}_K(m)$

141 : $K_{\mathsf{SKE}} = \mathsf{KDF}(K, \mathtt{SKE}); K_{\mathsf{PRF}} = \mathsf{KDF}(K, \mathtt{PRF});$
142 : $bwd.ctxt = \mathsf{SKE.Enc}_{K_{\mathsf{SKE}}}(\mathtt{AE}, m)$
143 : $bwd.mac = \mathsf{PRF}_{K_{\mathsf{PRF}}}(bwd.ctxt)$
144 : **return** $bwd = (bwd.ctxt, bwd.mac)$

$\mathsf{AE.Dec}_K(bwd)$

151 : $K_{\mathsf{SKE}} = \mathsf{KDF}(K, \mathtt{SKE}); K_{\mathsf{PRF}} = \mathsf{KDF}(K, \mathtt{PRF});$
152 : $\tau = \mathsf{PRF}_{K_{\mathsf{PRF}}}(bwd.ctxt)$
153 : **if** $\tau \neq bwd.mac$ **then return** $\perp$
154 : $m = \mathsf{SKE.Dec}_{K_{\mathsf{SKE}}}(\mathtt{AE}, bwd.ctxt)$
155 : **return** m

Figure 6: Pseudocode of EROR routines.

which makes using KEMs to generate the intermediate keys $K^i_{\rightarrow}$, $K^j_{\leftarrow}$ inconvenient, although it is, in principle, possible as we outline in Remark 5.4 below. However, if we allow FormOnion to output some state *decinfo* for the sender, and we give ProcOnion a list of *decinfo*s as optional input, then using KEMs gets easy.

For example, we can modify EROR as follows:

- Replace PKE with a KEM which generates $K^i_{\rightarrow}$ and $K^j_{\leftarrow}$.
- Use SKE to encrypt the meta-information ($\mathtt{HOP}, P^{i+1}$) and ($role, next$) in lines 35 and 36. (The security proof is unaffected as the MAC over the header and payload still ensures integrity.)
- Output $decinfo = (c^n_{\leftarrow}, (K^i_{\rightarrow})^{n_{\rightarrow}}_{i=1}, (K^j_{\leftarrow})^{n_{\leftarrow}}_{j=1})$.
- If ProcOnion enters the role SNDR branch (line 89), it looks for a matching *decinfo* and uses the respective keys to (un-)wrap the onion.

For security, we need to adapt the games $LU^+_{\rightarrow}$, $LU^+_{\leftarrow}$ and equivalence proofs from Section 4 to the stateful setting. If the sketched changes are implemented correctly, one obtains a secure *stateful* variant of EROR, which relies on a IND-CCA-secure KEM in a simple manner.

Remark 5.4 (KEM instead of PKE). It is possible to use KEMs instead of PKEs also in the stateless case: One generic approach is to use *omsk* to (re)derive the key encapsulation *randomness*, so that the sender can recover all encapsulated keys by recomputing the encapsulation. Another, less generic approach uses that in the security proof (after the above modifications) it should suffice if PKE is IND-CCA-secure for *random messages* (i.e., the challenges oracle picks a random message m^* and either gives m^* or fresh randomness to the adversary). This is a natural property, which many KEM schemes derived from Fujisaki–Okamoto-like transformations satisfy.

5.4.3 Using PRP-CCA-secure SKE. One goal of EROR was security even if SKE is only DLR\$-CPA-secure, which allows typical stream ciphers. If, as in prior works, e.g. [12, 2, 20], we use PRP-CCA-secure pseudorandom permutations (PRP) instead, then the special handling of the backwards payload can be simplified, as standard "padding schemes" suffice to obtain integrity protection from PRP-CCA-secure PRPs. Hence, the AE scheme and its special Wrap and Unwrap handling becomes superfluous. Additionally, some (at least heuristic) robustness against processing an onion more than once is obtained. The case where onions are processed more than once is not covered by our security proof, and indeed, this results in a form of nonce-reuse, which violates our security assumptions on SKE, potentially breaking its security entirely.

6 SECURITY OF EROR

In this section, we discuss the security of EROR.

Theorem 6.1. *Let* EROR *be as in Fig. 6. Suppose that*

- KDF *and* PRF *are secure pseudorandom functions.*
- SKE *is a DLR\$-CPA-secure (Definition A.3)nonce-based encryption scheme and permutation ciphers (Definition A.2).*
- MAC *is a SUF-CMA-secure MAC.*
- PKE *is a IND-CCA-secure PKE scheme.*

Then EROR *satisfies $LU^+_{\rightarrow}$ and $LU^+_{\leftarrow}$, in particular, it is a secure onion routing scheme according to Definition 4.6. If all building blocks are perfectly correct, so is* EROR.

Correctness follows directly from inspection of EROR. We give a high-level proof sketch for security. For this, we use the same notation for intermediate results in FormOnion as in the pseudocode. RecognizeOnion and ExtractPayload are defined as follows.

- RecognizeOnion$((i, dir), \overline{O}, m, (P^i_{\rightarrow})^{n_{\rightarrow}}_{i=1}, (P^j_{\leftarrow})^{n_{\leftarrow}}_{j=1}, \mathcal{R})$ where $dir = \rightarrow$ compares $\overline{c}$ with $c^i_{\rightarrow}$, where $(\overline{\tau}, \overline{c}) = \overline{B}_1 = \overline{O}.hdr[1]$ and $c^i_{\rightarrow}$ is the intermediate PKE ciphertext in $O_{\rightarrow}$ in the pseudocode (which can be recomputed with $\mathcal{R}$). If they are equal and $\overline{\tau}$ is valid, output 1 else 0. For $dir = \leftarrow$, use $c^i_{\leftarrow}$ from $O_{\leftarrow}$ instead.
- ExtractPayload$((i, \leftarrow), O, m, (P^i_{\rightarrow})^{n_{\rightarrow}}_{i=1}, (P^i_{\leftarrow})^{n_{\leftarrow}}_{i=1}, \mathcal{R})$: Recompute all ephemeral keys $K^j_{\leftarrow}$ (using $\mathcal{R}$) and Unwrap the payload until the backward receiver (i.e., original sender) is reached, i.e. until $j = n_{\leftarrow}$, then run the final ProcOnion processing (of the sender) to obtain the backward message.

With everything in place, we give a proof sketch.

Sketch. For simplicity, consider a stateful variant of EROR, where instead of deriving the keys $K^i_{\rightarrow}$, $K^j_{\leftarrow}$ from a master key, all keys are truly random. Moreover, let us pretend the derived subkeys $K_{\mathsf{SKE}}, K_{\mathsf{MAC}}, K_{\mathsf{PRF}}$, are truly random. In the full proof, this is easily achieved by suitably replacing derived keys with truly random ones, or undoing that, by reduction to PRF-security of KDF.

Game $LU^+_{\rightarrow}$ for $j < n_{\rightarrow}$. We first consider the $LU^+_{\rightarrow}$ game for $j < n_{\rightarrow}$, with $b = 0$, i.e., the "real word". We focus solely on the challenge onion O^1 and processing the challenge query O^j. Processing non-challenge queries is trivial. Since $b = 0$, we have $O =$ FormOnion$(m, (P^i_{\rightarrow})^{n_{\rightarrow}}_{i=1}, (P^i_{\leftarrow})^{n_{\leftarrow}}_{i=1})$. Now, we make following hybrid steps. (We omit the arrows for the onions, as they are all in forward direction.)

- If RecognizeOnion recognizes a query O as challenge, additionally compare header and forward payload to the challenge onion U^j. If they differ, output FAIL, otherwise, continue processing the query. [SUF-CMA of MAC.]
- Instead of processing the challenge query O^j, simply output $\overline{O}^{j+1} = (hdr, fwd, bwd)$, where $\overline{O}^{j+1}$ is a new variable, which is set to $hdr = U^{j+1}.hdr$, $fwd = U^{j+1}.fwd$ and bwd is the honest processing of $O^j.bwd$. [This change is only conceptual.]
- Replace $\overline{O}^{j+1}.bwd$ with true randomness. [PRF-security of PRF and DLR\$-CPA of SKE]
- In $\overline{O}^{j+1}$, replace the header blocks $B^{j+1}_{N-j+1}, \ldots, B^{j+1}_N$ by randomness $R^{j+1}_{N-j+1}, \ldots, R^{j+1}_N$. [DLR\$-CPA of SKE]
- In O^j, replace the header blocks $B^j_2, \ldots, B^j_{N-j+1}$ by randomness $R^j_2, \ldots, R^j_{N-j+1}$. [DLR\$-CPA of SKE]
- In $\overline{O}^{j+1}$, encrypt ($\mathtt{SNDR}, omsk, (n_{\rightarrow} - j, n_{\leftarrow})$) within $c^{n_{\leftarrow}}_{\leftarrow}$ in Onionize. [IND-CCA of PKE]
- At this point, one can see the $\overline{O}^{j+1}$ is distributed identically to FormOnion$(m, (P^i_{\rightarrow})^{n_{\rightarrow}}_{i=j+1}, (P^i_{\leftarrow})^{n_{\leftarrow}}_{i=1}; \mathcal{R})$. Formally, some more changes which use that SKE is a permutation cipher

are made, so that the header blocks which have been replaced by randomness are now random blocks wrapped in encryption layers. Since a permutation of a random block is again a random block, this does not affect the distribution of $\overline{O}^{j+1}$ and the change is conceptual.

- In O^j, encrypt $(\texttt{RCVR}, K^j_{\rightarrow}, \bot)$ in $c^j_{\rightarrow}$ in Onionize. [IND-CCA of PKE]
- In O^j, encrypt $(\bot, m')$ instead of $(O^1_{\leftarrow}, m)$ in $O^j.fwd$ for random m'. [DLR\$-CPA of SKE]
- At this point, O^j is modified so that O^1 is distributed as $\mathsf{FormOnion}(m', (P^i_{\rightarrow})^j_{i=1}, ())$, by arguing similar to $\overline{O}^{j+1}$.
- We reached $LU^+_{\rightarrow}$ with $b = 1$.

Cryptographically, all steps are simple and have straightforward reductions. The main difficulty of the proof is to realize that

- replacing header blocks $B^{j+1}_{N-j+1}, \dots, B^{j+1}_N$ and header blocks $B^j_2, \dots, B^j_{N-j+1}$ by randomness truly decouples the headers of O^1 and O^2, we provide a sketch of this situation in Fig. 5.
- the way that O^1 and $\overline{O}^{j+1}$ are eventually computed in the end is *syntactically exactly* as FormOnion.

This argument for syntactic equality with FormOnion is simple overall, but does some require care. Since the proof for case $j = n$ for $LU^+_{\rightarrow}$ is similar, we omit it in this sketch.

Game $LU^+_{\leftarrow}$ for $j < n_{\leftarrow}$. In large parts, the argument is analogous to $LU^+_{\leftarrow}$ (except with directions switched to backward). In particular, the same approach of "disconnecting" $O^j_{\leftarrow}$ and $\overline{O}^{j+1}_{\leftarrow}$ and obtaining syntactically new FormOnions applies in large parts. The major difference is how to correctly handle ExtractPayload and ensure integrity of the backward payload. We sketch the new steps now.

- In exception (1) (resp. (2)) of $LU^+_{\leftarrow}$, if RecognizeOnion recognizes a query, additionally compare header and forward payload to the challenge onion ($U^j_{\leftarrow}$ resp. $U^{n_\leftarrow}_{\leftarrow}$). If they differ, output $\texttt{FAIL}$, otherwise, continue processing normally. [SUF-CMA of MAC]
- In exception (2) of $LU^+_{\leftarrow}$, if the challenge is recognized and does not cause $\texttt{FAIL}$, and if ProcOnion outputs $(m, \bot)$ with $m \neq \bot$, then output the extracted m^* to the adversary. [PRF-security of PRF]
- In some steps, slightly change what is encrypted, e.g., $\overline{O}^{j+1}_{\leftarrow}$ encrypts $(\texttt{RCVR}, K, \bot)$ instead of using $\texttt{SNDR}$, etc.

The most important change is that in exception (2), we *always* output m^* (or $\bot$), where m^* was obtained through ExtractPayload. Here, it is crucial to ensure that if ExtractPayload outputs m^*, the adversary cannot maul the onion $\overline{O}^{j+1}_{\leftarrow}$ to some ciphertext which decrypts to any other $m \neq m^*$, even though $\mathcal{A}$ knows the end-to-end encryption keys. Recall that in AE the PRF PRF is used as follows:

- It is used as a deterministic and *unique* MAC in AE.Enc and AE.Dec.
- It is used to (un)mask the MAC with a PRF value in AE.Wrap and AE.Unwrap.

Now, consider ExtractPayload. It works as ProcOnion in the case $meta.role = \texttt{SNDR}$, that is, unwrapping the backward payload until $O^1_{\leftarrow}$ is reached, and then decrypting it. Observe that $\mathsf{AE.Wrap}(K, _)$ and $\mathsf{AE.Unwrap}(K, _)$ are permutations and AE.Wrap is the inverse of AE.Unwrap. In particular, if the adversary's query $O^{n_\rightarrow}_{\leftarrow}.bwd$ differs from an honest unwrapping of $\overline{O}^{j+1}_{\leftarrow}.bwd$, then rewrapping $O^{n_\rightarrow}_{\leftarrow}.bwd$ during ExtractPayload or ProcOnion necessarily arrives at an intermediate bwd^{j+1} which differs from $\overline{O}^{j+1}_{\leftarrow}.bwd$. Now, it is not hard to see that AE.Unwrap will force the MAC to become invalid with overwhelming probability. There are two cases:

- $\overline{O}^{j+1}_{\leftarrow}.bwd.ctxt \neq bwd^{j+1}.ctxt$: As unwrapping is a permutation, we observe that the AE.Dec not reject if only if $\mathsf{PRF}K_{\mathsf{PRF}}(\overline{O}^{j+1}_{\leftarrow}.bwd.ctxt) = \mathsf{PRF}_{K_{\mathsf{PRF}}}(bwd^{j+1}.ctxt)$ holds.
- $\overline{O}^{j+1}_{\leftarrow}.bwd.ctxt = bwd^{j+1}.ctxt$ holds, but $\overline{O}^{j+1}_{\leftarrow}.bwd.mac \neq bwd^{j+1}.mac$: Since unwrapping is a permutation the fully unwrapped MAC must differ, but unwrapping gives the same fully unwrapped ciphertexts. Thus, AE.Dec will reject (since MACs are unique).

Since the proof for case $j = 0$ for $LU^+_{\leftarrow}$ is very similar, and omitted we omit it in this sketch. □

7 PERFORMANCE

To evaluate the performance of EROR, we have assessed two aspects of our format: We evaluate the computational effort required to generate and process onion packets, and we evaluate the packet size in dependence of the payload size and the path length.

In both cases, we compare with Sphinx [12] as a format used in practice, and the repliable onion formats based on updatable encryption and SNARGs by Kuhn, Hofheinz, Rupp, and Strufe [20].

7.1 Implementation

For our experiments, we have implemented EROR in Rust. Rust was chosen because it provides a memory-safe, fast language, and implementations of existing mix formats are available to re-use. We have chosen the following building blocks for our implementation:

- For the asymmetric encryption scheme, a CCA-secure version of ElGamal [1] over the Curve25519 elliptic curve [4].
- For the symmetric encryption scheme, AES with 128 bit keys.
- For the message authentication code, HMAC together with SHA3 truncated to 128 bit.

Our choice of building blocks matches what is often used in practice and provides around 128 bits of security.

To provide reference values for Sphinx, we use the implementation of the Nym project[7] in comparable settings. In particular, we choose to represent addresses of mix nodes and recipients using 32 bytes of data, matching Nym's addressing scheme. While this adds more overhead than strictly needed, it makes the comparison fair and shows how EROR fares in practical situations. Additionally, we note that our implementation contains some overhead related to the used serialization libraries.

[7]https://github.com/nymtech/sphinx, accessed at 2023-12-19

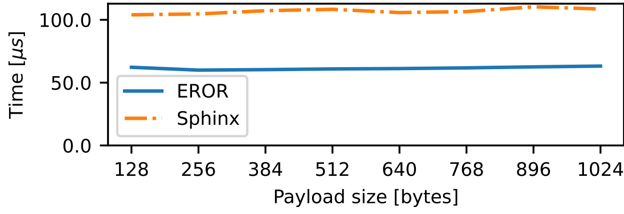

Figure 7: Benchmark of the onion processing time.

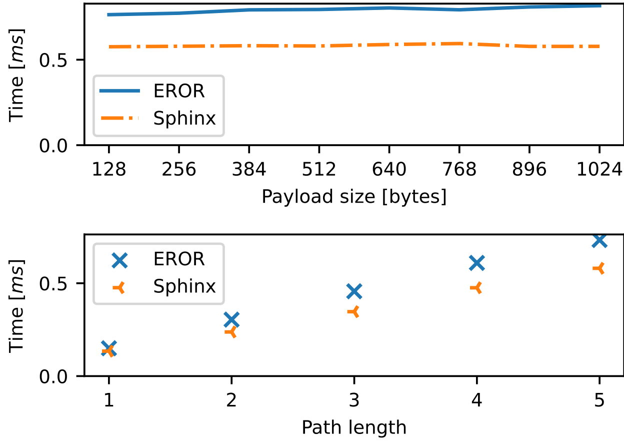

Figure 8: Benchmark of the onion creation time.

Both EROR and Sphinx were benchmarked using the `criterion`[8] tool on a Lenovo Thinkpad E14 AMD G4 with a Ryzen 5 5625U and 16 GiB of RAM.

7.2 Processing Benchmarks

We have measured the onion generation and processing times to assess the performance of EROR in practice. When processing onions, we expect EROR to be twice as fast as Sphinx: The processing time is dominated by expensive public-key operations, namely elliptic curve exponentiations, of which EROR needs one (to decrypt the header block), whereas Sphinx needs two (to derive the shared secret and to blind it for the next relay).

Our benchmarks confirm our expectation: EROR takes around 54 µs to process a 512 byte payload, whereas Sphinx takes 103 µs. For bigger payloads, the time increase is negligible. This is also shown in Figure 7.

For the onion creation, we can see that EROR is slightly slower than Sphinx: For a fixed path length of 5, onion creation takes around 0.73 ms for EROR and 0.57 ms for Sphinx, regardless of the tested payload size. When fixing the payload size and varying the path length, we can see a linear relationship between the number of hops and the time needed to form an onion for both EROR and Sphinx. This result is shown in Figure 8.

As there is no implementation available for the UE- and SNARG-based schemes, we use the approximations of Kuhn et al. [20, p. 8] to assess their performance. For UE, they claim that the performance is dominated by around 200 exponentiations on the elliptic curve, leading to a runtime increase by a factor of approximately 200 compared to EROR. For SNARGs, they expect an even higher processing cost.

Overall, we can say that EROR onion processing time is small, and mixes can process thousands of onions per second. In this regard, EROR is more efficient than Sphinx, but it comes at a slight decrease in efficiency for onion creation. Additionally, EROR is more efficient than the UE and SNARG based schemes by a large margin.

[8] https://crates.io/crates/criterion, accessed at 2023-12-19

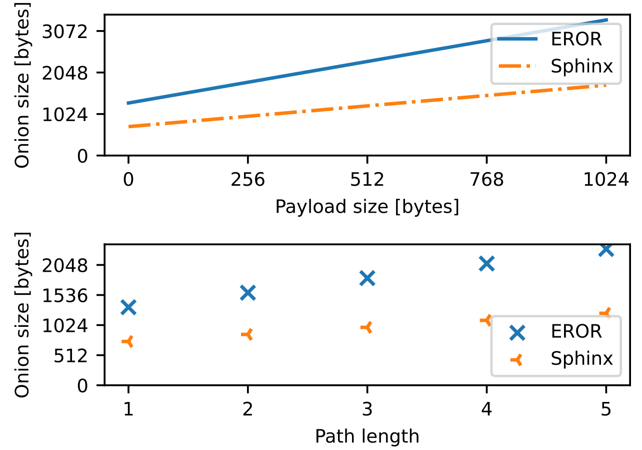

Figure 9: Onion sizes depending on the payload size with a fixed path length of 5 (upper image), and depending on the path length with a fixed payload size of 512 bytes (bottom image).

7.3 Onion Size

We have also evaluated the size of the resulting onions to assess EROR's space overhead. We expect that EROR onions are larger than corresponding Sphinx packets, as each onion contains two payloads: The forward payload and the backward payload, of which only one at a time contains valuable information. Further, Sphinx is optimized for a low overhead, whereas EROR embeds fresh key material for every relay.

We measure the onion sizes using the available implementations of Sphinx and EROR, and we see that the smallest EROR onion that contains 1 byte of payload and encodes a single hop has 305 bytes, whereas the smallest (repliable) Sphinx onion has 234 bytes. As expected, Sphinx headers are more compact, though 24 bytes of the overhead comes from implementation inefficiencies.

By increasing the path length, we see that EROR adds more overhead per encoded hop than Sphinx does: For every extra hop, EROR needs 248 bytes more header space, whereas Sphinx only needs 120 bytes. Further, the Sphinx onions grow linearly by a factor of 1 to the payload size, whereas the EROR onions grow by a factor of 2. Sphinx packets therefore stay more compact than EROR onions. These relationships are shown in Figure 9.

Payload size (bytes)	1	256	1024
EROR (this work)	1297	1808	3344
Sphinx [12]	714	969	1737
UE [20]	4928	42304	154432
SNARG [20]	7328	7584	8352

Table 1: Onion sizes (in bytes) for all formats and varying payload size. The path length is 5 in all cases.

To compare EROR with the UE- and SNARG-based formats, we use the performance estimate in [20, Appendix G] again. We simplify by rounding the sizes to whole bytes, and for their UE scheme we assume that the payload size grows linearly with the amount of group elements needed to represent the message.

From our calculations (shown in Table 1), we can see that the UE-based onions are larger than EROR onions, as the use of updatable encryption blows up the size of the payload. Further, the SNARG onions scale better with payload size than EROR, but they add more overhead per hop, as every hop needs a different SNARG. While for a fixed path length, SNARG onions are initially larger than EROR onions, SNARG will eventually become more efficient for large payloads. For example, a path length of 5 leads to SNARG and EROR onions of the same size for 6033 bytes of payload.

Overall, EROR stays within a factor of ≈ 2 compared to the compact Sphinx format, which we still consider practical. The comparison with the (computationally inefficient) UE and SNARG onions shows that EROR onions are smaller than UE onions, but SNARG onions can reach a similar or smaller size to EROR, depending on the path length and the payload size.

7.4 Concise comparison to related work

The prior (similarly secure) solutions from [20] are at least 100× more computationally expensive and 2.5× to 45× larger (for 1024 bytes payload) than EROR. Even the authors propose it as a "conceptual first step towards an efficient and secure solution" [20]. EROR provides a major improvement regarding performance and *is* the first practical solution.

Compared to Sphinx, EROR's size is "only" 2× larger. For computation, EROR onion generation is slightly slower (≈1.4×), but processing is 2× faster than Sphinx. But note here that Sphinx is insecure against the tagging attack. Here, the EROR solution provides a major improvement in security.

8 CONCLUSION

In this paper, we propose EROR, the first *efficient* and secure onion routing and mix network packet format in the integrated system model that simultaneously prevents against payload tagging attacks and provides request-reply indistinguishability. Our key insight is that it suffices to prevent the payload tagging attack only on forward communications (requests). Thus, by splitting (and duplicating) the payload into forward (request) and backward (reply) payload, we can still achieve request-reply indistinguishability.

Our implementation of EROR shows that onion processing is twice as fast as Sphinx (which is insecure in the integrated system model), at the cost of a larger header and a slightly longer onion creation time. In total, EROR produces onions that are roughly twice the size of Sphinx onions, but both the space overhead and the computational effort are within limits for a practical format. Importantly, EROR outperforms prior *secure* solutions in the integrated system model by a large margin.

Finally, to prove EROR secure, we *strengthen* the security model by including end-to-end integrity and simultaneously *simplify* the proof strategy for such packet formats by mildly strengthening the required properties ($LU^+_{\rightarrow}$ and $LU^+_{\leftarrow}$).

REFERENCES

[1] Masayuki Abe, Eike Kiltz, and Tatsuaki Okamoto. 2009. Compact CCA-secure encryption for messages of arbitrary length. In *PKC 2009* (LNCS). Vol. 5443. Springer, Heidelberg.

[2] Megumi Ando and Anna Lysyanskaya. 2021. Cryptographic shallots: A formal treatment of repliable onion encryption. In *TCC 2021, Part III* (LNCS). Vol. 13044. Springer, Heidelberg.

[3] Michael Backes, Ian Goldberg, Aniket Kate, and Esfandiar Mohammadi. 2012. Provably secure and practical onion routing. In *CSF 2012*. IEEE Computer Society Press.

[4] Daniel J. Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *PKC 2006* (LNCS). Vol. 3958. Springer, Heidelberg.

[5] Jan Camenisch and Anna Lysyanskaya. 2005. A formal treatment of onion routing. In *CRYPTO 2005* (LNCS). Vol. 3621. Springer, Heidelberg.

[6] Ran Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *42nd FOCS*. IEEE Computer Society Press.

[7] Dario Catalano, Mario Di Raimondo, Dario Fiore, Rosario Gennaro, and Orazio Puglisi. 2013. Fully non-interactive onion routing with forward secrecy. *INT J INF SECUR*.

[8] Dario Catalano, Dario Fiore, and Rosario Gennaro. 2009. Certificateless onion routing. In *ACM CCS 2009*. ACM Press.

[9] David L Chaum. 1981. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*.

[10] Chen Chen, Daniele E. Asoni, Adrian Perrig, David Barrera, George Danezis, and Carmela Troncoso. 2018. TARANET: Traffic-Analysis Resistant Anonymity at the NETwork layer. *IEEE EuroS&P*.

[11] Chen Chen, Daniele Enrico Asoni, David Barrera, George Danezis, and Adrian Perrig. 2015. HORNET: high-speed onion routing at the network layer. In *ACM CCS 2015*. ACM Press.

[12] George Danezis and Ian Goldberg. 2009. Sphinx: a compact and provably secure mix format. In *2009 IEEE S&P*. IEEE Computer Society Press.

[13] George Danezis and Ben Laurie. 2004. Minx: a simple and efficient anonymous packet format. In *WPES 2004*.

[14] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The second-generation onion router. Tech. rep. Naval Research Lab Washington DC.

[15] Joan Feigenbaum, Aaron Johnson, and Paul Syverson. 2012. Probabilistic analysis of onion routing in a black-box model. *ACM TISSEC*, 15, 3.

[16] Oded Goldreich. 2004. *The Foundations of Cryptography - Volume 2: Basic Applications*. 2004. Cambridge University Press. ISBN: 0-521-83084-2.

[17] David M Goldschlag, Michael G Reed, and Paul F Syverson. 1996. Hiding routing information. In *International workshop on information hiding*.

[18] Aniket Kate, Greg M. Zaverucha, and Ian Goldberg. 2010. Pairing-based onion routing with improved forward secrecy. *ACM TISSEC*, 13.

[19] Christiane Kuhn, Martin Beck, and Thorsten Strufe. 2020. Breaking and (partially) fixing provably secure onion routing. In *2020 IEEE S&P*. IEEE Computer Society Press.

[20] Christiane Kuhn, Dennis Hofheinz, Andy Rupp, and Thorsten Strufe. 2021. Onion routing with replies. In *ASIACRYPT 2021, Part II* (LNCS). Vol. 13091. Springer, Heidelberg.

[21] Sjouke Mauw, Jan Verschuren, and Erik P. de Vink. 2004. A formalization of anonymity and onion routing. In *ESORICS 2004* (LNCS). Vol. 3193. Springer, Heidelberg.

[22] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton. 2014. Reconsidering generic composition. In *EUROCRYPT 2014* (LNCS). Vol. 8441. Springer, Heidelberg.

[23] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. 2017. The loopix anonymity system. In *USENIX Security 2017*. USENIX Association.

[24] Philip Scherer, Christiane Weis, and Thorsten Strufe. 2023. Provable security for the onion routing and mix network packet format sphinx. (2023). arXiv: 2312.08028 [cs.CR].

[25] Erik Shimshock, Matt Staats, and Nicholas Hopper. 2008. Breaking and provably fixing minx. In *PETS 2008* (LNCS). Vol. 5134. Springer, Heidelberg.

A PRELIMINARIES AND FURTHER DISCUSSIONS

A.1 Cryptographic primitives

We recall the (standard) notions of cryptographic primitives which we use, and define specific notions tailored to our application.

A.1.1 Symmetric-key Primitives. In the following, all our symmetric encryption schemes are nonce-based and deterministic.

Definition A.1 (Nonce-based SKE). Let $\mathcal{K}$ be the key space, $\mathcal{N}$ the nonce space, $\mathcal{M}$ be the message space, and $\mathcal{C}$ be the ciphertext space, all subsets of bitstring. For simplicity, we assume $\mathcal{K} = \mathcal{N} = \{0,1\}^\lambda$ and $\mathcal{M} = \mathcal{C} = \{0,1\}^*$ unless noted otherwise. A (nonce-based) SKE is a tuple $(\mathsf{Enc}, \mathsf{Dec})$ of deterministic polynomial-time algorithms, where

- $\mathsf{SKE.Enc}_K(n, m) \rightarrow c$, given a secret key $K \in \mathcal{K}$, nonce $n \in \mathcal{N}$, and message $m \in \{0,1\}^*$, outputs a ciphertext c.
- $\mathsf{SKE.Dec}_K(n, c) \rightarrow m$, given a secret key $K \in \mathcal{K}$, nonce $n \in \mathcal{N}$, and ciphertext c, outputs a message m or $\perp$.

Instead of general SKEs, we will restrict to a notion of *(nonce-based) permutation cipher*, defined as follows.

Definition A.2 (Permutation cipher). Let SKE be a (nonce-based) SKE. Then SKE is a (nonce-based) *permutation cipher* if for all $K \in \{0,1\}^\lambda$, $n \in \{0,1\}^\lambda$, the map $\mathsf{SKE.Enc}_K(n, _)$ is a permutation on n-bit strings (for any n) and $\mathsf{SKE.Dec}_K(n, _)$ is its inverse.

We note that permutation ciphers as defined automatically have nice properties, e.g. they are perfectly correct, tidy [22] and perfectly length-regular. Note that permutation ciphers according to Definition A.2 need not be pseudo-random permutations. Indeed, most typical ciphers (which do not provide integrity) are permutation ciphers according to our definition, e.g. pseudo-random permutations, counter-mode stream ciphers, and many other modes of operation.

Our schemes must satisfy a strengthening of IND$-CPA security (i.e., indistinguishability from randomness under chosen plaintext attacks). Namely, in our security proofs, an adversary may either learn an encryption or a decryption of a message/ciphertext. One strict notion is the PRP-CCA property [2, 20], where the adversary has access to an Enc and Dec oracle. In our setting, a much weaker nonce-based notion suffices. Namely, for every nonce, the adversary may either request a single encryption or decryption (never both); nonce-reuse is forbidden. We define this below.

Definition A.3 (DLR$-CPA). Let $\mathsf{SKE} = (\mathsf{Enc}, \mathsf{Dec})$ be a nonce-based correct and tidy permutation cipher. The doubly LR$-CPA-secure (short: DLR$-CPA) experiment, is defined as follows:

(1) The challenger picks a secret key $K \leftarrow_R \mathcal{K}$ and a challenger bit b^*.
(2) The adversary gets access to oracles Enc, Dec where:
   - if $b^* = 0$: $\mathsf{Enc}(n, m) = \mathsf{Enc}_K(n, m)$ and $\mathsf{Dec}(n, c) = \mathsf{Dec}_K(n, c)$;
   - if $b^* = 1$: $\mathsf{Enc}(n, m)$ and $\mathsf{Dec}(n, c)$ always output fresh randomness.
(3) The adversary eventually outputs a guess b and the experiment outputs 1 if $b = b^*$, else 0.

The advantage of an adversary in the DLR$-CPA experiment for SKE is defined as

$$\mathsf{Adv}^{\mathsf{dbl\$\text{-}cpa}}_{\mathsf{SKE},\mathcal{A}} = \Pr[b = 1 | b^* = 0] - \Pr[b = 1 | b^* = 1].$$

A scheme SKE is DLR$-CPA secure if any PPT adversary has negligible advantage.

We note that any (stream) cipher with $\mathsf{Enc} = \mathsf{Dec}$ is obviously DLR$-CPA secure if it is IND$-CPA secure. Thus, typical stream ciphers are DLR$-CPA-secure. Any PRP-CCA-secure (nonce-based) PRP also yields a DLR$-CPA secure SKE.

Definition A.4 (PRF). Let $\mathcal{K}$ be the key space, $\mathcal{M}$ be the message space, and $\mathcal{Y}$ be the output space, all subsets of bitstring. For simplicity, we assume $\mathcal{K} = \mathcal{Y} = \{0,1\}^\lambda$ and $\mathcal{M} = \{0,1\}^*$ unless noted otherwise. A pseudo-random function (PRF) PRF takes as input a key and a message and outputs a value $y \in \mathcal{Y}$.

Let $\mathcal{A}$ be a distinguisher. We write

$$\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{PRF},\mathcal{A}}(\lambda) = \Pr_{K \leftarrow_R \mathcal{K}}(\mathcal{A}^{\mathsf{PRF}_K(_)} = 1) - \Pr_{\mathsf{RF} \leftarrow_R \mathcal{Y}^{\mathcal{M}}}(\mathcal{A}^{\mathsf{RF}(_)} = 1)$$

where RF is a truly random function. A PRF is secure, if for any PPT adversary the respective advantage is negligible.

A pseudorandom generator is effectively PRF with trivial message space $\mathcal{M} = \{*\}$ and $\mathcal{Y} = \{0,1\}^\ell$ where $\ell > \lambda$, i.e., a PRG only stretches a given key K to a longer pseudorandom string.

Definition A.5 (MAC). Let $\mathcal{K}$ be the key space, $\mathcal{M}$ be the message space, and $\mathcal{T}$ be the tag space, all subsets of bitstring. For simplicity, we assume $\mathcal{K} = \mathcal{T} = \{0,1\}^\lambda$ and $\mathcal{M} = \{0,1\}^*$ unless noted otherwise. In general, we assume that $\mathcal{T}$ has fixed bitlength.[9] A message authentication code (MAC) MAC is a tuple $(\mathsf{Sign}, \mathsf{Verify})$ of deterministic polynomial-time algorithms, where

- $\mathsf{MAC.Sign}_K(m) \rightarrow \tau$, given a secret key $K \in \mathcal{K}$ and message $m \in \{0,1\}^*$, outputs a tag τ.
- $\mathsf{MAC.Verify}_K(m, \tau) \rightarrow b$, given a secret key $K \in \mathcal{K}$, message m and purported tag τ, outputs a bit b.

A tag τ is *valid* on m (under key K) if $\mathsf{MAC.Verify}_K(m, \tau) = 1$. A MAC scheme is perfectly correct if any honestly generated MAC tag is valid. The SUF-CMA security game for MAC is as follows:

- The adversary selects a message $m^* \in \mathcal{M}$.
- The challenger samples $K \leftarrow_R \mathcal{K}$ and gives the adversary $\mathcal{A}$ access to a Sign and a Verify oracle.
- The adversary wins if it ever queries a forgery (m^*, τ^*) to Verify, i.e. if $\mathsf{Verify}_K(m^*, \tau^*) = 1$ but m^* was never signed by the challenger before.

The advantage $\mathsf{Adv}^{\mathsf{suf\text{-}cma}}_{\mathsf{MAC},\mathcal{A}}$ is the probability that $\mathcal{A}$ wins. We call MAC *selectively unforgeable under chosen messages attacks (SUF-CMA)* secure if any PPT adversary has negligible advantage. Moroever, we define the EUF-CMA experiment and security under *existentially unforgeable under chosen messages attacks (EUF-CMA)*, where the adversary does not need to select m^* at the beginning.

A MAC tag τ is called *valid* on m (regarding the implicit key K), if $\mathsf{MAC.Verify}_K(m, \tau) = 1$.

[9] Variable tag length is incompatible with our construction, which needs ciphertexts of fixed length.

Remark A.6 (PRF as MAC). A PRF gives a deterministic MAC by setting $\mathsf{MAC.Sign}_K(m) = \mathsf{PRF}_K(m)$ and $\mathsf{MAC.Verify}_K(m, \tau) = 1$ if $\mathsf{PRF}K(m) = \tau$, else 0. It has a unique valid tag for each message and the tags leak no information about the messages.

A.1.2 Public-key Encryption (PKE). A public-key encryption (PKE) scheme is a tuple $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ of efficient algorithms, where

- $\mathsf{Gen}(1^\lambda) \to (pk, sk)$, given a security parameter, outputs a pair (pk, sk) of public and secret keys.
- $\mathsf{PKE.Enc}_{pk}(m) \to c$, given a public key pk and message $m \in \{0,1\}^*$, outputs a ciphertext c.
- $\mathsf{PKE.Dec}_{sk}(c) \to m$, given a secret key sk and ciphertext $c \in \{0,1\}^*$, outputs a message m or $\perp$.

We require that schemes are *perfectly correct*, i.e. any honest encryption under any honest public key decrypts correctly for the respective secret key, and *length-regular*, i.e., the ciphertext bitlength depends deterministically on the message bitlength (and security parameter). We use the usual definition of CCA-security, see e.g. [16].

A.2 Correctness

An onion routing scheme is correct if the onions follow the paths and deliver the messages used as onion parameters during the construction of the onion resp. reply. We verbatimly follow [20]'s defintion for correctness except for changes in the notation.

Definition A.7 (Correctness). Suppose (G, FormOnion, ProcOnion, ReplyOnion) is a repliable OR scheme with maximal path length N. Then for all $n, n_\leftarrow \leq N$, $\lambda \in \mathbb{N}$, all choices of the public parameter p, all choices of the randomness $\mathcal{R}$, all choices of forward and backward paths $\mathcal{P}_\rightarrow = (P^1, \ldots, P^n)$ and $\mathcal{P}_\leftarrow = (P^1{}_\leftarrow, \ldots, P^{n_\leftarrow}_\leftarrow)$, all $(pk^i_{(\leftarrow)}, sk^i_{(\leftarrow)})$ generated by $G(1^\lambda, p, P^i{}_\leftarrow)$, all messages $m, m_\leftarrow$, all possible choices of internal randomness used by ProcOnion and ReplyOnion, the following needs to hold:

Correctness of forward path.

$Q^i = P^i$, for $1 \leq i \leq n$ and $Q^1 := P^1$, where
$O^1 \leftarrow \mathsf{FormOnion}(m, (P^1, \ldots, P^n), (P^1{}_\leftarrow, \ldots, P^{n_\leftarrow}_\leftarrow), (pk^1, \ldots, pk^n), (pk^1_\leftarrow, \ldots, pk^{n_\leftarrow}_\leftarrow); \mathcal{R})$,
$(O^{i+1}, Q^{i+1}) \leftarrow \mathsf{ProcOnion}(sk^i, O^i, Q^i)$.

Correctness of request reception.

$(m, \perp) = \mathsf{ProcOnion}(sk^n, O^n, P^n)$

Correctness of backward path.

$Q^i_\leftarrow = P^i_\leftarrow$, for $1 \leq i \leq n-1$ where
$(O^1_\leftarrow, Q^1_\leftarrow) \leftarrow \mathsf{ReplyOnion}(sk^n, m_\leftarrow, O^n, P^n)$,
$(O^{i+1}_\leftarrow, Q^{i+1}_\leftarrow) \leftarrow \mathsf{ProcOnion}(sk^i_\leftarrow, O^i_\leftarrow, Q^i_\leftarrow)$.

Correctness of reply reception.

$(m_\leftarrow, \perp) = \mathsf{ProcOnion}(sk^{n_\leftarrow}_\leftarrow, O^{n_\leftarrow}_\leftarrow, P^{n_\leftarrow}_\leftarrow)$

B FULL PROOF OF THEOREM 4.7

We verbatimly reuse the proof from [20], but adapt it to our new requirements and notation. The proof first describes the simulator $\mathcal{S}$ and then shows indistinguishability of the environment's view in the real and ideal world. Interestingly, the strategy used in the simulator is already sufficient even for our adaption. We hence only fix some minor inconsistencies for the simulator and the actual technical changes occur in the indistinguishability argument. New parts related to the adapted security requirements are highlighted in blue.

Constructing Simulator $\mathcal{S}$

$\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}$ 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 $(info, nextRelay, temp)$. Each entry in this list corresponds to a stage in processing an onion that belongs to a communication of an honest sender. 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 along with communication information, i.e., $(onion, currentRelay, nextRelay, information)$.
- The *Reply*-list containing reply information along with the forward id for communication with a corrupted sender (id_{fwd}, reply information).
- The C-list containing reply information together with the temp for communication with an honest sender $(P^i, reply, temp)$.

$\mathcal{S}$'s behavior on a message from $\mathcal{F}$. In case the received output belongs to an adversarial sender's communication[10]:

Case I: "*start* belongs to the onion from P^0 with $id, P^r, m, n, \mathcal{P}_\leftarrow, \mathcal{P}_\rightarrow, d$ as answer to id"; an honest node is replying to an onion of a corrupted sender. $\mathcal{S}$ knows that the next output "Onion *temp* in direction d from ..." includes the first part of this backward path, that he chose to consist of one adversarial node and just needed to give P^r (the backward sender) a chance to reply (as $\mathcal{S}$ did not know where the real reply path goes and does not need to know). $\mathcal{S}$ thus ignores this output and does not react with another Case on this. To construct the right real world reply onion, $\mathcal{S}$ looks up the reply information (id, reply info) for this id in the *Reply*-list and uses the information to construct an onion: $(O^1, P^1) \leftarrow \mathsf{ReplyOnion}(sk^r, m, \text{reply info}, P^r)$ and sends O^1 to P^1, if P^1 is adversarial or to $\mathcal{A}$'s party representing the link between the P^r and P^1, if P^1 is honest. (Note that P^r cannot be adversarial for this output as then both sender and receiver would be corrupt, which only activates cases VIII b and II (as it works without including any reply onion from the view of the ideal world).)

Case II: any output together with "*temp* belongs to onion from P^0 with $id, P^r, m, n, \mathcal{P}$" for $temp \notin \{start, end\}$. 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 III: "*end* belongs to onion from P^0 with $id, P^r, m, n, \mathcal{P}$". This means an honest relay is done processing an onion received

[10] $\mathcal{S}$ knows whether they belong to an adversarial sender from the output it gets.

from $\mathcal{A}$ that was not the protocol-conform processing of an honest sender's communication (processing that follows Case VIII). $\mathcal{S}$ finds ($onion, currentRelay, nextRelay, information$) with these inputs as $information$ in the O-list (notice that there has to be such an entry) and as $currentRelay$ sends the onion $onion$ to $nextRelay$ if it is an adversarial one, or it sends $onion$, as if it is transmitted, to $\mathcal{A}$'s party representing the link between the currently processing honest relay and the honest $nextRelay$.

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

Case IV: "Onion $temp$ from P^{O^i} routed through () to $P^{O^{i+1}}$". 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 $\mathcal{R}$, calculates $O^1 \leftarrow \mathsf{FormOnion}(m_{rdm}, \mathcal{P}, (), (pk)_{\mathcal{P}_{rdm}}, (); \mathcal{R})$, 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 ($info = (2, m_{rdm}, \mathcal{P}, (), (pk)_{\mathcal{P}_{rdm}}, (), \mathcal{R}), P^{O^{i+1}}, temp$) on the r-list.

Case V: "Onion $temp$ from P^{O^i} routed through $(P^{O^{i+1}}, \dots, P^{O^{j-1}})$ to P^{O^j}". $\mathcal{S}$ picks the path $\mathcal{P} = (P^{O^{i+1}}, \dots, P^{O^{j-1}})$, a randomness $\mathcal{R}$ and a message m_{rdm} and calculates $O^1 \leftarrow \mathsf{FormOnion}(m_{rdm}, \mathcal{P}, (), (pk)_{\mathcal{P}_{rdm}}, (); \mathcal{R})$ and sends the onion O^1 to $P^{O^{i+1}}$, as if it came from P^{O^i}. Then $\mathcal{S}$ stores ($info = (j - i, m_{rdm}, \mathcal{P}, (), (pk)_{\mathcal{P}_{rdm}}, (), \mathcal{R}), P^{O^j}, temp,$) on the r-list.

Case VI: "Onion $temp$ from P^{O^i} with message m for P^r routed through $(P^{O^{i+1}}, \dots, P^{O^n})$". Note that this output always occurs together with "temp's first part of the backward path is $\mathcal{P}_{\leftarrow}$" (P^r received a forward onion) [as otherwise P^r would receive a backward onion, the sender (=backward receiver) would be corrupt and hence the whole communication would be simulated by using cases VIII (b) and III (and VIII (a1) and I).]: $\mathcal{S}$ picks the path $\mathcal{P} = (P^{O^i}, \dots, P^{O^n}, P^r)$, randomness $\mathcal{R}$ and calculates $O^1 \leftarrow \mathsf{FormOnion}(m, \mathcal{P}_{rdm}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{rdm}}, (pk)_{\mathcal{P}_{\leftarrow}}; \mathcal{R})$ and sends the onion O^1 to $P^{O^{i+1}}$, as if it came from P^{O^i}. Further, $\mathcal{S}$ stores ($\mathcal{P}_{\leftarrow}.last, info, temp$) with $info = (n + \mathcal{P}_{\leftarrow}.lastPosition, m, \mathcal{P}_{rdm}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{rdm}}, (pk)_{\mathcal{P}_{\leftarrow}}, \mathcal{R})$ on the C-list. (Note that as this is an honest communication $\mathcal{P}_{\leftarrow}.last$ is honest.)

$\mathcal{S}$'s behavior on a message from $\mathcal{A}$. $\mathcal{S}$, as real world honest party P^i, received an onion O from $\mathcal{A}$ as adversarial player P^a.

Case VII: The onion is recognized (using $\mathsf{RecognizeOnion}$) with the information of an r-list entry ($information, P^i, temp$). In this case O is the protocol-conform processing of an onion from a communication of an honest sender. $\mathcal{S}$ calculates $\mathsf{ProcOnion}(SK(P^i), O, P^i)$. If it returns a fail ($O$ is a replay or modification that is detected and dropped), S does nothing. Otherwise, $\mathcal{S}$ sends the message (Deliver Message, $temp$) to $\mathcal{F}$.

Case VIII. The onion is not recognized (using $\mathsf{RecognizeOnion}$) with the information of any r-list entry ($information, P^i, temp$). $\mathcal{S}$ calculates $\mathsf{ProcOnion}(sk^i, O, P^i) = (O', P')$ (and aborts if this fails).

(a) $P' = \perp$: P^i is the recipient and O' contains a message and reply information; only a message (if send as reply or not repliable) or a fail symbol.

(a1) Contains a message and reply information. S sends the message "($ProcessNewOnion, P^i, O', (), \mathcal{P}_{\leftarrow}$) with $\mathcal{P}_{\leftarrow} = (P^a)$" (note that this is only one adversarial node) to $\mathcal{F}$ on P^a's behalf and as $\mathcal{A}$ already delivered this message to the honest party sends (Deliver Message, $temp$) for the belonging $temp$. Further, $\mathcal{S}$ stores (id, O') in the $Reply$-list (to later reply to this onion).

(a2) contains only a message m ($\mathcal{S}$ knows this as it can try to create a reply to it with P^i). This means the adversary possibly replied to an honest sender's forward onion. $\mathcal{S}$ checks for all ($P^i, reply, temp$) in C-List if the onion is recognized (using $\mathsf{RecognizeOnion}$) with any $reply$-info on this list. If so (it was a reply to $temp$), $\mathcal{S}$ sends the message ($ReplyOnion, m, temp$) to $\mathcal{F}$ on P^a's behalf and, as $\mathcal{A}$ already delivered this message to the honest party, sends (Deliver Message, $temp'$) for the belonging $temp'$. Otherwise $\mathcal{S}$ creates this onion in the $\mathcal{F}$ as it sends ($ProcessNewOnion, P^i, O', (), \perp$) and (Deliver Message, $temp$) for the corresponding $temp$. (Notice that $\mathcal{S}$ knows which $temp$ and id belongs to this communication as it is started at an adversarial party P^a).

(b) $P' \neq \perp$: $\mathcal{S}$ picks a message $m \in \mathcal{M}$. $\mathcal{S}$ sends on P^a's behalf the message, $Process_New_Onion(P^{a\prime}, m, (P^i), ())$ (notice that this is not repliable) and $Deliver_Message(temp)$ for the belonging $temp$ to $\mathcal{F}$ (notice that $\mathcal{S}$ knows the $temp$ as in case (a)). $\mathcal{S}$ adds the entry $(O', P^i, P', (P^a, id, P^{a\prime}, m, (P^i), ()))$ to the O-list.

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 node on the forward path (relay or receiver) are replaced with random onion layers embedding the same path and the onion after it is relaced with one with a shortened onion path. More precisely, this machine acts like $\mathcal{H}_0$ except that the consecutive onion layer $O^1, O^2, \dots, O^j$ from an honest sender P^0 to the next honest node P^j are replaced with $\bar{O}^1$ and its following processings by calculating (with honestly chosen randomness $\mathcal{R}$) $\bar{O}^1 \leftarrow \mathsf{FormOnion}(m_{rdm}, \mathcal{P}, (), (pk)_{\mathcal{P}}, (); \mathcal{R})$ where m_{rdm} is a random message, $\mathcal{P} = (P^1, \dots, P^j)$. $\mathcal{H}_1$ keeps an $\bar{O}$-list and stores ($info = (m, \mathcal{P}, \mathcal{P}_{\leftarrow}, (pk), (pk)_{\leftarrow}), P^j, (O_R^1, P^{j+1}), \mathcal{R}$) where $info$ are the parameters and randomness used for the original sender's onion creation and O_R^1 is calculated[11] as $O_R^1 \leftarrow \mathsf{FormOnion}(m, \tilde{\mathcal{P}}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\tilde{\mathcal{P}}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}};$

[11] As some parts of the onion are non-deterministic, we cannot assume that the sender and thus our machines know the onion layer after the honest node (only the deterministic part is known) and thus we have to replace it with an onion created as a close match due to the reproducability requirement.

$\tilde{\mathcal{R}}$), where the paths with $\tilde{\mathcal{P}}_{\rightarrow} = (P^{j+1}, \ldots, P^n)$ and message are chosen as in the original sender's call in $\mathcal{H}_0$.[12] If an onion $\tilde{O}$ is sent to P^j, the machine tests if processing results in a fail (replay or modification detected and dropped). If this is not the case, $\mathcal{H}_1$ uses $\mathsf{RecognizeOnion}((j, \rightarrow), \tilde{O}, m, \mathcal{P}, \mathcal{P}_{\leftarrow}, (pk), (pk)_{\leftarrow}, \mathcal{R})$ for all recognize information stored in the $\bar{O}$-list where the second entry is P^j. If it finds a match, the belonging O_R^1 is send to P^{j+1} as processing result of P^j. Otherwise, $\mathsf{ProcOnion}(sk^{P^j}, \tilde{O}, P^j)$ is used. As before, the environment is informed by honest parties about *temp* when they get an onion and about the included message if they are the final receiver – no matter whether this happened as a reaction to something on th $\bar{O}$-list, or not.

$\mathcal{H}_0 \approx_I \mathcal{H}_1$. The environment gets notified about *temp* when an honest party receives any onion layer. If the honest party is the onion's recipient, the environment further learns the message and whether it belongs to a request of the sender and if so, the request's ID. If there is at least one honest relay on the honest sender's path before the receiver, our replacement is introducing no change in the observations for the environment: The environment will still only see a freshly generated random *temp* at the honest relay. If there is no honest relay on the path, the second honest node is the receiver. In this case, as our replacement is reintroduing the original onion's processing at the second honest node, this node will behave as having received the originally included message, hence the environment again gets the same observation as before - namely the included message and the information that it is not a reply.

$\mathcal{A}$ observes the onion layers after P^0 and if it sends an onion to P^j, the result of the processing after the honest node. 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 $LU^{+}_{\rightarrow}$:

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.[13] For the other relays the attack decides on the keys as $\mathcal{A}$ (for corrupted) and the protocol (for honest) does. 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.[14] 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.

[12] $\mathcal{H}_1$ knows this as it simulates all honest senders and thus knows the parameters this honest sender picked.

[13] As both honest nodes are randomly drawn this does not change the success

[14] This includes that duplicates are dropped and onions are processed before they are replied (as assumed in Section 4.1).

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 node 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.]

$\mathcal{H}_1^{<x-1} \approx_I \mathcal{H}_1^{<x}$. Analogous to above. Apply argumentation of indistinguishability ($\mathcal{H}_0 \approx_I \mathcal{H}_1$) for every replaced subpath.[15]

Hybrid $\mathcal{H}_2$. In this hybrid, for the first forward communication, for which in the adversarial processing no recognition falsifying modification occurred and other modification does not result in a fail,[16] onion layers between two consecutive honest relays on the forward path are replaced with random onion layers embedding the same path. Additionally, for all forward communication replacements between the sender and the first relay happen as in $\mathcal{H}_1^*$. More precisely, this machine acts like $\mathcal{H}_1^*$ except that the processing of O^j; i.e. the consecutive onion layers $O^{j+1}, \ldots, O^{j'}$ from a communication of an honest sender, starting at the next honest node P^j to the next following honest node $P^{j'}$, are replaced with $\bar{O}^1, \ldots, \bar{O}^{j'-j}$ by sending $\bar{O}^1$. Thereby, for an honestly chosen randomness $\mathcal{R}$: $\bar{O}^1 \leftarrow \mathsf{FormOnion}(m_{rdm}, \mathcal{P}, (), (pk)_{\mathcal{P}}, (); \mathcal{R})$ where m_{rdm} is a random message, $\mathcal{P} = (P^j, \ldots, P^{j'})$ is the path between the honest nodes. $\mathcal{H}_2$ stores $(info = (m, \mathcal{P}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}, \mathcal{R}), P^{j'}, (O_R^1, P^{j'+1}))$,[17] where $O_R^1 \leftarrow \mathsf{FormOnion}(m, \tilde{\mathcal{P}}_{\rightarrow}, \mathcal{P}_{\leftarrow}, (pk)_{\tilde{\mathcal{P}}_{\rightarrow}}, (pk)_{\mathcal{P}_{\leftarrow}}; \tilde{\mathcal{R}})$, where messages and paths with $\tilde{\mathcal{P}}_{\rightarrow} = (P^{j+1}, \ldots, P^n)$ are chosen as the original sender would pick them in the original construction (of the complete onion),[18] 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, $\mathcal{H}_2$ checks $\mathsf{RecognizeOnion}((j'-j, \rightarrow), \tilde{O}, info)$ for any $info$ on the $\bar{O}$-list where the second entry is $P^{j'}$. If it finds a match, the belonging O_R^1 is used as processing result of $P^{j'}$. Otherwise, $\mathsf{ProcOnion}(sk^{P^{j'}}, \tilde{O}, P^{j'})$ is used. The environment is informed by the honest parties as before.

$\mathcal{H}_1^* \approx_I \mathcal{H}_2$. $\mathcal{H}_2$ replaces for one communication (and all its replays), the first subpath between two consecutive honest nodes 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 $\mathcal{H}_1^*$ and $\mathcal{H}_2$) 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 $\mathcal{H}_2$ (outputted by $\mathcal{H}_1^*$) or in case of the communication considered by $\mathcal{H}_2$, the newly drawn random replacement (generated by $\mathcal{H}_2$); and the processing after $P^{j'}$.

[15] Technically, we need the onion layers as used in $\mathcal{H}_1$ (with replaced onion layers between a honest sender and first honest node) in this case. Hence, slightly different than before, the attack needs to simulate the other communications not only by the oracle use and processing, but also by replacing some onion layers (between the honest sender and first honest node) with randomly drawn ones as $\mathcal{H}_1$ does.

[16] We treat modifying adversaries on other parts later in a generic way.

[17] $P^{j'+1}$ might be $\perp$ and O_R^1 the message m if $P^{j'}$ is the honest receiver.

[18] $\mathcal{H}_2$ can do this as it knows all parameters of the original onion and can link the current layer back to the original sending request of the honest sender.

The onions $\bar{O}^{earlier}$ are chosen independently at random by $\mathcal{H}_1^*$ and $\mathcal{H}_2$ such that they embed the original path between an honest sender and the first honest relay, but contain a random message. As they are replaced by other original onion layers after P^j (there was no recognition falsifying modification for this communication) and include a random message, onions $\bar{O}^{earlier}$ have no connection to onions output by P^j and hence can simply be generated for any distinguisher based on the knowledge and oracles an attacker on $LU^+_{\rightarrow}$ has access to.

Thus, all that is left are the original/replaced onion layer after the first honest node 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 $LU^+_{\rightarrow}$.

Counting explanation for $\mathcal{H}_2^{<x}$. Communication paths consist of possible multiple 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 node starting the selected subpath) did a recognition falsifying modification or a modification that results in a processing fail, the communication is skipped. As part of this replacement, if a communication involves an honest receiver, the reply onion will be replaced with a new forward onion, which we then treat as an additional forward communication in terms of the onion replacement in $\mathcal{H}_2^{<x}$. The outputs to the environment are still done by the Hybrid as for the original reply communication. Notice that the Hybrid knows all information necessary to link the reply back to the original request.

$\mathcal{H}_2^{<x-1} \approx_I \mathcal{H}_2^{<x}$. 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 until the honest (forward) sender is replaced and the onion before is replaced by one with a shorter path.

More precisely, this machine acts like $\mathcal{H}_2^*$ except for two replacements. First, the corresponding onion of the forward path is replaced by O^c with shortened backward path: $O^c \leftarrow \mathsf{FormOnion}(m, \mathcal{P}_{\rightarrow}, \tilde{\mathcal{P}}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\tilde{\mathcal{P}}_{\leftarrow}}; \tilde{\mathcal{R}})$ with $\mathcal{P}_{\rightarrow} = (P^{j\rightarrow}, \dots, P^n)$ and $\tilde{\mathcal{P}}_{\leftarrow} = (P^1_{\leftarrow}, \dots, P^j_{\leftarrow})$ starting at the last honest node $P^{j\rightarrow}$ on the forward paths.

Second, 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 $\mathcal{R}$): $\bar{O}^1 \leftarrow \mathsf{FormOnion}(m_{rdm}, \mathcal{P}, (), (pk)_{\mathcal{P}}, (); \mathcal{R})$ where m_{rdm} is a random message, $\mathcal{P} = (P^j_{\leftarrow}, \dots, P^{n_{\leftarrow}}_{\leftarrow})$ is the path from $P^j_{\leftarrow}$ to $P^{n_{\leftarrow}}_{\leftarrow}$. $\mathcal{H}_1^{\leftarrow}$ stores $(info, P^{n_{\leftarrow}}_{\leftarrow} = P^0, m)$, with $info$ being chosen according to the replacement onion (including m_{rdm}) and m being the message of the processing result of the original onion at the honest receiver, on the $\bar{O}$-list. When looking up entries (with RecognizeOnion) on the $\bar{O}$-list, $\mathcal{H}_1^{\leftarrow}$ checks the belonging last entry to be an onion before sending it to the next node and reporting *temp* to the environment. If it discovers a message instead of an onion, this means the node is the receiver. It treats this instance as if it received the reply onion to its request, i.e. reports to the environment. Notice that it has all necessary information as the message is included in $info$ as the processing result and $info$ also contains the original onions randomness and path to allow to match this reply to the corresponding request.

$\mathcal{H}_2^* \approx_I \mathcal{H}_1^{\leftarrow}$. The environment gets notified about *temp* when an honest party receives any onion layer. If the honest party is the onion's recipient, the environment further learns the message and if it belongs to a request of the sender, the request's ID. If there is at least one honest relay on the reply path before the receiver, our replacement is introducing no change in the observations for the environment: The environment still only sees a freshly generated random *temp* at the honest relay. If there is no honest relay on the path, the second honest node is the reply-receiver. In this case, as our replacement is reintroduing the original onion's processing at the second honest node, this node behaves as having received the originally included message, hence the environment again gets the same observation as before - namely the included message and the information that it is a reply to the corresponding request.

$\mathcal{A}$ observes the onion layers before $P^j_{\leftarrow}$ and if it sends an onion to $P^j_{\leftarrow}$ the result of the processing after the honest node. Depending on the behavior of $\mathcal{A}$ three cases occur: $\mathcal{A}$ drops the onion belonging to this communication before $P^j_{\leftarrow}$, $\mathcal{A}$ behaves protocol-conform and sends the expected onion to $P^j_{\leftarrow}$ or $\mathcal{A}$ modifies the expected onion before sending it to $P^j_{\leftarrow}$. 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 $\mathcal{H}_1^{\leftarrow}$ and construct a successful attack on $LU^+_{\leftarrow}$:

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 honest relay with the one that it got from the challenger.[19] For the other relays the attack decides on the keys as $\mathcal{A}$ (for corrupted) and the protocol (for honest) does. 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), so does our attack. The reply is processed (with the knowledge of the keys) until the honest node where the replaced onion layers start and this (processed) reply

[19] As both honest nodes are randomly drawn this does not change the success

is forwarded to the oracle of the challenger (Exception1) as O to process it.[20] The challenger returns[21] $\tilde{O}$. 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. As soon as $\mathcal{A}$ instructs the challenge onion to arrive at the honest reply-receiver, the attack uses the oracle (Exception2) to retrieve the output of the honest receiver for $\mathcal{D}$. In case that the oracle gives no output (because the payload was modified and this resulted in an error while processing at the receiver), the attack gives this information to $\mathcal{D}$. 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).[22]

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

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

Hybrid $\mathcal{H}_2^{\leftarrow}$. In this hybrid, for the first backward communication for which in the adversarial processing no recognition falsifying modification occurred and other modification did not lead to failed processing[23] 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. More precisely, this machine acts like $\mathcal{H}_1^{*\leftarrow}$ except for two replacements. First, the corresponding onion of the forward path is replaced by O^c with a shortened backward path: $O^c \leftarrow \mathsf{FormOnion}(m, \mathcal{P}_{\rightarrow}, \tilde{\mathcal{P}}_{\leftarrow}, (pk)_{\mathcal{P}_{\rightarrow}}, (pk)_{\tilde{\mathcal{P}}_{\leftarrow}}; \tilde{\mathcal{R}})$ with forward path $\mathcal{P}_{\rightarrow} = (P^{j\rightarrow}, \dots, P^n)$ and backward path $\tilde{\mathcal{P}}_{\leftarrow} = (P^1_{\leftarrow}, \dots, P^j_{\leftarrow})$ starting at the last honest node $P^{j\rightarrow}$ on the forward path.

Second, the processing of $O^j_{\leftarrow}$; i.e. the consecutive onion layers $O^{j+1}_{\leftarrow}, \dots, O^{j'}_{\leftarrow}$ from a backward communication of an honest (forward) sender, starting at the second last honest node $P^j_{\leftarrow}$ to the next following honest relay $P^{j'}_{\leftarrow}$ (on the backward path), are replaced with $\bar{O}^1, \dots, \bar{O}^{j'-j}$. Thereby for an honestly chosen $\mathcal{R}$; $\bar{O}^1 \leftarrow \mathsf{FormOnion}(m_{rdm}, \mathcal{P}, (), (pk)_{\mathcal{P}_{rdm}}, (); \mathcal{R})$ where m_{rdm} is a random message, $\mathcal{P} = (P^j_{\leftarrow}, \dots, P^{j'}_{\leftarrow})$ the path from $P^j_{\leftarrow}$ to $P^{j'}_{\leftarrow}$.

We now need to ensure that the last part of the onion path (which was already replaced in $\mathcal{H}_1^{*\leftarrow}$) gets used as the result of receiving this onion at $P^{j'}_{\leftarrow}$. Therefore this Hybrid uses ExtractPayload to retrieve the message contained in the received onion and otherwise constructs a replacement (including stored replacement information) for the $\bar{O}$-list just as in $\mathcal{H}_1^{*\leftarrow}$. Let the next relay and the resulting replacement onion for the next path part be $(\tilde{P}^{j'+1}_{\leftarrow}, \tilde{O}^k)$. The hybrid stores $(info, P^{j'}_{\leftarrow}, (\tilde{O}^k, \tilde{P}^{j'+1}_{\leftarrow}))$, with $info$ according to the currently added replacement part, to the $\bar{O}$-list. Notice that the path parts are known to the hybrid as this is an honest request sender's communication. The honest parties inform the environment as before.

$\mathcal{H}_1^{*\leftarrow} \approx_I \mathcal{H}_2^{\leftarrow}$. $\mathcal{H}_2^{\leftarrow}$ replaces for one backward communication, the last subpath between two consecutive honest nodes before an honest (forward) sender. The output to $\mathcal{A}$ includes the later (as in $\mathcal{H}_1^{*\leftarrow}$) replaced onion layers $\bar{O}^{later}$ after the second honest relay (these layers are identically generated in $\mathcal{H}_1^{*\leftarrow}$ and $\mathcal{H}_2^{\leftarrow}$) 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 $\mathcal{H}_2^{\leftarrow}$ (outputted by $\mathcal{H}_1^{*\leftarrow}$) or in case of the communication considered by $\mathcal{H}_2^{\leftarrow}$, the newly drawn random replacement (generated by $\mathcal{H}_2^{\leftarrow}$); and the processing before the first honest relay $P^j_{\leftarrow}$.

The onions $\bar{O}^{later}$ are chosen independently at random by $\mathcal{H}_1^{*\leftarrow}$ such that they embed the original path between the second considered honest relay and the honest (forward) sender, but contain a random message. As they are used as processing of the original onion layers before $P^{j'}_{\leftarrow}$ (there was no recognition falsifying modification for this communication) and include a random message, onions $\bar{O}^{later}$ are not connected to onions before $P^{j'}_{\leftarrow}$ and hence can simply be generated for any distinguisher based on the knowledge and oracles an attacker on $LU^+_{\leftarrow}$ has access to.

Thus, all that is left are the original/replaced onion layer after the honest node and the original layers before. This is the same output as in $\mathcal{H}_2^* \approx_I \mathcal{H}_1^{\leftarrow}$. Hence, the distinguisher between $\mathcal{H}_1^{*\leftarrow}$ and $\mathcal{H}_2^{\leftarrow}$ is similarly used to build an attack on $LU^+_{\leftarrow}$ except that the distinguisher does not get the output at the second honest relay, but the attack instead uses the information on the $\bar{O}$-list to continue the communication as is done in both, $\mathcal{H}_1^{*\leftarrow}$ and $\mathcal{H}_2^{\leftarrow}$.

Hybrid $\mathcal{H}_2^{<x\leftarrow}$. In this hybrid, for the first[24] $x-1$ honest subpaths on backward 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 node starting the selected subpath) did a recognition falsifying modification or a modification that results in a processing fail, the communication is skipped.

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

Hybrid $\mathcal{H}_3$**.** This machine acts the way that $\mathcal{S}$ acts in combination with $\mathcal{F}$. Note that $\mathcal{H}_2^{*\leftarrow}$ 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) $\mathcal{H}_2^{*\leftarrow}$ actually routes them through the real honest parties that do all the computation. $\mathcal{H}_3$, instead runs the way that $\mathcal{F}$ and $\mathcal{S}$ operate: there are no real honest parties, and the ideal honest parties do not do any crypto work. (b) $\mathcal{H}_2^{*\leftarrow}$ gets inputs directly from the

[20] In case of the (honest) forward receiver being $P^j_{\leftarrow}$, there is no such processing, but her answer $\tilde{O}$ is queried from the challenger by the attacker to simulate the honest communications that are happening.

[21] Unless the onion was no reply to the onion in question or processing failed, in which case we need to do nothing for $\mathcal{D}$

[22] This includes that duplicates are dropped and onions are processed before they are replied (as assumed in Section 4.1).

[23] We treat modifying adversaries on other parts of the onion later in a generic way.

[24] counted similarly to the forward path, but now starting from the backward receiver until the backward sender; again for the first backward communication until the last.

environment and gives output to it. In $\mathcal{H}_3$ the environment instead gives inputs to $\mathcal{F}$ and $\mathcal{S}$ gets the needed information (i.e. parts of path and the included message, if the receiver is corrupted) from outputs of $\mathcal{F}$ as the ideal world adversary. $\mathcal{F}$ gives the outputs to the environment as needed.

$\mathcal{H}_2^{*\leftarrow} \approx_I \mathcal{H}_3$. 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 node is done processing, it needs the path from it to the next honest node 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 $\mathcal{F}$.

Further, in $\mathcal{H}_2^{*\leftarrow}$, the environment is notified by the hybrid on behalf of honest nodes when they receive an onion *temp* as relays or when they receive a message and if the message is a reply to an earlier request also to which request the reply belongs. The same is done in the ideal functionality. Notice that the simulator ensures that every communication is simulated in $\mathcal{F}$ such that those notifications arrive at the environment without any difference. The simulator ensures that communications of honest senders are continued at the appropriate times in the ideal functionality, as well as that the replies are given belonging to the corresponding requests in the ideal functionality.

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 $\mathcal{H}_2^{*\leftarrow}$ and $\mathcal{H}_3$ (i.e. $\mathcal{S}+\mathcal{F}$) 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 1.2)]): All parts of the path are replaced with randomly drawn onion layers $\bar{O}^i$. The way those layers are chosen is identical for $\mathcal{H}_2^{*\leftarrow}$ and $\mathcal{H}_3$ (i.e. $\mathcal{S}+\mathcal{F}$). 1.2) Some recognition falsifying modification of the onion or a drop or insert happens: As soon as a recognition falsifying modification happens, both $\mathcal{H}_2^{*\leftarrow}$ and $\mathcal{H}_3$ continue to use the bit-identical onion for the further processing except that 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 $\mathcal{H}_3$. As we have shown indistinguishability in every step, we have indistinguishability in their views.

C FULL SECURITY PROOF OF EROR

This section contains the security proof for EROR. In Appendix C.1, we first fix some notational conventions. In Appendix C.2, we prove $LU^+_\rightarrow$ for honest relay at point $j < n_\rightarrow$, and in Appendix C.2 we show the case $j = n_\rightarrow$. Then, In Appendix C.4, we prove $LU^+_\leftarrow$ for honest relay at point $0 < j$, and in Appendix C.5 we show the case $j = 0$.

C.1 Notation and definitions of auxiliary algorithms

During FormOnion, ephemeral keys $K^i_\rightarrow, K^j_\leftarrow$ are generated. Moreover, in Onionize, intermediate results $U^i = (B^i_1, \ldots, B^i_N, fwd^i)$ are produced; we also write O^i if bwd^i is part of the onion in consideration, recall that bwd^i is only end-to-end authenticated, which will require special handling. We write U^i (or $U^i_\rightarrow$, $U^j_\leftarrow$ if the direction of the corresponding Onionize is not clear from the context) for these intermediate onions, we write c^j for these ciphertexts containing $meta^j = (role^j, P^j)$ or $meta^j = (role^j, next^j)$, and so on. In our proofs, we will modify these intermediate results, and let the algorithms (usually FormOnion or Onionize) continue with the modified values. This allows us to succinctly describe the steps in the security reduction.

We define RecognizeOnion and ExtractPayload needed in the security games via these intermediate results. That is,

- RecognizeOnion$((i, dir), \overline{O}, m, (P^i_\rightarrow)^{n_\rightarrow}_{i=1}, (P^i_\leftarrow)^{n_\leftarrow}_{i=1}, \mathcal{R})$ where $dir \in \{\rightarrow, \leftarrow\}$. Use $\mathcal{R}$ to recompute the onion. Then compare c^i_{dir} with $\overline{c}$, where $(\overline{\tau}, \overline{c}) = \overline{B}_1 = \overline{O}.hdr[1]$ and c^i_{dir} is the PKE ciphertext in U^i_{dir}. Output 1 if they are equal and $\overline{\tau}$ is valid, i.e., MAC.Verify$K^i_{\mathsf{MAC}}(\overline{\tau}, (\overline{O}.hdr, \overline{O}.fwd)) = 1$. Else output 0.
- ExtractPayload$((i, \rightarrow), O, m, (P^i_\rightarrow)^{n_\rightarrow}_{i=1}, (P^i_\leftarrow)^{n_\leftarrow}_{i=1}, \mathcal{R})$: Use $\mathcal{R}$ to recompute the ephemeral keys $K^j_\leftarrow$ and Unwrap the payload until the backward receiver, (i.e., original sender) is reached, i.e. until $j = n_\leftarrow$, then run the final ProcOnion processing (of the sender), which yields the backward message.

C.2 Security proof of $LU^+_\rightarrow$, $j < n$

We use the notation from Appendix C.1 in this section. Since we only need the forward direction, i.e. $K_\rightarrow$, $O_\rightarrow$, etc., we sometimes drop the $\rightarrow$ unless statements are ambiguous.

We will argue security via game hops. To describe our changes, we describe changes applied to the partial onions $U^i_\rightarrow$ as defined within the challenge execution[25] of FormOnion and processing of the challenge query $O^j_\rightarrow$. The following (partial) onions are of primary interest, which we will consider (and modify) them in the security proof:

- $O^1_\rightarrow$, i.e. the actual challenge onion.
- $U^j_\rightarrow$ and $U^{j+1}_\rightarrow$, the intermediate partial onions during computation of $O^1_\rightarrow$ in Onionize. Most modifications to $O^1_\rightarrow$ are done by modifying these intermediate values.
- $O^j_\rightarrow$, the recognized challenge query (onion) which $\mathcal{A}$ sends to Proc$(P_H, O^j_\rightarrow)$. Due to integrity protection, a successfully processed query O^j will agree with $U^j_\rightarrow$ (except for the backward payload).
- $\overline{O}^{j+1}_\rightarrow$, a newly variable introduced in the security proof to capture the modified output of Proc(P_H, O^j).

[25]We note, that these intermediate results correspond to FormOnion$(j > 1, \ldots)$ in [20]. However, we did not introduce FormOnion$(j, \ldots)$ for $j > 1$, as this is not required for our security experiments anymore.

Through game hops, we gradually modify the computation of $\overline{O}^{j+1}$, which initially is defined via honest processing with $\mathsf{ProcOnion}$, until it is a completely independent, freshly computed onion. We also modify the computation of O^1, the challenge onion, until eventually the $LU^+_{\rightarrow}$ game changes from $b = 0$ to $b = 1$. Most of the proof steps relate only to header and forward payload, i.e., to U^1 (resp. U^j) and $\overline{U}^{j+1}$, as the backward payload is garbage (and thus completely malleable by the adversary).

We give an overview of the security reduction (for $j < n$) in Table 2. There and in the proof below, unless noted otherwise, all changes are in the Onionize call of the challenge onion; non-challenge onions are always processed honestly. Moreover, missing direction subscripts/superscripts are forward ($\rightarrow$). E.g., by n we denote $n_{\rightarrow}$ (unless noted otherwise).

Game G_0. This game is the $LU^+_{\rightarrow}$ game with challenge bit 0.

Game $\mathsf{G}_{1.1}$. In the challenge execution of FormOnion make following changes: Replace $c^{n_{\leftarrow}}_{\leftarrow} \leftarrow \mathsf{PKE.Enc}_{\mathsf{PKI}[P^{n_{\leftarrow}}_{\leftarrow}]}(meta^{n_{\leftarrow}}_{\leftarrow})$ (in line 36) of $O_{\leftarrow} = \mathsf{Onionize}(\dots)$ by $c^{n_{\leftarrow}}_{\leftarrow} \leftarrow \mathsf{PKE.Enc}_{\mathsf{PKI}[P^{n_{\leftarrow}}_{\leftarrow}]}(\widetilde{meta})$, where $\widetilde{meta} = (\mathtt{SNDR}, K', (n_{\rightarrow}, n_{\leftarrow}))$ for $K' \leftarrow_{\mathrm{R}} \mathcal{K}$. When processing the (recognized) challenge onion, when recognizing $c^{n_{\leftarrow}}_{\leftarrow}$, instead of decrypting (in Unwrap) use the old $meta^{n_{\leftarrow}}_{\leftarrow}$, i.e. $meta^{n_{\leftarrow}}_{\leftarrow} = (\mathtt{SNDR}, K^{n_{\leftarrow}}_{\leftarrow}, (n_{\rightarrow}, n_{\leftarrow}))$.

This change removes the onion master secret key $omsk = K^{n_{\leftarrow}}_{\leftarrow}$ from $c^{n_{\leftarrow}}_{\leftarrow}$. Observe that PKE.Enc and PKE.Dec are always used black-box in Onionize and Unwrap. Thus, by a straightforward reduction, we obtain

$$\Pr[\mathsf{G}_{1.1}] - \Pr[\mathsf{G}_{1.2}] \leq \mathsf{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{PKE},\mathcal{B}_{1.2}}.$$

Game $\mathsf{G}_{1.2}$. Replace $c^j_{\rightarrow} \leftarrow \mathsf{PKE.Enc}_{\mathsf{PKI}[P^j_{\rightarrow}]}(meta^j_{\rightarrow})$ (in line 35) of $O_{\rightarrow} = \mathsf{Onionize}(\dots)$ by $c^j_{\rightarrow} \leftarrow \mathsf{PKE.Enc}_{\mathsf{PKI}[P^j_{\rightarrow}]}(\widetilde{meta})$, where $\widetilde{meta} = (\mathtt{RCVR}, K'', \bot)$ for $K'' \leftarrow_{\mathrm{R}} \mathcal{K}$.[26] When processing the (recognized) challenge onion, when recognizing $c^j_{\leftarrow}$, instead of decrypting $c^j_{\leftarrow}$ (in Unwrap) use the old $meta^j_{\rightarrow}$, i.e, $meta^j_{\rightarrow} = (\mathtt{HOP}, K^j_{\rightarrow}, P^{j+1})$.

This change removes the derived key $K^j_{\rightarrow}$ from $c^j_{\rightarrow}$. Completely analogous to game $\mathsf{G}_{1.1}$, we get

$$\Pr[\mathsf{G}_{1.1}] - \Pr[\mathsf{G}_{1.2}] \leq \mathsf{Adv}^{\mathsf{ind\text{-}cca}}_{\mathsf{PKE},\mathcal{B}_{1.2}}.$$

Game G_2. In the challenge execution of FormOnion, replace all keys $(K^i_{\rightarrow})^{n_{\rightarrow}}_{i=1}$ and $(K^i_{\leftarrow})^{n_{\leftarrow}-1}_{i=1}$ which are derived from $omsk = K^{n_{\leftarrow}}_{\leftarrow}$ (in FormOnion) by truly random keys.

The use of KDF with $omsk$ (in FormOnion, line 64) is black-box, and $omsk$ is chosen uniformly. Thus, by a straightforward reduction, we obtain

$$\Pr[\mathsf{G}_{1.2}] - \Pr[\mathsf{G}_2] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{B}_2}.$$

Game G_3. In the challenge execution of $O_{\rightarrow} = \mathsf{Onionize}(\dots)$, replace the derived keys $K^j_{\mathsf{SKE}}, K^j_{\mathsf{MAC}}, K^j_{\mathsf{PRF}}$ (in Onionize, Wrap and AE) by truly random choices, i.e. replace $\mathsf{KDF}(K^j_{\rightarrow}, _)$ by a truly random function.

All uses of KDF with $K^j_{\rightarrow}$ are black-box, and $K^j_{\rightarrow}$ is chosen uniformly due to $\mathsf{G}_{2.1}$. Thus, by a straightforward reduction, we obtain

$$\Pr[\mathsf{G}_2] - \Pr[\mathsf{G}_3] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{B}_3}.$$

Game G_4. We strengthen the RecognizeOnion check in step 7 as follows: If RecognizeOnion recognizes a query O, but $O.hdr \neq U^j.hdr$ or $O.fwd \neq U^j.fwd$, then output $\mathtt{FAIL}$ instead of processing the onion. In other words, if the ciphertext c^j is reused in a query O, then O must agree with U^j, except for backward payload.

Note that MAC.Sign and MAC.Verify are used only blackbox, and the key K_{MAC} is uniformly due to G_3. Thus, by a straightforward reduction, we obtain

$$\Pr[\mathsf{G}_3] - \Pr[\mathsf{G}_4] \leq \mathsf{Adv}^{\mathsf{suf\text{-}cma}}_{\mathsf{MAC},\mathcal{B}_4}.$$

Game G_5. We introduce $\overline{O}^{j+1}$ as a separate variable, which is now the output $\mathcal{A}$ receives after the challenge Proc request. That is, in step 7, when query O^j is recognized as challenge, we introduce a new onion, denoted $\overline{O}^{j+1} := (hdr', fwd', bwd')$, and defined as

- $hdr' = U^{j+1}.hdr$
- $fwd' = U^{j+1}.fwd$
- $bwd' = \mathsf{AE.Unwrap}(K^j_{\rightarrow}, O^j.bwd)$ (but use the truly random $K^j_{\mathsf{MAC}}, K^j_{\mathsf{PRF}}$ in AE instead of deriving them from KDF).

We note that the recognized challenge query O^j and the intermediate partial onion U^j during FormOnion (of the challenge onion) coincide (by game 4).[27] Hence, processing of bwd' is unchanged, whereas hdr' and fwd' simply reuses the intermediate partial onion U^{j+1}, which is the output of processing U^j. The change in this game is merely conceptual, thus

$$\Pr[\mathsf{G}_4] = \Pr[\mathsf{G}_5].$$

In the following games, games $\mathsf{G}_{6.1}$ to $\mathsf{G}_{6.4}$, we disconnect the computation of $\overline{O}^{j+1}$ (resp. U^{j+1}) and the challenge onion O^1 (resp. U^j).

Game $\mathsf{G}_{6.1}$. In header of U^{j+1}, we replace blocks $B^{j+1}_{N-j+1}, \dots, B^{j+1}_N$ by randomness $R^{j+1}_{N-j+1}, \dots, R^{j+1}_N$. This makes sure that the *header* of $\overline{O}^{j+1}$ (resp. U^{j+1}) is independent from previous keys $(K^i_{\mathsf{SKE}})^j_{i=1}$, whose dependency is only in these garbage terms. A sketch of this processing is provided in Fig. 5.

Note that Enc/Dec are always used blackbox, and the key K^j_{SKE} is truly random since game G_3. For the reduction to DLR\$-CPA, we generate the onion $O^1_{\rightarrow}$ by making oracle calls for the encryptions (resp. decryption) which are under K^j_{SKE}. More concretely, during Onionize, when computing the garbage terms in lines 37 to 43, and when Wrapping U^{j+1} to get U^j in line 57, the reduction to DLR\$-CPA uses the challenge oracle to encrypt or decrypt under key $K^j_{\rightarrow}$. Moreover, the plaintext-ciphertext pairs are cached (and used to answer encryption/decryption queries).[28] To answer Proc calls of the adversary, we observe that thanks to the checks introduced in game G_3, if the onion $O = ((B_1, \dots, B_N), fwd, bwd)$

[26] We set the next relay to $\bot$ already, as we eventually have make O^1 an onion without reply information in game $LU^+_{\rightarrow}$ with $b = 1$.

[27] As the backward payload $O^j.bwd$ is malleable, it need not coincide with the honest processing of the initial backward payload $O^1.bwd$.

[28] For caching, we implicitly use that SKE is a (perfectly correct) permutation cipher.

Table 2: Overview of proof for $LU^+_{\rightarrow}$, $j < n+1$. Unless noted otherwise, all changes are in the Onionize call of the challenge. Missing direction subscripts are forward ($\rightarrow$).

Game	Description/Changes	Reduction
(0)	The $LU^+_{\rightarrow}$ game with challenge bit chosen as 0.	
(1)	Get rid of challenge keys in $c^j_{\rightarrow}$ and $c^{n_\leftarrow}_{\leftarrow}$	
(1.1)	Replace the encryption $c^{n_\leftarrow}_{\leftarrow}$ of $meta^{n_\leftarrow}_{\leftarrow} = (\texttt{SNDR}, K^{n_\leftarrow}_{\leftarrow}, (n_\rightarrow, n_\leftarrow))$ by an encryption of $(\texttt{SNDR}, K', (n_\rightarrow, n_\leftarrow))$, for fresh $K' \leftarrow_R \mathcal{K}$. (Still use the old $meta^{n_\leftarrow}_{\leftarrow}$ to process the challenge in ProcOnion.)	IND-CCA
(1.2)	Replace the encryption $c^j_{\rightarrow}$ of $meta^j_{\rightarrow} = (\texttt{HOP}, K^j_{\rightarrow}, P^{j+1})$ by an encryption of $(\texttt{RCVR}, K'', \perp)$, for fresh $K'' \leftarrow_R \mathcal{K}$. (Still use the old $meta^j_{\rightarrow}$ to process.)	IND-CCA
(2)	Replace all keys $(K^i_{\rightarrow})^{n_\rightarrow}_{i=1}$ and $(K^i_{\leftarrow})^{n_\leftarrow -1}_{i=1}$ in challenge FormOnion by truly random keys (instead of deriving them from $omsk = K^{n_\leftarrow}_{\leftarrow}$).	PRF
(3)	Instead of deriving $K^j_{\mathsf{MAC}}, K^j_{\mathsf{SKE}}, K^j_{\mathsf{PRF}}$ from $K^j_{\rightarrow}$, use truly random keys when processing challenge onion.	PRF
(4)	In step 7 of $LU^+_{\rightarrow}$, if RecognizeOnion recognizes query O, but $O.hdr \neq U^j.hdr$ or $O.fwd \neq U^j.fwd$, then output $\texttt{FAIL}$ instead of processing the onion. (Does not change the computation of onions.)	SUF-CMA
(5)	In step 7 of $LU^+_{\rightarrow}$, when query O^j is recognized as challenge, output $\overline{O}^{j+1} := (hdr, fwd, bwd)$, where instead of processing O^j, $\overline{U}^{j+1}$ is set to U^{j+1}, and $\overline{O}^{j+1}.bwd = bwd^{j+1}$ is processed honestly (via AE.Unwrap).	—
(6)	"Disconnect" O^j and $\overline{O}^{j+1}$ by separating usage of keys	
(6.1)	In U^{j+1} (hence $\overline{O}^{j+1}$), remove use of $(K^i_{\mathsf{SKE}})^j_{i=1}$ (in garbage added by $(P^i_{\rightarrow})^j_{i=1}$), by replacing blocks $B^{j+1}_{N-j+1}, \ldots, B^{j+1}_N$ by randomness $R^{j+1}_{N-j+1}, \ldots, R^{j+1}_N$.	DLR\$-CPA
(6.2)	In computation of payload of $\overline{O}^{j+1}$, remove use of K^j_{SKE} and K^j_{SKE} by replacing the backward payload $\overline{O}^{j+1}.bwd$ by randomness. (Still choose $O^1.bwd$ uniformly and independently.)	DLR\$-CPA + PRF
(6.3)	In U^j, remove the use of keys $(K^i_{\mathsf{SKE}}, K^i_{\mathsf{MAC}}, K^i_{\mathsf{PRF}})^{n_\rightarrow}_{i=j+1}$ in header blocks $B^j_2, \ldots, B^j_{N-j+1}$ by replacing these blocks with randomness $R^j_2, \ldots, R^j_{N-j+1}$	DLR\$-CPA
(6.4)	In U^j, remove the use of $(K^i_{\mathsf{SKE}})^{n_\rightarrow}_{i=j+1}$ in fwd^j, by replacing the value fwd^j with randomness during wrapping in Onionize.	DLR\$-CPA
(7)	Compute $\overline{O}^{j+1}$ as $\mathsf{FormOnion}(m, (P^i_{\rightarrow})^{n_\rightarrow}_{i=j+1}, (P^i_{\leftarrow})^{n_\leftarrow}_{i=1}; \mathcal{R})$	
(7.1)	In the header of $\overline{O}^{j+1}$, replace (random) blocks $B^{j+1}_{N-j+1}, \ldots, B^{j+1}_N$ by wrapped fresh randomness, i.e., $(\mathsf{Enc}_{K^{j+1}}(\ldots \mathsf{Enc}_{K^{n_\rightarrow}}(R^{n_\rightarrow}_i) \ldots))^{N-(n_\rightarrow - j)+1}_{i=N-n_\rightarrow - j+1}$ for $R^{n_\rightarrow}_i \leftarrow_R \$$.	SKE is permutation
(7.2)	Derive keys $K^{j+1}_{\rightarrow}, \ldots, K^{n_\rightarrow}_{\rightarrow}$ used in $\overline{O}^{j+1}$ from $\overline{omsk}$	PRF
(7.3)	Encrypt $(\texttt{SNDR}, \overline{omsk}, (n_\rightarrow - j, n_\leftarrow))$ in $\overline{O}.c^{n_\leftarrow}_{\leftarrow}$. (If $n_\leftarrow = 0$, this does nothing.)	IND-CCA
(7.4)	Use $\mathsf{FormOnion}(m, (P^i_{\rightarrow})^{n_\rightarrow}_{i=j+1}, (P^i_{\leftarrow})^{n_\leftarrow}_{i=1}; \mathcal{R})$ to compute $\overline{O}^{j+1}$	—
(8)	Compute O^1 as $\mathsf{FormOnion}(m', (P^i_{\rightarrow})^j_{i=1}, (); \mathcal{R}')$	
(8.1)	Make $P^j_{\rightarrow}$ the receiver of a proper forward payload, by replacing fwd^j in O^j by payload $\mathsf{Enc}_{K^j_{\mathsf{SKE}}}(\texttt{FWD}, (O^{\leftarrow}.header = \perp, m'))$, where $m' \leftarrow_R \mathcal{M}$	DLR\$-CPA
(8.2)	Revert the change of Game (4) (but still return $\overline{O}^{j+1}$ if challenge onion is recognized)	SUF-CMA
(8.3)	Revert the change in Game (3) by deriving $K^j_{\mathsf{MAC}}, K^j_{\mathsf{SKE}}, K^j_{\mathsf{PRF}}$ from $K^j_{\rightarrow}$ again	PRF
(8.4)	Derive keys $(K^i_{\rightarrow})^j_{i=1}$ used in O^1 from freshly chosen $omsk'$	PRF
(8.5)	In U^j, make block B^j_1 a real (receiver) block by encrypting $(\texttt{RCVR}, K^j_{\rightarrow}, \perp)$ in c^j.	IND-CCA
(8.6)	Use $\mathsf{FormOnion}(m', (P^i_{\rightarrow})^j_{i=1}, (); \mathcal{R}')$ for freshly chosen $\mathcal{R}'$ to compute O^1	—
(9)	The $LU^+_{\rightarrow}$ game with challenge bit chosen as 1.	

is recognized, i.e. if $B_1 = (\tau, c)$ with $c = c^j$, then Proc only processes if $((B_1, \dots, B_N), fwd) = ((B_1^j, \dots, B_N^j), fwd^j)$ otherwise it outputs FAIL. In particular, the header is identical to the respective intermediate challenge onion header generated via the (modified) FormOnion process. Thus, no encryption or decryption queries are required to process a recognized onion query. Hence, there is no nonce-reuse, and the reduction yields a valid adversary. By a hybrid argument, we obtain

$$\Pr[\mathsf{G}_5] - \Pr[\mathsf{G}_{6.1}] \leq \mathsf{Adv}^{\text{dbl\$-cpa}}_{\mathsf{SKE},\mathcal{B}_{6.1}}.$$

Game $\mathsf{G}_{6.2}$. Instead of $\overline{O}^{j+1}.bwd = \mathsf{AE.Unwrap}(K^j_{\mathsf{MAC}}, K^j_{\mathsf{PRF}}, O^j.bwd)$, we compute $\overline{O}^{j+1}.bwd$ for recognized challenge query O^j honestly, choose $\overline{O}^{j+1}.bwd$ uniformly randomly. This makes sure that from now on, $\overline{O}^{j+1}$ *is independent from* $K^j_{\rightarrow}$, and hence all previous keys $(K^i_{\rightarrow})^j_{i=1}$, whose dependency in U^{j+1} (and hence $\overline{O}^{j+1}$) is only in these garbage terms. (Observe that $\overline{U}^{j+1}.fwd = U^{j+1}.fwd$ is independent of $(K^i_{\rightarrow})^j_{i=1}$ by its very definition (as Onionize wrapping goes from $n_{\rightarrow}$ down to $j+1$), and $U^{j+1}.hdr$ was treated in the previous step.)

By a reduction analogous to game $\mathsf{G}_{6.1}$ for *bwd.ctxt* and a PRF reduction for *bwd.mac*, we obtain

$$\Pr[\mathsf{G}_{6.1}] - \Pr[\mathsf{G}_{6.2}] \leq \mathsf{Adv}^{\text{dbl\$-cpa}}_{\mathsf{SKE},\mathcal{B}_{6.2}} + \mathsf{Adv}^{\text{prf}}_{\mathsf{SKE},\mathcal{B}'_{6.2}}.$$

Game $\mathsf{G}_{6.3}$. During computation U^1 from U^j in Onionize, replace blocks $B^j_2, \dots, B^j_{N-j+1}$ by randomness $R^j_2, \dots, R^j_{N-j+1}$. This makes sure that *the header* of U^j, hence the header of O^1, is independent from the keys $(K^i_{\rightarrow})^{n_{\rightarrow}}_{i=j+1}$, whose dependency in U^{j+1} (and hence $\overline{O}^{j+1}$) is only in these garbage terms.

The reduction is a hybrid argument, very similar to game $\mathsf{G}_{6.1}$. One irrelevant difference is, that instead of replacing decryption (of the constrained garbage terms) by randomness, now encryptions (of the unconstrained garbage terms) are replaced by randomness. Refer to Fig. 5 for a sketch of the situation. Overall, we obtain, completely analogous to $\mathsf{G}_{6.1}$, a reduction such that

$$\Pr[\mathsf{G}_{6.2}] - \Pr[\mathsf{G}_{6.3}] \leq \mathsf{Adv}^{\text{dbl\$-cpa}}_{\mathsf{SKE},\mathcal{B}_{6.3}}.$$

Game $\mathsf{G}_{6.4}$. During computation U^1 from U^j (in Onionize), replace fwd^j by true randomness. Now, header and payload the challenge onion O^1 are independent of the keys $(K^i_{\rightarrow})^{n_{\rightarrow}}_{i=j+1}$ as well as *omsk*.

The reduction is a direct reduction, completely analogous to the previous ones. Thus we find

$$\Pr[\mathsf{G}_{6.3}] - \Pr[\mathsf{G}_{6.4}] \leq \mathsf{Adv}^{\text{dbl\$-cpa}}_{\mathsf{SKE},\mathcal{B}_{6.4}}.$$

Game $\mathsf{G}_{7.1}$. In the header of $\overline{O}^{j+1}$, we replace the (random) blocks $B^{j+1}_{N-j+1}, \dots, B^{j+1}_N$ by encrypted fresh randomness, i.e., replace them by $(\mathsf{Enc}_{K^{j+1}}(\dots \mathsf{Enc}_{K^{n_{\rightarrow}}}(R^{n_{\rightarrow}}_i) \dots))^{N-(n_{\rightarrow}-j)+1}_{i=N-n_{\rightarrow}-j+1}$, for randomly chosen $R^{n_{\rightarrow}}_i$. Observe that this is as in FormOnion, with a path of length $n_{\rightarrow} - j$.

This change is merely conceptual, as SKE is a permutation, the distribution of $B^{j+1}_{N-j+1}, \dots, B^{j+1}_N$ in $\overline{O}^{j+1}_{\rightarrow}$ is still uniform. (Because a permutation of a uniformly random distribution is again uniformly random.) We have

$$\Pr[\mathsf{G}_{6.4}] = \Pr[\mathsf{G}_{7.1}].$$

Game $\mathsf{G}_{7.2}$. In this game, we derive the keys $K^{j+1}_{\rightarrow}, \dots, K^{n-j}_{\rightarrow}$ used in $\overline{O}^{j+1}$ from a uniformly random $\overline{omsk} \leftarrow \mathcal{K}$. That is, we undo the change of game G_3, but only for $\overline{O}^{j+1}$ and using an independent $\overline{omsk}$.

As in game G_3, this is a straightforward reduction to PRF security, and we find

$$\Pr[\mathsf{G}_{7.2}] - \Pr[\mathsf{G}_{7.2}] \leq \mathsf{Adv}^{\text{prf}}_{\mathsf{KDF},\mathcal{B}_{7.2}}.$$

Game $\mathsf{G}_{7.3}$. Modify the change from game G_1 to $c^{n_{\leftarrow}}_{\leftarrow}$ further by encrypting $meta^{bwd}_{\leftarrow} = (\mathtt{SNDR}, \overline{omsk}, (n_{\rightarrow} - j, n_{\leftarrow}))$, where $\overline{omsk}$ is again the onion master secret key. Additionally, the encrypted length of the forward path is reduced to $n_{\rightarrow} - j$. A completely analogous reduction as for game $\mathsf{G}_{1.1}$ shows

$$\Pr[\mathsf{G}_{7.2}] - \Pr[\mathsf{G}_{7.3}] \leq \mathsf{Adv}^{\text{ind-cca}}_{\mathsf{PKE},\mathcal{B}_{7.3}}.$$

Game $\mathsf{G}_{7.4}$. We compute onion $\overline{O}^{j+1}_{\rightarrow}$ by running $\mathsf{FormOnion}(m, (P^i_{\rightarrow})^{n_{\rightarrow}}_{i=j+1}, (P^i_{\leftarrow})^{n_{\leftarrow}}_{i=1}; \overline{\mathcal{R}})$ (with fresh randomness). This change is merely conceptual:

- By definition $\overline{O}^{j+1}.bwd$ is uniformly random.
- The changes made above ensure that $\overline{U}^{j+1}$ is computed via the modification to $\overline{U}^{j+1}$ where all garbage terms $B^{n_{\rightarrow}}_2, \dots, B^{N-j+1}_2$ are uniformly random, as in Onionize with forward path $(P^i_{\rightarrow})^{n_{\rightarrow}}_{i=j+1}$ as input.
- The forward payload $\overline{U}^{j+1}.fwd$ was never changed.
- The remaining computation is as in Onionize with forward path $(P^i_{\rightarrow})^{n_{\rightarrow}}_{i=j+1}$ as input.

We have yet to handle the backward onion contained in $\overline{O}^{j+1}.fwd$. This was almost unchanged in the sense that we only modified $c^{n_{\leftarrow}}_{bwd}$, and in game $\mathsf{G}_{7.3}$. There, we set it to the proper value to make $\overline{O} = \mathsf{FormOnion}((P^i_{\rightarrow})^{n_{\rightarrow}}_{i=j+1}, (P^i_{\leftarrow})^{n_{\leftarrow}}_{i=1}, m; \overline{\mathcal{R}})$ hold true. Also note, that after the changes in games $\mathsf{G}_{6.1}$ to $\mathsf{G}_{6.4}$, all terms in $\overline{O}^{j+1}$ and $O^{1]}$ independent.

We leave a more detailed verification of this to the reader. As the change was conceptual, we find

$$\Pr[\mathsf{G}_{7.3}] = \Pr[\mathsf{G}_{7.4}].$$

Game $\mathsf{G}_{8.1}$. We make P_j the receiver of a proper forward payload (namely, message m' in a non-repliable onion) by replacing fwd^j (which was random since $\mathsf{G}_{6.4}$) with $\mathsf{Enc}_{K^j_{\mathsf{SKE}}}(\mathtt{FWD}, (\perp, m'))$, where $O_{\leftarrow}.header = \perp$ since the (eventually) formed onion will be non-repliable.

Completely analogous to game $\mathsf{G}_{6.4}$, we exploit the blackbox use of encryption (resp. decryption) and the key K^j_{SKE} to reduce to DLR\$-CPA security of SKE. We find

$$\Pr[\mathsf{G}_{7.4}] - \Pr[\mathsf{G}_{8.1}] \leq \mathsf{Adv}^{\text{dbl\$-cpa}}_{\mathsf{SKE},\mathcal{B}_{8.1}}.$$

Game $\mathsf{G}_{8.2}$. We revert the change of game G_4, that is, we remove the additional check on when RecognizeOnion recognizes a challenge for an oracle call Proc. We still process a challenge query O^j as before, in particular, we return $\overline{O}^{j+1}$ when the challenge onion is recognized (by comparing it B_1^j and assuming processing of O^j does not fail).

Completely analogous to game G_4, this change reduces to SUF-CMA security of MAC. Thus we find

$$\Pr[\mathsf{G}_{8.1}] - \Pr[\mathsf{G}_{8.2}] \leq \mathsf{Adv}^{\text{suf-cma}}_{\mathsf{MAC},\mathcal{B}_{8.2}}.$$

Game $\mathsf{G}_{8.3}$. We revert the change of game G_3, that is, instead of truly random keys K^j_{MAC}, K^j_{SKE}, and K^j_{PRF}, we derive these keys from $K^j_{\rightarrow}$.

Completely analogous to game G_3, this change reduces to the PRF security of KDF, and we find

$$\Pr[\mathsf{G}_{8.2}] - \Pr[\mathsf{G}_{8.3}] \leq \mathsf{Adv}^{\text{prf}}_{\mathsf{KDF},\mathcal{B}_{8.3}}.$$

Game $\mathsf{G}_{8.4}$. We partially revert the changes in game G_2 as follows: We derive the keys $K^1_{\rightarrow}, \ldots, K^j_{\rightarrow}$ used in O^1 from freshly chosen $omsk'$ (instead of uniformly at random, as we do since game G_2).

Completely analogous to game G_1, this reduces to PRF security of KDF and we obtain

$$\Pr[\mathsf{G}_{8.3}] - \Pr[\mathsf{G}_{8.4}] \leq \mathsf{Adv}^{\text{prf}}_{\mathsf{KDF},\mathcal{B}_{8.4}}.$$

Game $\mathsf{G}_{8.5}$. We adapt the changes in game G_2 as follows: We modify the message in the ciphertext c^j again (i.e. the ciphertext in block B_1^j of U^j in Onionize), from $meta^j = (\texttt{RCVR}, K'', \perp)$ to $meta^j = (\texttt{RCVR}, K^j_{\rightarrow}, \perp)$. This makes P_j the receiver (of an in game $\mathsf{G}_{8.1}$ suitably modified forward).

Completely analogous to game G_2, this reduces to IND-CCA security of PKE, we obtain

$$\Pr[\mathsf{G}_{8.4}] - \Pr[\mathsf{G}_{8.5}] \leq \mathsf{Adv}^{\text{ind-cca}}_{\mathsf{PKE},\mathcal{B}_{8.5}}.$$

Game $\mathsf{G}_{8.6}$. We compute O^1 as $\mathsf{FormOnion}(m', (P^i_{\rightarrow})^j_{i=1}, (); \mathcal{R}')$. Observe that:

- Garbage blocks $B_2^j, \ldots, B^j_{N-j+1}$ are uniformly random (due to game $\mathsf{G}_{6.3}$).
- $O^j.fwd$ is set as in Onionize for input $(P^i_{\rightarrow})^j_{i=1}$.
- $O^1.bwd$ is uniformly random (independent of $\overline{O}^{j+1}$).
- c^j is set as for a non-repliable onion.

Overall it is easy to see that O^1 can now equivalently be computed as $O^1 = \mathsf{FormOnion}(m', (P^i_{\rightarrow})^j_{i=1}, (); \mathcal{R}')$. Consequently,

$$\Pr[\mathsf{G}_{8.5}] = \Pr[\mathsf{G}_{8.6}].$$

It is worth noting, that in this case the onion master secret key $omsk'$ is never encrypted under PKE, because there is no backward path, and thus $O_{\leftarrow} = \perp$ is encrypted in the forward payload.

Game G_9. This is the $LU^+_{\rightarrow}$ game, with challenge bit chosen as 1. Again, this change is merely conceptual. Indeed we have changed the computation of O^1 and the output $\overline{O}^{j+1}$ of Proc for the challenges onion in steps (7) and (8), i.e. games $\mathsf{G}_{7.3}$ and $\mathsf{G}_{8.6}$, to independent executions of FormOnion with suitably truncated paths and payload, such that indeed, $\mathsf{G}_{8.6}$ is (up to syntactical differences) identical to G_9, i.e. the $LU^+_{\rightarrow}$ game with challenge bit 1. Consequently, $\Pr[\mathsf{G}_{8.6}] = \Pr[\mathsf{G}_9]$.

C.3 Security proof of $LU^+_{\rightarrow}$, $j = n$

If $j = n$, i.e., if the receiver is honest, we have following changes to the $LU^+_{\rightarrow}$ game (compared to $j < n$):

- Instead of $\overline{O}^{j+1}$, $\mathcal{A}$ receives $(m, \perp)$ from the game (via the oracle $\mathsf{Proc}(P_H, O)$).
- Additionally, $\mathcal{A}$ has access to $\mathsf{Reply}(P_H, O, m^{\leftarrow})$, through which it receives
  - $P^1_{\rightarrow}$, the next relay;
  - $O^1_{\leftarrow}.hdr$ the backward onion header;
  - $O^1_{\leftarrow}.fwd = \mathsf{PRG}(\mathsf{KDF}(K^{n_{\rightarrow}}_{\rightarrow}, \texttt{PRG}))$, the pseudo-random forward payload of the reply;
  - $O^1_{\leftarrow}.bwd = \mathsf{AE.Enc}K^{n_{\rightarrow}}_{\rightarrow}(m^{\leftarrow})$, the E2E encrypted backward payload $\overline{m}$.

By interpreting the forward layer $n_{\leftarrow} + 1$ as backward layer 1, e.g., setting $K^{n_{\rightarrow}+1}_{\rightarrow} = K^1_{\leftarrow}$, $U^{n_{\rightarrow}+1}_{\rightarrow} = U^1_{\leftarrow}$, etc., the security proof for $LU^+_{\rightarrow}$ with $j = n_{\rightarrow}$ turns out mostly analogous to the case $j < n_{\rightarrow}$, and in fact simpler to some extent. Unfortunately, notationally, this setting is less convenient since we have both forward onion and backward onion parts. We let $\overline{O}^1_{\leftarrow}$ denote a new variable for the backward onion, analogous to $\overline{O}^{j+1}$ in the case $j < n$. As in the proof for $LU^+_{\rightarrow}$ with $j < n$, we first set these variables to the honest computations (as in $b = 0$), and gradually modify them until we arrive at $b = 1$.

We briefly sketch the changes which are made to handle $j = n_{\rightarrow}$ instead of $j < n_{\rightarrow}$.

- Game 0 is $LU^+_{\rightarrow}$ for $j = n_{\rightarrow}$ with $b = 1$.
- Game 1.1: Replace $(\texttt{RCVR}, K^{n_{\rightarrow}}_{\rightarrow}, P^1_{\leftarrow})$ with $(\texttt{RCVR}, K'', \perp)$, instead of replacing $(\texttt{HOP}, K^j_{\rightarrow}, P^{j+1})$ with $(\texttt{RCVR}, K'', \perp)$.
- Game 2 is unchanged.
- Game 3 is unchanged, except that K_{PRG} is also replaced by a truly random key.
- Game 4 is unchanged. (The change to RecognizeOnion now applies to tests for Proc and Reply.)
- Game 5 now corresponds to two changes, one for Proc queries and one for Reply queries.
  - Proc: Here it corresponds to $\mathcal{A}$ learning $(m, \perp)$ (instead of O^{j+1}). The game is modified to *always output* m if the challenge was recognized and decryption succeeds. At this point, this is a conceptual change.
  - Reply: Here we replace the AE-encryption of $m^{\leftarrow}$ in $\overline{O}^1_{\leftarrow}.bwd$ with randomness.
- Game 6 only has one interesting step, namely Game 6.4 where the forward payload, which contains in particular $U^1_{\leftarrow}$, is randomized. With only this step, $O^1_{\leftarrow}$ is already fully separated from $O^1_{\rightarrow}$ (and m is hidden), since, unlike for $j < n$, $U^1_{\rightarrow}$ and $U^1_{\leftarrow}$ only "share" derived key material, which was replaced by truly random keys in prior steps.
- Game 7 computes $O^1_{\leftarrow}$ as $\mathsf{FormOnion}(m^{\leftarrow}, (P^i_{\leftarrow})^{n_{\leftarrow}}_{i=1}, (); \mathcal{R})$, where $m^{\leftarrow}$ is the adversarially chosen message. Since $U^1_{\leftarrow}$ is already independent of $U^1_{\rightarrow}$, the main change are:

- We need an extra step to replace the pseudorandom forward payload $U^1_{\leftarrow}.fwd = \mathsf{PRG}(K_{\mathsf{PRG}})$ with $\overline{U}^1_{\leftarrow}.fwd = \mathsf{SKE.Enc}_{K^1_{\leftarrow,\mathsf{SKE}}}((\bot, m^{\leftarrow}))$. (Because $\overline{O}_{\leftarrow}$ has to be a *non-repliable forward* onion containing $m^{\leftarrow}$.)
- Game 7.3: Change $\overline{O}_{\leftarrow}.c^{n_{\leftarrow}}_{\rightarrow}$ from encrypting ($\mathtt{SNDR}, K'$, $(n_{\rightarrow}, n_{\leftarrow})$) to encrypting ($\mathtt{RCVR}, K', \bot$).
- Game 7.4: Derive all keys for $\overline{O}^1_{\leftarrow}$ again (from some *omsk'*), instead of picking truly random keys. (Note that, according to FormOnion, now only forward keys are derived for $\overline{O}^1_{\leftarrow}$; there is no backward header included in $\overline{O}^1_{\leftarrow}$; the forward payload of $\overline{O}^1_{\leftarrow}$ encrypts $m^{\leftarrow}$; the backward payload is random (since Game 5).)

- Game 8 is unchanged. Now, we compute $O^1_{\rightarrow}$ as $\mathsf{FormOnion}(m', (P^i_{\rightarrow})^{n_{\rightarrow}}_{i=1}, (); \mathcal{R}')$, where m' is random.
- Game 9 is $LU^+_{\rightarrow}$ for $j = n_{\rightarrow}$ with $b = 1$.

The reductions for each of these steps are simple and analogous to those in Appendix C.2 and thus left to the reader.

C.4 Security proof of $LU^+_{\leftarrow}$, $j > 0$

The security proof for Strong Backward Layer-Unlinkability is very similar to the proof for Strong Forward Layer-Unlinkability. Thus, we fist sketch the similarities and differences. Then we describe in more detail how adaptations are made.

Basically, the proof of $LU^+_{\rightarrow}$ was concerned with "splitting" (the forward path of) an onion at an honest party P_H, so that the new onion ends at P_H (and contains a forward message), and P_H sends out a fresh onion which ends at the actual destination and contains the actual forward message. While $LU^+_{\leftarrow}$ is concerned with the backward path, first observe that the onion headers are constructed almost identically using Onionize, with only the minor difference in the role SNDR of the final receiver, a pseudorandom forward payload, and a AE-encrypted backward payload. As such, the steps we used to split the onion path in the header in the $LU^+_{\rightarrow}$ game apply almost verbatim, with $O_{\rightarrow}$ replaced by $O_{\leftarrow}$, in this case. Overall, the "onion-splitting" can simply be copied from our proof of $LU^+_{\rightarrow}$. At a few points, we have to take into account that the game $LU^+_{\leftarrow}$ is a bit different, e.g. there are two steps where RecognizeOnion is used, and we need to ensure that ExtractPayload can be computed throughout all steps. However, this will be easy to see. Moreover, we need to ensure integrity of the backward payload. For this, we rely on the PRF which is used as a MAC for AE and the masking of the MAC by xoring a PRF value derived from the ciphertext *bwd.ctxt*. This makes sure that, if the *bwd.ctxt* was modified, then the MAC is completely randomized, hence will be rejected with overwhelming probability at decryption.

Remark C.1. The definition of $\mathsf{ExtractPayload}((i, \leftarrow), O^i, m, (P^i_{\rightarrow})^{n_{\rightarrow}}_{i=1}, (P^i_{\leftarrow})^{n_{\leftarrow}}_{i=1}, \mathcal{R})$ given in Appendix C.1 is based on unwrapping the onion. There are two abort events: Firstly, whether processing aborts (because the MAC over onion header and forward payload fails to verify). Secondly, whether the MAC on the final AE decryption of the final backward payload fails. Observe that instead of repeatedly running ProcOnion until reaching $\overline{O}^n_{\leftarrow}$, i.e., completing the unwrapping, and then decrypting (via re-wrapping as in ProcOnion case SNDR), we could instead just use the decryption routine (i.e., re-wrap) directly. That is, ExtractPayload could just check the MAC tag on $\overline{O}^i_{\leftarrow}$, and if valid, (re-)wrap the $bwd^i := O^i.bwd$ to obtain bwd^1 (as it happens in ProcOnion loop in line 92). By perfect correctness of the scheme, this yields the same output.

The detailed game hops are presented in Table 3. Since most of the steps in Table 3 can be argued exactly as for $LU^+_{\rightarrow}$ in Table 2, we only focus on the interesting steps in more detail below.

Game G_4. Due to the difference in the games for $LU^+_{\rightarrow}$ and $LU^+_{\leftarrow}$, we need to adapt this step. In $LU^+_{\rightarrow}$, this step strengthens the check of RecognizeOnion to identify the challenge onion, by forcing both header and forward payload of the query O to be identical to $U^j_{\leftarrow}$. As we will use the same arguments to change onion computation as done in $LU^+_{\rightarrow}$, we also impose these changes on RecognizeOnion.

We note that to apply the reduction to SUF-CMA for exception (2), we additionally need to replace $K^{n_{\leftarrow}}_{\mathsf{MAC}} = \mathsf{KDF}(K^{n_{\leftarrow}}_{\leftarrow}, \mathtt{MAC})$ by a truly random key and undo that change again, via the usual game hops. Thus, we obtain

$$\Pr[\mathsf{G}_3 = 1] - \Pr[\mathsf{G}_4 = 1] \leq 2 \cdot \mathsf{Adv}^{\text{dbl\$-cpa}}_{\mathsf{MAC},\mathcal{B}_4} + 2 \cdot \mathsf{Adv}^{\text{prf}}_{\mathsf{KDF},\mathcal{B}'_4}.$$

Game $\mathsf{G}_{4'}$. In this game, we change the output in exception (2) of the $LU^+_{\leftarrow}$ experiment as follows: Suppose $O^j_{\leftarrow}$ is *the*[29] challenge query in exception (1). Let $m^* = \mathsf{ExtractPayload}((j, \leftarrow), O^j_{\leftarrow}, m, (P^i_{\rightarrow})^{n_{\rightarrow}}_{i=1}, (P^i_{\leftarrow})^{n_{\leftarrow}}_{i=1}, \mathcal{R})$ as computed in exception (1). If m^* is not defined (because exception (1) did not yet occur or the extracted payload was $\bot$), let $m^* = \bot$. When RecognizeOnion in exception (2) recognizes the query $O^n_{\leftarrow}$ as challenge, let $(m', \bot)$ be the output of the (final) ProcOnion call (i.e., the decryption).

- If $m' = \bot$, then output $(\bot, \bot)$ to the adversary.
- If $m' \neq \bot$, then output $(m^*, \bot)$ to the adversary. (If $m^* = \bot$, this becomes $(\bot, \bot)$.)

In other words, we always output $(m^*, \bot)$ to the adversary, unless the processing of $O^n_{\leftarrow}$ failed.

Indistinguishability of this change follows by reduction to the PRF-security, using that K_{PRF} is truly random due to game (3). To compute ProcOnion in exception (1), if $O^j_{\leftarrow}$ is recognized as the challenge onion (and not aborted with FAIL), use the PRF oracle to obtain $r = \mathsf{PRF}K_{\mathsf{PRF}}(O^j_{\leftarrow}.bwd.ctxt)$ and let $\overline{O}^{j+1}.bwd.mac = O^j.bwd.\tau \oplus r$. Note that, if onion $O^j_{\leftarrow}$ is recognized as a challenge, then header and forward payload of $O^j_{\leftarrow}$ agrees with $U^j_{\leftarrow}$, or further processing is aborted with FAIL (as ensured in game (4)). Hence, there is at most one call to ProcOnion which triggers processing exception (2). It will decrypt the query by unwrapping the queried challenge onion O^n. If decryption failed, i.e. ProcOnion outputs $(\bot, \bot)$, then this does not change the output to the adversary. So suppose decryption produced some $(m', \bot)$ with $m' \neq \bot$. The unwrapping of query O during ProcOnion yields some intermediate backward payload $\overline{bwd}^{j+1}$ (in pseudocode line 92). There are three (non-disjoint) cases:

- If $\overline{bwd}^{j+1} = \overline{O}^{j+1}_{\leftarrow}.bwd$, then $m' = m^*$. (Recall that, by Remark C.1, ExtractPayload can just (re)wrap O^i to obtain the message m^* (or $\bot$). In this case, this clearly means $m' = m^*$.)

[29] By the changes in game G_4, the challenge query header must be $U^j_{\leftarrow}.hdr$. As onions the same header are only processed once, we have a at most one challenge query $O^j_{\leftarrow}$

Table 3: Overview of proof for $LU^{+}_{\leftarrow}$, $j > 0$. Unless noted otherwise, all changes are in the Onionize call of the challenge. Missing direction subscripts are backward ($\leftarrow$). Important changes compared to Table 2 are highlighted.

Game	Description/Changes	Reduction
(0)	The $LU^{+}_{\leftarrow}$ game with challenge bit chosen as 0.	
(1)	Get rid of challenge keys in $c^{j}_{\rightarrow}$ and $c^{n_{\leftarrow}}_{\leftarrow}$	
(1.1)	Replace the encryption $c^{n_{\leftarrow}}_{\leftarrow}$ of $meta^{n_{\leftarrow}}_{\leftarrow} = (\texttt{SNDR}, K^{n_{\leftarrow}}_{\leftarrow}, (n_{\rightarrow}, n_{\leftarrow}))$ by an encryption of $(\texttt{SNDR}, K', (n_{\rightarrow}, n_{\leftarrow}))$, for fresh $K' \leftarrow_{R} \mathcal{K}$. (Still use the old $meta^{n_{\leftarrow}}_{\leftarrow}$ to process the challenge in ProcOnion.)	IND-CCA
(1.2)	Replace the encryption $c^{j}_{\leftarrow}$ of $meta^{j}_{\leftarrow} = (\texttt{HOP}, K^{j}_{\leftarrow}, P^{j+1})$ by an encryption of $(\texttt{RCVR}, K'', \perp)$, for fresh $K'' \leftarrow_{R} \mathcal{K}$. (Still use the old $meta^{j}_{\leftarrow}$ to process.)	IND-CCA
(2)	Replace all keys $(K^{i}_{\rightarrow})^{n_{\rightarrow}}_{i=1}$ and $(K^{i}_{\leftarrow})^{n_{\leftarrow}-1}_{i=1}$ in challenge FormOnion by truly random keys (instead of deriving them from $omsk = K^{n_{\leftarrow}}_{\leftarrow}$).	PRF
(3)	Instead of deriving $K^{j}_{\mathsf{MAC}}, K^{j}_{\mathsf{SKE}}, K_{\mathsf{PRF}}$ from $K^{j}_{\leftarrow}$, use truly random keys when processing challenge onion.	PRF
(4)	In exception (1) (resp. (2)) of $LU^{+}_{\leftarrow}$, if RecognizeOnion recognizes query O, but $O.hdr \neq U^{j}_{\leftarrow}.hdr$ or $O.fwd \neq U^{j}_{\leftarrow}.fwd$ (resp. $O.hdr \neq U^{n_{\leftarrow}}_{\leftarrow}.hdr$ or $O.fwd \neq U^{n_{\leftarrow}}_{\leftarrow}.fwd$), then output $\texttt{FAIL}$ instead of processing the onion.	SUF-CMA + PRF
(4’)	Exception (2) of $LU^{+}_{\leftarrow}$, if RecognizeOnion recognizes the challenge onion, and if ProcOnion outputs $(m, \perp)$, always output m^{*} (unless $m = \perp$).	PRF
(5)	In exception (1) of $LU^{+}_{\leftarrow}$, but not exception (2), when $O^{j}_{\leftarrow}$ is recognized as challenge, output $\overline{O}^{j+1}_{\leftarrow} := (hdr, fwd, bwd)$, where instead of processing $O^{j}_{\leftarrow}$, $\overline{O}^{j+1}_{\leftarrow}$ is set to $U^{j+1}_{\leftarrow}$, and $\overline{O}^{j+1}_{\leftarrow}.bwd = bwd^{j+1}$ is processed honestly (via AE.Unwrap).	—
(6)	“Disconnect” $O^{j}_{\leftarrow}$ and $\overline{O}^{j+1}_{\leftarrow}$ by separating usage of keys	
(6.1)	In $U^{j+1}_{\leftarrow}$ (hence $\overline{O}^{j+1}_{\leftarrow}$), remove use of $(K^{i}_{\mathsf{SKE}})^{j}_{i=1}$ (in garbage added by $(P^{i}_{\leftarrow})^{j}_{i=1}$), by replacing blocks $B^{j+1}_{N-j+1}, \ldots, B^{j+1}_{N}$ by randomness $R^{j+1}_{N-j+1}, \ldots, R^{j+1}_{N}$	DLR\$-CPA
(6.2)	In computation of payload of $\overline{O}^{j+1}_{\leftarrow}$, remove use of K^{j}_{SKE} and K^{j}_{PRF} by replacing the backward payload $\overline{O}^{j+1}_{\leftarrow}.bwd$ by randomness	DLR\$-CPA + PRF
(6.3)	In $U^{j}_{\leftarrow}$, remove the use of $(K^{i}_{\mathsf{SKE}}, K^{i}_{\mathsf{MAC}}, K^{i}_{\mathsf{PRF}})^{n_{\leftarrow}}_{i=j+1}$ in header blocks $B^{j}_{2}, \ldots, B^{j}_{N-j+1}$ by replacing these blocks with randomness $R^{j}_{2}, \ldots, R^{j}_{N-j+1}$	DLR\$-CPA
(6.4)	In $U^{j}_{\leftarrow}$, remove the use of $(K^{i}_{\mathsf{SKE}})^{n_{\leftarrow}}_{i=j+1}$ in fwd^{j}, by replacing the value fwd^{j} with randomness during wrapping in Onionize.	DLR\$-CPA
(6.5)	This step is not applicable. (The challenge backward payload is adversarially chosen.)	—
(7)	Compute $\overline{O}^{j+1}_{\leftarrow}$ as FormOnion$(\overline{m}, (P^{i}_{\leftarrow})^{n_{\leftarrow}}_{i=j}, (); \mathcal{R})$	
(7.1)	In the header of $\overline{O}^{j+1}_{\leftarrow}$, replace (random) blocks $B^{j+1}_{N-j+1}, \ldots, B^{j+1}_{N}$ by wrapped fresh randomness, i.e., $(\mathsf{Enc}_{K^{j+1}}(\ldots \mathsf{Enc}_{K^{n_{\leftarrow}}}(R^{n_{\leftarrow}}_{i}) \ldots))^{N-(n_{\leftarrow}-j)+1}_{i=N-n_{\leftarrow}-j+1}$ for $R^{n_{\rightarrow}}_{i} \leftarrow_{R} \$$.	SKE is permutation
(7.1’)	In $\overline{O}^{j+1}_{\leftarrow}$, replace the forward payload by an encryption $\mathsf{Enc}_{K^{j+1}}(\ldots \mathsf{Enc}_{K^{n_{\leftarrow}}}(m) \ldots)$ of $m = (O_{\leftarrow}.hdr = \perp, \overline{m})$.	DLR\$-CPA + PRF
(7.2)	Derive keys $K^{j+1}_{\leftarrow}, \ldots, K^{n_{\leftarrow}}_{\leftarrow}$ used in $\overline{O}^{j+1}$ from $\overline{omsk}$.	PRF
(7.3)	Encrypt $(\texttt{RCVR}, \overline{omsk}, (n_{\leftarrow} - j, 0))$ in $\overline{O}.c^{n_{\leftarrow}}_{\leftarrow}$.	IND-CCA
(7.4)	Use FormOnion$(\overline{m}, (P^{i}_{\leftarrow})^{n_{\leftarrow}}_{i=j}, (); \mathcal{R})$ to compute $\overline{O}^{j+1}_{\leftarrow}$.	—
(8)	Compute $O^{1}_{\rightarrow}$ as FormOnion$(m, (P^{i}_{\rightarrow})^{n_{\rightarrow}}_{i=1}, (P^{i}_{\rightarrow})^{j}_{i=1}; \mathcal{R}')$	
(8.1)	This step is not applicable. (Its counterpart is step (7.1’))	—
(8.2)	Revert the change of Game (4) (but still return $\overline{O}^{j+1}_{\leftarrow}$ if challenge onion is recognized).	SUF-CMA + PRF
(8.3)	Revert the change in Game (3) by deriving $K^{j}_{\mathsf{MAC}}, K^{j}_{\mathsf{SKE}}, K^{j}_{\mathsf{PRF}}$ from $K^{j}_{\leftarrow}$ again	PRF
(8.4)	Derive all keys $(K^{i}_{\rightarrow})^{n_{\rightarrow}}_{i=1}$ and $(K^{i}_{\leftarrow})^{j}_{i=1}$ used in $O^{1}_{\leftarrow}$ from freshly chosen $omsk_{\leftarrow}'$	PRF
(8.5)	In $U^{j}_{\leftarrow}$, make block B^{j}_{1} a real “backward receiver” block by encrypting $(\texttt{SNDR}, K^{j}_{\leftarrow}, omsk')$ in c^{j}. (Note that $K^{j}_{\leftarrow}$ is, by definition, $omsk'$.)	IND-CCA
(8.6)	Use FormOnion$(m, (P^{i}_{\rightarrow})^{n_{\rightarrow}}_{i=1}, (P^{i}_{\rightarrow})^{j}_{i=1}; \mathcal{R}')$ for freshly chosen $\mathcal{R}'$ to compute O^{1}	—
(9)	The $LU^{+}_{\leftarrow}$ game with challenge bit chosen as 1.	

- If $\overline{bwd}^{j+1}.mac \neq \overline{O}_{\leftarrow}^{j+1}.bwd.mac$ holds, while at the same time $\overline{bwd}^{j+1}.ctxt = \overline{O}_{\leftarrow}^{j+1}.bwd.ctxt$, then this implies that $\overline{bwd}^{j} \neq \overline{O}_{\leftarrow}^{j}.bwd$ and $\overline{bwd}^{j}.ctxt = \overline{O}_{\leftarrow}^{j}.bwd.ctxt$ (because AE.Wrap is a permutation), and eventually $\overline{bwd}^{1} \neq \overline{O}_{\leftarrow}^{1}.bwd$ and $\overline{bwd}^{1}.ctxt = \overline{O}_{\leftarrow}^{1}.bwd.ctxt$. Note that AE.Dec (in pseudocode line 96) is just another AE.Wrap and a PRF equality check. The PRF-based MAC is *unique*, i.e., there is only a single valid MAC $bwd^1.mac$ for $bwd^1.ctxt$, but $\overline{bwd}^{1}$ differs. Thus, decryption must fail.
- If $\overline{bwd}^{j+1}.ctxt \neq \overline{O}_{\leftarrow}^{j+1}.bwd.ctxt$ holds, then this implies $\overline{bwd}^{j}.mac = \overline{r} \oplus \overline{bwd}^{j+1}.mac$ for $\overline{r} = \mathsf{PRF}K_{\mathsf{PRF}}(\overline{bwd}^{j}.ctxt)$ and $\overline{bwd}^{j} = \mathsf{Dec}_{K^j_{\mathsf{SKE}}}(\mathtt{BWD}, \overline{bwd}^{j+1}.ctxt)$. By an argument analogous to the argument above, there is at most one choice of $\overline{r}$ for which the MAC check succeeds. Since $\overline{r}$ is a PRF output, after we replace the PRF with a truly random function, the probability that the MAC check succeeds is at most $2^{-\lambda}$, as the output length of PRF is λ bits.

As $\overline{r} = 0$ is the only failure case, and this happens with probability at most $2^{-\lambda}$ if the PRF were replaced by a truly random function, we find that

$$\Pr[\mathsf{G}_4 = 1] - \Pr[\mathsf{G}_{4'} = 1] \leq 2 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{B}_{4'}} + 2^{-\lambda}.$$

Game $\mathsf{G}_{6.5}$. In the $LU^+_{\rightarrow}$ game, it was necessary to choose the backward payload randomly in game $\mathsf{G}_{6.5}$. Because, by definition of FormOnion, this is how the backward payload is constructed for a *forward* onion. However, as we handle a backward onion here (and never changed the honest generation of the forward onion part), and the backward payload is chosen adversarially (since $j > 0$ means that $\mathcal{A}$ controls the receiver), there is nothing to do here — the backward payload of the challenge was already honestly generated and processed until this point.

Game $\mathsf{G}_{7.1'}$. In this game, in $\overline{O}^{j+1}$, the random forward payload is replaced by an encryption $\mathsf{Enc}_{K^{j+1}_{\mathsf{SKE}}}(\dots \mathsf{Enc}_{K^{n_\leftarrow}_{\mathsf{SKE}}}(m)\dots)$ of $m = (\perp, \overline{m})$, where $\perp$ is due to $O_{\leftarrow}.hdr = \perp$ being encrypted in the forward payload of a non-repliable onion. Indistinguishability of this change follows from a few simple game hops: Since $K^{n_\leftarrow}_{\leftarrow}$ is truly random, we can first replace $K^{n_\leftarrow}_{\mathsf{SKE}} = \mathsf{KDF}(K^{n_\leftarrow}_{\leftarrow}, \mathtt{SKE})$ by a truly random key. Then apply a reduction to the DLR\$-CPA security of SKE, to replace the ciphertext by an encryption of $m = (\perp, \overline{m})$. Then derive $K^{n_\leftarrow}_{\mathsf{SKE}}$ again. Thus, we find

$$\Pr[\mathsf{G}_{7.1} = 1] - \Pr[\mathsf{G}_{7.1'} = 1] \leq \mathsf{Adv}^{\mathsf{dbl\$\text{-}cpa}}_{\mathsf{SKE},\mathcal{B}_{7.1'}} + 2 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{B}'_{7.1'}}.$$

Game $\mathsf{G}_{7.3}$. This game differs slightly from game $\mathsf{G}_{7.2}$ in $LU^+_{\rightarrow}$ because we must generate a *non-repliable* forward onion in $\overline{O}^{j+1}$. Thus, instead of SNDR, we need to encrypt RCVR for the final relay. Moreover, the final relay is the honest sender (since this is his reply). In particular, $j < n_\leftarrow$, since the honest sender cannot be corrupted. Thus, $n_\leftarrow - j \geq 1$ and $\overline{O}_{\leftarrow}.c_{\rightarrow}^{n_\leftarrow - j}$ is always defined (unlike in $LU^+_{\rightarrow}$, where the challenge onion may be non-repliable). Analogous to the $LU^+_{\rightarrow}$ game, this step ensures that $\overline{O}^{j+1}$ will correspond to a fresh $\mathsf{FormOnion}(\overline{m}, (P^i_{\leftarrow})^{n_\leftarrow}_{i=j}, (); \mathcal{R})$.

Game $\mathsf{G}_{8.5}$. This game differs slightly from game $\mathsf{G}_{8.5}$ in $LU^+_{\rightarrow}$ because we must generate a *repliable* as challenge onion in O^1. Thus, instead of RCVR, we need to encrypt SNDR for the final relay. Note that we handle only the case $j > 0$, and hence the shortened backward path is not empty. Analogous to $LU^+_{\rightarrow}$, this step ensures that O^{j+1} will correspond to a fresh $\mathsf{FormOnion}(m, (P^i_{\rightarrow})^{n_\rightarrow}_{i=1}, (P^i_{\rightarrow})^{j}_{i=1}; \mathcal{R}')$.

Game G_9. By appropriate syntactical transformations, we see that game $\mathsf{G}_{8.5}$ is identical to the $LU^+_{\leftarrow}$ game with challenge bit $b = 1$.

C.5 Security proof of $LU^+_{\leftarrow}$, $j = 0$

The proof steps are summarized in Table 4. The hybrid reductions are analogous to prior ones and left to the reader.

Table 4: Overview of proof for $LU^{+}_{\leftarrow}$, $j=0$. Unless noted otherwise, all changes are in the Onionize call of the challenge. Missing direction subscripts are backward ($\leftarrow$). Important changes compared to Table 3 are highlighted.

Game	Description/Changes	Reduction
(0)	The $LU^{+}_{\leftarrow}$ game with challenge bit chosen as 0.	
(1)	Get rid of challenge keys in $c^{n_\rightarrow}_{\rightarrow}$ and $c^{n_\leftarrow}_{\leftarrow}$	
(1.1)	Replace the encryption $c^{n_\leftarrow}_{\leftarrow}$ of $meta^{n_\leftarrow}_{\leftarrow} = (\texttt{SNDR}, K^{n_\leftarrow}_{\leftarrow}, (n_\rightarrow, n_\leftarrow))$ by an encryption of $(\texttt{SNDR}, K', (n_\rightarrow, n_\leftarrow))$, for fresh $K' \leftarrow_R \mathcal{K}$. (Still use the old $meta^{n_\leftarrow}_{\leftarrow}$ to process the challenge in ProcOnion.)	IND-CCA
(1.2)	Replace the encryption $c^{n_\rightarrow}_{\rightarrow}$ of $meta^{n_\rightarrow}_{\rightarrow} = (\texttt{HOP}, K^{n_\rightarrow}_{\rightarrow}, P^1_{\leftarrow})$ by an encryption of $(\texttt{RCVR}, K'', \perp)$, for fresh $K'' \leftarrow_R \mathcal{K}$. (Still use the old $meta^{n_\rightarrow}_{\rightarrow}$ to process.)	IND-CCA
(2)	Replace all keys $(K^i_{\rightarrow})^{n_\rightarrow}_{i=1}$ and $(K^i_{\leftarrow})^{n_\leftarrow -1}_{i=1}$ in challenge FormOnion by truly random keys (instead of deriving them from $omsk = K^{n_\leftarrow}_{\leftarrow}$).	PRF
(3)	Instead of deriving $K^{n_\rightarrow}_{\text{MAC}}, K^{n_\rightarrow}_{\text{SKE}}, K^{n_\rightarrow}_{\text{PRF}}, K_{\text{PRG}}$ from $K^{n_\rightarrow}_{\rightarrow}$, use truly random keys when processing challenge onion.	PRF
(4)	In exception (1) (resp. (2)) of $LU^{+}_{\leftarrow}$, if RecognizeOnion recognizes query O, but $O.hdr \neq U^{n_\rightarrow}_{\rightarrow}.hdr$ or $O.fwd \neq U^{n_\rightarrow}_{\rightarrow}.fwd$ (resp. $O.hdr \neq U^{n_\leftarrow}_{\leftarrow}.hdr$ or $O.fwd \neq U^{n_\leftarrow}_{\leftarrow}.fwd$), then output FAIL instead of processing the onion.	SUF-CMA + PRF
(4')	In Exception (2) of $LU^{+}_{\leftarrow}$, if RecognizeOnion recognizes the challenge onion, and if ProcOnion outputs $(m, \perp)$, always output m^* (unless $m = \perp$).	PRF
(5)	In exception (1) of $LU^{+}_{\leftarrow}$, but not exception (2), when $O^{n_\rightarrow}_{\rightarrow}$ is recognized as challenge, output $\overline{O}^1_{\leftarrow} := (hdr, fwd, bwd)$, where instead of processing $O^{n_\rightarrow}_{\rightarrow}$ (ReplyOnion), $\overline{O}^1_{\leftarrow}.hdr$ is set to $U^1_{\leftarrow}.hdr$, $\overline{O}^1_{\leftarrow}.fwd$ to $U^1_{\leftarrow}.fwd$ and $\overline{O}^1_{\leftarrow}.bwd$ to an authenticated encryption of $m^{\leftarrow}$ (for $m^{\leftarrow}$ provided in the ReplyOnion request) using $K^{n_\rightarrow}_{\text{SKE}}$ and $K^{n_\rightarrow}_{\text{PRF}}$.	—
(6)	"Disconnect" $O^{n_\rightarrow}_{\rightarrow}$ and $\overline{O}^1_{\leftarrow}$ by separating usage of keys	
(6.1)	This step is not applicable. ($\overline{O}^1_{\leftarrow}$ does not contain garbage added by $P^i_{\leftarrow}$)	—
(6.2)	In computation of payload of $\overline{O}^1_{\leftarrow}$, remove use of $K^{n_\rightarrow}_{\text{SKE}}$ and $K^{n_\rightarrow}_{\text{PRF}}$ by replacing the backward payload $\overline{O}^1_{\leftarrow}.bwd$ by randomness	DLR\$-CPA + PRF
(6.3)	In the forward payload of $O^{n_\rightarrow}_{\rightarrow}$, remove the use of $(K^i_{\text{SKE}}, K^i_{\text{MAC}}, K^i_{\text{PRF}})^{n_\leftarrow}_{i=1}$ by replacing $m' = (U^1_{\leftarrow}.hdr, m)$ with $m' = (\perp, m)$ in the forward payload $fwd^n = \text{Enc}_{K^{n_\rightarrow}_{\text{SKE}}}(m')$.	DLR\$-CPA
(6.4)	This step is obsolete given the change in Game (6.3).	—
(6.5)	This step is not applicable. (The challenge backward payload is adversarially chosen.)	—
(7)	Compute $\overline{O}^1_{\leftarrow}$ as FormOnion$(\overline{m}, (P^i_{\leftarrow})^{n_\leftarrow}_{i=1}, (); \mathcal{R})$	
(7.1)	This step is not applicable. ($\overline{O}^1_{\leftarrow}$ does not contain garbage added by $P^i_{\leftarrow}$)	—
(7.1')	In $\overline{O}^1_{\leftarrow}$, replace the forward payload by an encryption $\text{Enc}_{K^1}(\dots \text{Enc}_{K^{n_\leftarrow}}(m') \dots)$ of $m' = (O_{\leftarrow}.hdr = \perp, \overline{m})$.	DLR\$-CPA + PRF + PRG
(7.2)	Derive keys $K^1_{\leftarrow}, \dots, K^{n_\leftarrow}_{\leftarrow}$ used in $\overline{O}^1$ from $\overline{omsk}$.	PRF
(7.3)	Encrypt $(\texttt{RCVR}, \overline{omsk}, (n_\leftarrow, 0))$ in $\overline{O}.c^{n_\leftarrow}_{\leftarrow}$.	IND-CCA
(7.4)	Use FormOnion$(\overline{m}, (P^i_{\leftarrow})^{n_\leftarrow}_{i=1}, (); \mathcal{R})$ to compute $\overline{O}^1_{\leftarrow}$.	—
(8)	Compute $O^1_{\rightarrow}$ as FormOnion$(m, (P^i_{\rightarrow})^{n_\rightarrow}_{i=1}, (); \mathcal{R}')$	
(8.1)	This step is not applicable. (Its counterpart is step (7.1'))	—
(8.2)	Revert the change of Game (4) (but still return $\overline{O}^1_{\leftarrow}$ if challenge onion is recognized).	SUF-CMA + PRF
(8.3)	Revert the change in Game (3) by deriving $K^{n_\rightarrow}_{\text{MAC}}, K^{n_\rightarrow}_{\text{SKE}}, K^{n_\rightarrow}_{\text{PRF}}, K_{\text{PRG}}$. from $K^{n_\rightarrow}_{\rightarrow}$ again	PRF
(8.4)	Derive all keys $(K^i_{\rightarrow})^{n_\rightarrow}_{i=1}$ from freshly chosen $omsk'$ (note that no backward keys are contained in $O^1_{\rightarrow}.fwd$).	PRF
(8.5)	This step is not applicable. (Backward path in $O^1_{\rightarrow}$ is empty)	—
(8.6)	Use FormOnion$(m, (P^i_{\rightarrow})^{n_\rightarrow}_{i=1}, (); \mathcal{R}')$ for freshly chosen $\mathcal{R}'$ to compute O^1	—
(9)	The $LU^{+}_{\leftarrow}$ game with challenge bit chosen as 1.	