Onion Routing with Replies

Christiane Kuhn[1], Dennis Hofheinz[2], Andy Rupp[3], and Thorsten Strufe[1]

[1] Karlsruhe Institute of Technology, KASTEL firstname.lastname@kit.edu
[2] ETH Zürich lastname@inf.ethz.ch
[3] University of Luxembourg firstname.lastname@uni.lu

Abstract. Onion routing (OR) protocols are a crucial tool for providing anonymous internet communication. An OR protocol enables a user to anonymously send requests to a server. A fundamental problem of OR protocols is how to deal with replies: ideally, we would want the server to be able to send a reply back to the anonymous user without knowing or disclosing the user's identity.
Existing OR protocols do allow for such replies, but do not provably protect the payload (i.e., message) of replies against manipulation. Kuhn et al. (IEEE S&P 2020) show that such manipulations can in fact be leveraged to break anonymity of the whole protocol.
In this work, we close this gap and provide the first framework and protocols for OR with protected replies. We define security in the sense of an ideal functionality in the universal composability model, and provide corresponding (less complex) game-based security notions for the individual properties.
We also provide two secure instantiations of our framework: one based on updatable encryption, and one based on succinct non-interactive arguments (SNARGs) to authenticate payloads both in requests and replies. In both cases, our central technical handle is an *implicit* authentication of the transmitted payload data, as opposed to an explicit, but insufficient authentication (with MACs) in previous solutions. Our results exhibit a new and surprising application of updatable encryption outside of long-term data storage.

Keywords: Privacy, Anonymity, Updatable Encryption, SNARGs

This is the extended version to "Onion Routing with Replies" published at Asiacrypt 2021.

1 Introduction

Onion routing. Whenever we are communicating online without further security measures, personal information is leaked. While encryption can protect the content of the communication, metadata (like who communicates with whom) still allows an adversary to learn extensive sensitive information about her victim [22]. Mix [10] and Onion Routing (OR) Networks, like Tor [15], are crucial tools to protect communication metadata for example when accessing information on

web servers or during personal chat or email communication. Intuitively, in an OR protocol, the sender encrypts the message several times (e.g. using a public-key encryption scheme), which results in an "onion".[4] This onion is then sent along a path of OR relays chosen from an overlay network. Each relay removes (only) one layer of encryption and then forwards the partially-processed onion to the next relay. The last relay as the final receiver[5] removes the innermost layer of encryption and thus retrieves the plaintext.

This technique provides a certain degree of anonymity: the first relay knows the sender, but neither message plaintext nor final receiver, while the last relay as receiver knows only the message, but not the sender. These guarantees hold even if some relays are corrupt (i.e., under control of an adversary). In fact, as long as one of the involved relays is honest (and does not share its secrets), an onion cannot be distinguished from any other onion (with a possibly different sender and/or receiver).

Open problem: onion routing with replies. Most natural use cases for internet communication are bidirectional, i.e., require a receiver to respond to the sender. However, in the above simplified description, a receiver of an OR-transmitted message has no obvious way to send a *reply* back to an anonymous sender. Note that adding a sender address in plain to the payload message would of course defeat the purpose of OR. Even encrypting the sender address, say, with the public key of the receiver (so that the receiver can use another OR communication to reply), is not appropriate as we may not always trust the receiver to protect the sender's privacy (e.g., like a newspaper agency being forced to reveal whistleblowers).

Perhaps surprisingly, this problem of "OR with replies" has not been formally addressed in the OR literature with sufficient generality (with one recent exception).[6] Hence, our goal in this work is to provide definitions and instantiations for OR protocols "with an anonymous back envelope". That is, we attempt to formalize and construct OR protocols which allow the receiver to reply to the sender *without* revealing the identity of the sender to anyone.

Related work. Before detailing our own contribution, we first start by giving context. OR and Mixing have been introduced in the early years of anonymous communication. Chaum presented the first idea of a mix network, which randomly adds delays to each message at the forwarding relays to hinder linking based on timing information to the basic concept of layered encryption and source routing [10]; Goldschlag, Reed and Syverson proposed a clever setup procedure together with the same basic ideas, but decided against random delays

[4] This name stems from the fact that in order to get to the message, several layers of encryption have to be "peeled".

[5] There are also OR protocols that allow the receiver to be unaware of the protocol and provide anonymization as a service. In such a protocol, the last relay recovers both the plaintext and the receiver address. We however focus our work on the model with a protocol aware receiver.

[6] The one exception is a work by Ando and Lysyanskaya [3]. We discuss their work, and why we believe that their solution is not sufficient, below.

[19], which later on led to the development of the best known anonymous communication network, Tor [15]. In the following years many solutions applied the same technique [11,12,13,14,32,31]. With increasing importance of OR and understanding of the subtleties, which allow for attacks, theoretical and formal models for OR were developed. Thereby, the problem of secure onion routing and mixing is usually divided in two subproblems [6,13]: The definition of a secure onion routing/mixing packet format (to avoid simple observing and tagging attacks) and additional measures, like methods to detect dropping adversaries (against traffic analysis attacks).

For the scope of this paper, we concentrate on the first subproblem. Thus, we ignore attacks based on timings or dropping of onions, but instead aim to construct a secure packet format, which can later be combined with different measures against timing and traffic analysis attacks.[7]

Early on, Mauw et al. [29] modeled and analyzed OR. However, this work does not contain any proposal to prove future systems secure. Backes et al.'s ideal functionality [4] models Tor and hence includes sessions and reply channels. However, it is very specific to Tor and thus rather complex and can hardly be reused for general onion routing and mix networks. The Black-Box Model of Feigenbaum et al. [16] on the other hand, oversimplifies the problem and cannot support replies either. Further approaches [8,9,23] propose some security properties, but do not give any ideal functionality or similar concept that would allow to understand their concrete implications for the users' privacy.

As the most prominent formalization without replies, Camenisch and Lysanskaya [6] defined an ideal functionality in the UC-Framework and showed properties an onion routing protocol needs to fulfill to prove its security. Proving these properties has become the preferred way to prove mix and onion routing networks secure. It has been used to prove the correction [32] of Minx [14], as well as for the security proof of Sphinx [13], a fundamental protocol for onion routing and mix networks. Sphinx splits the onion in a header and payload part and elegantly includes the keys for every forwarding relay in the header, while the payload is modified with these keys at each hop. By decoupling header and payload Sphinx allows to precalculate a header for the backward direction, which can be included in the payload to send a reply. The most eminent, recently proposed practical onion routing and mix networks [11,12,31] build on Sphinx as solution for the first subproblem to subsequently tackle the second subproblem of dropping and timing attacks, while proving the security of their adaptions to Sphinx still based on the properties of Camenisch and Lysanskaya.

However, Kuhn et al. [26] recently recognized and corrected flaws in those definitions that allowed for sincere practical consequences in the form of a *malleability attack*:

[7] For example, we accept a packet format solution that transforms a modification attack into a dropping attack, e.g. by recognizing the modification and dropping the according onion. As dropping attacks can be solved with additional measures, this does not weaken the protocol.

An adversary that controls the internet service provision of the sender, or the first relay as well as the receiver can easily mark the onion for later recognition by flipping bits in the payload of the onion sent by her victim. She then checks if the receiver receives some message that is not in the usual message space (e.g. not containing English language). This allows the adversary to easily learn that the victim wanted to contact the receiver (if such an unusual message was received). Otherwise, the victim must be communicating with someone else. This attack requires a very weak adversary only, but ultimately breaks the anonymity of the sender, defeating the entire purpose of the protocol.

While the corrected properties are easy enough to be used, Kuhn et al. only partially fix the situation as the models do not include support for reply messages (to the anonymous sender). Sphinx and the improved version of Minx make adaptions to the properties of Camenisch and Lysyanskaya to account for replies to some extent, but thereby they build on the flawed properties and do not treat replies correctly, thus limiting the achieved privacy [26]. Even worse, nearly all of the eminent recent network proposals claim to support anonymity for replies, while relying on the flawed properties for which the practical attack [26] was introduced at the example of HORNET [11], an OR network that was proposed as the successor of Tor.

On the work of Ando and Lysyanskaya. In a recent work [3], Ando and Lysyanskaya define an ideal functionality for repliable onion routing. They also propose corresponding properties and a protocol that satisfies their ideal functionality. In this, they partition onions in header and payload and realize replies by having the senders pre-compute another header for the reply path. They alter the header and payload deterministically and check the header's integrity at each hop on the path, but they do not check the payload's integrity until the onion is at its final destination.

Thus the above malleability attack still works: Assume an adversarial first relay P_1 and receiver. P_1 modifies the payload of the (forward) onion, i.e. replaces the payload ciphertext C_2 in [3] with randomness. The next honest relays process the onion as usual without noticing this, as [3] uses *mere end-to-end* integrity protection for the payload. Only the adversarial receiver can notice the payload manipulation as the verification fails. This is the signal for the adversary that this is the onion she tampered with earlier. While the message is lost, the adversary learns critical metadata: who wanted to contact the receiver (e.g. the regime, hosting own relays and pressuring newspapers, learns who tried to anonymously contact the newspaper), unacceptable for practical protocols like [11,12].

While Ando and Lysyanskaya target the same setting, they avoid the challenge imposed by the above malleability attack [26] by explicitly allowing it in their ideal functionality, which allows them to work with traditional malleability protection: a "postponed" integrity check at the destination. This however even strengthens the simple, yet effective de-anonymizing malleability attack on the payload, as the receiver now realizes the failed integrity check and does not even have to compare with the expected message space. Therefore, the question of a *secure*, repliable OR scheme is still unanswered.

A technical challenge with practical relevance. The goal we are aiming at is not only useful, but also technically difficult to achieve. First of all, practically prominent protocols and packet formats [11,13,32] require that *any reply is indistinguishable from any forward request*, except at the sender and receiver. In particular, all parts of the onions in both directions should look alike, and must be treated according to the same processing rules. This is necessary to provide senders that expect replies with a sufficiently large anonymity set even if there is only a small amount of reply packets, because they are hidden under all forward traffic.

To prevent the malleability attack [26], tampering by a potentially corrupt relay must become detectable. Theoretically, conventional payload authentication for all forward layers, e.g., with MACs precalculated by the sender, is sufficient. However, both Ando and Lysyanskaya [3] and practical proposals [11] require indistinguishability of forward and reply onions. Extending authentication also to the reply payload is challenging: The original sender cannot precalculate those authentication tags as the reply payload is unknown and we cannot necessarily assume that the receiver is honest. Letting the reply sender (= original receiver) precalculate the authentication tags enables an attack similar to the malleability attack: the malicious reply sender (= receiver) together with the last relay can recognize the reply onion (without modifying its payload on the way) simply based on the known authentication tags; thus letting the attacker learn the same metadata as in the malleability attack. Hence, payload protection in the reply setting is the real challenge towards a practical solution.

Our contribution. In this work, we present a framework for repliable OR, along with two different instantiations (with different properties). Our framework protects against malleability attacks on the payload, while even guaranteeing that replies are indistinguishable from original requests. In our approach, hence, both requests and replies are authenticated *implicitly* (i.e., without MACs) at each step along the way.

From a definitional point of view, we express these requirements by an ideal functionality (in the UC framework) which does not reveal the onion's path, message or direction to the adversary (unless all involved routers are corrupt; further a corrupt receiver of course learns the message and direction). This translates to strong game-based properties, which are proven to imply the security in the sense of that ideal functionality.

We also present two protocols that realize this ideal functionality. Both of our OR protocols are in fact similar to existing protocols, and are partially inspired by the popular Sphinx approach [13] and the Shallot scheme of Ando and Lysyanskaya [3]. The main conceptual difference to previous work is that the authentication of the (encrypted) payload happens *implicitly* in our case.

Our first protocol uses updatable encryption (UE), a variant of symmetric[8] encryption that provides both re-randomizable ciphertexts (and in fact RCCA security [7] and plaintext integrity) and re-randomizable keys, as a central prim-

[8] Although being a variant of symmetric encryption, UE schemes typically make use of public-key techniques to achieve updatability through malleability.

itive. Intuitively, using UE for encrypting the payload message (in both communication directions) enables a form of "implicit authentication" of ciphertexts, and hence thwarts malleability attacks without explicit MACs on the payload.

Our second protocol is based on succinct non-interactive arguments (SNARGs [30,5]), a variant of zero-knowledge arguments with compact proofs. Intuitively, SNARGs enable every relay *and* the receiver to prove that they have processed (or replied to) their input onion according to the protocol. This way, no explicit authentication of the payload data is necessary, since the SNARGs guarantee that no "content-changing" modification of the payload took place.

Neither of our protocols indeed are competitive in efficiency with existing OR protocols. Further, our protocols require a trusted setup. This is due to the introduction of new concepts and techniques for qualitatively stronger security properties. Our work however represents an important conceptual first step towards an efficient *and* secure solution.

A closer look at our UE-based protocol. We start by outlining the basic ideas of our protocol based on updatable encryption (UE).

UE originally targets the scenario of securely outsourcing data to a semi-trusted cloud server. To enable efficient key rotation, i.e., updating the stored ciphertexts to a freshly chosen key, UE schemes allow the generation of update tokens based on the old and new key. Given such a token the server can autonomously lift a ciphertext encrypted with the old key to a ciphertext encrypted with the new key. Of course, the token itself may not leak information about the old nor the new key to the cloud server. Despite this update feature, UE schemes should satisfy security properties similar to regular authenticated symmetric encryption schemes like IND-CPA/RCCA/CCA type of security along with INT-PTXT or INT-CTXT security (when excluding trivial wins resulting from corrupting certain keys and tokens). An additional property, which make them especially interesting for our purposes is that some provide unlinkability of ciphertext updates, i.e., an updated ciphertext does not provide information about its old version (even given the old key).

The basic idea to exploit UE for secure onion routing with replies is simple: the sender encrypts its request using an UE scheme and provides each relay, using a header construction similar to Sphinx, with an update token to unlinkably transform this request. Similarly, the receiver is equipped with the corresponding decryption key and a fresh encryption key for the backward path.

More precisely, each onion $O = (\eta, \delta)$ consists of two components:

- a *header* η which contains encrypted key material with which routers can process and (conventionally) authenticate the header itself, and
- the UE-encrypted *payload* δ; each router will use a UE update token to re-randomize and re-encrypt δ under a different (hidden) key.

The structure of η is similar to the Sphinx and Shallot protocols. Namely, each layer contains a public-key encryption (under the public key of the respective relay) of ephemeral keys that encrypt the next layer (including the address of the next relay), and authentication information with which to verify this layer.

Additionally, in our case each layer also contains an encrypted token which can be used to update the payload ciphertext δ and we include the backward path in the header as well. All of this header information can be precomputed by the sender for both communication directions (i.e., for the path from sender to receiver, and for the return path).

After processing the header (i.e., decrypting, verifiying, and applying the UE token to the payload), each relay pads the so-extracted header for the next relay suitably with randomness, so that it is not clear how far along the onion has been processed. At some point, the decrypted header will contain a receiver symbol and a UE decryption key to indicate that processing of the onion in the forward direction has finished.

The header will then contain also a UE encryption key and enough information for the receiver to prepare a "backwards onion", i.e., an onion with the same format for the return path. We stress that all header parts of this "backwards onion", including UE tokens and authentication parts, are precomputed by the initial sender. The receiver merely UE-encrypts the payload and adds padding similarly to relays during processing.

Processing on the return path works similarly, only that eventually, the initial sender is contacted with the (still UE-encrypted) reply payload. The sender can then decrypt the payload using a precomputed UE key.

We stress that there is no explicit check that the payload δ is still intact at any point. However, the demanded UE security guarantees that re-encryption of invalid UE ciphertexts will fail.

More precisely, we require an UE scheme with strong properties, namely RCCA security, plaintext integrity and perfect re-encryption under *ciphertext-independent* re-encryption of *arbitrary* (i.e., even maliciously formed) ciphertexts. RCCA security and plaintext integrity ensure that valid payloads from an honest sender cannot be modified or replaced by adversarially generated payloads. Perfect re-encryption ensures that a payload encryption observed along the forward/backward path does not leak the position in the path. This property also implies the unlinkability of ciphertext updates. Allowing re-encryption of arbitrary ciphertexts in the UE security games (what many UE schemes do not consider) is crucial in our scenario as the relays who will perform re-encryption might be easily confronted with adversarially crafted ciphertexts which they need to reject. Also the property that update tokens can be generated independently of the ciphertext to be updated is essential for our application as otherwise the anonymous sender could not pre-compute the tokens for the backward path.

Considering the requirements from above, we are currently only aware of a single suitable UE scheme which is a construction by Klooss, Lehmann, and Rupp [24] based on (the malleability of) Groth-Sahai proofs. Unfortunately, instantiating our protocol with their UE scheme leads to payload parts of the onion which are comparatively large: Their underlying algebraic structure is a pairing-based group setting $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. To encrypt a single $\mathbb{G}_1$-element, the payload part contains 58 $\mathbb{G}_1$-and 44 $\mathbb{G}_2$-elements. For realistic group (bit)sizes of, say, $|\mathbb{G}_1| = 256$ and $|\mathbb{G}_2| = 512$, we obtain a payload size of about 4.5 kilo-

bytes for 256 bits of communicated message. The header part of the onion is about half as large for small pathlengths, and using conventional state-of-the art building blocks, a full onion (including header and payload) comes out at about $4.5 + N$ kilobytes, where N is the maximal length of a path, i.e., the number of hops between sender and receiver (cf. Appendix G.1 for details). Processing an onion at a relay is dominated by the cost to perform the re-encryption of the payload which requires about 110 $\mathbb{G}_1$- and 90 $\mathbb{G}_2$-exponentiations [24].

A closer look at our SNARG-based protocol. Our SNARG-based protocol works conceptually similarly, but with two differences:

- First, the payload is enclosed by multiple symmetric encryption layers (one for each relay). This is very similar to previous approaches [13,3], but also opens the door to malleability attacks.
- Second, in order to prevent such malleability attacks, each layer contains a concise SNARG proof on top of header and payload, which proves that this onion is the result of (a) a fresh onion as constructed by a sender, (b) a fresh backwards onion as constructed by a receiver, or (c) a legitimate processing of another onion (with valid SNARG proof). In essence, this SNARG proof avoids malleability attacks by inductively proving that this onion has gone only through valid onion generation or processing steps.

We note that the SNARG proof may need to show that this onion is the result of an honest processing of another onion *with valid SNARG proof*. Hence, we need to be careful in designing the corresponding SNARG language in a recursive way while avoiding circularities. This recursive and self-referential nature of our language is also the reason why we use SNARGs (as opposed to "regular" zero-knowledge techniques with larger proofs).

Our SNARG-based onions are in fact smaller (for small pathlengths N) than the ones from our UE-based protocol. Using the SNARKs of Groth and Maller [21] (and state-of-the-art conventional building blocks), we obtain onions with an additive overhead (over the message size) of $128N^2 + 448N + 192(2N - 1) + 160$ bytes (cf. Appendix G.2 for details). The perhaps surprising quadratic term in the maximal pathlength N stems from the fact that we require additional encryptions of *all* previous onion headers to enable a recursive extraction of previous onion states.

However, due to our somewhat complex SNARG language, we expect that the actual processing time of our SNARG-based approach (which involves constructing SNARG proofs at each processing step) will be considerably higher than the one from our UE-based protocol.

Performance in comparison to Ando and Lysyanskaya. While Ando and Lysyanskaya do not provide concrete efficiency calculations, conceptually their and our (time and space) overhead are similar *except* for the parts related to updatable encryption, resp. SNARGs. For realistic security parameters, these parts dominate the header overhead. This is the price one has to pay for preventing the malleability attack from [26] while making reply onions indistinguishable from request onions as desired by practical protocols like HORNET [11].

After all, this is the first paper providing immunity in this strong sense, and while we do not claim optimality of our constructions, we are convinced they are the basis for a real-world improvement in communication privacy.

2 Notation

We use the superscript $x^{\leftarrow}$ to denote the corresponding entity on the backwards path. For example, while P_1 is the first router on the forwards path, $P_1{}^{\leftarrow}$ is the first router on the backwards path. Further, we use the notation as summarized in the following table:

Notation	Meaning
$\|$	concatenation of strings
λ	the security parameter
$\mathscr{P}$	an onion path
m	a message
P_i	for the i-th router name on the forward path, P_0 (usually $= P_{n^{\leftarrow}+1}^{\leftarrow}$) is the forward sender and $P_{n+1} (= P_0^{\leftarrow})$ the forward receiver
PK_i	public key of P_i
SK_i	private key of P_i
O_i	$= (\eta_i, \delta_i)$ is the i-th forward onion layer to be processed by P_i
η_i	the header of O_i
δ_i	the payload of O_i
FormOnion	the function to build a new onion as a sender.
ProcOnion	the function to process an onion at a relay.
ReplyOnion	the function to reply to a received onion as receiver.

3 Model and Ideal Functionality

We first define our assumptions and model for repliable OR and then describe our desired security as the ideal functionality. Our model extends the OR scheme definition of [6] as used in [26] by adding an algorithm to create replies. Our ideal functionality extends the one of [6] as used in [26] and has similarities to [3], but is strictly stronger as it requires protection against malleability attacks on the payload.

3.1 Assumptions

We make the following assumptions that result from commonly used techniques to ensure unlinkability of onion layers on criteria other than the concrete representation of the onion.

As in earlier examples, we assume the existence of public keys PK for all relays.

Assumption 1 *The sender knows the (authentic) public keys PK_i of all relays P_i it uses (e.g., by means of a PKI).*

To ensure that packets cannot be linked based on their size, all onions are padded to the same, fixed size (otherwise the largest incoming onion could trivially be linked to the largest outgoing onion). As the path information has to be encoded in the onion, fixing the size also entails an upper bound for the length of the routing path.

Assumption 2 *The protocol's maximum path length is N.*

To ensure that packets cannot be linked based on the included routing path, the sender includes the routing information encrypted for each forwarder, such that any forwarder only learns the next hop of the routing path. We assume that the routing information is included in a header, while the message is included in the payload of the onion.

Assumption 3 *Each onion O consists of a header η and a payload δ.*

To ensure that packets cannot be linked based on duplicate attacks, i.e. the onion of the victim is duplicated at the first corrupted relay and observed twice at the corresponding receiver, duplicates have to be detected and dropped. We support duplicate detection with deterministically evolving headers, which allows to also protect from duplicated replies. Thus, even though the (forward) receiver is allowed to decide on her answer arbitrarily, we can still detect if she tries to send multiple different answers to the same request.[9] As some related work wrongly adapted proof strategies for OR schemes where the duplicate detection is solely based on *parts of the onion*, we deliberately build this model for OR-schemes allowing for such protocols.[10]

Assumption 4 *Duplicates, i.e. onions O_i, O_i' with the same header $\eta_i = \eta_i'$, lead to a fail for every but the first such onion that is given to* $\mathrm{ProcOnion}(SK_i, O_i, P_i)$ *except with negligible probability.*

To ensure the best chances that an honest relay is on the path, the honest sender will pick a path without any repetition in the relays (acyclic).[11]

Assumption 5 *Each honestly chosen path $\mathcal{P}$ is acyclic.*

While true for, to our knowledge, all protocols, we use the following processing order as an assumption in our proofs:

Assumption 6 *Each onion is processed by the receiver, before it is replied to.*

[9] Note that our scope is a secure message format. Traffic analysis protection, like e.g. recognizing duplicated onions, has to happen *additionally* to our message format, but assuming that such a protection is in place allows for simplified proofs even for the message format.

[10] Practically, this assumption is often ensured by storing the seen headers in an efficient way, e.g. Bloom filters, until a router's key pair is changed or the current epoch expires if the protocol works in time epochs. The change of key pairs can be expressed in our framework by replacing a router identity by a fresh one ("Bob2020" becomes "Bob2025").

[11] Note that our adversary model trusts the sender and hence this assumption is merely a restriction of how the protocol works and the sender does not need to prove a correct choice to anyone.

3.2 Modeling Replies

We extend the definition of an onion routing scheme [6] as used in [26] with an algorithm to send replies, similar to [3].

Definition 1 (Repliable OR Scheme). *A Repliable OR Scheme is a tuple of PPT algorithms* $(G, \text{FormOnion}, \text{ProcOnion}, \text{ReplyOnion})$ *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. $\text{FormOnion}(i, \mathcal{R}, m, \mathcal{P}^\rightarrow, \mathcal{P}^\leftarrow, (PK)_{\mathcal{P}^\rightarrow}, (PK)_{\mathcal{P}^\leftarrow})$ *returns an* i*-th*[12] *onion layer* O_i *(*$i = 1$ *for sending) on input of* $i \leq n+n^\leftarrow+2$ *(for* $i > n+1$*,* m *is the reply message and* O_i *the backward onion layer), randomness* $\mathcal{R}$*, message* m*, a forward path* $\mathcal{P}^\rightarrow = (P_1, \ldots, P_{n+1})$*, a backward path* $\mathcal{P}^\leftarrow = (P_1^\leftarrow, \ldots, P_{n^\leftarrow+1}^\leftarrow)$*, public keys* $(PK)_{\mathcal{P}^\rightarrow} = (PK_1, \ldots, PK_{n+1})$ *of the relays on the forward path, and public keys* $(PK)_{\mathcal{P}^\leftarrow} = (PK'_1, \ldots, PK'_{n^\leftarrow+1})$ *of the relays on the backward path. The backward path can be empty if the onion is not intended to be repliable.*

Forwarding an onion. $\text{ProcOnion}(SK, O, P)$ *outputs the next onion layer and router identity* (O', P') *on input of an onion layer* O*, a router identity* P *and* P*'s secret key* SK*.* (O', P') *equals* $(\perp, \perp)$ *in case of an error or* $(m, \perp)$ *if* P *is the recipient.*

Replying to an onion. $\text{ReplyOnion}(m^\leftarrow, O, P, SK)$ *returns a reply onion* $O^\leftarrow$ *along with the next router* $P^\leftarrow$ *on input of a received (forward) onion* O*, a reply message* $m^\leftarrow$*, the receiver identity* P *and its secret key* SK*.* $O^\leftarrow$ *and* $P^\leftarrow$ *attains* $\perp$ *in case of an error.*

Correctness. We want the onions to take the paths and deliver the messages that were chosen as the input to FormOnion resp. ReplyOnion.

Definition 2 (Correctness). *Let* $(G, \text{FormOnion}, \text{ProcOnion}, \text{ReplyOnion})$ *be a repliable OR scheme with maximal path length* N*. Then for all* $n, n^\leftarrow < N$*,* $\lambda \in \mathbb{N}$*, all choices of the public parameter* p*, all choices of the randomness* $\mathcal{R}$*, all choices of forward and backward paths* $\mathcal{P}^\rightarrow = (P_1, \ldots, P_{n+1})$ *and* $\mathcal{P}^\leftarrow = (P_1^\leftarrow, \ldots, P_{n^\leftarrow+1}^\leftarrow)$*, 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* $Q_1 := P_1$*,* $O_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m, (P_1, \ldots, P_{n+1}), (P_1^\leftarrow, \ldots, P_{n^\leftarrow+1}^\leftarrow), (PK_1, \ldots, PK_{n+1}), (PK_1^\leftarrow, \ldots, PK_{n^\leftarrow+1}^\leftarrow))$*,* $(O_{i+1}, Q_{i+1}) \leftarrow \text{ProcOnion}(SK_i, O_i, Q_i)$*.*

Correctness of request reception. $(m, \perp) = \text{ProcOnion}(SK_{n+1}, O_{n+1}, P_{n+1})$

Correctness of backward path. $Q_i^\leftarrow = P_i^\leftarrow$*, for* $1 \leq i \leq n$ *and* $(O_1^\leftarrow, Q_1^\leftarrow) \leftarrow \text{ReplyOnion}(m^\leftarrow, O_{n+1}, P_{n+1}, SK_{n+1})$*,* $(O_{i+1}^\leftarrow, Q_{i+1}^\leftarrow) \leftarrow \text{ProcOnion}(SK_i^\leftarrow, O_i^\leftarrow, Q_i^\leftarrow)$*.*

Correctness of reply reception. $(m^\leftarrow, \perp) = \text{ProcOnion}(SK_{n^\leftarrow+1}^\leftarrow, O_{n^\leftarrow+1}^\leftarrow, P_{n^\leftarrow+1}^\leftarrow)$

[12] During normal operation only $i = 1$ is used. The possibility to form onion layers for $i > 1$ (without using ProcOnion) is needed for our security definitions and proofs.

Recognizing onions. To define our security properties, we need a way to recognize if an onion O provided by the adversary resulted from processing a given onion O^*. To this end, we define the algorithm RecognizeOnion$(i, O, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})$, which uses the given inputs (that have been used to create the onion O^* in the first place) to form the i-th layer of the onion O_i^* using FormOnion and then compares the header of O_i^* to the header of the onion O in question. If the headers[13] are identical, it returns *True*, otherwise *False*.

Note that the "correctness" of FormOnion and RecognizeOnion for $i > 1$ is defined implicitly as part of our security properties in Section 4.

3.3 Ideal Functionality

Informally, as long as the sender is honest we want that the adversary can only learn the parts of the onion's path (and associated reply's path), where she corrupted all relays. This includes especially the following three facts:

1. The adversary cannot link onion layers before and after any honest relay.
2. The adversary cannot learn the included message, unless she controls the receiver.
3. The adversary cannot distinguish whether onions are on the forward or backward path, unless she controls the receiver and the onion is either the last layer before (forward) reception, or the first layer of her reply.

Note that this especially includes that she also cannot link layers based on malleability attacks on the payload. See Appendix B for technical details.

4 New Properties

We now define our security properties and show that if they are fulfilled, our ideal functionality is realized.

The ideal functionality requires that the adversary only learns parts of the onion's real path; the subpaths from each honest relay until the next honest relay. Our idea is to replace any real sequence of onion layers that is observed on such a subpath, with a random sequence that only is equal in the information learned by the adversary, i.e., the allowed leakage of the ideal functionality. More precisely, this information relates to the subpath the adversary controls. It extends to the plaintext of the message, and the fact if the onion is at forward or backward layers, if she also controls the receiver. For replacement, we distinguish three types of subpaths and introduce one property for each type, challenging the adversary to distinguish the real and a random layer sequence for this specific subpath type. The types are: a subpath that is part of the forward path (Forwards Layer-Unlinkability), one that is part of the backward path (Backwards Layer-Unlinkability) and one that includes parts of the forward and backward path as the receiver is corrupted (Repliable Tail-Indistinguishability).

[13] We define RecognizeOnion and the duplicate detection on the header as this is common practice.

Forward Path: We first require that the layers on the forward path can be replaced by *random* ones. Therefore, we extend Layer-Unlinkability from [26] with oracles for the creation of replies and illustrate the property in Figure 1.

Thereby, we challenge the adversary to distinguish between (a) an onion created according to her choices from (b) a random onion that takes the same path from the sender to the first honest relay. We use oracles to allow for processing of and replying to (other) onions at the honest relays. Due to duplicate checks, these oracles only return a processed onion if no onion with this header was processed before (Assumption 4) and only return a reply onion if the onion was processed before (Assumption 6). Further, the oracle after the challenge has to treat the challenge onion with care: if it is processed or replied to (depending if the honest relay is an intermediate relay or the receiver), a onion fitting to the original choice is constructed with FormOnion and returned.

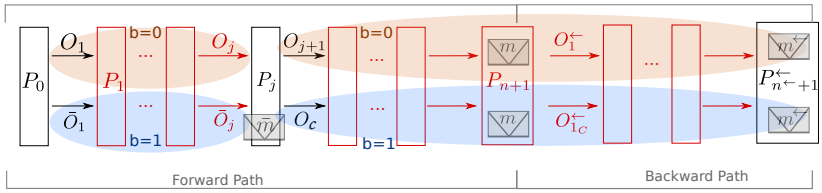

Fig. 1. Forwards Layer-Unlinkability illustrated: Red boxes are corrupted relays, black honest relays, orange ellipses are the $b = 0$ and the blue the $b = 1$ case. $\bar{m}$ is a random message. The main idea is that the adversary cannot distinguish between real and random onions *before* P_j.

Definition 3 (Forwards Layer-Unlinkability $LU^{\rightarrow}$). *Forwards Layer Unlinkability is defined as:*

1. *The adversary receives the router names P_H, P_S and challenge public keys PK_S, PK_H, chosen by the challenger by letting $(PK_H, SK_H) \leftarrow G(1^\lambda, p, P_H)$ and $(PK_S, SK_S) \leftarrow G(1^\lambda, p, P_S)$.*
2. *Oracle access: The adversary may submit any number of* Proc *and* Reply *requests for P_H or P_S to the challenger. For any* Proc(P_H, O)*, the challenger checks whether η is on the η^H- list. If not, it sends the output of* ProcOnion(SK_H, O, P_H)*, stores η on the η^H-list and O on the O^H-list. For any* Reply(P_H, O, m) *the challenger checks if O is on the O^H- list and if so, the challenger sends* ReplyOnion(m, O, P_H, SK_H) *to the adversary. (Similar for requests on P_S with the η^S-list).*
3. *The adversary submits a message m, a position j with $1 \leq j \leq n+1$, a path $\mathcal{P}^{\rightarrow} = (P_1, \ldots, P_j, \ldots, P_{n+1})$ with $P_j = P_H$,a path $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{n^{\leftarrow}+1}^{\leftarrow} = P_S)$ and public keys for all nodes PK_i ($1 \leq i \leq n+1$ for the nodes on the path and $n+1 < i$ 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* $PK^{\leftarrow}_{n^{\leftarrow}+1} = PK_S$ *and picks* $b \in \{0,1\}$ *at random.*
5. *The challenger creates the onion with the adversary's input choice and honestly chosen randomness* $\mathcal{R}$*:* $O_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})$ *and a replacement onion with the first part of the forward path* $\mathcal{P}^{\rightarrow} = (P_1, \ldots, P_j)$*, a random message* $\bar{m} \in \mathcal{M}$*, another honestly chosen randomness* $\bar{\mathcal{R}}$*, and an empty backward path* $\bar{\mathcal{P}}^{\leftarrow} = ()$*:* $\bar{O}_1 \leftarrow \text{FormOnion}(1, \bar{\mathcal{R}}, \bar{m}, \bar{\mathcal{P}}^{\rightarrow}, \bar{\mathcal{P}}^{\leftarrow}, (PK)_{\bar{\mathcal{P}}^{\rightarrow}}, (PK)_{\bar{\mathcal{P}}^{\leftarrow}})$
6. *If* $b = 0$*, the challenger gives* O_1 *to the adversary.*
   Otherwise, the challenger gives $\bar{O}_1$ *to the adversary.*
7. *Oracle access:*
   If $b = 0$*, the challenger processes all oracle requests as in step 2).*
   Otherwise, the challenger processes all requests as in step 2) except for:
   - *If* $j < n+1$*:* Proc(P_H, O) *with* $\text{RecognizeOnion}(j, O, \bar{\mathcal{R}}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}) = True$*,* η *is not on the* η^H*-list and* $\text{ProcOnion}(SK_H, O, P_H) \neq \perp$*:*
     The challenger outputs (P_{j+1}, O_c) *with* $O_c \leftarrow \text{FormOnion}(j+1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})$ *and adds* η *to the* η^H*-list and* O *to the* O^H*-list.*
   - *If* $j = n+1$*:*
     * Proc(P_H, O) *with* $\text{RecognizeOnion}(j, O, \bar{\mathcal{R}}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}) = True$*,* η *is not on the* η^H*-list and* $\text{ProcOnion}(SK_H, O, P_H) \neq \perp$*:*
       The challenger outputs $(m, \perp)$ *and adds* η *to the* η^H*-list and* O *to the* O^H*-list.*
     * Reply$(P_H, O, m^{\leftarrow})$ *with* $\text{RecognizeOnion}(j, O, \bar{\mathcal{R}}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}) = True$*,* O *is on the* O^H*- list and has not been replied before and* $\text{ReplyOnion}(m^{\leftarrow}, O, P_H, SK_H) \neq \perp$*:*
       The challenger outputs $(P_1^{\leftarrow}, O_c)$ *with* $O_c \leftarrow \text{FormOnion}(j+1, \mathcal{R}, m^{\leftarrow}, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})$
8. *The adversary produces guess* b'*.*

$LU^{\rightarrow}$ *is achieved if any probabilistic polynomial time (PPT) adversary* $\mathcal{A}$*, cannot guess* $b' = b$ *with a probability non-negligibly better than* $\frac{1}{2}$*.*

Note that by using the real processing for the oracle in step 7 for $b = 0$ and the recognition and a newly formed onion layer for $j+1$ in $b = 1$, it follows that both RecognizeOnion and FormOnion have to adhere to their intuition, i.e. with overwhelming probability only the challenge onion is recognized and the newly formed layer has to be indistinguishable to the real processing.

Backward Path: Additionally, we build a reverse version of Layer-Unlinkability for the backward path and illustrate the property *Backwards Layer-Unlinkability* $LU^{\leftarrow}$ in Figure 2. This definition is similar to $LU^{\rightarrow}$, but the challenge is to distinguish a reply from randomness. We thus return the challenge onion in a special case of the second oracle (step 7 in $LU^{\rightarrow}$) and the forward onion is always

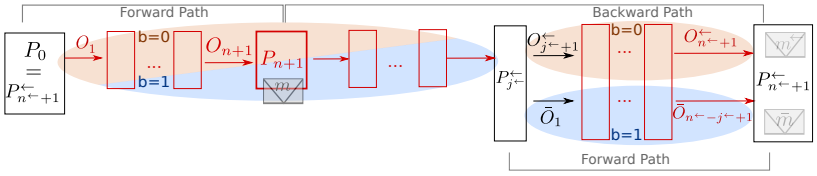

Fig. 2. Backwards Layer-Unlinkability illustrated: Red boxes are corrupted relays, black honest relays, orange ellipses are the $b = 0$ and the blue the $b = 1$ case. The main idea is that the adversary cannot distinguish between real and random onions *after* $P^{\leftarrow}_{j\leftarrow}$.

constructed to the adversary's choice (instead of step 6 in $LU^{\rightarrow}$). The challenge onion either contains the layers of the reply constructed to the adversary's choices (including the chosen reply message) or random *forward* layers with a random message. As these two cases are trivially distinguishable by processing the challenge onion at the honest original sender (i.e. backwards receiver), we ensure that the oracle denies to do this final processing of the challenge onion. This corresponds to the real world in which our trusted sender does not share any received message with the adversary. For a formal definition of this property see Appendix C.1.

Notice that we pick the random replacements to be *forward onion layers*. Thus the property $LU^{\leftarrow}$ implies indistinguishability between forward and backward onions for intermediates (otherwise the adversary could distinguish the real (backward) onion from the fake (forward) onion).

Between forward and backward path: Finally, we want to replace the

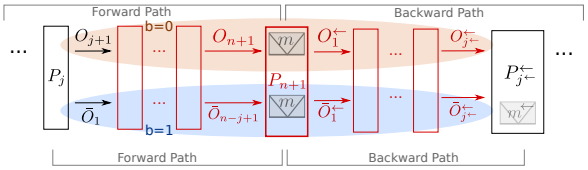

Fig. 3. Repliable Tail-Indistinguishability illustrated: Red boxes are corrupted relays, black honest relays, orange ellipses are the $b = 0$ and the blue the $b = 1$ case. While the adversary can learn the behavior between *between* P_j and $P^{\leftarrow}_{j\leftarrow}$ she cannot connect it to anything before P_j and after $P^{\leftarrow}_{j\leftarrow}$.

layers between the last honest relay on the forward and the first honest relay on the backward path with random ones. Note that the replaced part of the path contains an adversarial receiver. For the replacement in this case, we extend Tail-Indistinguishability from [26] with oracles for the creation of replies and illustrate the property *Repliable Tail-Indistinguishability* $TI^{\leftrightarrow}$ in Figure 3. As

we can already replace all other layers before and after this part of the path with random ones due to the Layer Unlinkability properties, the $TI^{\leftrightarrow}$ property does not output anything for these layers. We thus, start outputting the challenge layers only *after* the honest relay P_j on the forward path and refuse processing of the challenge onion at the honest relay on the backwards path $P^{\leftarrow}_{j\leftarrow}$ in our oracle (similar to $LU^{\leftarrow}$). The challenge onion hence either contains the layers after P_j of an onion build according to the adversary's choices or random layers that take the same part of the path and carry the same message, but for an onion that actually starts at P_j and ends (with the backwards path) at $P^{\leftarrow}_{j\leftarrow}$. For a formal definition of this property see Appendix C.2.

Properties imply ideal functionality As argued in the beginning of this section, we built the properties to step by step replace the real onion layers between honest relays with random ones that only coincide with the real ones in information that the ideal functionality allows to leak. By applying one property for each subpath between honest relays at a time, similar to earlier proofs [6,26], we show that these properties imply the ideal functionality in Appendix D. From here on, we call any OR scheme that fulfills our properties a *secure, repliable OR scheme*.

5 Our UE-based Scheme

5.1 Building Blocks

Our construction makes use of the generic building blocks listed below. Due to the page limit, we restrict to only elaborate on the less common and more complex building block of updatable encryption while referring to Appendix A for formal definitions of the more common building blocks.

- an asymmetric CCA2-secure encryption scheme (to encrypt ephemeral keys) with encryption and decryption algorithms denoted by $\mathsf{PK.Enc}_{PK_i}$ and $\mathsf{PK.Dec}_{SK_i}$ when used with public key PK_i and secret key SK_i,
- a PRP-CCA secure symmetric encryption scheme (to encrypt routing information) of length L_1 with encryption and decryption algorithms denoted by $\mathsf{PRP.Enc}_{k^\eta}$ and $\mathsf{PRP.Dec}_{k^\eta}$ when used with the symmetric key k^η,
- an SUF-CMA secure message authentication code (to protect the header) with tag generation and verification algorithm denoted by MAC_{k^γ} and Ver_{k^γ} when used with the symmetric key k^γ,
- a sufficiently secure (see below) updatable encryption scheme (to protect the payload) with encryption, decryption and re-encryption algorithms denoted by $\mathsf{UE.Enc}_{k^\Delta}$, $\mathsf{UE.Dec}_{k^\Delta}$ and $\mathsf{UE.ReEnc}_{\Delta}$ when used with keys k^Δ and tokens Δ . We assume that keys and tokens are of the same length or padded to the same length. Further, all messages are padded to the same length.

Updatable Encryption. Roughly speaking, an updatable encryption (UE) scheme is a symmetric encryption scheme with an extra re-encryption functionality moving ciphertexts from an old to a new key. In the following, we recapit-

ulate the definitions of an UE scheme providing RCCA security and plaintext integrity given in [24].

Security for UE is defined based on a notion of time which evolves in epochs. Data is encrypted with respect to a specific epoch e (starting with $e = 1$) using key k_e. When time advances from epoch e to $e + 1$, first a new key k_{e+1} is generated using $\mathsf{UE.GenKey}$ and then a token Δ_e is created using $\mathsf{UE.GenTok}$ on input of k_e and k_{e+1}. This token allows to update all ciphertexts from epoch e to $e + 1$ using the re-encryption algorithm $\mathsf{UE.ReEnc}$.

Definition 4 (Updatable Encryption [24]). *An* ***updatable encryption scheme*** UE *is a tuple* $(\mathsf{GenSP}, \mathsf{GenKey}, \mathsf{GenTok}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{ReEnc})$ *of PPT algorithms defined as:*

$\mathsf{UE.GenSP}(pp)$ *is given the public parameters and returns some system parameters sp. We treat sp as implicit input to all other algorithms.*
$\mathsf{UE.GenKey}(sp)$ *is the key generation algorithm which on input of the system parameters outputs a key* $k \in \mathcal{K}_{sp}$.
$\mathsf{UE.GenTok}(k_e, k_{e+1})$ *is given two keys* k_e *and* k_{e+1} *and outputs some update token* Δ_e.
$\mathsf{UE.Enc}(k_e, M)$ *is given a key* k_e *and a message* $M \in \mathcal{M}_{sp}$ *and outputs some ciphertext* $C_e \in \mathcal{C}_{sp}$ *(or* $\perp$ *in case* $M = \perp$*).*
$\mathsf{UE.Dec}(k_e, C_e)$ *is given a key* k_e *and a ciphertext* C_e *and outputs some message* $m \in \mathcal{M}_{sp}$ *or* $\perp$.
$\mathsf{UE.ReEnc}(\Delta_e, C_e)$ *is given an update token* Δ_e *and a ciphertext* C_e *and returns an updated ciphertext* C_{e+1} *or* $\perp$.

Given UE*, we call* $\mathsf{SKE} = (\mathsf{GenSP}, \mathsf{GenKey}, \mathsf{Enc}, \mathsf{Dec})$ *the* ***underlying (standard) encryption scheme****.* UE *is called* ***correct*** *if* SKE *is correct and it holds that* $\forall sp \leftarrow \mathsf{GenSP}(pp), \forall k^{\text{old}}, k^{\text{new}} \leftarrow \mathsf{GenKey}(sp), \forall \Delta \leftarrow \mathsf{GenTok}(k^{\text{old}}, k^{\text{new}})$, $\forall C \in \mathcal{C}\colon \mathsf{Dec}(k^{\text{new}}, \mathsf{ReEnc}(\Delta, C)) = \mathsf{Dec}(k^{\text{old}}, C)$.

RCCA Security. RCCA is a relaxed version of CCA where the decryption oracle ignores queries for ciphertexts containing the challenge messages m_0 or m_1. In particular, these ciphertexts could be re-randomizations of the challenge ciphertext. In the updatable encryption setting, the adversary is additionally given access to a re-encryption oracle and an oracle to adaptively corrupt secret keys and tokens of the current and past epochs. Trivial wins by means of corruption or re-encryption need to be excluded by the definition.

Definition 5 (UP-IND-RCCA [24]). UE *is called UP-IND-RCCA secure if for any PPT adversary* $\mathcal{A}$ *the following advantage is negligible in* κ*:*

$$\mathsf{Adv}^{\textsf{up-ind-rcca}}_{\mathsf{UE},\mathcal{A}}(pp) := \left|\Pr[\mathsf{Exp}^{\textsf{up-ind-rcca}}_{\mathsf{UE},\mathcal{A}}(pp, 0) = 1] - \Pr[\mathsf{Exp}^{\textsf{up-ind-rcca}}_{\mathsf{UE},\mathcal{A}}(pp, 1) = 1]\right|.$$

Experiment $\mathsf{Exp}^{\textsf{up-ind-rcca}}_{\mathsf{UE},\mathcal{A}}(pp, b)$
$(sp, k_1, \Delta_0, \mathbf{Q}, \mathbf{K}, \mathbf{T}, \mathbf{C}^*) \leftarrow \mathsf{Init}(pp)$
$(M_0, M_1, state) \leftarrow_{\text{R}} \mathcal{A}^{\mathsf{Enc},\mathsf{Dec},\mathsf{Next},\mathsf{ReEnc},\mathsf{Corrupt}}(sp)$
proceed only if $|M_0| = |M_1|$ and $M_0, M_1 \in \mathcal{M}_{sp}$

$C^* \leftarrow_{\mathrm{R}} \mathsf{UE.Enc}(k_e, M_b)$, $\mathrm{M}^* \leftarrow (M_0, M_1)$, $\mathbf{C}^* \leftarrow \{e\}$, $e^* \leftarrow e$
$b' \leftarrow_{\mathrm{R}} \mathcal{A}^{\mathsf{Enc},\mathsf{Dec},\mathsf{Next},\mathsf{ReEnc},\mathsf{Corrupt}}(C^*, state)$
return b' if $\mathbf{K} \cap \widehat{\mathbf{C}}^* = \emptyset$, i.e. $\mathcal{A}$ did not trivially win. (Else abort.)

In the above definition, the global state $(sp, k_e, \Delta_{e-1}, \mathbf{Q}, \mathbf{K}, \mathbf{T}, \mathbf{C}^*)$ is initialized by $\mathsf{Init}(pp)$ as follows:

$\mathsf{Init}(pp)$**:** Returns $(sp, k_1, \Delta_0, \mathbf{Q}, \mathbf{K}, \mathbf{T}, \mathbf{C}^*)$ where $e \leftarrow 1$, $sp \leftarrow_{\mathrm{R}} \mathsf{UE.GenSP}(pp)$, $k_1 \leftarrow_{\mathrm{R}} \mathsf{UE.GenKey}(sp)$, $\Delta_0 \leftarrow \perp$, $\mathbf{Q} \leftarrow \emptyset, \mathbf{K} \leftarrow \emptyset$, $\mathbf{T} \leftarrow \emptyset$ and $\mathbf{C}^* \leftarrow \emptyset$.

The list $\mathbf{Q}$ contains "legitimate" ciphertexts the adversary has obtained through Enc or ReEnc calls. The challenger also keeps track of epochs in which $\mathcal{A}$ corrupted a secret key ($\mathbf{K}$), token ($\mathbf{T}$), or obtained a re-encryption of the challenge-ciphertext ($\mathbf{C}^*$).

Moreover, the oracles given to the adversary are defined as follows:

$\mathsf{Next}()$**:** Runs $k_{e+1} \leftarrow_{\mathrm{R}} \mathsf{UE.GenKey}(sp)$, $\Delta_e \leftarrow_{\mathrm{R}} \mathsf{UE.GenTok}(k_e, k_{e+1})$, adds (k_{e+1}, Δ_e) to the global state and updates the current epoch to $e \leftarrow e + 1$.
$\mathsf{Enc}(M)$**:** Returns $C \leftarrow_{\mathrm{R}} \mathsf{UE.Enc}(k_e, M)$ and sets $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{(e, M, C)\}$.
$\mathsf{Dec}(C)$**:** If $\mathsf{isChallenge}(k_e, C) = \mathtt{false}$, it returns $m \leftarrow \mathsf{UE.Dec}(k_e, C)$, else `invalid`.
$\mathsf{ReEnc}(C, i)$**:** Returns C_e iteratively computed as $C_\ell \leftarrow_{\mathrm{R}} \mathsf{UE.ReEnc}(\Delta_{\ell-1}, C_{\ell-1})$ for $\ell = i + 1, \ldots, e$ and $C_i \leftarrow C$. It also updates the global state depending on whether the queried ciphertext is the challenge ciphertext or not:
- If $(i, M, C) \in \mathbf{Q}$ (for some m), then set $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{(e, M, C_e)\}$.
- Else, if $\mathsf{isChallenge}(k_i, C) = \mathtt{true}$, then set $\mathbf{C}^* \leftarrow \mathbf{C}^* \cup \{e\}$.

$\mathsf{Corrupt}(\{\mathsf{key}, \mathsf{token}\}, i)$**:** Allows corruption of keys and tokens, respectively:
- Upon input (key, i), the oracle sets $\mathbf{K} \leftarrow \mathbf{K} \cup \{i\}$ and returns k_i.
- Upon input (token, i), the oracle sets $\mathbf{T} \leftarrow \mathbf{T} \cup \{i\}$ and returns Δ_{i-1}.

The $\mathsf{isChallenge}$ predicate (used by Dec and ReEnc) is defined as:

$\mathsf{isChallenge}(k_i, C)$: If $\mathsf{UE.Dec}(k_i, C) \in \mathrm{M}^*$, return `true`. Else, return `false`.

To exclude trivial wins, we need to define the set of *challenge-equal epochs* containing all epochs in which the adversary obtains a version of the challenge ciphertext, either through oracle queries or by up/downgrading[14] the challenge ciphertext herself using a corrupted token.

$$\widehat{\mathbf{C}}^* \leftarrow \{e \in \{1, \ldots, e_{\mathsf{end}}\} \mid \mathsf{challenge\text{-}equal}(e) = \mathtt{true}\}$$
$$\text{and } \mathtt{true} \leftarrow \mathsf{challenge\text{-}equal}(e) \text{ iff: } (e \in \mathbf{C}^*) \vee$$
$$(\mathsf{challenge\text{-}equal}(e-1) \wedge e \in \mathbf{T}) \ \vee \ (\mathsf{challenge\text{-}equal}(e+1) \wedge e+1 \in \mathbf{T})$$

The adversary can trivially win UP-IND-RCCA by corrupting the key in any challenge-equal epoch. This is excluded by the UP-IND-RCCA definition.

Perfect Re-encryption. Intuitively, perfect re-encryption demands that fresh and re-encrypted ciphertexts are indistinguishable. This is defined by requiring that decrypt-then-encrypt has the same distribution as re-encryption.

[14] We assume that a token Δ_e also enables *downgrades* of ciphertexts from epoch $e+1$ to epoch e.

Definition 6 (Perfect Re-encryption [24]). *Let* UE *be an updatable encryption scheme where* $\mathsf{UE.ReEnc}$ *is* probabilistic. *We say that re-encryption (of* UE*) is* ***perfect****, if for all* $sp \leftarrow_R \mathsf{UE.GenSP}(pp)$*, all keys* $k^{\text{old}}, k^{\text{new}} \leftarrow_R \mathsf{UE.GenKey}(sp)$*, token* $\Delta \leftarrow_R \mathsf{UE.GenTok}(k^{\text{old}}, k^{\text{new}})$*, and all ciphertexts* C*, we have*

$$\mathsf{UE.Enc}(k^{\text{new}}, \mathsf{UE.Dec}(k^{\text{old}}, C)) \stackrel{\text{dist}}{\equiv} \mathsf{UE.ReEnc}(\Delta, C).$$

In particular, note that $\mathsf{ReEnc}(\Delta, C) = \bot \Leftrightarrow \mathsf{Dec}(k^{\text{old}}, C) = \bot$.

Plaintext Integrity. Plaintext integrity demands that the adversary cannot produce a ciphertext decrypting to a message for which she does not trivially know an encryption.

Definition 7 (UP-INT-PTXT [24]). UE *is called UP-INT-PTXT secure if for any PPT adversary* $\mathcal{A}$ *the following advantage is negligible in* κ*:*
$\mathsf{Adv}^{\textit{up-int-ptxt}}_{\mathsf{UE},\mathcal{A}}(pp) := \Pr[\mathsf{Exp}^{\textit{up-int-ptxt}}_{\mathsf{UE},\mathcal{A}}(pp) = 1]$.

Experiment $\mathsf{Exp}^{\mathsf{up\text{-}int\text{-}ptxt}}_{\mathsf{UE},\mathcal{A}}(pp)$
$(sp, k_1, \Delta_0, \mathbf{Q}, \mathbf{K}, \mathbf{T}) \leftarrow \mathsf{Init}(pp)$
$c^* \leftarrow_{\text{R}} \mathcal{A}^{\mathsf{Enc},\mathsf{Dec},\mathsf{Next},\mathsf{ReEnc},\mathsf{Corrupt}}(sp)$
return 1 if $\mathsf{UE.Dec}(k_{e_{\mathsf{end}}}, c^*) = m^* \neq \bot$ and $(e_{\mathsf{end}}, m^*) \notin \mathbf{Q}^*$,
and $\nexists e \in \mathbf{K}$ where $i \in \mathbf{T}$ for $i = e$ to e_{end}; i.e. if $\mathcal{A}$ does not trivially win.

The oracles provided to the adversary are defined as follows:

$\mathsf{Next}()$, $\mathsf{Corrupt}(\{\mathsf{key}, \mathsf{token}\}, i)$**:** as in CCA game
$\mathsf{Enc}(M)$**:** Returns $C \leftarrow_{\text{R}} \mathsf{UE.Enc}(k_e, M)$ and sets $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{(e, M)\}$.
$\mathsf{Dec}(C)$**:** Returns $m \leftarrow \mathsf{UE.Dec}(k_e, C)$ and sets $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{(e, M)\}$.
$\mathsf{ReEnc}(C, i)$**:** Returns C_e, the re-encryption of C from epoch i to the current epoch e. It also sets $\mathbf{Q} \leftarrow \mathbf{Q} \cup \{(e, M)\}$ where $M \leftarrow \mathsf{UE.Dec}(k_e, C_e)$.

To exclude trivial wins, we define the set $\mathbf{Q}^*$ which contains all plaintexts (and epochs) the adversary has received a ciphertext for by means of Enc and ReEnc queries or by upgrading a ciphertext herself using a corrupted token.

for each $(e, m) \in \mathbf{Q}$:
set $\mathbf{Q}^* \leftarrow \mathbf{Q}^* \cup (e, m)$, and $i \leftarrow e + 1$
while $i \in \mathbf{T}$: set $\mathbf{Q}^* \leftarrow \mathbf{Q}^* \cup (i, m)$ and $i \leftarrow i + 1$

The adversary trivially wins if her output decrypts to a message m such that (e_{end}, m) is contained in this set or if she has corrupted a secret key and all following tokens, as this allows to create valid ciphertexts for any plaintext.

5.2 Scheme Description

The basic idea is to share the update tokens for the payload with intermediate relays and the encryption key with the receiver. So, the payload in each layer is encrypted under a different key that only the sender knows (see Fig. 4). To realize this, we need to construct a header that transports the tokens and routing

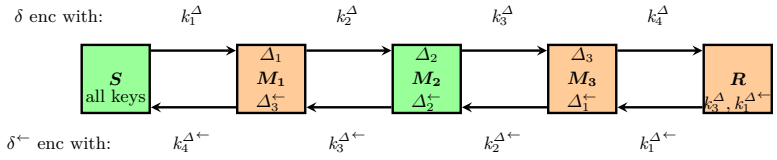

Fig. 4. Overview of the basic idea for the payload δ. Each relay gets an updatable encryption token Δ to change the key k^{Δ} under which the payload is encrypted.

information while ensuring that headers of different layers of the same onion cannot be linked to each other.

Setup To setup the system, $\mathsf{UE.GenSP}(pp)$ needs to be run on the public parameters pp (which, e.g., may contain a description of the group setting) by an honest party (or by using multi-party computation). The resulting system parameters sp (which, e.g., may contain a Groth-Sahai CRS) need to be made public and used by all participating parties. Usually they would be distributed along with the software package.

Header Construction. Each onion layer O_i, which is sent from P_{i-1} to P_i, is a tuple of header η_i and payload δ_i: $O_i = (\eta_i, \delta_i)$. Constructing the header is inspired by the Sphinx approach [13] and the Shallot scheme [3]. Contrary to the existing works, we however treat the payload with sufficiently secure updatable encryption.

Each header η_i is a tuple of encrypted temporary keys and tokens in E_i, encrypted routing information and keys for the current router P_i and later routers $P_{>i}$ in B_i^j and a MAC over the header in γ_i: $\eta_i = (E_i, B_i^1, B_i^2, \ldots, B_i^{2N-1}, \gamma_i)$. We describe a non-repliable header first and later on extend it to be repliable. The first layer's header η_1 contains:

$$\eta_1 = (\quad E_1, \quad B_1^1, \quad B_1^2 \quad, \ldots, \quad B_1^{2N-1}, \quad \gamma_1 \quad)$$

$$\eta_1 = (\mathsf{PK.Enc}_{PK_1}(k_1^{\eta}, k_1^{\gamma}, \Delta_1), \mathsf{PRP.Enc}_{k_1^{\eta}}(P_2, E_2, \gamma_2), \mathsf{PRP.Enc}_{k_1^{\eta}}(B_2^1), \ldots, \mathsf{PRP.Enc}_{k_1^{\eta}}(B_2^{2N-2}), \mathsf{MAC}_{k_1^{\gamma}}(E_1, B_1^1, \ldots, B_1^{2N-1}))$$

The second layer's header η_2 has padding added by the first relay in B_2^{2N-1}:

$$\eta_2 = (\quad E_2, \quad B_2^1, \quad B_2^2 \quad, \ldots, \quad B_2^{2N-1}, \quad \gamma_2 \quad)$$

$$\eta_2 = (\mathsf{PK.Enc}_{PK_2}(k_2^{\eta}, k_2^{\gamma}, \Delta_2), \mathsf{PRP.Enc}_{k_2^{\eta}}(P_3, E_3, \gamma_3), \mathsf{PRP.Enc}_{k_2^{\eta}}(B_3^1), \ldots, \mathsf{PRP.Dec}_{k_1^{\eta}}(\mathbf{0}\ldots\mathbf{0}), \mathsf{MAC}_{k_2^{\gamma}}(E_2, B_2^1, \ldots, B_2^{2N-1}))$$

The already existing relay padding is further decrypted for later layers:

$$\eta_3 = (\ldots, \quad B_3^{2N-3} \quad , \quad B_3^{2N-2} \quad , \quad B_3^{2N-1} \quad, \ldots)$$

$$\eta_3 = (\ldots, \mathsf{PRP.Enc}_{k_3^{\eta}}(B_4^{2N-4}), \mathsf{PRP.Dec}_{k_2^{\eta}}(\mathsf{PRP.Dec}_{k_1^{\eta}}(\mathbf{0}\ldots\mathbf{0})), \mathsf{PRP.Dec}_{k_2^{\eta}}(0\ldots0), \ldots)$$

The message is destined for the current processing relay P_{n+1} if $(\perp, \perp, \perp)$ is encrypted in B_{n+1}^1. All later $B_{n+1}^{>1}$ contain random bit strings chosen by the sender resp. the padding added by the earlier relays (see Figure 5). The blocks with sender chosen padding are used for the reply path in repliable onions later.

To construct η_1 for a path $\mathcal{P} = (P_1, \ldots, P_{n+1}), n+1 \leq N-1$, the sender builds the onion from the center, i.e. calculates the layer for the receiver first:

1. Pick keys $k_1^{\eta}, \ldots, k_{n+1}^{\eta}$ for the block cipher, $k_1^{\Delta}, \Delta_1, \ldots, \Delta_n, k_{n+1}^{\Delta}$ for the UE and $k_1^{\gamma}, \ldots, k_{n+1}^{\gamma}$ for the MAC randomly.

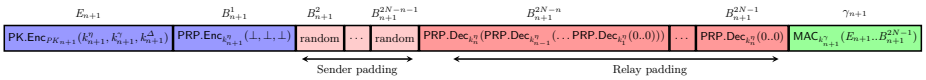

Fig. 5. Non-repliable receiver header illustrated

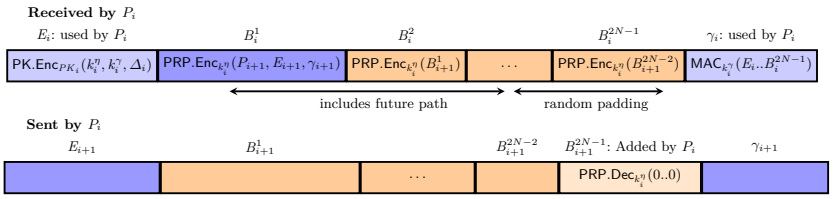

Fig. 6. Processing illustrated

2. Construct η_{n+1}:

$$E_{n+1} = \mathsf{PK.Enc}_{PK_{n+1}}(k^{\eta}_{n+1}, k^{\gamma}_{n+1}, k^{\Delta}_{n+1})$$
$$B^1_{n+1} = \mathsf{PRP.Enc}_{k^{\eta}_{n+1}}(\bot, \bot, \bot)$$
$$B^{2N-i}_{n+1} = \mathsf{PRP.Dec}_{k^{\eta}_n}(\mathsf{PRP.Dec}_{k^{\eta}_{n-1}}(\ldots \mathsf{PRP.Dec}_{k^{\eta}_{n+1-i}}(0 \ldots 0)))$$
for $1 \le i \le n$ (blocks appended by relays)
$$B^{2N-i}_{n+1} \leftarrow^R \{0,1\}^{L_1} \text{ for } n+1 \le i \le 2N-2$$
(blocks as path length padding calculated by sender)
$$\gamma_{n+1} = \mathsf{MAC}_{k^{\gamma}_{n+1}}(E_{n+1}, B^1_{n+1}, B^2_{n+1}, \ldots, B^{2N-1}_{n+1})$$

3. Construct $\eta_i, i < n+1$ recursively (from $i = n$ to $i = 1$):

$$E_i = \mathsf{PK.Enc}_{PK_i}(k^{\eta}_i, k^{\gamma}_i, \Delta_i)$$
$$B^1_i = \mathsf{PRP.Enc}_{k^{\eta}_i}(P_{i+1}, E_{i+1}, \gamma_{i+1})$$
$$B^j_i = \mathsf{PRP.Enc}_{k^{\eta}_i}(B^{j-1}_{i+1}) \text{ for } 2 \le i \le 2N-1$$
$$\gamma_i = \mathsf{MAC}_{k^{\gamma}_i}(E_i, B^1_i, B^2_i, \ldots, B^{2N-1}_i)$$

Payload Construction. Let m be a message of the fixed message length to be sent. We add a 0 bit to the message to signal that it is not repliable $m' = 0 \| m$: $\delta_i = \mathsf{UE.ReEnc}_{\Delta_{i-1}}(\ldots(\mathsf{UE.ReEnc}_{\Delta_1}(\mathsf{UE.Enc}_{k^{\Delta}_1}(m'))\ldots)$.

Onion Processing. The same processing is used for any forward or backward, repliable or not-repliable onion. If P_i receives an onion $O_i = (\eta_i = (E_i, B^1_i, B^2_i, \ldots, B^{2N-1}_i, \gamma_i), \delta_i))$, it takes the following steps (see Fig. 6):

1. Decrypt the first part of the header $(k^{\eta}_i, k^{\gamma}_i, \Delta_i) = \mathsf{PK.Dec}_{PK_i}(E_i)$ [resp. k^{Δ}_{n+1} instead of Δ_i, if P_i is the receiver]
2. Check the MAC γ_i of the received onion (and abort if it fails)
3. Decrypt the second part of the header $(P_{i+1}, E_{i+1}, \gamma_{i+1}) = \mathsf{PRP.Dec}_{k^{\eta}_i}(B^1_i)$ [if $P_{i+1} = E_{i+1} = \gamma_{i+1} = \bot$ (P_i is the receiver), skip processing of the header and process the payload (and check for replies as explained below)]

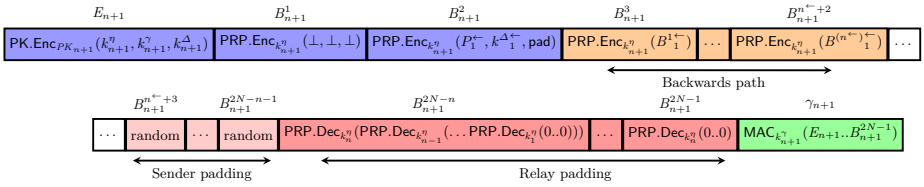

Fig. 7. Repliable receiver header illustrated

4. Decrypt the rest of the header $B_{i+1}^{j-1} = \mathsf{PRP.Dec}_{k_i^\eta}(B_i^j)$ for $j \geq 2$
5. Pad the new header $B_{i+1}^{2N-1} = \mathsf{PRP.Dec}_{k_i^\eta}(0 \dots 0)$
6. Construct the new payload $\delta_{i+1} = \mathsf{UE.ReEnc}_{\Delta_i}(\delta_i)$ [resp. retrieve the message in case of being the receiver ($\delta_{i+1} = \mathsf{UE.Dec}_{k_{n+1}^\Delta} = 0 \| m$ if no reply)] and abort if this fails
7. Send the new onion $O_{i+1} = ((E_{i+1}, B_{i+1}^1, \dots, B_{i+1}^{2N-1}, \gamma_{i+1}), \delta_{i+1})$ to the next relay P_{i+1}

Constructing a Repliable Onion. Let m be the message for the receiver, $\mathcal{P}^\leftarrow = (P_1^\leftarrow, \dots, P_{n^\leftarrow+1}^\leftarrow), n^\leftarrow + 1 \leq N-1$ the backward path. To send a repliable onion, the sender performs the following steps:

1. Construct a (non-repliable) header $\eta_1^\leftarrow$ with path $\mathcal{P}^\leftarrow$. Let the chosen keys be $k^{\eta}{}_1^\leftarrow, \dots, k^{\eta}{}_{n^\leftarrow+1}^\leftarrow$ and $k^{\Delta}{}_1^\leftarrow, \Delta_1^\leftarrow, \dots, \Delta_{n^\leftarrow}^\leftarrow, k^{\Delta}{}_{n^\leftarrow+1}^\leftarrow$.
2. Construct the (repliable) header η_1 by starting to construct η_{n+1} for the receiver as before in the non-repliable case, but with the following differences (see Fig. 7) with pad being padding to the fixed blocklength:
   - Set $B_{n+1}^2 = \mathsf{PRP.Enc}_{k_{n+1}^\eta}(P_1^\leftarrow, k^{\Delta}{}_1^\leftarrow, \mathsf{pad})$.
   - Set $B_{n+1}^i = \mathsf{PRP.Enc}_{k_{n+1}^\eta}(B^{(i-2)}{}_1^\leftarrow)$ for $3 \leq i \leq n^\leftarrow + 2$.
   - Store the key $k^{\Delta}{}_{n^\leftarrow+1}^\leftarrow$
3. Evolve the header η_{n+1} as before to create η_1
4. Construct the message for the repliable onion as $m' = 1 \| m$.
5. Construct the payload δ_1 for m' as before.
6. The repliable onion is (η_1, δ_1).

Sending a reply. After recognizing to be the receiver (due to $(\perp, \perp, \perp)$ in B^1) of an repliable message (due to the starting bit), the receiver retrieves $P_1^\leftarrow$ and $k^{\Delta}{}_1^\leftarrow$ from B_{n+1}^2. Let $m^\leftarrow$ be the reply message padded to the fixed message length. To send the reply the receiver performs the following steps:

1. Calculate $\delta_1^\leftarrow = \mathsf{UE.Enc}_{k^{\Delta}{}_1^\leftarrow}(m^\leftarrow)$
2. Evolve the header (as before but shifting the header by two blocks):
   - $B^{(j-2)}{}_1^\leftarrow = \mathsf{PRP.Dec}_{k_{n+1}^\eta}(B_{n+1}^j)$ for $j \geq 3$
   - $B^{(2N-1)}{}_1^\leftarrow = \mathsf{PRP.Dec}_{k_{n+1}^\eta}(0 \dots 0)$ (i.e. receiver padding)
   - $B^{(2N-2)}{}_1^\leftarrow = \mathsf{PRP.Dec}_{k_{n+1}^\eta}(1 \dots 1)$ (i.e. receiver padding)
3. Send the onion $O_1^\leftarrow = (\eta_1^\leftarrow, \delta_1^\leftarrow)$ to $P_1^\leftarrow$

Decrypting a reply. After recognizing to be the receiver (due to $(\perp, \perp, \perp)$ in B^1), the relay checks whether the included key k_{n+1}^{Δ} for her matches a stored $k^{\Delta}{}_{n^{\leftarrow}+1}^{\leftarrow}$s (it indeed is a reply) or not (it is just a new message). She uses the key and decrypts the message: $m = \mathsf{UE.Dec}_{k_{n+1}^{\Delta}}(\delta)$.

6 Security of Our Repliable OR Scheme

In this section, we prove that our scheme is secure:

Theorem 1. *Let us assume a PK-CCA2 secure PKE, a PRP-CCA secure SKE, a UP-IND-RCCA, and UP-INT-PTXT secure UE scheme with perfect Re-Encryption (of arbitrary ciphertexts), and a SUF-CMA secure MAC are given. Then our construction described in Section 5 satisfies $LU^{\rightarrow}$ security.*

Intuitively, the PK-CCA2 secure PKE ensures that the temporary keys for each relay are only learned by the intended relay, and the PRP-CCA secure SKE that the header is rerandomized and can be padded in the processing at a relay (so incoming and outgoing onions cannot be linked based on the header). Further, the SUF-CMA secure MAC protects the header against modifications. The UE scheme takes care of the payload: the UP-IND-RCCA ensures that the message is hidden and that the payload is rerandomized during the processing at a relay (so incoming and outgoing onions cannot be linked based on the payload), UP-INT-PTXT security that the payload cannot be maliciously modified (as in the malleability attack), while Perfect Re-Encryption guarantees that the adversary does not learn how far on the path the onion has already traveled.

Formally, we first describe FormOnion for later layers and show a detailed proof sketch for $LU^{\rightarrow}$. As the proofs for $LU^{\leftarrow}$ and $TI^{\leftrightarrow}$ are similar to the one of $LU^{\rightarrow}$, we only quickly sketch them here. All detailed proofs are provided in Appendix E. Further, correctness follows from inspection of our scheme.

FormOnion - later layers. FormOnion for $i > 1$ uses the k_i^{Δ} belonging to the corresponding epoch to create the payload $\delta = \mathsf{UE.Enc}_{k_i^{\Delta}}(m)$ and creates the other onion parts deterministically as described in the protocol for the current layer (with the randomness, all used keys are known and the deterministic parts of all layers can be built). For reply layers ($i > n+1$) it combines the deterministically computed header and payload with the encrypted new message[15] (as all randomness is known, all temporary keys are).

Forwards Layer Unlinkability. Our proof for $LU^{\rightarrow}$ follows a standard hybrid argument. We distinguish the cases that the honest node is a forward relay ($j < n+1$) and that it is the receiver ($j = n+1$).

Case 1 – Honest Relay ($j < n+1$). We first replace the temporary keys of the honest party included in the header, to be able to change the blocks of the header and the payload corresponding to the $b = 1$ case. For the oracles we further need to ensure, that RecognizeOnion does not mistreat any processing of

[15] We use the parameter m of FormOnion for the reply message if $i > n+1$, as the forward message is not needed to construct the reply.

e.g. modified onions. Therefore, we leverage the UE properties for the payload protection and the MAC for the header. Table 2 and Table 3 in Appendix E provide overviews of the proof.

Proof Sketch We assume a fixed, but arbitrary PPT algorithm $\mathcal{A}_{LU^{\rightarrow}}$ as adversary against the $LU^{\rightarrow}$ game and use a sequence of hybrid games $\mathcal{H}$ for our proof. We show that the probability of $\mathcal{A}_{LU^{\rightarrow}}$ outputting $b' = 1$ in the first and last hybrid are negligibly close to each other.

Hybrid 1) $LU^{\rightarrow}_{(b=0)}$**.** The $LU^{\rightarrow}$ game with b chosen as 0.

Hybrid 2) replaces the keys and token included in E_j with $0 \dots 0$ before encrypting them and adapts the oracle of step 7 such that RecognizeOnion checks for the adapted header, but still uses the original keys as decryption of E_j.

We reduce this to the PK-CCA2 security of our PK encryption: We either embed $0 \dots 0$ or the keys and token as the $CCA2$ challenge message and process other onions (for the step 7 oracle) by using the $CCA2$ decryption oracle.

Hybrid 3) rejects all onions that reuse E_j, but differ in another part of the header, in the oracle of step 7.

Due to the SUF-CMA of our MAC a successful processing of a modified header can only occur with negligible probability.

Hybrid 4) replaces the blocks (with information and keys for the future path of the onion) with random blocks and adapts the oracle of step 7 such that RecognizeOnion checks for the adapted header, but still uses the original blocks as processing result.

We reduce this to the PRP-CCA security of the PRP, by embedding the PRP-CCA challenge into these blocks, while continuing to treat these same blocks during processing as if they had the original content. (Other blocks in onions using E_j are rejected in the oracle of step 7.)

Hybrid 5) replies with a fail to all step 7 oracle requests, that use the challenge onion's header, but modified the message included in its payload.

We reduce this to the UP-INT-PTXT of our UE: First, we carefully construct the secrets of the challenge onion until it is at the honest relay with the help of the UP-INT-PTXT-oracles. Then we wait for an onion with the challenge header to be given to the oracle in step 7. We use the payload of this onion as the ciphertext to break UP-INT-PTXT. Note that we do not have to answer this oracle request in our reduction, but only oracle requests for onions with a different header, which we can easily process with the knowledge of the secret keys (only the keys for the challenge onion are partially unknown in the reduction).

Hybrid 6) replaces the processing result of the challenge onion (recognized based on the header, with an unchanged message in payload) with a newly formed onion (FormOnion) that includes the same rest of the path and message.

FormOnion constructs the header deterministically as before, the only difference is the re-encryption (Hybrid 5) and the fresh encryption (Hybrid 6) of the same message in the payload. Due to the perfect Re-Encryption of our UE scheme those are indistinguishable.

Hybrid 7) replaces the message included in the payload with a random message and adapts the oracle in step 7 to expect this random message as payload, but still replies with the newly formed onion including the original message as before.

We reduce this to the UP-IND-RCCA security of our UE: We carefully construct the secrets of the challenge onion until the honest relay with the help of the UP-IND-RCCA-oracles and either embed the original or a random message as the UP-IND-RCCA challenge message. To answer the step 7 oracle, we use the knowledge of the secret keys if the requested onion does not have the challenge onion's header. If it has, we use the decryption oracle of UP-IND-RCCA to detect whether the payload was maliciously modified (the UP-IND-RCCA oracle returns another message m') or not (the UP-IND-RCCA oracle does not process the payload). In the first case, we return a fail (as introduced in Hybrid 5), in the second we return a newly formed onion (as introduced in Hybrid 6).

Hybrid 8) - Hybrid 12) revert the hybrids 5)-2) (similar argumentation).

Case 2 – Honest Receiver ($j = n + 1$): We sketch the proof in Table 4 and 5 of Appendix E. The steps are the same as for the first case of $LU^{\rightarrow}$, but in Hybrid 6) we need to treat Reply and Proc requests separately. As the FormOnion behavior simulating the receiver is exactly the same as in the real protocol, we do not need to rely on Perfect Re-Encryption, but just on correctness of the decryption in this step. Note further that the earlier restrictions on the oracle work both for Reply and Proc requests.

Other Properties. We sketch the proofs in Table 6 – 11 of Appendix E.

Theorem 2. *Let us assume a PK-CCA2 secure PKE, a PRP-CCA secure SKE and a UP-IND-RCCA secure UE scheme with perfect Re-Encryption (of arbitrary ciphertexts), and a SUF-CMA secure MAC are given. Then our construction described in Section 5 satisfies $LU^{\leftarrow}$ security.*

Backwards Layer Unlinkability. The steps are similar to the ones for $LU^{\rightarrow}$ Case 1: We replace the temporary keys of honest routers, before we exclude bad events (header manipulations) at the oracles and finally set the header and payload parts to correspond to the $b = 1$ case. However, this time we need to replace parts for both at the forward and backward path, as the forward layers also include information about the backward layers (but not the other way round). Notice that we can skip the steps related to the modification of the payload (and thus UP-INT-PTXT). As the forward message is known to the adversary anyways and the backward message (as the final processing) is never given to the adversary, she cannot exploit payload modification at the oracles to break $LU^{\leftarrow}$.

Theorem 3. *Let us assume a PK-CCA2 secure PKE, a PRP-CCA secure SKE, and a SUF-CMA secure MAC are given. Then our construction described in Section 5 satisfies $TI^{\leftrightarrow}$ security.*

Tail Indistinguishability. This is similar to $LU^{\leftarrow}$, except that we can skip more steps. For the same reasons as before, we do not need the payload protection in $TI^{\leftrightarrow}$. Further, the adversary does not obtain any leakage related to k_j^{η} and thus the blocks in the forward header can be replaced right away.

7 Our SNARG-based Scheme

We now present an alternative instantiation of a secure, repliable OR scheme based on SNARGs, instead of updatable encryption.

7.1 Building Blocks and Setting

We make use of the following cryptographic building blocks and emphasize the differences compared to the UE-based scheme (see Appendix A for details):

- an asymmetric CCA2-secure encryption scheme with encryption and decryption algorithms denoted by $\mathsf{PK.Enc}_{PK_i}$ and $\mathsf{PK.Dec}_{SK_i}$ when used with public key PK_i and secret key SK_i.
- an SUF-CMA secure message authentication code with tag generation algorithm denoted by MAC_{k^γ} when used with the symmetric key k^γ.
- *two* PRP-CCA secure symmetric encryption schemes of short length L_1 (for the header) and long length L_2 (for the payload) with encryption and decryption algorithms denoted by $\mathsf{PRP.Enc}_{k^\eta}$ and $\mathsf{PRP.Dec}_{k^\eta}$ resp. $\mathsf{PRP2.Enc}_{k^\delta}$ and $\mathsf{PRP2.Dec}_{k^\delta}$ when used with the symmetric key k^η resp. k^δ.
- a *re-randomizable IND-CPA secure asymmetric encryption scheme*, with encryption, decryption, and re-randomization algorithms denoted by $\mathsf{PKM.Enc}_{PK^{\mathrm{M}}}$, $\mathsf{PKM.Dec}_{SK^{\mathrm{M}}}$, $\mathsf{PKM.ReRand}_{PK^{\mathrm{M}}}$ when used with public key PK^{M} and secret key SK^{M}. We require that re-randomization is invertible, in the sense that knowing the random coins of $\mathsf{PKM.ReRand}$ allows to retrieve the original ciphertext.
- a *simulation-sound SNARG* with proof generation, verification, and simulation algorithms denoted by $\mathsf{Prove}_{\mathsf{ZK}}$, $\mathsf{Vfy}_{\mathsf{ZK}}$, and $\mathsf{Sim}_{\mathsf{ZK}}$.

We assume that all keys of honest participants are chosen independently at random.

Regarding the setting, we assume additionally that

- a master public key PK^{M} (for the re-randomizable IND-CPA secure encryption) and a common reference string CRS (for the SNARG) are known to all participants, while the corresponding SNARG trapdoor and secret key SK^{M} are not known to anyone.[16]

We will use PK^{M} to let participants encrypt secrets "to the sky", and the corresponding secret key SK^{M} will only be used as an extraction trapdoor in our proof. Hence, it is crucial that in an implementation of our scheme, both PK^{M} and CRS are chosen such that noone knows their trapdoors. (However, at least in the case of CRS, subversion-zero-knowledge SNARKs [17] are a promising tool to allow for adversarially chosen CRS.)

[16] Those public parameters can be either chosen by a trusted party, agreed upon with an initial multi-party computation, or, if SNARG and the re-randomizable encryption scheme have dense keys, be derived from a public source of trusted randomness (like, e.g., sunspots).

7.2 Scheme Description

Overview Each router (publicly) proves at each step of the protocol that the *whole* current onion (including payload) is consistent, in the sense that it is the result of a faithful processing of a previous onion. This proof is realized with a succinct non-interactive argument of knowledge (SNARG [5]). This in fact presents us with a minor technical challenge, since now proving consistency involves proving that a previous onion *with a valid consistency proof* exists.

Why we do not use SNARK extraction. In our security proofs, such a consistency proof will be used to reconstruct previous onions (and in fact the whole past of an onion) by using the soundness of the SNARG. We stress that we will not be using any extractability properties from the SNARG (i.e., we do not rely on any knowledge soundness properties) at this point, since this would need to extract recursively. Indeed, in our proofs, we will need to simulate a ProcOnion oracle on adversarial inputs (i.e., onions) without knowing the underlying secret key. Instead, we will "reverse-process" the given onion until its creation with FormOnion, and then extract all future onions from the initial FormOnion inputs.

Our crucial tool to enable this "reverse-processing" is the soundness of the used SNARG. Intuitively, it seems possible to use a SNARK (i.e., a succinct argument of *knowledge*, which allows extraction of a witness) to prove that this onion has been created or processed honestly, with the witness being the corresponding FormOnion, resp. ProcOnion input. The problem with this approach is that each proof only certifies a single processing step, and so we would have to extract SNARK witnesses multiple times, and in fact *recursively extract* (which is notoriously difficult).

Instead, each onion will carry enough encrypted information to recreate previous onions, and the corresponding SNARG will certify the validity of that (encrypted) information. (Since the size of onions should not grow during processing, we will not be able to *fully* reconstruct the previous onion. However, we will still be able to implement the above strategy.) Like before, we rely on using MACs for a more "fine-grained" (and, most importantly, deterministic) authentication and progression of onion *headers*.

Viewed from a higher level, these consistency proofs provide a whole authentication chain for both requests and replies even with an intermediate receiver that replies with an arbitrary (and a-priori unknown) payload. This authentication chain protects against malleability attacks and payload changes along the way.

More details In our protocol, each onion $O = (\eta, \sigma, \delta)$ consists of three main components:

- a *header* η which contains encrypted key material with which routers can process and (conventionally) authenticate the onion,
- the (SNARG-related) *authentication part* σ,
- the multiply encrypted *payload* δ; each router will decrypt one layer during processing.

While η and δ are similar to the Sphinx and Shallot protocols, σ contains several SNARG proofs $\pi_1, \ldots, \pi_N$ and an encrypted ring buffer (that consists of ciphertexts $C_1, \ldots, C_N$). Here, N denotes the maximal path length in the scheme. Intuitively, the C_i contain information that is required to reverse-process O, and the π_i prove that the information encrypted in C_1 is accurate. More specifically:

- C_1 contains a public-key encryption of the $\pi'_1, \ldots, \pi'_N$ from the *previous* onion O', as well as the last router's long-term secret key SK'. The public key used is a public parameter of the OR scheme, such that the secret key is not known by anyone. Of course, this last property is crucial to the security of the scheme. We will use this secret key as a trapdoor that allows to reverse-process onions during the proof.
- $C_2, \ldots, C_N$ are the values $C'_1, \ldots, C'_{N-1}$ from O'. Note that this implies that C'_N is lost during processing and cannot be reconstructed.
- π_i is a SNARG proof that proves that η, δ, and $C_1, \ldots, C_{N-i}$ are the result of an honest processing of some previous onion. The reason for N proofs π_i (and not just a single one) is that during repeated reverse-processing of a given onion, more and more C_i will unavoidably be lost. To check the integrity of such incomplete onions, we will use π_i in the i-th reverse-processing step.

Header Construction Each onion layer O_i is a tuple of header η_i, SNARG-Information σ_i and payload δ_i: $O_i = (\eta_i, \sigma_i, \delta_i)$. We construct the header η_i as in the UE-based solution (see Section 5.2), except that instead of the Δ_i resp. k_i^{Δ} we now include k_i^{δ} of the second PRP-CCA secure symmetric encryption scheme for the relays.

SNARG Construction: The SNARG-Information σ_i consists of a ring buffer $C_i = (C_i^1, \ldots, C_i^N)$ and the SNARGs $\pi_i = (\pi_i^1, \ldots \pi_i^N)$: $\sigma_i = (C_i, \pi_i)$.

Ring buffer. The ring buffer C_i is calculated similarly to B_i, but reversed. The ring buffer for forward onions includes all information needed to undo the processing of the onion or reconstruct all input to FormOnion, encrypted under the master public key. On the reply path, we overwrite old (forward) information in C_is, as this is sufficient to achieve the forward-backward indistinguishability.

$$C_1^1 = \mathsf{PKM.Enc}_{PK^{\mathrm{M}}}(\mathcal{I}) \text{ with } \mathcal{I} = (\texttt{form}, (R, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}))$$

$$C_1^j \leftarrow^R \{0,1\}^{L_3} \setminus \{\texttt{sim}\} \text{ with } \texttt{sim} \text{ being a special symbol and } L_3 \text{ the fixed length of ring buffer elements}$$

$$C_i^1 = \mathsf{PKM.Enc}_{PK^{\mathrm{M}}}(\mathcal{I}) \text{ with } \mathcal{I} = (\texttt{proc}, (SK_{i-1}, \pi_{i-1}^1, \ldots, \pi_{i-1}^N, E_{i-1}, P_{i-1}))$$

$$C_i^j = \mathsf{PKM.ReRand}_{PK^{\mathrm{M}}}(C_{i-1}^{j-1})$$

Note that the onion O_i is created by P_{i-1} and thus the information included in C_i is known at the time of creation. Further, information encrypted in C_i does not include the payload message or the MAC, as both an be reconstructed given the current onion layer. Finally, all C_i^j are padded to the fixed length L_3.

SNARGs. The SNARG π_i^j is calculated by P_{i-1} for the language $\mathcal{L}^j$, which consists of all partial onions $X = (\eta_i, (C_i^1, \ldots, C_i^{N-j}), \delta_i)$ for which the following

holds: namely, there should exist R, M such that $C_i^1 = \mathsf{Enc}(PK^{\mathrm{M}}, M; R)$, and such that M fulfills the following:

1. If M is of the form $M = (\mathtt{form}, I)$, then I is some parameter list $I = (1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})$ (including random coins $\mathcal{R}$) for which FormOnion(I) outputs an onion $O^* = (\eta^*, \sigma^*, \delta^*)$ with $\eta^* = \eta_i$ and $\delta^* = \delta_i$. In other words, in this case, M explains X as an immediate FormOnion output for a particular message m.
2. If M is of the form $M = (\mathtt{proc}, (SK_{i-1}, \pi_{i-1}^1, \ldots, \pi_{i-1}^N, E_{i-1}, P_{i-1}, \mathcal{R}))$, then
   (a) all π_{i-1}^{N-k} (for $k > j$) are valid, in the sense that π_{i-1}^{N-k} shows that $(\eta_i, (C_i^1, \ldots, C_i^{N-k}), \delta_i) \in \mathcal{L}^k$. (Note that this is a well-defined statement if we define $\mathcal{L}^j$ for larger j first.)
   (b) $\mathrm{ProcOnion}^j_{\mathrm{partial}}(SK_{i-1}, (\eta_{i-1}, (C_{i-1}^1, \ldots, C_{i-1}^{N-j-1}), \delta_{i-1}), P_{i-1}; \mathcal{R}) = (\eta_i, (C_i^1, \ldots, C_i^{N-j}), \delta_i)$, where $\mathrm{ProcOnion}^j_{\mathrm{partial}}$ is the upcoming ProcOnion algorithm restricted to header, payload, and (partial) ring buffer processing (i.e., without any SNARG proof checks or creations), and η_{i-1}, δ_{i-1}, and the C_{i-1}^j are the previous header, payload, and (partial) ring buffer that are reverse-processed from X, SK_{i-1}, and random coins $\mathcal{R}$.[17]
3. M of any other form are not allowed.

The intuition behind $\mathcal{L}^j$ is simple: partial onions in $\mathcal{L}^j$ feature a ciphertext C_i^1 that allows to "reverse-process" the given onion to some extent. In particular, either the onion in question is the immediate output of either a FormOnion or a ProcOnion query. In case of a ProcOnion output, the whole onion cannot be reconstructed or checked (since some information in the C_i ring buffer is necessarily lost during processing). However, given an onion O_i and the secret key SK^{M}, the validity of π_i^N guarantees that a large portion of O_{i-1} can be reconstructed. In fact, only C_{i-1}^N cannot possibly be retrieved. However, going further, the reconstructed π_{i-1}^{N-1} now makes a statement about that "incomplete onion" O_{i-1}, and the reverse-processing can be continued.

Payload Construction. For message m, we again signal that it is not repliable by prepending a 0-bit: $m' = 0\|m$ and construct the payload as multiple encryption: $\delta_1 = \mathsf{PRP2.Enc}_{k_1^\delta}(\mathsf{PRP2.Enc}_{k_2^\delta}(\ldots \mathsf{PRP2.Enc}_{k_{n+1}^\delta}(m')\ldots))$

Onion Processing The processing of the header is done as in the UE-based scheme (see Section 5.2). However, the processing also checks the SNARG and treats the payload with PRP2.Dec:

If P_i receives an onion $O_i = (\eta_i = (E_i, B_i^1, B_i^2, \ldots, B_i^{2N-1}, \gamma_i), (C_i, \pi_i), \delta_i))$, it does the following steps differently:

1. Check the SNARG-Sequence π_i of the received onion (and abort if it fails)
2. Decrypt the header to retrieve $(k_i^\eta, k_i^\gamma, k_i^\delta)$ and new header blocks for $i+1$, check the MAC, pad the header as before (see Section 5.2)

[17] We will describe ProcOnion only below, but it will be clear that the header, payload, and partial ring buffer part of the processing can be reversed with the secret key SK_{i-1} of the processing party. We additionally run $\mathrm{ProcOnion}_{\mathrm{partial}}$ to re-check MAC values.

3. Construct the new payload $\delta_{i+1} = \mathsf{PRP2.Dec}_{k_i^\delta}(\delta_i)$ [resp. retrieve the message in case of being the receiver ($\delta_{i+1} = 0 \| m$ if no reply)]
4. Rerandomize and shift ring buffer: $C_{i+1}^{j+1} = \mathsf{PKM.ReRand}_{PK^{\mathrm{M}}}(C_i^j)$
5. Replace first ring buffer entry: $C_{i+1}^1 = \mathsf{PKM.Enc}_{PK^{\mathrm{M}}}(\mathcal{J})$
6. Build the new SNARG-Sequence π_{i+1}
7. Send the new onion $O_{i+1} = ((E_{i+1}, B_{i+1}^1, \ldots, B_{i+1}^{2N-1}, \gamma_{i+1}), (C_{i+1}, \pi_{i+1}), \delta_{i+1})$ to the next relay P_{i+1}

Constructing a Repliable Onion. works as for the UE-based scheme before, except that we include $k^{\delta}{}_1^{\leftarrow}$ in the header and store all chosen $k^{\delta}{}_i^{\leftarrow}$ for later use.

Sending a reply. Processing the repliable onion, the receiver stores $P_1^{\leftarrow}, \eta_1^{\leftarrow}$ and k_R^δ. To reply with m (padded to the fixed message length), the receiver does the following steps:

1. Calculate $\delta_1 = \mathsf{PRP2.Enc}_{k_R^\delta}(m)$
2. Construct the SNARG-Sequence π_1,
3. Pick the ring buffer elements randomly $C_1^j \leftarrow^R \{0,1\}^{L_3} \setminus \{\mathtt{sim}\}$ for all j
4. Send the onion $O_1 = (\eta_1^{\leftarrow}, (C_1, \pi_1), \delta_1)$ to $P_1^{\leftarrow}$

Decrypting a reply. After recognizing to have received a reply (by checking the stored $k_{n^{\leftarrow}+1}^\delta$), the reply is "decrypted":
$m = \mathsf{PRP2.Dec}_{k_1^\delta}(\mathsf{PRP2.Enc}_{k_2^\delta}(\ldots(\mathsf{PRP2.Enc}_{k_{n^{\leftarrow}}^\delta}(\mathsf{PRP2.Enc}_{k_{n^{\leftarrow}+1}^\delta}(\delta))\ldots)))$

7.3 Security

The proofs of our onion routing properties are similar to the ones for the UE-based scheme, except that they rely on the SNARGs to protect the payload. We detail them in Appendix F.

Theorem 4. *Our SNARG-based OR Scheme is a* secure, repliable OR scheme.

Acknowledgements This work was supported by funding from the topic Engineering Secure Systems (Subtopic 46.23.01) of the Helmholtz Association (HGF), by the KASTEL Security Research Labs, by the Cluster of Excellence 'Centre for Tactile Internet with Human-in-the-Loop' (EXC 2050/1, Project ID 390696704), and by the ERC grant 724307.

References

1. Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, Nov. 2001.
2. Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 202, U.S. Department of Commerce, 2015.
3. M. Ando and A. Lysyanskaya. Cryptographic shallots: A formal treatment of repliable onion encryption. *eprint, https://eprint.iacr.org/2020/215.pdf*, 2020.

4. M. Backes et al. Provably secure and practical onion routing. In *Computer Security Foundations Symposium*, pages 369–385, 2012.
5. N. Bitansky et al. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, 2012.
6. J. Camenisch and A. Lysyanskaya. A formal treatment of onion routing. In *Annual International Cryptology Conference*, 2005.
7. R. Canetti, H. Krawczyk, and J. B. Nielsen. Relaxing chosen-ciphertext security. In D. Boneh, editor, *Proc. CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 565–582. Springer, 2003.
8. D. Catalano, M. Di Raimondo, D. Fiore, R. Gennaro, and O. Puglisi. Fully non-interactive onion routing with forward secrecy. *INT J INF SECUR*, 2013.
9. D. Catalano, D. Fiore, and R. Gennaro. A certificateless approach to onion routing. *International Journal of Information Security*, 16(3):327–343, 2017.
10. D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 1981.
11. C. Chen, D. E. Asoni, D. Barrera, G. Danezis, and A. Perrig. HORNET: High-speed onion routing at the network layer. In *ACM CCS*, 2015.
12. C. Chen et al. TARANET: Traffic-Analysis Resistant Anonymity at the NETwork layer. *IEEE EuroS&P*, 2018.
13. G. Danezis and I. Goldberg. Sphinx: A compact and provably secure mix format. In *IEEE S&P*, 2009.
14. G. Danezis and B. Laurie. Minx: A simple and efficient anonymous packet format. In *WPES*, 2004.
15. R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.
16. J. Feigenbaum, A. Johnson, and P. Syverson. Probabilistic analysis of onion routing in a black-box model. *ACM TISSEC*, 15(3):14, 2012.
17. G. Fuchsbauer. Subversion-zero-knowledge SNARKs. In M. Abdalla and R. Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 315–347. Springer, Heidelberg, Mar. 2018.
18. T. E. Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Information Theory*, 31(4):469–472, 1985.
19. D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding routing information. In *International workshop on information hiding*, 1996.
20. J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326, 2016.
21. J. Groth and M. Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *CRYPTO*, 2017.
22. M. Hayden. The price of privacy: Re-evaluating the nsa. Johns Hopkins Foreign Affairs Symposium, Apr. 2014.
23. A. Kate, G. M. Zaverucha, and I. Goldberg. Pairing-based onion routing with improved forward secrecy. *ACM TISSEC*, 13, 2010.
24. M. Klooß, A. Lehmann, and A. Rupp. (R)CCA secure updatable encryption with integrity protection. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 68–99. Springer, Heidelberg, May 2019.
25. A. E. Kosba et al. How to use snarks in universally composable protocols. IACR ePrint Archive, 2015. `http://eprint.iacr.org/2015/1093`.
26. C. Kuhn, M. Beck, and T. Strufe. Breaking and (partially) fixing provably secure onion routing. In *IEEE Symposium on Security and Privacy*, 2020.

27. K. Kurosawa and Y. Desmedt. A new paradigm of hybrid encryption scheme. In M. Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 426–442. Springer, Heidelberg, Aug. 2004.
28. H. Lipmaa. Simulation-extractable snarks revisited. IACR ePrint Archive, report 2019/612, 2019. `http://eprint.iacr.org/2019/612`.
29. S. Mauw, J. H. Verschuren, and E. P. de Vink. A formalization of anonymity and onion routing. In *ESORICS*, 2004.
30. S. Micali. CS proofs (extended abstracts). In *35th FOCS*, pages 436–453. IEEE Computer Society Press, Nov. 1994.
31. A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis. The loopix anonymity system. In *USENIX*, 2017.
32. E. Shimshock, M. Staats, and N. Hopper. Breaking and provably fixing minx. In *PETS*, 2008.

Appendix

A Definition of Building Blocks

For our construction we make use of SUF-CMA secure MACs, PRP-CCA secure symmetric encryption, re-randomizable IND-CPA secure public-key encryption, CCA secure public-key encryption, and simulation-sound SNARGs which are all defined below.

Definition 8. *A* message authentication code (MAC) *scheme consists of three PPT algorithms* $(\mathsf{Gen}, \mathit{MAC}, \mathit{Ver})$ *with the following syntax:*

Key generation. $\mathsf{Gen}(1^\lambda)$ *outputs a key* k.
MAC generation. $\mathit{MAC}(k, M)$ *computes a tag* γ *for a message* $M \in \{0,1\}^*$ *under key* k.
MAC Verification. $\mathit{Ver}(k, M, \gamma)$ *outputs a bit on input of a key* k*, a message* m*, and a tag* γ.

We require correctness in the sense that for all $\lambda \in \mathbb{N}$*, all* $k \leftarrow \mathsf{Gen}(1^\lambda)$*, all* $M \in \{0,1\}^*$*, it holds that* $\mathit{Ver}(k, M, \mathit{MAC}(k, M)) = 1$.

Finally, we say that the MAC scheme is SUF-CMA secure if for all PPT adversaries $\mathcal{A}$ *it holds that the success probability defined by*

$$\Pr\left[\begin{array}{c} \mathit{Ver}(k, M^*, \gamma^*) = 1 \\ \wedge \\ (M^*, \gamma^*) \notin \{(M_1, \gamma_1), \ldots, (M_q, \gamma_q)\} \wedge \end{array} \middle| \begin{array}{c} k \leftarrow \mathsf{Gen}(1^\lambda) \\ (M^*, \gamma^*) \leftarrow \mathcal{A}^{\mathit{MAC}(k,\cdot)}(1^\lambda) \end{array}\right]$$

is negligible in λ*, where* $\mathit{MAC}(k, \cdot)$ *is an oracle that, on input* M*, returns* $\mathit{MAC}(k, M)$*,* $M_1, \ldots, M_q$ *denotes the messages queried by* $\mathcal{A}$ *to its oracle, and* $\gamma_1, \ldots, \gamma_q$ *the respective replies.*

Furthermore, we will make use of PRP-CCA secure symmetric encryption for constructing onion headers which is defined as follows:

Definition 9. *A* symmetric encryption scheme *consists of three polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with the following syntax:*

Key generation. $\mathsf{Gen}(1^\lambda)$ *is a probabilistic algorithm which outputs a key* k.
Encryption. $\mathsf{Enc}(k, M)$ *is a deterministic algorithm which takes a key* k *and a plaintext message* $M \in X \subset \{0,1\}^*$ *and outputs a ciphertext* C.
Decryption. $\mathsf{Dec}(k, C)$ *is a deterministic algorithm takes a key* k *and a ciphertext* C *and outputs a plaintext message* m.

We require correctness in the sense that for all $\lambda \in \mathbb{N}$, *all* $k \leftarrow \mathsf{Gen}(1^\lambda)$, *all* $M \in X$, *it holds that* $\mathsf{Dec}(k, \mathsf{Enc}(k, M)) = M$.

We restrict to encryption schemes defining a permutation, i.e., for all $\lambda \in \mathbb{N}$ *and* $k \leftarrow \mathsf{Gen}(1^\lambda)$ *the function* $\mathsf{Enc}(k, \cdot)$ *is a permutation on* X. *We call such an encryption scheme PRP-CCA secure if for every PPT* $\mathcal{A}$, *the advantage*

$$\Pr\left[b' = b \;\middle|\; \begin{array}{c} k \leftarrow \mathsf{Gen}(1^\lambda) \\ P \leftarrow Perm(X) \\ b \leftarrow \{0,1\} \\ (F, F^{-1}) := \begin{cases} (\mathsf{Enc}(k,\cdot), \mathsf{Dec}(k,\cdot)), & b = 0 \\ (P, P^{-1}), & b = 1 \end{cases} \\ b' \leftarrow \mathcal{A}^{F(\cdot), F^{-1}(\cdot)}(1^\lambda) \end{array} \right] - \frac{1}{2}$$

is negligible in λ, *where* P *is a uniformly chosen permutation on* X *and* P^{-1} *is its inverse. Note that if an encryption scheme is PRP-CCA secure then the above property also holds if we swap* Enc *and* Dec *(i.e.,* Dec *is also a strong pseudo-random permutation on* X*).*

Moreover, to enable payload integrity along with SNARGs, we will make use of IND-CCA secure and re-rerandomizable IND-CPA secure public-key encryption.

Definition 10. *An* asymmetric (or public-key) encryption scheme *consists of three PPT algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with the following syntax:*

Key generation. $\mathsf{Gen}(1^\lambda)$ *outputs a public key PK and a secret key SK. We assume that PK defines an efficiently decidable and samplable message space* $\mathcal{M} \subseteq \{0,1\}^*$.
Encryption. $\mathsf{Enc}(PK, M)$ *encrypts a message* M *under a public key PK to a ciphertext* C.
Decryption. $\mathsf{Dec}(SK, C)$ *decrypts a ciphertext* C *under a secret key SK to a message* $M \in \mathcal{M} \cup \{\mathtt{rej}\}$, *where* `rej` *is a special symbol that indicates that* C *was rejected.*

We require correctness in the sense that for all λ, *all* $(PK, SK) \leftarrow \mathsf{Gen}(1^\lambda)$, *all* $M \in \mathcal{M}$, *and all* $C \leftarrow \mathsf{Enc}(PK, M)$, *we always have* $\mathsf{Dec}(SK, C) = M$.

Furthermore, a ciphertext is called valid *(for PK) iff it lies in the range of* $\mathsf{Enc}(PK, \cdot)$. *We say that the scheme has efficiently recognizable ciphertexts iff the set of valid ciphertexts (for a given PK) can be efficiently recognized.*

Finally, we say that the scheme is IND-CPA secure iff no PPT adversary $\mathcal{A}$ *can distinguish the following two experiments with non-negligible advantage:*

- *$\mathcal{A}$ gets a fresh public key PK, selects two equal-length messages $M_0, M_1 \in \mathcal{M}$, and receives $C^* \leftarrow \mathsf{Enc}(PK, M_0)$.*
- *$\mathcal{A}$ gets a fresh public key PK, selects two equal-length messages $M_0, M_1 \in \mathcal{M}$, and receives $C^* \leftarrow \mathsf{Enc}(PK, M_1)$.*

We say that the scheme is IND-CCA secure if the above experiments are indistinguishable even when $\mathcal{A}$ gets access to a decryption oracle $\mathsf{Dec}(SK, \cdot)$ (with the provision that $\mathsf{Dec}(SK, \cdot)$ does not decrypt C^* after C^* is defined).

Definition 11. *A* rerandomizable asymmetric encryption scheme *is an IND-CPA secure PKE scheme $(\mathsf{Gen}_{RR}, \mathsf{Enc}_{RR}, \mathsf{Dec}_{RR})$ with efficiently recognizable ciphertext in the sense of Def. 10 for which a PPT algorithm Rerand exists that inputs a public key PK and a ciphertext C, and outputs another ciphertext C'. We require that no PPT adversary $\mathcal{A}$ can distinguish the following two experiments with non-negligible advantage:*

- *$\mathcal{A}$ gets a fresh public key PK, selects C, and receives $C' \leftarrow \mathsf{Rerand}(PK, C)$.*
- *$\mathcal{A}$ gets a fresh public key PK, selects C, and receives $C' \leftarrow \mathsf{Enc}_{RR}(PK, R)$, where R is a fresh random message from the scheme's message space.*

Here, we only quantify adversaries $\mathcal{A}$ that always output valid ciphertexts (in the sense of Def. 10).

We stress that rerandomizability requires that even adversarially generated (but valid) ciphertexts can be rerandomized. An example of a rerandomizable asymmetric encryption scheme is the ElGamal scheme [18]. (The corresponding Rerand homomorphically adds a fresh encryption of the neutral group element to C.)

In the following, we present a variant of the SNARK definition from [20]. We will assume a language $L \subseteq \{0,1\}^*$ (that may depend on the security parameter and additional random choices), and an efficiently computable witness relation R for L. Hence, R takes as input $x \in \{0,1\}^*$ and a potential witness $w \in \{0,1\}^{p(|x|)}$ (for a fixed polynomial p), and gives a binary output. We require that $x \in L \Leftrightarrow (\exists w \in \{0,1\}^{p(|x|)} : R(x,w) = 1)$. We also assume a canonical description of R, e.g., as a Boolean circuit.

Definition 12 (SNARG). *A succinct non-interactive argument (SNARG) for a relation R consists of four PPT algorithms:*

- *Key generation: $\mathsf{Setup}_{\mathsf{ZK}}(1^\lambda, R)$ outputs a common reference string CRS and a simulation trapdoor τ.*
- *Proofs: $\mathsf{Prove}_{\mathsf{ZK}}(R, CRS, x, w)$, for $R(x,w) = 1$, outputs a proof π.*
- *Verification: $\mathsf{Vfy}_{\mathsf{ZK}}(R, CRS, x, \pi)$ outputs a binary verdict.*
- *Simulation: $\mathsf{Sim}_{\mathsf{ZK}}(R, \tau, x)$ outputs a simulated proof π.*

We require the following properties:

- *Succinctness: the bitlength $|\pi|$ of π is polynomial in the security parameter λ, and the runtime of $\mathsf{Vfy}_{\mathsf{ZK}}$ is polynomial in $\lambda + |x|$.*

- *(Perfect) completeness: for all* λ *and* x, w *with* $R(x,w) = 1$*, it is* $\mathsf{Vfy}_{\mathsf{ZK}}(R, CRS, x, \mathsf{Prove}_{\mathsf{ZK}}(R, CRS, x, w)) = 1$ *always.*
- *(Perfect) zero-knowledge: for all* λ *and* x, w *with* $R(x,w) = 1$*, the outputs of* $\mathsf{Prove}_{\mathsf{ZK}}(R, CRS, x, w)$ *and* $\mathsf{Sim}_{\mathsf{ZK}}(R, \tau, x)$ *are identically distributed.*
- *Simulation-soundness: for every PPT* $\mathcal{A}$*, the following probability is negligible in* λ*:*

$$\Pr\left[\begin{array}{c}\mathsf{Vfy}_{\mathsf{ZK}}(R, CRS, x, \pi) = 1 \\ \nexists w : R(x,w) = 1\end{array}\middle|\begin{array}{c}(CRS, \tau) \leftarrow \mathsf{Setup}_{\mathsf{ZK}}(1^\lambda, R) \\ (x, \pi) \leftarrow \mathcal{A}^{\mathsf{Sim}_{\mathsf{ZK}}(R,\tau,\cdot)}(1^\lambda, R, CRS) \\ \mathsf{Sim}_{\mathsf{ZK}} \text{ never queried with } x\end{array}\right]$$

where the probability is over the random coins of $\mathsf{Setup}_{\mathsf{ZK}}$*, the random coins of* $\mathcal{A}$ *and* $\mathsf{Vfy}_{\mathsf{ZK}}$*, and possibly over the choice of the relation* R *itself.*

We note that the simulation-soundness game above is not necessarily efficient (because of the condition $\nexists w : R(x,w) = 1$), but still implied by a property called "simulation-extractability" [25,21]. Efficient simulation-extractable SNARGs (i.e., SNARKS) can be constructed from knowledge assumptions [21,28].

B Ideal Functionality

We show the ideal functionality in Algorithm 1 and 2. As in earlier works, the honest nodes inform the environment about the *temp* whenever they receive an onion. Further, they now additionally include the information whether the onion is repliable. We highlight the changes compared to the ideal functionality of the simple one-way sending [26] in teal. We especially add two data structures:

- *Back*: to store the mapping from *temp*s (labels of the onions) to the corresponding path and forward onion id. This mapping is used to find the right path when a reply onion is constructed.
- ID_{fwd}: to store the mapping from backward *id*s to forward *id*s. This mapping is used to allow corrupted senders (i.e. backward receivers) to learn all information of the backward onion; including to which forward onion she belongs.

We assume that a corrupted sender (i.e. backward receiver) can learn and link all onion layers. Further, an onion can be replied to multiple times. We stress that this is a useful security definition as single use reply blocks can be created with the help of duplicate protection (on the header), which also prevents other traffic analysis attacks that are considered an orthogonal problem [26].

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
// $\mathcal{P} = (P_{o_1}, \ldots, P_{o_n})$: Onion path, ($\mathcal{P}^{\rightarrow}$ forwards, $\mathcal{P}^{\leftarrow}$ backwards)
// $O = (id, P_s, P_r, m, \mathcal{P}, \mathcal{P}', i, d)$: 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_s
  // P_s creates and sends a new onion (either instructed by $\mathcal{Z}$ if honest or $\mathcal{S}$ if corrupted)
  if $|\mathcal{P}| > N$; // selected path too long
  then
    Reject;
  else
    $id \leftarrow^R$ session ID ; // pick random session ID
    $O \leftarrow (id, P_s, P_r, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, 0, f)$; // create new onion
    Output_Corrupt_Sender$(P_s, id, P_r, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, \text{start}, f)$;
    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
    $Back(temp) = (P_s, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, P_r, id')$; // lookup the corresponding path
    $id \leftarrow^R$ session ID ; // pick random session ID
    Store id' under $ID_{fwd}(id)$; // add ID linking to mapping
    $O \leftarrow (id, P_r, P_s, m, \mathcal{P}^{\leftarrow}, \mathcal{P}^{\rightarrow}, 0, b)$; // create new onion
    Output_Corrupt_Sender$(P_r, id, P_s, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, \text{start}, b)$;
    Process_Next_Step(O);

Procedure Output_Corrupt_Sender$(P_s, id, P_r, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, temp, d)$
  // Give all information about onion to adversary if sender is corrupt
  if $P_s \in Bad$ **then**
    Send "$temp$ belongs to onion from P_s 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`$(O = (id, P_s, P_r, m, \mathcal{P}, \mathcal{P}'i, d))$
- `// 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 $(P_{o_{i+1}}, \ldots, P_{o_n})$" to $\mathcal{S}$;
  - **if** $d = f$ **then**
    - Store $(P_s, \mathcal{P}, \mathcal{P}', P_r, id)$ 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`$(P_s, id, P_r, m, \mathcal{P}, \mathcal{P}', temp, f)$;
  - **else**
    - `Output_Corrupt_Sender`$(P_r, id, P_s, m, \mathcal{P}', \mathcal{P}, temp, b)$;
- **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 $(P_{o_{i+1}}, \ldots, P_{o_{j-1}})$ 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`$(P_s, id, P_r, m, \mathcal{P}, \mathcal{P}', temp, f)$;
  - **else**
    - `Output_Corrupt_Sender`$(P_r, id, P_s, m, \mathcal{P}', \mathcal{P}, temp, b)$;
    - **if** $P_s \in Bad$ *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 $(temp, O = (sid, P_s, P_r, m, \mathcal{P}, \mathcal{P}', i), j)$ from L;
  - $O \leftarrow (sid, P_s, P_r, m, \mathcal{P}, \mathcal{P}', j)$; `// 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 to P_r";
      - **if** $\mathcal{P}' \neq ()$ *and* $d = f$ **then**
        - add "that is repliable" to the message for P_r ;
        - Store $(P_s, \mathcal{P}, \mathcal{P}', P_r, id)$ 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);

C Other OR Property Definitions

C.1 Backwards Layer-Unlinkability

Definition 13 (Backwards Layer-Unlinkability $LU^{\leftarrow}$). *Backwards Layer-Unlinkability is defined as:*

1. *The adversary receives the router names P_H, P_S and challenge public keys PK_S, PK_H, chosen by the challenger by letting $(PK_H, SK_H) \leftarrow G(1^\lambda, p, P_H)$ and $(PK_S, SK_S) \leftarrow G(1^\lambda, p, P_S)$.*
2. *Oracle access: The adversary may submit any number of* Proc *and* Reply *requests for P_H or P_S to the challenger. For any* Proc(P_H, O)*, the challenger checks whether η is on the η^H- list. If not, it sends the output of* ProcOnion(SK_H, O, P_H)*, stores η on the η^H-list and O on the O^H-list. For any* Reply(P_H, O, m) *the challenger checks if O is on the O^H- list and if so, the challenger sends* ReplyOnion(m, O, P_H, SK_H) *to the adversary. (Similar for requests on P_S with the η^S-list).*
3. *The adversary submits*
   - *message m,*
   - *a position $j^{\leftarrow}$ with $0 \leq j^{\leftarrow} \leq n^{\leftarrow} + 1$,*
   - *a path $\mathcal{P}^{\rightarrow} = (P_1, \ldots, P_j, \ldots, P_{n+1})$, where $P_{n+1} = P_H$, if $j^{\leftarrow} = 0$,*
   - *a path $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{j^{\leftarrow}}^{\leftarrow}, \ldots, P_{n^{\leftarrow}+1}^{\leftarrow} = P_S)$ with the honest node P_H at backward position $j^{\leftarrow}$, if $1 \leq j^{\leftarrow} \leq n^{\leftarrow} + 1$, and the second honest node P_S at position $n^{\leftarrow} + 1$*
   - *and public keys for all nodes PK_i ($1 \leq i \leq n + 1$ for the nodes on the path and $n + 1 < i$ 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+1} if $j^{\leftarrow} = 0$), $PK_{n^{\leftarrow}+1}^{\leftarrow} = PK_S$ and sets bit b at random.*
5. *The challenger creates the onion with the adversary's input choice and honestly chosen randomness $\mathcal{R}$:*

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

   and sends O_1 to the adversary.
6. *The adversary gets oracle access as in step 2) except if:*

   Exception 1) The request is ...
   - *for $j^{\leftarrow} > 0$:* Proc(P_H, O) *with* RecognizeOnion$((n+1)+j^{\leftarrow}, O, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}) = True$, η *is not on the η^H-list and* ProcOnion$(SK_H, O, P_H) \neq \perp$*: stores η on the η^H and O on the O^H-list and ...*
   - *for $j^{\leftarrow} = 0$:* Reply$(P_H, O, m^{\leftarrow})$ *with* RecognizeOnion$((n+1), O, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}) = True$, O *is on the O^H- list and no onion with this η has been replied to before and* ReplyOnion$(m^{\leftarrow}, O, P_H, SK_H) \neq \perp$*:*

.. then: The challenger picks the rest of the return path $\bar{\mathcal{P}}^{\rightarrow} = (P^{\leftarrow}_{j^{\leftarrow}+1}, \ldots, P^{\leftarrow}_{n^{\leftarrow}+1})$*, an empty backward path* $\bar{\mathcal{P}}^{\leftarrow} = ()$*, and a random message* $\bar{m}$*, another honestly chosen randomness* $\bar{\mathcal{R}}$*, and generates:*

$$\bar{O}_1 \leftarrow \text{FormOnion}(1, \bar{\mathcal{R}}, \bar{m}, \bar{\mathcal{P}}^{\rightarrow}, \bar{\mathcal{P}}^{\leftarrow}, (PK)_{\bar{\mathcal{P}}^{\rightarrow}}, (PK)_{\bar{\mathcal{P}}^{\leftarrow}})$$

- *If* $b = 0$*, the challenger calculates* $(O_{j^{\leftarrow}+1}, P^{\leftarrow}_{j^{\leftarrow}+1}) = \text{ProcOnion}(SK_H, O, P^{\leftarrow}_{j^{\leftarrow}})$ *(for* $j^{\leftarrow} > 0$*) resp.* $(O_{j^{\leftarrow}+1}, P^{\leftarrow}_{j^{\leftarrow}+1}) = \text{ReplyOnion}(m^{\leftarrow}, O, P^{\leftarrow}_{j^{\leftarrow}}, SK_H)$ *(for* $j^{\leftarrow} = 0$*) and gives* $O_{j^{\leftarrow}+1}$ *for* $P^{\leftarrow}_{j^{\leftarrow}+1}$ *to the adversary.*
- *Otherwise, the challenger gives* $\bar{O}_1$ *for* $P^{\leftarrow}_{j^{\leftarrow}+1}$ *to the adversary.*

Exception 2) Proc(P_S, O) *with* O *being the challenge onion as processed for the final receiver on the backward path, i.e.:*

- *for* $b = 0 : \text{RecognizeOnion}((n+1) + (n^{\leftarrow}+1), O, \mathcal{R}) = True$
- *for* $b = 1 : \text{RecognizeOnion}((n^{\leftarrow}+1) - j^{\leftarrow}, O, \bar{\mathcal{R}}, \bar{m}, \bar{\mathcal{P}}^{\rightarrow}, \bar{\mathcal{P}}^{\leftarrow}, (PK)_{\bar{\mathcal{P}}^{\rightarrow}}, (PK)_{\bar{\mathcal{P}}^{\leftarrow}}) = True$

.. then the challenger outputs nothing.

7. *The adversary produces guess* b' *.*

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

C.2 Repliable Tail-Indistinguishability

Definition 14 (Repliable Tail-Indistinguishability $TI^{\leftrightarrow}$). *Repliable Tail-Indistinguishability is defined as:*

1. *The adversary receives the router names* $P_H, P^{\leftarrow}_H, P_S$ *and challenge public keys* $PK_S, PK_H, PK^{\leftarrow}_H$*, chosen by the challenger by letting* $(PK_H, SK_H) \leftarrow G(1^{\lambda}, p, P_H)$*,* $(PK^{\leftarrow}_H, SK^{\leftarrow}_H) \leftarrow G(1^{\lambda}, p, P^{\leftarrow}_H)$*,* $(PK_S, SK_S) \leftarrow G(1^{\lambda}, p, P_S)$*.*
2. *Oracle access: The adversary may submit any number of* Proc *and* Reply *requests for* $P_H, P^{\leftarrow}_H$ *or* P_S *to the challenger. For any* Proc(P_H, O)*, the challenger checks whether* η *is on the* η^H*- list. If not, it sends the output of* $\text{ProcOnion}(SK_H, O, P_H)$*, stores* η *on the* η^H*-list and* O *on the* O^H*-list. For any* Reply(P_H, O, m) *the challenger checks if* O *is on the* O^H*- list and if so, the challenger sends* $\text{ReplyOnion}(m, O, P_H, SK_H)$ *to the adversary. (Similar for requests on* $P^{\leftarrow}_H, P_S$*).*
3. *The adversary submits a message* m*, a path* $\mathcal{P}^{\rightarrow} = (P_1, \ldots, P_j, \ldots, P_{n+1})$ *with the honest node* P_H *or* $P^{\leftarrow}_H$ *at position* j*,* $1 \leq j < n+1$*, a path* $\mathcal{P}^{\leftarrow} = (P^{\leftarrow}_1, \ldots, P^{\leftarrow}_{n^{\leftarrow}+1})$ *with the honest node* $P^{\leftarrow}_H$ *at position* $1 \leq j^{\leftarrow} \leq n^{\leftarrow}+1$ *and public keys for all nodes* PK_i *(*$1 \leq i \leq n+1$ *for the nodes on the path and* $n+1 < i$ *for the other relays).*
4. *The challenger checks that the given 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* $PK_j = PK_H$ *(or* $PK_j = PK^{\leftarrow}_H$*, if the adversary chose* $P^{\leftarrow}_H$ *at this position as well) ,* $PK^{\leftarrow}_{j^{\leftarrow}} = PK^{\leftarrow}_H$ *,*$PK^{\leftarrow}_{n^{\leftarrow}+1} = PK_S$ *and sets bit* b *at random.*

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

$$O_{j+1} \leftarrow \text{FormOnion}(j+1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})$$

and a replacement onion with the path from the honest relay P_H to the corrupted receiver $\bar{\mathcal{P}}^{\rightarrow} = (P_{j+1}, \ldots, P_{n+1})$ and the backward path from the corrupted receiver starting at position 0 ending at $j^{\leftarrow}$: $\bar{\mathcal{P}}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{j^{\leftarrow}}^{\leftarrow})$; and another honestly chosen randomness $\bar{\mathcal{R}}$:

$$\bar{O}_1 \leftarrow \text{FormOnion}(1, \bar{\mathcal{R}}, m, \bar{\mathcal{P}}^{\rightarrow}, \bar{\mathcal{P}}^{\leftarrow}, (PK)_{\bar{\mathcal{P}}^{\rightarrow}}, (PK)_{\bar{\mathcal{P}}^{\leftarrow}})$$

6. *If $b = 0$: The challenger sends O_{j+1} to the adversary.*
   Otherwise: The challenger sends $\bar{O}_1$ to the adversary.
7. *Oracle access: the challenger processes all requests as in step 2) except if...*
   ... $\mathsf{Proc}(P_H^{\leftarrow}, O)$ with O being the challenge onion as processed for the honest relay on the backward path, i.e.:
   - *for $b = 0$: $\text{RecognizeOnion}((n+1) + j^{\leftarrow}, O, \mathcal{R}) = True$ or*
   - *for $b = 1$: $\text{RecognizeOnion}((n-j)+j^{\leftarrow}, O, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}) = True$*

   .. then the challenger outputs nothing.
8. *The adversary produces guess b'.*

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

D Proof of UC-realization

D.1 Overview

This argumentation extends the one from [26] for the replies.

***** Informally: For corrupted sender (= backward receiver), all information about the communication are leaked in the ideal functionality. Thus, no protection is needed.

For honest senders (= backward receivers), we want to ensure that only the subpaths between honest relays and (if the receiver is corrupted) the messages can be learned by the adversary. Therefore, we start by replacing the onion layers on the first part of the path, i.e. from the honest sender to the first honest relay on the forward path, with random ones that take the same path. Due to $LU^{\rightarrow}$, we know that the adversary cannot notice the difference. We continue, one onion and subpath at the time, until all subpaths between honest relays on the forward path are replaced.

Next, we replace the last part of the backward path, i.e. from the last honest relay on the backward path to the honest sender (= backward path receiver). Due to $LU^{\leftarrow}$, we know that the adversary cannot notice the difference. We continue, one onion and subpath at the time, until all subpaths between honest relays on the backward path are replaced.

If the receiver is honest, the steps above already replaced everything, as then the receiver is an honest relay. If the receiver is corrupted, we still need to replace the subpath between the last honest relay on the forward and the first honest relay on the backward path. We can do this without the adversary noticing any change due to $TI^{\leftrightarrow}$. Thus, we replaced the onion layers on all subpaths.

***** Formally: We assume that the public keys are already distributed and define any secure, repliable OR scheme to fulfill our properties:

Definition 15. *A secure repliable OR scheme is a quadruple of polynomial-time algorithms* $(G, FormOnion, ProcOnion, ReplyOnion)$ *(Section 3.2) that achieves Onion-Correctness (Def. 2), Repliable Tail-Indistinguishability (Def. 14), Forwards Layer-Unlinkability (Def. 3) and Backwards Layer-Unlinkability (Def. 13).*

Similarly to [6,26], we say that a OR protocol is build from the OR scheme with an additional ideal functionality for the assumed key distribution $\mathcal{F}_{RKR}$.

Definition 16. *OR protocol Π is a secure repliable OR protocol (in the $\mathcal{F}_{RKR}$-hybrid model), iff it is based on a secure OR scheme* $(G, \text{FormOnion}, \text{ProcOnion}, \text{ReplyOnion})$ *and works as follows:*

Setup: *Each node P_i generates a key pair $(SK_i, PK_i) \leftarrow G(1^\Lambda)$ and publishes PK_i by using $\mathcal{F}_{RKR}$.*

Sending a Message: *If P_S wants to send $m \in \mathcal{M}$ to P_R over path $P_1, \ldots, P_n$ with $n < N$ and wants to allow a reply over the path $P_1^{\leftarrow}, \ldots, P_{n^{\leftarrow}}^{\leftarrow}$ with $n^{\leftarrow} < N$ and $P_{n^{\leftarrow}}^{\leftarrow} = P_S$, he chooses a randomness $\mathcal{R}$ and sends the following O_1 to P_1.*

$$O_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m, (P_1, \ldots, P_n, P_R), (P_1^{\leftarrow}, \ldots, P_{n^{\leftarrow}}^{\leftarrow}), (PK_1, \ldots, PK_n, PK_R), (PK_1^{\leftarrow}, \ldots, PK_{n^{\leftarrow}}^{\leftarrow}))$$

Replying an Onion: *If P_R wants to reply to an onion O with message $m^{\leftarrow}$, he sends $O_1^{\leftarrow}$ to $P_1^{\leftarrow}$ which are calculated as*

$$(O_1^{\leftarrow}, P_1^{\leftarrow}) \leftarrow \text{ReplyOnion}(m^{\leftarrow}, O, P_R, SK_R).$$

Processing an Onion: *If P_i received O_i, he calculates:*

$$(O_j, P_j) \leftarrow \text{ProcOnion}(SK_i, O_i, P_i)$$

If $P_j = \perp$, P_i outputs "Received $(m, Reply) = O_j$" in case $O_j \neq \perp$ and reports a fail if $O_j = \perp$. Otherwise P_j is a valid relay name and P_i generates a random temp and stores $(temp, (O_j, P_j))$ in its outgoing buffer and notifies the environment about temp.

Sending an Onion: *When the environment instructs P_i to forward temp, P_i looks up temp in its buffer. If P_i does not find such an entry, it aborts. Otherwise, it found $(temp, (O_j, P_j))$ and sends O_j to P_j.*

We now show that our properties are sufficient for the ideal functionality:

Theorem 5. *A secure repliable onion routing protocol following Definition 16 UC-realizes $\mathcal{F}$ in the $(\mathcal{F}_{RKR})$-hybrid model.*

Therefore, we describe a simulator that translates any attack on the secure, repliable OR protocol to an attack in the ideal functionality.

Simulator Overview

The simulator uses the knowledge of honest keys to process adversarial onions (FormOnion called by an adversarial sender or modified at the adversarial relay) just as the protocol does. For honest onions (FormOnion called by an honest sender) our simulator uses the information it gets from the ideal functionality to build the random replacement onions for each subpath. This information are the part of the path (and if the receiver is adversarial, the message and repliability). The correct relaying of honest onions is recognized by the simulator with *RecognizeOnion*. With the help of our security properties, we can show that the adversary cannot notice the change. We give an overview over standard hybrid argument in Table 1.

D.2 Proof - Detailed

Our proof follows in large parts the argumentation from [26], which in turn adapted the one of [6]. For UC-realization, we show that every attack on the real world protocol Π can be simulated by an ideal world attack without the environment being able to distinguish those. We first describe the simulator $\mathcal{S}$. Then we show indistinguishability of the environment's view in the real and ideal world.

Constructing Simulator $\mathcal{S}$

$\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}$ 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 $(r_{temp}, nextRelay, temp)$. 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 together with the information about the communication $(onion, nextRelay, information)$.
- The *Reply*-list containing reply information together with the forward id for communications with a corrupted sender (id_{fwd}, reply information).

Table 1. Overview Proof: Properties imply Ideal Functionality

Hybrid	Description	Reduction
$\mathcal{H}_0$	Machine using the real world protocol to interact with the real world adversary $\mathcal{A}$ and the environment	
$\mathcal{H}_1 = \mathcal{H}_1^{<2}$	As $\mathcal{H}_0$ but for one forward communication of an honest sender: The onion layers between this sender and the next honest node (relay or receiver) are replaced by the layers of a newly formed onion taking this part of this path but carrying a random message for the next honest node.	$LU^{\rightarrow}$
$\mathcal{H}_1^{<x}$	As $\mathcal{H}_1^{<x-1}$ but for one forward communication of an honest sender, where the onion layers between the first two honest nodes are not yet replaced: Replace as in $\mathcal{H}_1$	$(LU^{\rightarrow})$
$\mathcal{H}_2 = \mathcal{H}_2^{<2}$	As $\mathcal{H}_1^*$ (=first part for all honest forwards communications replaced) but for one forward communication of an honest sender where no modification happened: The onion layers between the next two honest node (relay or receiver) is replaced as in $\mathcal{H}_1$	$LU^{\rightarrow}$
$\mathcal{H}_2^{<x}$	As $\mathcal{H}_2^{<x-1}$ but for one forward communication of an honest sender where no modification happened and these layers are not yet replaced: Replace as in $\mathcal{H}_2$	$(LU^{\rightarrow})$
$\mathcal{H}_1^{\leftarrow} = \mathcal{H}_1^{<2\leftarrow}$	As $\mathcal{H}_2^*$ (=all are replaced) but for one backward communication of an honest sender: The onion layers between the last honest node (relay or forward receiver) are replaced by the layers of a newly formed onion taking this part of this path but carrying a random message for the honest backward receiver (=forward sender).	$LU^{\leftarrow}$
$\mathcal{H}_1^{<x\leftarrow}$	As $\mathcal{H}_1^{<x-1\leftarrow}$ but for one backward communication of an honest sender, where the onion layers between the last honest node and (backward) receiver are not yet replaced: Replace as in $\mathcal{H}_1^{\leftarrow}$	$(LU^{\leftarrow})$
$\mathcal{H}_2^{\leftarrow} = \mathcal{H}_2^{<2\leftarrow}$	As $\mathcal{H}_1^{*\leftarrow}$(=all are replaced) but for one backward communication of an honest sender where no modification happened: The onion layers between the (next) last two honest nodes (relay or forward receiver) are replaced as in $\mathcal{H}_1^{\leftarrow}$	$LU^{\leftarrow}$
$\mathcal{H}_2^{<x\leftarrow}$	As $\mathcal{H}_2^{<x-1\leftarrow}$ but for one forward communication of an honest sender where no modification happened and these layers are not yet replaced: Replace as in $\mathcal{H}_2^{\leftarrow}$	$(LU^{\leftarrow})$
$\mathcal{H}_3 = \mathcal{H}_3^{<2}$	As $\mathcal{H}_2^{*\leftarrow}$(=all are replaced) but for one forward communication of an honest sender where no modification happened but no other honest relay exists(i.e. receiver is corrupt): The onion layers between the last honest node on the forward path and the receiver are replaced with the ones generated by a newly formed onion for this part of the path, carrying the same message	$TI^{\leftrightarrow}$
$\mathcal{H}_3^{<x}$	As $\mathcal{H}_3^{<x-1}$ but for one forward communication of an honest sender where no modification happened and these layers are not yet replaced: Replace as in $\mathcal{H}_3$	$(TI^{\leftrightarrow})$

– The C-list containing reply information together with the temp for communications 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*[18]:

Case I: "start belongs to onion from P_S 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 $(id, \text{reply info})$ for this id in the *Reply*-list and uses the information to construct an onion: $(O_1, P_1) \leftarrow \text{ReplyOnion}(m, replyinfo, P_r, SK_r)$ 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: "start belongs to onion from P_S with $sid, P_r, m, n, \mathcal{P}$". 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: any output together with "*temp* belongs to onion from P_S with $sid, P_r, m, n, \mathcal{P}$" for $temp \notin \{\text{start}, \text{end}\}$. This means an honest relay is done processing an onion received from $\mathcal{A}$ that was not the protocol-conform processing of an honest sender's communication (processing that follows Case VII). $\mathcal{S}$ finds $(onion, nextRelay, information)$ with this inputs as $information$ in the O-list (notice that there has to be such an entry) and 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 $P_{o_{i+1}}$: $\mathcal{S}$ the path $\mathcal{P} = (P_{o_i}, P_{o_{i+1}})$, and random message m_{rdm}. $\mathcal{S}$ honestly picks a randomness $\mathcal{R}$ and calculates

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

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 $(info = (2, \mathcal{R}, m_{rdm}, \mathcal{P}, (), (PK)_{\mathcal{P}_{rdm}}, ()), P_{o_{i+1}}, temp)$ on the r-list.

Case V: "Onion *temp* from P_{o_i} routed through $(P_{o_{i+1}}, \ldots, P_{o_{j-1}})$ to P_{o_j}". $\mathcal{S}$ picks the path $\mathcal{P} = (P_{o_{i+1}}, \ldots, P_{o_{j-1}})$, a randomness $\mathcal{R}$ and a message m_{rdm}

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

and calculates

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

and sends the onion O_1 to $P_{o_{i+1}}$, as if it came from P_{o_i}. $\mathcal{S}$ stores $(info = (j-i, R, m_{rdm}, \mathcal{P}, ()(PK)_{\mathcal{P}_{rdm}}, ()), P_{o_j}, temp)$ on the r-list.

Case VI: "Onion *temp* from P_{o_i} with message m for P_r routed through $(P_{o_{i+1}}, \ldots, P_{o_n})$". 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 II (and VIII a1 and R).]: $\mathcal{S}$ picks the path $\mathcal{P} = (P_{o_i}, \ldots, P_{o_n}, P_r)$, randomness $\mathcal{R}$ and calculates

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

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 $info = (R, n+1+\mathcal{P}^{\leftarrow}.lastPosition, m, \mathcal{P}_{rdm}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}_{rdm}}, (PK)_{\mathcal{P}^{\leftarrow}})$ 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 $O = (\tilde{\eta}, \tilde{\pi}, \tilde{\delta})$ from $\mathcal{A}$ as adversarial player P_a.

Case VII: $(\tilde{\eta}, P_i, temp)$ is on the r-list for some *temp*. In this case O is the protocol-conform processing of an onion from a communication of an honest sender. $\mathcal{S}$ calculates $\text{ProcOnion}(SK(P_i), O, P_i)$. If it returns a fail ($O$ is a replay or modification that is detected and dropped by Π), S does nothing. Otherwise, $\mathcal{S}$ sends the message (Deliver Message, *temp*) to $\mathcal{F}$.

Case VIII. $(\tilde{\eta}, P_i, temp)$ is not on the r-list for any *temp*. $\mathcal{S}$ calculates $\text{ProcOnion}(SK(P_i), O, P_i) = (O', P')$ (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 thus 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 we can try to create a reply to it with P_i). This means the adversary possibly replied to an honest senders forward onion. $\mathcal{S}$ checks for all $(P_i, reply, temp)$ tuples in the C-List to see if $\tilde{\eta}$ matches 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 $\mathcal{F}$) 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,

$$Process\_New\_Onion(P', m, n, ())$$

(notice that this is not repliable) from P_i 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 $(O', P', (P_a, id, P', m, n, ()))$ to the O-list.

Example Case Combinations for the Simulator For easier understanding of the simulator, we sketch the cases that are used together:

- Corrupt Sender
  - Corrupt Receiver
    - Forward: Case VIII (b) and Case III [repeatedly if honest relays involved] until receiver receives correctly unwrapped onion (outside of the scope of the simulator)
    - Backward: Case VIII(b) and Case III [repeatedly if honest relays involved] until sender receives reply onion (outside of the scope of the simulator)
  - Honest Receiver
    - Forward: Case VIII (b) and Case III [repeatedly if honest relays involved] until Case VIII (a1) [receiver receives onion]
    - Backward: Case I [receiver replies], then Case VIII(b) and Case III [repeatedly if honest relays involved] until sender receives reply onion (outside of the scope of the simulator)
- Honest Sender
  - Corrupt Receiver
    - Forward: Case IV or V and Case VII [repeatedly if honest relays involved] until case VI [receiver receives]
    - Backward: Case VIII(a2) [receiver replies] and Case IV or V and Case VII [repeatedly over honest relays until sender]
  - Honest Receiver
    - Forward: Case IV or V and Case VII [repeatedly over honest relays until receiver]
    - Backward: Case IV or V and Case VII [repeatedly over honest relays until sender]

Modified onions are treated as corrupt sender communications.

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. More precisely, this machine acts like $\mathcal{H}_0$ except that the consecutive onion layer $O_1, O_2, \ldots, O_j$ 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 $\mathcal{R}$) $\bar{O}_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m_{rdm}, \mathcal{P}, (), (PK)_{\mathcal{P}}, ())$ where m_{rdm} is a random message, $\mathcal{P} = (P_1, \ldots, P_j)$. $\mathcal{H}_1$ keeps an $\bar{O}$-list and stores $(info = (\mathcal{R}, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, (PK), (PK)^{\leftarrow}), P_j, (O_1^R, P_{j+1}))$ where $info$ are the randomness and parameters used for the original senders onion creation and O_1^R is calculated[19] as

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

where the randomness, paths and message are chosen as in the original sender's call in $\mathcal{H}_0$.[20] If an onion $\tilde{O}$ is sent to P_j, the machine tests if processing results in a fail (replay/modification detected and dropped). If it does not, $\mathcal{H}_1$ uses $RecognizeOnion(\tilde{O}, j, \mathcal{R}, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, (PK), (PK)^{\leftarrow})$ 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_1^R is send to P_{j+1} as processing result of P_j. Otherwise, ProcOnion(SK_{P_j}, $\tilde{O}, P_j$) is used.

$\mathcal{H}_0 \approx_I \mathcal{H}_1$. The environment gets notified when an honest party receives an onion layer (and about their repliability) and inputs when this party is done. As we just exchange onion layers by others (with the same repliability), the behavior to the environment is indistinguishable for both machines.

$\mathcal{A}$ observes the onion layers after P_0 and if it sends an onion to P_j, the result of the processing after the honest 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 $LU^{\rightarrow}$:

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

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

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.[21] 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.[22] 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.

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.[23]

Hybrid $\mathcal{H}_2$. In this hybrid, for the first forward communication, for which in the adversarial processing no recognition falsifying modification (i.e. on η) occurred and other modification does not result in a fail,[24] onion layers between two consecutive honest relays on the forward path (the second might be the receiver) are replaced with random onion layers embedding the same path. Additionally, for all forward communications 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 a honestly chosen randomness $\mathcal{R}$: $\bar{O}_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m_{rdm}, \mathcal{P}, (), (PK)_{\mathcal{P}}, ())$ 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 $(info = (\mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}^{\rightarrow}}, PK_{\mathcal{P}^{\leftarrow}}), P_{j'}, (O_1^R, P_{j'+1}))$,[25]

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

[22] This includes that duplicates are dropped (Assumption 4) and onions are processed before they are replied (Assumption 6).

[23] 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.

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

[25] $P_{j'+1}$ might be $\perp$ and O_1^R the message m if $P_{j'}$ is the honest receiver.

where O_1^R is calculated as $O_1^R \leftarrow \text{FormOnion}(j'+1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}^{\rightarrow}}, PK_{\mathcal{P}^{\leftarrow}})$, where the randomness, paths and message are chosen as the original sender would pick them in the original construction (of the complete onion),[26] 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 $RecognizeOnion(\tilde{O}, j'-j, info)$ for any $info$ on the $\bar{O}$-list where the second entry is $P_{j'}$. If it finds a match, the belonging O_1^R is used as processing result of $P_{j'}$. Otherwise, $ProcOnion(SK_{P_{j'}}, \tilde{O}, P_{j'})$ is used.

$\mathcal{H}_1^* \approx_I \mathcal{H}_2$. $\mathcal{H}_2$ 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 $P_{j'}$.

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 $\mathcal{H}_1^*$ and $\mathcal{H}_2$ 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) modified η of an onion layer in this communication or modifies other parts such that processing fails, the communication is skipped.

[26] $\mathcal{H}_2$ 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.

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

Replacing between Honest - Backward Onion

On the backward path, we replace the last onion layers first, then the second last and so on. Each machine only starts replacing at a certain point and if a message does not come that far (it is modified or dropped), they simply do not use any replacement.

For all following hybrids the replacements on the forward path are done as in $\mathcal{H}_2^*$

Hybrid $\mathcal{H}_1^{\leftarrow}$. Similar to $\mathcal{H}_1$, but this time one backward communication between the last honest node (relay or forwards receiver) until the honest (forwards) sender is replaced.

More precisely, this machine acts like $\mathcal{H}_2^*$ except that the consecutive onion layers $O_{j+1}^{\leftarrow}, \ldots, O_{n^{\leftarrow}+1}^{\leftarrow}$ from a reply to an honest (forward) sender from the last honest relay $P_j^{\leftarrow}$ to the (forward) sender $P_{n^{\leftarrow}+1}^{\leftarrow} = P_0$ are replaced with $\bar{O}_1, \ldots, \bar{O}_{n^{\leftarrow}-j+1}$ with (for a honestly chosen $\mathcal{R}$:) $\bar{O}_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m_{rdm}, \mathcal{P}, (), (PK)_{\mathcal{P}}, ())$ where m_{rdm} is a random message, $\mathcal{P} = (P_j^{\leftarrow}, \ldots, P_{n^{\leftarrow}+1}^{\leftarrow})$ is the path from $P_j^{\leftarrow}$ to $P_{n^{\leftarrow}+1}^{\leftarrow}$. $\mathcal{H}_1^{\leftarrow}$ stores $(info, P_{n^{\leftarrow}+1}^{\leftarrow} = P_0, m_{rdm})$, 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.

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

$\mathcal{A}$ 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.[27] 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+1}), 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 is

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

forwarded to the oracle of the challenger as O to process it.[28] The challenger returns[29] $\tilde{O}$. The attack sends $\tilde{O}$, as the processing of the answer, to $\mathcal{A}$ as soon as $\mathcal{D}$ instructs to do so. To simulate further for $\mathcal{D}$ it uses $\tilde{O}$ to calculate the later layers and does any actions $\mathcal{A}$ does on the onion. Further, the attack simulates also all other communications with the oracles and knowledge of the protocol and keys (or the random replacement onions, if replaced before).[30]

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 (=backwards 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 (and all its replays) for which in the adversarial processing no recognition falsifying modification occurred and other modification did not lead to failed processing[31] 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 that the processing of $O_j^{\leftarrow}$; i.e. the consecutive onion layers $O_{j+1}^{\leftarrow}, \ldots, O_{j'}^{\leftarrow}$ 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 $\bar{O}_1, \ldots, \bar{O}_{j'-j}$. Thereby for an honestly chosen $\mathcal{R}$; $\bar{O}_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m_{rdm}, \mathcal{P}, (), (PK)_{\mathcal{P}_{rdm}}, ())$ where m_{rdm} is a random message, $\mathcal{P} = (P_j^{\leftarrow}, \ldots, P_{j'}^{\leftarrow})$ the path from $P_j^{\leftarrow}$ to $P_{j'}^{\leftarrow}$.

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

$\mathcal{H}_1^{*\leftarrow} \approx_I \mathcal{H}_2^{1\leftarrow}$. $\mathcal{H}_2^{\leftarrow}$ 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 (by $\mathcal{H}_1^{*\leftarrow}$) replaced onion layers $\bar{O}_{later}$ af-

[28] 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.

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

[30] This includes that duplicates are dropped (Assumption 4) and onions are processed before they are replied (Assumption 6).

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

ter 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, if there exists a distinguisher between $\mathcal{H}_1^{*\leftarrow}$ and $\mathcal{H}_2^{\leftarrow}$ there exists an attack on $LU^{\leftarrow}$.

Hybrid $\mathcal{H}_2^{<x\leftarrow}$. In this hybrid, for the first[32] $x-1$ honest subpaths on backwards communications are replaced with a random onion sharing the path and the other replacements calculated as before and all are stored on the $\bar{O}$-list. If $\mathcal{A}$ previously (i.e. in onion layers up to the honest node starting the selected subpath) modified η of an onion layer in this communication or modified another part such that processing fails, the communication is skipped.

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

Onion replacement for corrupted receivers

We replace the missing part between the onion layers already replaced on the forward path and the onion layers already replaced on the backward path.

Hybrid $\mathcal{H}_3$. In this hybrid, for the first forward communication for which in the adversarial processing no recognition falsifying modification (i.e. a modification on η) occurred (and no other modification caused the processing to fail) so far, forward onion layers from its last honest relay to the corrupted receiver are replaced with random onions sharing this path and message and the first part of the reply-path. More precisely, this machine acts like $\mathcal{H}_2^{*\leftarrow}$ except that the processing of O_j; i.e. the consecutive onion layers $O_{j+1}, \ldots, O_{n+1}$ from a communication of an honest sender, starting at the last honest node P_j to the corrupted receiver P_{n+1} are replaced with $\bar{O}_1, \ldots, \bar{O}_{n-j+1}$. Thereby for a honestly chosen $\mathcal{R}$; $\bar{O}_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}}, (PK)_{\mathcal{P}^{\leftarrow}})$ where m is the message of this communication,[33] $\mathcal{P} = (P_j, \ldots, P_{n+1})$ is the path from P_j

[32] 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.

[33] $\mathcal{H}_3$ knows this message as it simulates the honest sender.

to P_{n+1} and $\mathcal{P}^{\leftarrow}$ is the first part of the reply-path (until the first honest node), that an reply to the original onion would have taken.[34]

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

$\mathcal{H}_2^{*\leftarrow} \approx_I \mathcal{H}_3$. Similar to $\mathcal{H}_1^* \approx_I \mathcal{H}_2$ the forward onion layers before P_j are independent and hence can be simulated for the distinguisher by an attack on $TI^{\leftrightarrow}$. Similar to $\mathcal{H}_1^{*\leftarrow} \approx_I \mathcal{H}_2^{\leftarrow}$ the backward onion layers after $\mathcal{P}^{\leftarrow}.last$ are independent and hence can be simulated for the distinguisher by an attack on $TI^{\leftrightarrow}$. The remaining outputs suffice to construct an attack on $TI^{\leftrightarrow}$ similar to the one on $LU^{\rightarrow}$ in $\mathcal{H}_1^*$ and $\mathcal{H}_2$.

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

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

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

$\mathcal{H}_3^* \approx_I \mathcal{H}_4$. For the interaction with the environment from the protocol/ideal functionality, it is easy to see that the simulator directly gets the information it needs from the outputs of the ideal functionality to the adversary: whenever an honest 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 the real protocol, the environment is notified by honest nodes when they receive an onion together with some random ID that the environment sends back to signal that the honest node is done processing the onion. The same is done in the ideal functionality. Notice that the simulator ensures that every communication is simulated in $\mathcal{F}$ such that those notifications arrive at

[34] $\mathcal{H}_3$ knows this reply path as the forward onion was constructed by an honest party.

the environment without any difference (this includes them having the same repliability).

For the interaction with the real world adversary, we distinguish the outputs in communications from honest and corrupted senders. 0) Corrupted (forward) senders: In the case of a corrupted sender both $\mathcal{H}_3^*$ and $\mathcal{H}_4$ (i.e. $\mathcal{S}+\mathcal{F}$) 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 $\mathcal{H}_3^*$ and $\mathcal{H}_4$ (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 $\mathcal{H}_3^*$ and $\mathcal{H}_4$ 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 $\mathcal{H}_4$. As we have shown indistinguishability in every step, we have indistinguishability in their views.

E Proof Sketches of Further Properties for our UE Scheme

E.1 Forwards Layer-Unlinkability, Honest Relay

We assume a fixed, but arbitrary PPT algorithm $\mathcal{A}_{LU^{\rightarrow}}$ as adversary against the $LU^{\rightarrow}$ game and use a sequence of hybrid games $\mathcal{H}$ for our proof. Let $\mathcal{X}_i$ be the event that $\mathcal{A}_{LU^{\rightarrow}}$ outputs $b' = 1$ in the i-th hybrid game $\mathcal{H}_i$. We start with the $LU^{\rightarrow}$ game with $b = 0$ as first hybrid and transform it to the $LU^{\rightarrow}$ game with $b = 1$, while showing that the probability of $\mathcal{X}$ in the first and last hybrid are negligibly close to each other.

Hybrid 1) $LU^{\rightarrow}_{(b=0)}$**.** The $LU^{\rightarrow}$ game with b chosen as 0.

Hybrid 2). As Hybrid 1), but with differences in the following steps:

5. The challenger creates the onion as before, but encrypts $\bar{0}\ldots\bar{0}$ instead of $k_j^\eta, k_j^\gamma, \Delta_j$ for E_j (but still encrypts other blocks of the header with the real k_j^η, the payload with the real Δ_j and MACs with k_j^γ):

$$E_j = \mathsf{PK.Enc}_{PK_j}(0,\ldots,0)$$

$$B_j^1 = \mathsf{PRP.Enc}_{k_j^\eta}(P_{j+1}, E_{j+1}, \gamma_{j+1})$$

$$B_j^i = \mathsf{PRP.Enc}_{k_j^\eta}(B_{j+1}^{i-1}) \text{ for } 2 \leq i \leq 2N-1$$

The challenger calculates the new MAC for the blocks. All the later layers $E_{\geq j+1}, B^i_{\geq j+1}$ are constructed as before but using the replacements for the calculations, i.e. the onion layer O_j is wrapped as before.

Table 2. Overview Proof for $LU^{\rightarrow}$, $j < n+1$

Hybrid	Description	Reduction
1)	The $LU^{\rightarrow}$ game with challenge bit chosen as 0	
2)	We replace the temporary keys $k_j^{\eta}, k_j^{\gamma}, \Delta_j$ at the honest relay by 0..0 before they are encrypted in E_j (and adapt recognizeOnion to the new header), but still use the real keys for the processing.	PK-CCA2
3)	We let the oracles in step 7 output a fail, if the challenge E_j is recognized, but other parts of the header differ.	SUF-CMA
4)	We replace the blocks $B_j^1, ..., B_j^{2N-j}$ by $R_1, R_2, \ldots, R_{2N-j}$ with R_i being randomly chosen (and adapt recognizeOnion to the new header), but use the real blocks for the processing.	PRP-CCA
5)	We let the oracles in step 7 output a fail, if the challenge header η_j is recognized, but the payload does not include the correct plaintext.	UP-INT-PTXT
6)	We let the Proc oracle in step 7 output the replicated layer $j+1$:$(FormOnion(j+1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}))$, if the challenge η_j is recognized, the payload matches, and real processing of the given onion would not fail.	Perfect Re-Encryption
7)	We replace the content δ_j by a random string of the same length.	UP-IND-RCCA
8)	We revert the changes made in Game 5).	UP-INT-PTXT
9)	We replace the block B_j^1 by $(\perp, \perp, \perp)$ (and adapt recognizeOnion to the new header).	PRP-CCA
10)	We revert the changes made in Game 3).	SUF-CMA
11)	We revert the changes made in Game 2).	PK-CCA2
12)	We use FormOnion with the parameter of the $b = 1$ case to generate the first challenge onion layer. This is the $LU^{\rightarrow}$ game with challenge bit chosen as 1.	Same behavior except for new draw of randomness

6. The challenger gives the final O_1 to the adversary.
7. RecognizeOnion now checks for the adapted header (as constructed above) and if E_j is reused in a not recognized onion, the original keys $k_j^{\eta}, k_j^{\gamma}, \Delta_j$ are returned as decryption.

Hybrid 1) $\approx_{IND}$ Hybrid 2). Assume there exists a distinguisher $\mathcal{D}$ that can distinguish Hybrid 1) and 2). We can build an attack $\mathcal{A}_{CCA2}$ on the CCA2 security of the PK encryption scheme:

1. $\mathcal{A}_{CCA2}$ gets the public key PK from the challenger Ch_{CCA2}, picks an honest router's name P_j as $Ch_{LU\rightarrow}$ would and gives both to $\mathcal{D}$.
2. $\mathcal{A}_{CCA2}$ keeps an η-list and answers all queries from the $\mathcal{D}$ (as $Ch_{LU\rightarrow}$ would; including rejecting already seen headers). To decrypt ciphertexts under PK (possible in E of the header), $\mathcal{A}_{CCA2}$ uses the decryption oracle provided by the challenger Ch_{CCA2}.
3. $\mathcal{A}_{CCA2}$ gets the challenge choices from $\mathcal{D}$.
4. $\mathcal{A}_{CCA2}$ checks the challenge choices from $\mathcal{D}$ as $Ch_{LU\rightarrow}$ would.
5. $\mathcal{A}_{CCA2}$ sends $m_0 = (k_j^{\eta}, k_j^{\gamma}, \Delta_j)$ and $m_1 = (0 \ldots 0)$ to the challenger Ch_{CCA2} and receives the ciphertext c that $\mathcal{A}_{CCA2}$ uses as $E_j = c$ and calculates the MAC for it. Other than that $\mathcal{A}_{CCA2}$ forms the onion O_1 just as the challenger $Ch_{LU\rightarrow}$ in the hybrids would.[35]

[35] Note that anything except for E_j is constructed exactly in the same way in both hybrids.

Table 3. Overview Hybrids for $LU^{\rightarrow}$, $j < n+1$

	O_1:	$k_j^\eta, k_j^\gamma, \Delta_j$ in E_j	$B_j^1, \ldots, B_j^{2N-j}$	δ_j	Oracle
1)	param. $b=0$	real $k_j^\eta, k_j^\gamma, \Delta_j$	contains path after P_j	contains m	honest proc.
2)		$(0,\ldots,0)$			
3)					fail, if $E_1 = exp$, η modif.
4)			$R_1, R_2, \ldots, R_{2N-j}$		
5)					fail, if $\eta_1 = exp$, δ modif.
6)					recog+ FormOnion($i>1$)
7)				rdm $\bar{m}$	
8)					proc, if $\eta_1 = exp$, δ modif.
9)			$\mathsf{PRP.Enc}(\perp,\perp,\perp), R_2, \ldots, R_{2N-j}$		
10)					proc, if $E_1 = exp$, η modif.
11)		real $k_j^\eta, k_j^\gamma, \Delta_j$			
12)	param. $b=1$	(real $k_j^\eta, k_j^\gamma, \Delta_j$)	(receiver signal and rdm blocks)	(contains $\bar{m}$)	(recog.+ FormOnion($i>1$))

6. $\mathcal{A}_{CCA2}$ gives O_1 to $\mathcal{D}$.
7. $\mathcal{A}_{CCA2}$ answers the oracles for $\mathcal{D}$ just as $Ch_{LU\rightarrow}$ would. If $\mathcal{A}_{CCA2}$ needs to decrypt $E_j = c$ under PK, it uses the original keys $k_j^\eta, k_j^\gamma, \Delta_j$ as result. For all other requests; to decrypt ciphertexts under PK, $\mathcal{A}_{CCA2}$ uses the decryption oracle of Ch_{CCA2}.
8. $\mathcal{A}_{CCA2}$ receives the guess from $\mathcal{D}$ and uses it as its own guess.

Note that $\mathcal{A}_{CCA2}$ simulates Hybrid 1) for $b=0$ and Hybrid 2) for $b=1$ and thus wins the CCA2 game with the same advantage as $\mathcal{D}$ distinguishes the hybrids.

Security loss: $|\Pr(\mathcal{X}_1) - \Pr(\mathcal{X}_2)| \leq \epsilon_{\text{PK-CCA2}}$ with $\epsilon_{\text{PK-CCA2}}$ being the $CCA2$-advantage of some efficient adversary against our PK encryption scheme (which is negligible according to our choice).

Hybrid 3). As Hybrid 2) but with differences in the following step:

7. If an onion is handed to the oracles that reuses E_j, but changes another part of the header, i.e. is not the recognized as challenge onion processing, processing fails.

Hybrid 2) $\approx_{IND}$ Hybrid 3). Due to the SUF-CMA of our MAC and the already replaced MAC key, a successful processing of $(E_j, B, \gamma) \neq (E_j, B_j, \gamma_j)$, i.e. an onion with reused E_j but modified header, can only happen with negligible probability and except for these cases the hybrids are identical.

Security loss: $|\Pr(\mathcal{X}_2) - \Pr(\mathcal{X}_3)| \leq \epsilon_{\text{SUF-CMA}}$ with $\epsilon_{\text{SUF-CMA}}$ being the SUF-CMA-advantage of some efficient adversary against our used MAC scheme (which is negligible according to our choice).

Hybrid 4). As Hybrid 3), but with differences in the following step:

5. The challenger creates the onion as before *but the blocks $B_j^1, \ldots, B_j^{2N-j}$ are replaced with $R_1, R_2, \ldots R_{2N-j}$ with R_i being randomly chosen blocks to calculate* O_1:

   $E_j = \mathsf{PK.Enc}_{PK_j}(0,\ldots,0)$

   $B_j^i = (R_i)$ for $1 \leq i \leq 2N-j$ with R_i being the randomly generated

   The challenger continues to wrap the onion to create O_1.

7. The challenger answers oracles as in Hybrid 3), except if the header reuses E_j and B_j^i. In this case, the challenger replaces the header with the one first calculated for this position and processes it as usual.

Hybrid 3) $\approx_{IND}$ Hybrid 4). Assume there exists a distinguisher $\mathcal{D}$ that can distinguish Hybrid 3) and 4). We can build an attack $\mathcal{A}_{CCA}$ on the PRP-CCA security of the PRP:

1. $\mathcal{A}_{CCA}$ picks an honest router's name P_j and public key PK as $Ch_{LU\rightarrow}$ would and gives both to $\mathcal{D}$.
2. $\mathcal{A}_{CCA}$ answers the oracle queries from $\mathcal{D}$ as $Ch_{LU\rightarrow}$ would (including rejecting already seen headers).
3. $\mathcal{A}_{CCA}$ gets the challenge choices from $\mathcal{D}$.
4. $\mathcal{A}_{CCA}$ checks the challenge choices from $\mathcal{D}$ as $Ch_{LU\rightarrow}$ would.
5. $\mathcal{A}_{CCA}$ constructs the onion O_1 as before and sends the blocks $(P_{j+1}, E_{j+1}, \gamma_{j+1}), B_{j+1}^1, \ldots, B_{j+1}^{2N-j-1}$ to the challenger Ch_{CCA}. The challenger replies with blocks $\tilde{B}_j^1, \ldots, \tilde{B}_j^{2N-j}$ as encryption. $\mathcal{A}_{CCA}$ replaces the calculated blocks $B_j^1, \ldots, B_j^{2N-j}$ with the ones received from the challenger and continues to calculates O_1 from it by wrapping and adapting the MAC.
6. $\mathcal{A}_{CCA}$ gives O_1 to $\mathcal{D}$.
7. $\mathcal{A}_{CCA}$ answers the oracles for $\mathcal{D}$ by processing it, if it does not use E_j. Otherwise (it uses E_j): if some of the blocks $\tilde{B}_j^1, \ldots, \tilde{B}_j^{2N-j}$ are changed, $\mathcal{A}_{CCA}$ returns a fail (as introduced in Hybrid 3)). If all blocks are as expected, it calculates the answer as in step 7 of Hybrid 4).
8. $\mathcal{A}_{CCA}$ receives the guess from $\mathcal{D}$ and uses it as its own guess.

$\mathcal{A}_{CCA}$ simulates Hybrid 3) for $b = 0$ and Hybrid 4) for $b = 1$ and thus wins the PRP-CCA game with the same advantage as $\mathcal{D}$ distinguishes the hybrids.

Security loss: $|\Pr(\mathcal{X}_3) - \Pr(\mathcal{X}_4)| \leq \epsilon_{\text{PRP-CCA}}$ with $\epsilon_{\text{PRP-CCA}}$ being the CCA-advantage of some efficient adversary against our used PRP (which is negligible according to our choice).

Hybrid 5). As Hybrid 4) but with differences in the following step:

7. The challenger replies with a fail to all $\mathsf{Proc}(P_H, O)$ requests with $\eta = \eta_j$, i.e. RecognizeOnion is true - the header corresponds to the one we created and $\mathsf{UE.Dec}_{k_j^\Delta}(\delta) \neq m$, i.e. the payload was modified.[36] Otherwise, the challenger processes the onions for the oracles as before.

Hybrid 4) $\approx_{IND}$ Hybrid 5). Let $\mathcal{B}$ be the (bad) event that an $\mathsf{Proc}(P_H, O)$ request with $\eta = \eta_j$ and $\mathsf{UE.Dec}_{k_j^\Delta}(\delta) \neq m$ is not replied with a fail in Hybrid 4). Hybrid 4) and 5) work on the same underlying probability space. Thus both games indeed are identical, except if $\mathcal{B}$ happens.

Idea: If $\mathcal{B}$ happens, we can use the payload of the corresponding onion to break UP-INT-PTXT. We therefore assume an distinguisher $\mathcal{D}$ distinguishing

[36] Note that the challenger can check this as it knows all UE keys because it created the onion.

Hybrid 4) and 5). Note that we can recognize the header belonging to $\mathcal{B}$ and $\mathcal{D}$ has only one try with this header at the oracle (due to the duplicate check with the η-list). We thus do *not* have to answer $\mathcal{D}$'s oracle request during $\mathcal{B}$ but instead use the included payload to break UP-INT-PTXT. Therefore, we carefully only progress until the epoch j (corresponding to the input for the honest relay) in the UP-INT-PTXT game, only request tokens (no keys) and create later keys and tokens with GenKey and GenTok. *Precise steps of* $\mathcal{A}_{UP\text{-}INT\text{-}PTXT}$*:*

1. Pick random router names P_H, P_S and generate corresponding key pairs $(PK_H, SK_H) \leftarrow G(1^\lambda, p, P_H)$, $(PK_S, SK_S) \leftarrow G(1^\lambda, p, P_S)$. Send P_H, P_S and PK_H, PK_S to $\mathcal{D}$.
2. Oracle access: Upon receiving onions O and messages m_i from $\mathcal{D}$, create and update the η-list to detect replicates, and if it is no replicate; use SK_H resp. SK_S to ProcOnion(SK_H, O, P_H) resp. ReplyOnion(m_i, O, P_H, SK_H).
3. Receive message m, paths $\mathcal{P}^{\rightarrow}$, $\mathcal{P}^{\leftarrow}$ and public keys PK_i for both path directions from $\mathcal{D}$.
4. Check validity of names and set $PK_j = PK_H$ and $PK^{\leftarrow}_{n^{\leftarrow}+1} = PK_S$.
5. Construct O_1 carefully using the UP-INT-PTXT oracles as follows:
   (a) $\delta_1 = \mathsf{Enc}(m)$
   (b) Use Next oracles $j-1$ times until epoch $e = j$.
   (c) Get tokens $\Delta_1, \ldots, \Delta_{j-1}$ with $\mathsf{Corrupt}(token, 2), \ldots, \mathsf{Corrupt}(token, j)$
   (d) Create $k^{\Delta}_{j+1}, \ldots, k^{\Delta}_{n+1}$ as $k^{\Delta}_i \leftarrow \mathsf{UE.GenKey}(sp)$ for all $j+1 \leq i \leq n+1$ (if $\mathcal{P}^{\leftarrow} \neq \{\}$: Create $k^{\Delta^{\leftarrow}}_1, \ldots, k^{\Delta^{\leftarrow}}_{n^{\leftarrow}+1}$ similar)
   (e) Create tokens $\Delta_{j+1}, \ldots, \Delta_n$ as $\Delta_i \leftarrow \mathsf{UE.GenTok}(k^{\Delta}_i, k^{\Delta}_{i+1})$ for all $j+1 \leq i \leq n$ (if $\mathcal{P}^{\leftarrow} \neq \{\}$: Create $\Delta^{\leftarrow}_1, \ldots, \Delta^{\leftarrow}_{n^{\leftarrow}}$ similar)
   (f) Pick keys $k^{\eta}_1, \ldots, k^{\eta}_{n+1}$ for the block cipher and $k^{\gamma}_1, \ldots, k^{\gamma}_{n+1}$ for the MAC randomly. (If $\mathcal{P}^{\leftarrow} \neq \{\}$, similar for keys on the backwards path)
   (g) Create the header η_1 with the keys $k^{\eta}_1, \ldots, k^{\eta}_{n+1}, \Delta_1, \ldots, \Delta_{j-1}, \Delta_{j+1}, \ldots, \Delta_n, k^{\Delta}_{n+1}$ (resp. additionally with the keys for the backward path, if $\mathcal{P}^{\leftarrow} \neq \{\}$) as the protocol does, but replace $E_j = \mathsf{PK.Enc}_{PK_j}(0, \ldots, 0)$ and $B^i_j = (R_i)$ (as in the Hybrids before). Note that we can do this without knowing Δ_j, as it has been replaced with $0 \ldots 0$ earlier.

   Construct $\bar{O}_1$: $\bar{O}_1 \leftarrow \text{FormOnion}(1, \bar{\mathcal{R}}, \bar{m}, \bar{\mathcal{P}}^{\rightarrow}, \bar{\mathcal{P}}^{\leftarrow}, (PK)_{\bar{\mathcal{P}}^{\rightarrow}}, (PK)_{\bar{\mathcal{P}}^{\leftarrow}})$
6. Send $O_1 = (\eta_1, \delta_1)$ to $\mathcal{D}$
7. Oracle access: Upon receiving $\mathsf{Proc}(P, O)$ from $\mathcal{D}$, check if $\eta = \eta_j$ and $P = P_H$. If not or it is a Reply request, process/reply with knowledge of secret keys as before. Otherwise, output δ_j as c^* and stop.

Note that the new generation of UE keys and tokens uses the same generation functions as FormOnion and thus $\mathcal{A}_{\text{UP-INT-PTXT}}$ simulates the hybrids perfectly until $\mathcal{B}$ occurs.

$\mathcal{A}_{UP\text{-}INT\text{-}PTXT}$ is valid: As UE.ReEnc does not output a fail, UE.Dec (with the corresponding key) does not output a fail. This follows from Perfect Re-Encryption.[37] Further, this is no trivial win, as $Q*$ only contains m: $Q^* = \{(0, m), (1, m), \ldots, (j-1, m)\}$ and K contains no keys at all $K = \emptyset$.

[37] Assume, UE.ReEnc does not fail, but UE.Dec results in a fail, i.e. output $\perp$. This means UE.Enc(UE.Dec()) would also output a fail $\perp$ (see Note in Perfect Re-

$\mathcal{A}_{UP\text{-}INT\text{-}PTXT}$ *is successful:* Unless $\mathcal{B}$ happens, Hybrid 4) and 5) are identical. If $\mathcal{B}$ happens, $\mathcal{A}_{\text{UP-INT-PTXT}}$ wins.

Security loss: $|\Pr(\mathcal{X}_4) - \Pr(\mathcal{X}_5)| \leq \epsilon_{\text{UP-INT-PTXT}}$ with $\epsilon_{\text{UP-INT-PTXT}}$ being the UP-INT-PTXT-advantage of some efficient adversary against our used UE scheme (which is negligible according to our choice).

Hybrid 6). As Hybrid 5), but with differences in the following step:

7. The challenger uses P_{j+1} and FormOnion$(j+1, \mathcal{R}, m, \mathscr{P}^{\rightarrow}, \mathscr{P}^{\leftarrow}, (PK)_{\mathscr{P}^{\rightarrow}}, (PK)_{\mathscr{P}^{\leftarrow}})$ to answer $\mathsf{Proc}(P_H, O)$ requests with $\eta = \eta_j$, i.e. RecognizeOnion is true; if η is not on the η^H-list and processing of O would not have failed (this includes failing because of the wrong content as in Hybrid 5.). Otherwise, the challenger processes the onions for the oracles as before.

Hybrid 5) $\approx_{IND}$ Hybrid 6). The hybrids only differ in the replied layers for $\mathsf{Proc}(P_H, O)$ requests with $\eta = \eta_j$. FormOnion constructs the header deterministically just as before, hence the header of the replied layers are equal. The payload carries the same content, as otherwise the output would fail both in Hybrid 5) and 6). Thus the only difference is the re-encryption (Hybrid 5)/fresh encryption (Hybrid 6) of the plaintext for the payload. Those encryptions are indistinguishable due to the perfect Re-Encryption property of the UE-scheme (note that perfect Re-Encryption holds also for multiple times re-encrypted ciphertexts): $\eta_j^{\mathcal{H}_5} = \eta_j^{\mathcal{H}_6}$ (header is deterministic),

$$\delta_j^{\mathcal{H}_5} = \mathsf{UE.ReEnc}^{j}_{\Delta_{i,1\leq i\leq j-1}}(m) \overset{\text{dist}}{\equiv} \delta_j^{\mathcal{H}_6} = \mathsf{UE.Enc}_{k_j^{\Delta}}(m)$$

$$\Leftrightarrow \mathsf{UE.Enc}(k^{\text{new}}, \mathsf{UE.Dec}(k^{\text{old}}, C)) \overset{\text{dist}}{\equiv} \mathsf{UE.ReEnc}(\Delta, C)$$

Security loss: $|\Pr(\mathcal{X}_5) - \Pr(\mathcal{X}_6)| = 0$

Hybrid 7). As Hybrid 6), but with differences in the following step:

5. The challenger creates the onion as before but *with random payload*:
   $\delta_1 = (\mathsf{UE.Enc}_{k_1^{\Delta}}(R))$ with R being a random message
7. The expected plaintext encrypted in the payload of the recognized onion is adapted to be R. Otherwise, the oracle works as before.

Hybrid 6) $\approx_{IND}$ Hybrid 7). Assume there exists a distinguisher $\mathcal{D}$ that can distinguish Hybrid 6) and 7). We build an attack $\mathcal{A}_{CCA}$ on the UP-IND-RCCA security of the UE scheme: *Precise steps of* $\mathcal{A}_{CCA}$:

1. Pick random router names P_H, P_S and generate corresponding key pairs $(PK_H, SK_H) \leftarrow G(1^{\lambda}, p, P_H)$, $(PK_S, SK_S) \leftarrow G(1^{\lambda}, p, P_S)$. Send P_H, P_S and PK_H, PK_S to $\mathcal{D}$.
2. Oracle access: Use SK_H resp. SK_S to answer the oracles as in the original game.
3. Receive message m, paths $\mathscr{P}^{\rightarrow}$, $\mathscr{P}^{\leftarrow}$ and public keys PK_i for both path directions from $\mathcal{D}$.
4. Check validity of names and set $PK_j = PK_H$ and $PK^{\leftarrow}_{n^{\leftarrow}+1} = PK_S$.

Encryption) and thus $\mathsf{UE.ReEnc}$ has to fail as well ($\overset{\text{dist}}{\equiv}$), completing the indirect argument.

5. Construct O_1 using the UP-IND-RCCA oracles and challenge as follows:
   (a) Send $M_0 = m, M_1 = R \leftarrow_R \mathcal{M}_{sp}$ with $|m| = |R|$ to Ch_{CCA} and receive $C^* \leftarrow_R \mathsf{UE.Enc}(k_1^{\Delta}, M_b)$ from Ch_{CCA}. It sets $\delta_1 = C^*$.
   (b) Use Next oracles $j-1$ times until epoch $e = j$.
   (c) Get tokens $\Delta_1, \ldots, \Delta_{j-1}$ with $\mathsf{Corrupt}(token, 2), \ldots, \mathsf{Corrupt}(token, j)$
   (d) Create $k_{j+1}^{\Delta}, \ldots, k_{n+1}^{\Delta}$ as $k_i^{\Delta} \leftarrow \mathsf{UE.GenKey}(sp)$ for all $j+1 \leq i \leq n+1$ (if $\mathcal{P}^{\leftarrow} \neq \{\}$: Create $k^{\Delta^{\leftarrow}}_1, \ldots, k^{\Delta^{\leftarrow}}_{n^{\leftarrow}+1}$ similar)
   (e) Create tokens $\Delta_{j+1}, \ldots, \Delta_n$ as $\Delta_i \leftarrow \mathsf{UE.GenTok}(k_i^{\Delta}, k_{i+1}^{\Delta})$ for all $j+1 \leq i \leq n$ (if $\mathcal{P}^{\leftarrow} \neq \{\}$: Create $\Delta_1^{\leftarrow}, \ldots, \Delta_{n^{\leftarrow}}^{\leftarrow}$ similar)
   (f) Pick keys $k_1^{\eta}, \ldots, k_{n+1}^{\eta}$ for the block cipher and $k_1^{\gamma}, \ldots, k_{n+1}^{\gamma}$ for the MAC randomly. (If $\mathcal{P}^{\leftarrow} \neq \{\}$, similar for keys on the backwards path)
   (g) Create the header η_1 with the keys $k_1^{\eta}, \ldots, k_{n+1}^{\eta}, \Delta_1, \ldots, \Delta_{j-1}, \Delta_{j+1}, \ldots, \Delta_n, k_{n+1}^{\Delta}$ (resp. additionally with the keys for the backward path, if $\mathcal{P}^{\leftarrow} \neq \{\}$) as the protocol does, but replace $E_j = \mathsf{PK.Enc}_{PK_j}(0, \ldots, 0)$ and $B_j^i = (R_i)$ (as in the Hybrids before). Store the intermediate result η_{j+1} for later use. Note that we can do this without knowing Δ_j, as it has been replaced with $0 \ldots 0$ earlier.

   Construct $\bar{O}_1$: $\bar{O}_1 \leftarrow \text{FormOnion}(1, \bar{\mathcal{R}}, \bar{m}, \bar{\mathcal{P}}^{\rightarrow}, \bar{\mathcal{P}}^{\leftarrow}, (PK)_{\bar{\mathcal{P}}^{\rightarrow}}, (PK)_{\bar{\mathcal{P}}^{\leftarrow}})$
6. Send $O_1 = (\eta_1, \delta_1)$ to $\mathcal{D}$
7. Oracle access: Answer all Reply as before. Upon receiving $\mathsf{Proc}(P, O = (\eta, \delta))$ from $\mathcal{D}$:
   – check if $\eta = \eta_j$ and $P = P_H$:
     • If not, process/reply with knowledge of secret keys as before (if it is not on the η-List).
     • If so, ($\eta = \eta_j$, i.e. the challenge is recognized):
       (a) Request $\mathsf{Dec}(\delta)$ from Ch_{CCA} and output a fail if a message m' is returned. (This is, the header is equal, but the payload has been modified which requires a fail since Hybrid 5).)
       (b) If no fail was output: (we use[38] $FormOnion(j+1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})$ as introduced in Hybrid 6.)
          ∗ $\delta_{+1} = \mathsf{UE.Enc}(k_{j+1}^{\Delta}, m)$.(k_{j+1}^{Δ} was created in Step 5.(d).)
          ∗ $\eta_{+1} = \eta_{j+1}$. (η_{j+1} was stored during Step 5.(g).)
          ∗ Give P_{j+1} and $O_{+1} = (\delta_{+1}, \eta_{+1})$ as reply to $\mathcal{D}$.
8. Receive the guess from $\mathcal{D}$ and return it as own guess.

Note that the keys and tokens during FormOnion and in our simulation are generated with the same functions and parameters. Thus $\mathcal{A}_{CCA}$ indeed simulates Hybrid 6) for $b = 0$ and Hybrid 7) for $b = 1$ and thus wins the UP-IND-RCCA game with the same advantage as $\mathcal{D}$ distinguishes the hybrids.

Security loss: $|\Pr(\mathcal{X}_6) - \Pr(\mathcal{X}_7)| \leq \epsilon_{\text{UP-IND-RCCA}}$ with $\epsilon_{\text{UP-IND-RCCA}}$ being the UP-IND-RCCA-advantage of some efficient adversary against our used UE scheme (which is negligible according to our choice).

[38] We do not know all random coins $\mathcal{R}$ used for all onion layers of this communication. To create the $j+1$ -th layer we however only need the right randomness for the keys that we generated ourselves during 5. (d) -(f) and we can pick the remaining parts of $\mathcal{R}$ arbitrarily as they will not be used for layer $j+1$.

Hybrid 8) until Hybrid 12). Those hybrids merely revert earlier hybrids. Thus a similar argumentation as in the original hybrid transition applies.

Total Security loss: $|\Pr(\mathcal{X}_1) - \Pr(\mathcal{X}_{12})| \leq 2 \cdot \epsilon_{\text{PK-CCA2}} + 2 \cdot \epsilon_{\text{SUF-CMA}} + 2 \cdot \epsilon_{\text{PRP-CCA}} + 2 \cdot \epsilon_{\text{UP-INT-PTXT}} + \epsilon_{\text{UP-IND-RCCA}}$ with ϵ defined as the advantage of some efficient adversary against the corresponding primitive, which are all negligible due to our choices.

E.2 Forwards Layer-Unlinkability, Honest Receiver: $LU^{\rightarrow}, j = n+1$:

The hybrids work exactly as before, except for a small change in Hybrid 6, which we detail below.

Table 4. Overview Proof for $LU^{\rightarrow}, j = n + 1$

Hybrid	Description	Reduction
1)	The $LU^{\rightarrow}$ game with challenge bit chosen as 0	
2)	We replace the temporary keys $k_j^{\eta}, k_j^{\gamma}, k_{j=n+1}^{\Delta}$ at the honest relay by 0..0 before they are encrypted in E_j (and adapt recognizeOnion to the new header), but still use the real keys for the processing.	PK-CCA2
3)	We let the oracles in step 7 output a fail, if the challenge E_j is recognized, but other parts of the header differ.	SUF-CMA
4)	We replace the blocks $B_j^1, ..., B_j^{2N-j}$ by $R_1, R_2, \ldots, R_{2N-j}$ with R_i being randomly chosen (and adapt recognizeOnion to the new header), but use the real blocks for the processing.	PRP-CCA
5)	We let oracles in step 7 output a fail, if the challenge header η_j is recognized, but the payload does not include the correct plaintext.	UP-INT-PTXT
6)	We let the oracles in step 7 output the replicated layer $j+1$:$(FormOnion(j+1, \mathcal{R}, m^{\leftarrow}, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}))$ for the Reply$(P_H, O, m^{\leftarrow})$ request and we output $(\perp, m)$ for the Proc(P_H, O), if the challenge η_j is recognized, the payload matches, and real processing of the given onion would not fail.	Reply: Same behavior as before, Proc: UE-Correctness
7)	We replace the content δ_j by a random string of the same length.	UP-IND-RCCA
8)	We revert the changes made in Game 5).	UP-INT-PTXT
9)	We replace the block B_j^1 by PRP.Enc$(\perp, \perp, \perp)$ (and adapt recognizeOnion to the new header).	PRP-CCA
10)	We revert the changes made in Game 6).	SUF-CMA
11)	We revert the changes made in Game 5).	PK-CCA2
12)	We use FormOnion with the parameter of the $b = 1$ case to generate the first challenge onion layer. The $LU^{\rightarrow}$ game with challenge bit chosen as 1	Same behavior except for new draw of randomness

$\mathcal{H}_4 \rightarrow \mathcal{H}_5$: Note that even for the receiver this works as before, as onions have to be processed before they can be replied (Assumption 6 and included in $LU^{\rightarrow}$).

Precise steps of $\mathcal{A}_{UP\text{-}INT\text{-}PTXT}$:

1. Pick random router names P_H, P_S and generate corresponding key pairs $(PK_H, SK_H) \leftarrow G(1^{\lambda}, p, P_H)$, $(PK_S, SK_S) \leftarrow G(1^{\lambda}, p, P_S)$. Send P_H, P_S and PK_H, PK_S to $\mathcal{D}$.

Table 5. Overview Hybrids for $LU^{\rightarrow}$, $j = n+1$

	O_1:	$k_j^{\eta}, k_j^{\gamma}, k_j^{\Delta}$ in E_j	$B_j^1, \ldots, B_j^{2N-j}$	δ_j	Oracle
1)	param. $b = 0$	real $k_j^{\eta}, k_j^{\gamma}, k_j^{\Delta}$	contains path after P_j	contains m	honest proc.
2)		$(0, \ldots, 0)$			
3)					fail, if $E_1 = exp$, η modif.
4)			$R_1, R_2, \ldots, R_{2N-j}$		
5)					fail, if $\eta_1 = exp$, δ modif.
6)					recog+ FormOnion($i > n+1$)
7)				rdm $\bar{m}$	
8)					proc, if $\eta_1 = exp$, δ modif.
9)			$\mathsf{PRP.Enc}(\perp, \perp, \perp), R_2, \ldots, R_{2N-j}$		
10)					proc, if $E_1 = exp$, η modif.
11)		real $k_j^{\eta}, k_j^{\gamma}, k_j^{\Delta}$			
12)	param. $b = 1$	(real $k_j^{\eta}, k_j^{\gamma}, k_j^{\Delta}$)	(receiver signal and rdm blocks)	(contains $\bar{m}$)	(recog.+FormOnion($i > n+1$))

2. Oracle access: Use SK_H resp. SK_S to answer the requests as in the original game.
3. Receive message m, paths $\mathcal{P}^{\rightarrow}$, $\mathcal{P}^{\leftarrow}$ and public keys PK_i for both path directions from $\mathcal{D}$.
4. Check validity of names and set $PK_j = PK_H$ and $PK^{\leftarrow}_{n^{\leftarrow}+1} = PK_S$.
5. Construct O_1 carefully using the UP-INT-PTXT oracles as follows:
   (a) $\delta_1 = \mathsf{Enc}(m)$
   (b) Use Next oracles n times until epoch $e = n+1$.
   (c) Get tokens $\Delta_1, \ldots, \Delta_n$ with $\mathsf{Corrupt}(token, 2), \ldots, \mathsf{Corrupt}(token, n+1)$
   (d) If $\mathcal{P}^{\leftarrow} \neq \{\}$ create keys $k^{\Delta^{\leftarrow}}$ and tokens $\Delta^{\leftarrow}$ for the backwards path with $\mathsf{UE.GenKey}$ and $\mathsf{UE.GenTok}$.
   (e) Pick keys $k_1^{\eta}, \ldots, k_{n+1}^{\eta}$ for the block cipher and $k_1^{\gamma}, \ldots, k_{n+1}^{\gamma}$ for the MAC randomly. (If $\mathcal{P}^{\leftarrow} \neq \{\}$, similar for keys on the backwards path)
   (f) Create the header with the keys $k_1^{\eta}, \ldots, k_{n+1}^{\eta}, \Delta_1, \ldots, \Delta_n$ (resp. additionally with the keys for the backward path, if $\mathcal{P}^{\leftarrow} \neq \{\}$) as the protocol does, but replace $E_j = \mathsf{PK.Enc}_{PK_j}(0, \ldots, 0)$ and $B_j^i = (R_i)$ (as in the Hybrids before). Note that we can do this without knowing k_{n+1}^{Δ}, as it has been replaced with $0 \ldots 0$ in the earlier Hybrids.

   Construct $\bar{O}_1$ as before:

$$\bar{O}_1 \leftarrow \text{FormOnion}(1, \bar{\mathcal{R}}, \bar{m}, \bar{\mathcal{P}}^{\rightarrow}, \bar{\mathcal{P}}^{\leftarrow}, (PK)_{\bar{\mathcal{P}}^{\rightarrow}}, (PK)_{\bar{\mathcal{P}}^{\leftarrow}})$$

6. Send O_1 to $\mathcal{D}$
7. Oracle access: Upon receiving $\mathsf{Proc}(P, O)$ from $\mathcal{D}$, check if $\eta = \eta_j$ and $P = P_H$. If not, process/reply with knowledge of secret keys as before (if it is not on the η-List). Otherwise, output δ_j as c^* and stop.

Hybrid 6). As Hybrid 5), but with differences in the following step:

7. The challenger uses $P_{j=n+1}$ and FormOnion($j+1, \mathcal{R}, m^{\leftarrow}, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}$, $(PK)_{\mathcal{P}^{\leftarrow}}$) to answer $\mathsf{Reply}(P_H, O, m^{\leftarrow})$ requests with $\eta = \eta_j$, i.e. RecognizeOnion is true; if O is on the O^H-list. Further, for a $\mathsf{Proc}(P_H, O)$request, we check that $\eta = \eta_j$ is not on the η^H-list, the payload matches, (if so) output $(\perp, m)$ and store η on the η^H-list and O on the O^H-list. Otherwise, the challenger processes the onions for the oracle as before.

$\mathcal{H}_5 \to \mathcal{H}_6$: As before, except for using Correctness: Since the correct message is included and the expected challenge header is used, FormOnion outputting $(\perp, m)$ for the receiver is what would also happen during real processing as long as the UE scheme has correctness.

E.3 Backwards Layer-Unlinkability, Honest Receiver: $LU^{\leftarrow}, j^{\leftarrow} = 0$

Similar to the $LU^{\rightarrow}$ proofs, we now use $\mathcal{X}_i$ as the event that $\mathcal{A}_{LU^{\leftarrow}}$ outputs $b' = 1$ in the i-th hybrid game $\mathcal{H}_i$.

Table 6. Overview Proof for $LU^{\leftarrow}$, $j^{\leftarrow} = 0$

Hybrid	Description	Reduction
1)	The $LU^{\leftarrow}$ game with challenge bit chosen as 0	
2)	We replace the temporary keys $k^{\eta}_{n+1}, k^{\gamma}_{n+1}, k^{\Delta}_{n+1}$ on the forward path at the honest receiver with $0 \ldots 0$ in their encryption for E_{n+1} (and adapt recognizeOnion to the new header), but still use the real keys for the processing.	PK-CCA2
3)	We let the oracles in step 6 output a fail, if the challenge E_{n+1} is recognized, but other parts of the header differ.	SUF-CMA
4)	We replace the blocks $B^1_{n+1}, \ldots B^{2N-1}_{n+1}$ by random values when forming the challenge onion (and adapt recognizeOnion to the new header), but use the real block for the processing. (In particular, in this way we get rid of $k^{\Delta^{\leftarrow}_1}$ and all $k^{\eta^{\leftarrow}_{>j\leftarrow}}, k^{\gamma^{\leftarrow}_{>j\leftarrow}}, \Delta^{\leftarrow}_{>j\leftarrow}$)	PRP-CCA
5)	We let the keys $k^{\Delta^{\leftarrow}_1}, k^{\eta^{\leftarrow}_{>j\leftarrow}}, k^{\gamma^{\leftarrow}_{>j\leftarrow}}, \Delta^{\leftarrow}_{>j\leftarrow}$ and random padding used in step 6 be freshly chosen and use these for exception 2 of the game.	Games are equivalent
6)	We replace the content $\delta^{\leftarrow}_1$ by a random string of the same length during ReplyOnion.	UP-IND-RCCA
7)	We revert the changes made in Game 4).	PRP-CCA
8)	We revert the changes made in Game 3).	SUF-CMA
9)	We revert the changes made in Game 2).	PK-CCA2
10)	The $LU^{\leftarrow}$ game with challenge bit chosen as 1	Same Behavior

Hybrid 1

1. The adversary receives the router names P_H, P_S and challenge public keys PK_S, PK_H, chosen by the challenger by letting $(PK_H, SK_H) \leftarrow G(1^{\lambda}, p, P_H)$ and $(PK_S, SK_S) \leftarrow G(1^{\lambda}, p, P_S)$.
2. Oracle access: The adversary may submit any number of Proc and Reply requests for P_H or P_S to the challenger. For any Proc(P_H, O), the challenger checks whether η is on the η^H- list. If not, it sends the output of ProcOnion(SK_H, O, P_H), stores η on the η^H-list and O on the O^H-list. For any Reply(P_H, O, m) the challenger checks if O is on the O^H- list and if so, the challenger sends ReplyOnion(m, O, P_H, SK_H) to the adversary. (Similar for requests on P_S with the η^S-list).
3. The adversary submits
   - message m,
   - a position $j^{\leftarrow}$ with $0 \le j^{\leftarrow} \le n^{\leftarrow} + 1$,
   - a path $\mathcal{P}^{\rightarrow} = (P_1, \ldots, P_j, \ldots, P_{n+1})$, where $P_{n+1} = P_H$,

Table 7. Overview Hybrids for $LU^{\leftarrow}$, $j^{\leftarrow} = 0$ with "bw" denoting parts of the backward onion

	δ_{n+1}	$B^1_{n+1}, \ldots, B^{2N-1}_{n+1}$	$k^{\eta}_{n+1}, k^{\gamma}_{n+1}, k^{\Delta}_{n+1}$	$\delta^{\leftarrow}_1$ (bw)	Oracle (Reply)
1)	real	PRP.Enc$(\perp, \perp, \perp)$, bw path, sender & relay padding	real	contains $m^{\leftarrow}$ (adv. chosen)	recog.+ honest reply
2)			$(0, \ldots, 0)$		(adapt recog.)
3)					fail, if $E_{n+1} = exp$, η modif.
4)		random			(treat B^1_{n+1} as path-end)
5)					use fresh $k^{\Delta^{\leftarrow}_1}, k^{\eta^{\leftarrow}_{>j}}, k^{\gamma^{\leftarrow}_{>j}}, \Delta^{\leftarrow}_{>j}$
6)				random	
					if $\eta_1 = exp$, δ modif.
7)		PRP.Enc$(\perp, \perp, \perp)$, bw path sender & relay padding			use actual content of B^1_{n+1}
8)					verify MAC, if $E_{n+1} = exp$, η modif.
9)			real		
10)	real	PRP.Enc$(\perp, \perp, \perp)$, bw path sender & relay padding	real	random	use fresh $k^{\Delta^{\leftarrow}_1}, k^{\eta^{\leftarrow}_{>j}}, k^{\gamma^{\leftarrow}_{>j}}, \Delta^{\leftarrow}_{>j}$

- a path $\mathscr{P}^{\leftarrow} = (P^{\leftarrow}_1, \ldots, P^{\leftarrow}_{j^{\leftarrow}}, \ldots, P^{\leftarrow}_{n^{\leftarrow}+1} = P_S)$ with the second honest node P_S at position $n^{\leftarrow} + 1$
- and public keys for all nodes PK_i $(1 \leq i \leq n+1$ for the nodes on the path and $n+1 < i$ for the other relays).

4. The challenger checks that the chosen paths are acyclic, the router names and public keys are valid and that the same key is chosen if the router names are equal, and if so, sets $PK_{n+1} = PK_H$, $PK^{\leftarrow}_{n^{\leftarrow}+1} = PK_S$ and sets bit b at random.
5. The challenger creates the onion with the adversary's input choice and honestly chosen randomness $\mathscr{R}$:

$$O_1 \leftarrow \text{FormOnion}(1, \mathscr{R}, m, \mathscr{P}^{\rightarrow}, \mathscr{P}^{\leftarrow}, (PK)_{\mathscr{P}^{\rightarrow}}, (PK)_{\mathscr{P}^{\leftarrow}})$$

and sends O_1 to the adversary.

6. The adversary gets oracle access as in step 2) except if:

Exception 1) The request is ...

- Reply$(P_H, O, m^{\leftarrow})$ with RecognizeOnion$((n+1), O, \mathscr{R}, m, \mathscr{P}^{\rightarrow}, \mathscr{P}^{\leftarrow}, (PK)_{\mathscr{P}^{\rightarrow}}, (PK)_{\mathscr{P}^{\leftarrow}}) = True$, O is on the O^H- list and no onion with this η has been replied to before and ReplyOnion$(m^{\leftarrow}, O, P_H, SK_H) \neq \perp$:

.. then: The challenger picks the remaining return path $\bar{\mathscr{P}}^{\rightarrow} = (P^{\leftarrow}_{j^{\leftarrow}+1}, \ldots, P^{\leftarrow}_{n^{\leftarrow}+1})$, an empty backward path $\bar{\mathscr{P}}^{\leftarrow} = ()$, and a random message $\bar{m}$, another honestly chosen randomness $\bar{\mathscr{R}}$, and generates:

$$\bar{O}_1 \leftarrow \text{FormOnion}(1, \bar{\mathscr{R}}, \bar{m}, \bar{\mathscr{P}}^{\rightarrow}, \bar{\mathscr{P}}^{\leftarrow}, (PK)_{\bar{\mathscr{P}}^{\rightarrow}}, (PK)_{\bar{\mathscr{P}}^{\leftarrow}})$$

- The challenger calculates $(O_{j^{\leftarrow}+1}, P^{\leftarrow}_{j^{\leftarrow}+1}) = \text{ReplyOnion}(m^{\leftarrow}, O, P^{\leftarrow}_{j^{\leftarrow}}, SK_H)$ and gives $O_{j^{\leftarrow}+1}$ for $P^{\leftarrow}_{j^{\leftarrow}+1}$ to the adversary.

Exception 2) Proc(P_S, O) with O being the challenge onion as processed for the final receiver on the backward path, i.e.:

- RecognizeOnion$((n+1)+(n^{\leftarrow}+1), O, \mathcal{R}) = True$

.. then the challenger outputs nothing.

7. The adversary produces guess b' .

Hybrid 2 As Hybrid 1), but with differences in the following step:

5. The challenger creates the onion as before, but encrypts $0 \ldots 0$ instead of $k^{\eta}_{n+1}, k^{\gamma}_{n+1}, k^{\Delta}_{n+1}$ for E_{n+1} (but still encrypts other blocks of the header with the real k^{η}_{n+1}, the payload with the real k^{Δ}_{n+1} and MACs with k^{γ}_{n+1}):

$$E_{n+1} = \mathsf{PK.Enc}_{PK_{n+1}}(0, \ldots, 0)$$

The challenger calculates the new MAC for the blocks. All the other layers are constructed as before but using the replacements for the calculations, i.e. the onion layer O_{n+1} is wrapped as before. The challenger gives the final O_1 to the adversary.

6. RecognizeOnion now checks for the adapted header (as constructed above) and if E_{n+1} is reused in a not recognized onion, the original keys $k^{\eta}_{n+1}, k^{\gamma}_{n+1}$, k^{Δ}_{n+1} are returned as decryption.

$\mathcal{H}_1 \Rightarrow \mathcal{H}_2$:

$$|\Pr(\mathcal{X}_1) - \Pr(\mathcal{X}_2)| \leq \epsilon_{\text{PK-CCA2}}$$

with $\epsilon_{\text{PK-CCA2}}$ being the $CCA2$-advantage of some efficient adversary against our PK encryption scheme; argued similarly to $LU^{\rightarrow}, j < n+1, \mathcal{H}_1 \Rightarrow \mathcal{H}_2$.

Hybrid 3 As Hybrid 2) but with differences in the following step:

7. If an onion is handed to the oracles that reuses E_{n+1}, but changes another part of the header, i.e. is not the recognized as challenge onion processing, processing fails.

$\mathcal{H}_2 \Rightarrow \mathcal{H}_3$: *Security loss:*

$$|\Pr(\mathcal{X}_2) - \Pr(\mathcal{X}_3)| \leq \epsilon_{SUF-CMA}$$

with $\epsilon_{SUF-CMA}$ being the $SUF-CMA$-advantage of some efficient adversary against our used MAC scheme; argued similarly to $LU^{\rightarrow}, j < n+1, \mathcal{H}_2 \Rightarrow \mathcal{H}_3$.

Hybrid 4 As Hybrid 3), but with differences in the following step:

5. The challenger creates the onion as before *but the blocks* $B^1_{n+1}, \ldots, B^{2N-1}_{n+1}$ *are replaced with* $R_1, \ldots, R_{2N-1}$ *with* R_i *being randomly chosen blocks to calculate* O_1:

$$E_j = \mathsf{PK.Enc}_{PK_j}(0, \ldots, 0)$$
$$B^i_{n+1} = (R_i) \text{ for } 1 \leq i \leq 2N-1 \text{ with } R_i \text{ being the randomly generated}$$

The challenger continues to wrap the onion with the replaced B_{n+1} to create O_1.

6. The challenger answers oracles as in Hybrid 3), except if the header reuses E_{n+1} and B^i_{n+1}. In this case, the challenger replaces the header with the one first calculated for this position and processes it as usual.

$\mathcal{H}_3 \Rightarrow \mathcal{H}_4$*: Security loss:*

$$|\Pr(\mathcal{X}_3) - \Pr(\mathcal{X}_4)| \leq \epsilon_{\text{PRP-CCA}}$$

with $\epsilon_{\text{PRP-CCA}}$ being the CCA-advantage of some efficient adversary against our used PRP; argued similarly to $LU^{\rightarrow}, j < n+1, \mathcal{H}_3 \Rightarrow \mathcal{H}_4$.

Hybrid 5 As Hybrid 4), but with differences in the following step:

6. The challenger creates the reply onion in Exception 1 as before but with freshly chosen $k^{\Delta^{\leftarrow}_1}, k^{\eta^{\leftarrow}_{>j^{\leftarrow}}}, k^{\gamma^{\leftarrow}_{>j^{\leftarrow}}}, \Delta_{>j^{\leftarrow}}$, i.e. a freshly constructed backward header. Exception 2 is adapted to match these new header layers.

$\mathcal{H}_4 \rightarrow \mathcal{H}_5$*:* This step is new compared to what we saw before in $LU^{\rightarrow}$. Note however, that the keys after $j^{\leftarrow}$ are not included in the payload of the forward onion anymore (they originally were included as part of the backward header), because the payload was replaced by randomness. Further $\mathcal{H}_7$ creates the keys with the generation functions, just as the original FormOnion-call did that is responsible for there backward header in $\mathcal{H}_6$. So, this is merely a change of the point in time in which the keys are generated.

Security loss:

$$|\Pr(\mathcal{X}_4) - \Pr(\mathcal{X}_5)| = 0$$

Hybrid 6 As Hybrid 5), but with differences in the following step:

6. The challenger creates the reply onion in Exception 1 as before but *with random payload*:

$$\delta^{\leftarrow}_1 = (\mathsf{UE.Enc}_{k^{\Delta^{\leftarrow}_1}}(R)) \text{ with } R \text{ being a random message}$$

$\mathcal{H}_5 \rightarrow \mathcal{H}_6$*:* As this is the first time that we replace the backward payload, we argue this transition in more details below:

Precise steps of $\mathcal{A}_{CCA}$:

1. Pick random router names P_H, P_S and generate corresponding key pairs $(PK_H, SK_H) \leftarrow G(1^\lambda, p, P_H)$, $(PK_S, SK_S) \leftarrow G(1^\lambda, p, P_S)$. Send P_H, P_S and PK_H, PK_S to $\mathcal{D}$.
2. Oracle access: Use SK_H resp. SK_S to answer as in the original game.
3. Receive message m, paths $\mathcal{P}^{\rightarrow}$, $\mathcal{P}^{\leftarrow}$ and public keys PK_i for both path directions from $\mathcal{D}$.
4. Check validity of names and set $PK_{n+1} = PK_H$ and $PK^{\leftarrow}_{n^{\leftarrow}+1} = PK_S$.
5. Construct O_1 as before (Note that the backwards part of the header has been replaced with random bocks before and thus we do not need to involve any UP-IND-RCCA oracles for this construction.)
6. Send O_1 to $\mathcal{D}$

7. Oracle access: Process Proc requests normally. Upon receiving a Reply$(P, O = (\eta, \delta), m^{\leftarrow})$ from $\mathcal{D}$:
   - check if $\eta = \eta_n$ and $P = P_H$:
     - If not: process/reply with knowledge of secret keys and adaptations to the processing as before (if it is not on the η-List and if it not violates exception 2 (is the reply processed at the original sender)).
     - If so: Construct the reply onion carefully using the UP-IND-RCCA oracles and challenge as follows:
       (a) Sends $M_0 = m^{\leftarrow}, M_1 = R \leftarrow_{\mathrm{R}} \mathcal{M}_{sp}$ with $|m^{\leftarrow}| = |R|$ to Ch_{CCA} and receives $C^* \leftarrow_{\mathrm{R}} \mathsf{UE.Enc}(k_1^{\Delta}, M_b)$ from Ch_{CCA}. It sets $\bar{\delta}_1 = C^*$.
       (b) Use Next oracles $n^{\leftarrow}$ times until epoch $e = n^{\leftarrow} + 1$.
       (c) Get tokens $\Delta_1, \ldots, \Delta_{n^{\leftarrow}}$ with Corrupt$(token, 2), \ldots,$ Corrupt$(token, n^{\leftarrow} + 1)$
       (d) Create $k_{n^{\leftarrow}+1}^{\Delta} \leftarrow \mathsf{UE.GenKey}(sp)$
       (e) Pick keys $k_1^{\eta}, \ldots, k_{n+1}^{\eta}$ for the block cipher and $k_1^{\gamma}, \ldots, k_{n+1}^{\gamma}$ for the MAC randomly.
       (f) Create the non-repliable (backwards) header $\bar{\eta}_1$ with the keys $k_1^{\eta}, \ldots, k_{n+1}^{\eta}$, $\Delta_1, \ldots, \Delta_{n^{\leftarrow}}, k_{n^{\leftarrow}+1}^{\Delta}$ as the protocol does.
       (g) Return onion $(\bar{\eta}_1, \bar{\delta}_1)$ for $P_1^{\leftarrow}$.
       (h) Adapt exception 2 to work for the newly constructed header (the sender will not process this onion in the oracle).
8. Receive the guess from $\mathcal{D}$ and return it as own guess.

Security loss:

$$|\Pr(\mathcal{X}_5) - \Pr(\mathcal{X}_6)| \leq \epsilon_{\text{UP-IND-RCCA}}$$

with $\epsilon_{\text{UP-IND-RCCA}}$ being the UP-IND-RCCA-advantage of some efficient adversary against our used UE scheme (which is negligible according to our choice).

Remaining hybrids: The remaining hybrids just revert changes done in earlier hybrids and are argued similarly to these.

E.4 Backwards Layer-Unlinkability, Honest Relay: $LU^{\leftarrow}, j^{\leftarrow} > 0$:

Similar to the $LU^{\rightarrow}$ proofs, we now use $\mathcal{X}_i$ as the event that $\mathcal{A}_{LU^{\leftarrow}}$ outputs $b' = 1$ in the i-th hybrid game $\mathcal{H}_i$.

Hybrid 1

1. The adversary receives the router names P_H, P_S and challenge public keys PK_S, PK_H, chosen by the challenger by letting $(PK_H, SK_H) \leftarrow G(1^{\lambda}, p, P_H)$ and $(PK_S, SK_S) \leftarrow G(1^{\lambda}, p, P_S)$.
2. Oracle access: The adversary may submit any number of Proc and Reply requests for P_H or P_S to the challenger. For any Proc(P_H, O), the challenger checks whether η is on the η^H- list. If not, it sends the output of ProcOnion(SK_H, O, P_H), stores η on the η^H-list and O on the O^H-list. For any Reply(P_H, O, m) the challenger checks if O is on the O^H- list and if so, the challenger sends ReplyOnion(m, O, P_H, SK_H) to the adversary. (Similar for requests on P_S with the η^S-list).

Table 8. Overview Proof for $LU^{\leftarrow}$, $j^{\leftarrow} > 0$

Hybrid	Description	Reduction
1)	The $LU^{\leftarrow}$ game with challenge bit chosen as 0	
2)	We replace the temporary keys $k^{\eta}_{j\leftarrow}, k^{\gamma}_{j\leftarrow}, \Delta^{\leftarrow}_{j\leftarrow}$ at the honest relay with $0 \dots 0$ in their encryption for $E^{\leftarrow}_{j\leftarrow}$ (and adapt recognizeOnion to the new header) as part of the payload of O_1, but still use the real keys for the processing.	PK-CCA2
3)	We let the oracles in step 6 output a fail, if the challenge $E^{\leftarrow}_{j\leftarrow}$ is recognized, but other parts of the header differ.	SUF-CMA
4)	We replace all $B^{1\leftarrow}_{j\leftarrow}, ..., B^{2N-1\leftarrow}_{j\leftarrow}$ while constructing the header with randomness, but use the real header for $j^{\leftarrow}+1$ as answer to the corresponding Proc request. Note that replacing all $B^{\leftarrow}_{j\leftarrow}$s results in not including any keys for $> j^{\leftarrow}$ in the earlier header.	PRP-CCA
5)	We let the keys $k^{\Delta\leftarrow}_{j\leftarrow+1}, k^{\eta\leftarrow}_{>j\leftarrow}, k^{\gamma\leftarrow}_{>j\leftarrow}, \Delta^{\leftarrow}_{>j\leftarrow}$, that are used in step 6 to generate the layer for $j^{\leftarrow}+1$ when given the challenge header, be freshly chosen and also pick new randomness for the padding of the blocks without path information and use these for exception 2 of the game.	Perfect ReEncryption
6)	We replace the content $\delta^{\leftarrow}_{j}$ with a random string of the same length during ProcOnion at P_H.	UP-IND-RCCA
7)	We revert the changes made in Game 4).	PRP-CCA
8)	We revert the changes made in Game 3).	SUF-CMA
9)	We revert the changes made in Game 2).	PK-CCA2
10)	The $LU^{\leftarrow}$ game with challenge bit chosen as 1	Same Behavior

3. The adversary submits
   - message m,
   - a position $j^{\leftarrow}$ with $0 \leq j^{\leftarrow} \leq n^{\leftarrow}+1$,
   - a path $\mathcal{P}^{\rightarrow} = (P_1, \dots, P_j, \dots, P_{n+1})$,
   - a path $\mathcal{P}^{\leftarrow} = (P^{\leftarrow}_1, \dots, P^{\leftarrow}_{j\leftarrow}, \dots, P^{\leftarrow}_{n^{\leftarrow}+1} = P_S)$ with the honest node P_H at backward position $j^{\leftarrow}$, if $1 \leq j^{\leftarrow} \leq n^{\leftarrow}+1$, and the second honest node P_S at position $n^{\leftarrow}+1$
   - and public keys for all nodes PK_i ($1 \leq i \leq n+1$ for the nodes on the path and $n+1 < i$ for the other relays).
4. The challenger checks that the chosen paths are acyclic, the router names and public keys are valid and that the same key is chosen if the router names are equal, and if so, sets $PK^{\leftarrow}_{j\leftarrow} = PK_H$, $PK^{\leftarrow}_{n^{\leftarrow}+1} = PK_S$ and sets bit b at random.
5. The challenger creates the onion with the adversary's input choice and honestly chosen randomness $\mathcal{R}$:

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

   and sends O_1 to the adversary.
6. The adversary gets oracle access as in step 2) except if:

   Exception 1) The request is ...
   - Proc(P_H, O) with RecognizeOnion$((n+1)+j^{\leftarrow}, O, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}) = True$, η is not on the η^H-list and ProcOnion$(SK_H, O, P_H) \neq \perp$: stores η on the η^H and O on the O^H-list and ...

Table 9. Overview Hybrids for $LU^{\leftarrow}$, $j^{\leftarrow} > 0$ with "bw" denoting parts of the backward onion

	$B^{1\leftarrow}_{j\leftarrow}, \ldots$ $B^{2N-1\leftarrow}_{j\leftarrow}$ (bw)	$k^{\eta\leftarrow}_{j\leftarrow}, k^{\gamma\leftarrow}_{j\leftarrow}, \Delta^{\leftarrow}_{j\leftarrow}$ in $E^{\leftarrow}_{j\leftarrow}$ (bw)	$B^{(n-j^{\leftarrow}+2)\leftarrow}_{j^{\leftarrow}+1}, \ldots$ $B^{(2N-1)\leftarrow}_{j^{\leftarrow}+1}$ (bw)	$\delta^{\leftarrow}_{j}$ (bw)	Oracle (bw)
1)	real	real $k^{\eta}_{j\leftarrow}, k^{\gamma}_{j\leftarrow}, \Delta^{\leftarrow}_{j\leftarrow}$	$\mathsf{PRP.Dec}^{\iota}(0,\ldots,0)$ (Relay padding)	contains $m^{\leftarrow}$ (adv. chosen)	recog.+ honest proc
2)		$(0,\ldots,0)$			use $k^{\eta}_{j\leftarrow}, k^{\gamma}_{j\leftarrow}, k^{\vartheta}_{j\leftarrow}$, if $E^{\leftarrow}_{j\leftarrow}$ is recognized
3)					fail, if $E^{\leftarrow}_{j\leftarrow} = exp$, η modif.
4)	random				use sender's $k^{\eta}_{>j\leftarrow}, k^{\gamma}_{>j\leftarrow}, \Delta^{\leftarrow}_{>j\leftarrow}$ for challenge onion
5)			random		use fresh $k^{\eta}_{>j\leftarrow}, k^{\gamma}_{>j\leftarrow}, \Delta^{\leftarrow}_{>j\leftarrow}$ and fresh rdm padding for challenge onion
6)				random	
7)	real				
8)					proc, if $E^{\leftarrow}_{j\leftarrow} = exp$, η modif.
9)		real $k^{\eta}_{j\leftarrow}, k^{\gamma}_{j\leftarrow}, \Delta^{\leftarrow}_{j\leftarrow}$			
10)	real	real $k^{\eta}_{j\leftarrow}, k^{\gamma}_{j\leftarrow}, \Delta^{\leftarrow}_{j\leftarrow}$	random (Sender padding)	random	recog.+ $\bar{O}_1$ (new fw onion)

.. then: The challenger picks the rest of the return path $\bar{\mathcal{P}}^{\rightarrow} = (P^{\leftarrow}_{j^{\leftarrow}+1}, \ldots, P^{\leftarrow}_{n^{\leftarrow}+1})$, an empty backward path $\bar{\mathcal{P}}^{\leftarrow} = ()$, and a random message $\bar{m}$, another honestly chosen randomness $\bar{\mathcal{R}}$, and generates:

$$\bar{O}_1 \leftarrow \text{FormOnion}(1, \bar{\mathcal{R}}, \bar{m}, \bar{\mathcal{P}}^{\rightarrow}, \bar{\mathcal{P}}^{\leftarrow}, (PK)_{\bar{\mathcal{P}}^{\rightarrow}}, (PK)_{\bar{\mathcal{P}}^{\leftarrow}})$$

- The challenger calculates $(O_{j^{\leftarrow}+1}, P^{\leftarrow}_{j^{\leftarrow}+1}) = \text{ProcOnion}(SK_H, O, P^{\leftarrow}_{j^{\leftarrow}})$ and gives $O_{j^{\leftarrow}+1}$ for $P^{\leftarrow}_{j^{\leftarrow}+1}$ to the adversary.

Exception 2) $\mathsf{Proc}(P_S, O)$ with O being the challenge onion as processed for the final receiver on the backward path, i.e.:

- $\text{RecognizeOnion}((n+1)+(n^{\leftarrow}+1), O, \mathcal{R}) = True$

.. then the challenger outputs nothing.

7. The adversary produces guess b' .

Hybrid 2 As Hybrid 1), but with differences in the following step:

5. The challenger creates the onion as before, but encrypts $0\ldots0$ instead of $k^{\eta\leftarrow}_{j\leftarrow}, k^{\gamma\leftarrow}_{j\leftarrow}, \Delta^{\leftarrow}_{j\leftarrow}$ for $E^{\leftarrow}_{j\leftarrow}$ (but still encrypts other blocks of the header with the real $k^{\eta\leftarrow}_{j\leftarrow}$, the payload with the real $\Delta^{\leftarrow}_{j\leftarrow}$ and MACs with $k^{\gamma\leftarrow}_{j\leftarrow}$):

$$E^{\leftarrow}_{j\leftarrow} = \mathsf{PK.Enc}_{PK_{j\leftarrow}}(0,\ldots,0)$$

The challenger calculates the new MAC for the blocks. All the other layers are constructed as before but using the replacements for the calculations, i.e. the onion layer $O'_{j\leftarrow}$ is wrapped as before and the new backward header embedded in O_1. The challenger gives the final O_1 to the adversary.

6. RecognizeOnion now checks for the adapted header (as constructed above) and if $E^{\leftarrow}_{j\leftarrow}$ is reused in a not recognized onion, the original keys $k^{\eta}{}^{\leftarrow}_{j\leftarrow}, k^{\gamma}{}^{\leftarrow}_{j\leftarrow}$, $\Delta^{\leftarrow}_{j\leftarrow}$ are returned as decryption.

$\mathcal{H}_1 \Rightarrow \mathcal{H}_2$*:*

$$|\Pr(\mathcal{X}_1) - \Pr(\mathcal{X}_2)| \leq \epsilon_{\text{PK-CCA2}}$$

with $\epsilon_{\text{PK-CCA2}}$ being the $CCA2$-advantage of some efficient adversary against our PK encryption scheme; argued similarly to $LU^{\rightarrow}, j < n+1, \mathcal{H}_1 \Rightarrow \mathcal{H}_2$.

Hybrid 3 As Hybrid 2) but with differences in the following step:

7. If an onion is handed to the oracles that reuses $E^{\leftarrow}_{j\leftarrow}$, but changes another part of the header, i.e. is not the recognized as challenge onion processing, processing fails.

$\mathcal{H}_2 \Rightarrow \mathcal{H}_3$*: Security loss:*

$$|\Pr(\mathcal{X}_2) - \Pr(\mathcal{X}_3)| \leq \epsilon_{SUF-CMA}$$

with $\epsilon_{SUF-CMA}$ being the $SUF-CMA$-advantage of some efficient adversary against our used MAC scheme; argued similarly to $LU^{\rightarrow}, j < n+1, \mathcal{H}_2 \Rightarrow \mathcal{H}_3$.

Hybrid 4 As Hybrid 3), but with differences in the following step:

5. The challenger creates the onion as before *but the block* $B^{1}{}^{\leftarrow}_{j\leftarrow}, ..., B^{2N-1}{}^{\leftarrow}_{j\leftarrow}$ *are replaced with* R_i *being randomly chosen blocks to calculate* O_1:

$$B^{i}_{j\leftarrow} = (R_i) \text{ for } 1 \leq i \leq 2N-1 \text{ with } R_i \text{ being the randomly generated}$$

6. The challenger answers oracles as in Hybrid 3), except if the header matches $\eta_{j\leftarrow}$ and the request Proc. In this case, the original (backwards) header for $j^{\leftarrow}$ (as before the random replacement) is output as processing.

$\mathcal{H}_3 \Rightarrow \mathcal{H}_4$*: Security loss:*

$$|\Pr(\mathcal{X}_3) - \Pr(\mathcal{X}_4)| \leq \epsilon_{\text{PRP-CCA}}$$

with $\epsilon_{\text{PRP-CCA}}$ being the CCA-advantage of some efficient adversary against our used PRP; argued similarly to $LU^{\rightarrow}, j < n+1, \mathcal{H}_3 \Rightarrow \mathcal{H}_4$.

Hybrid 5 As Hybrid 4), but with differences in the following step:

6. The challenger creates the next onion layer in Exception 1 as before but with freshly chosen $k^{\Delta}{}^{\leftarrow}_{j^{\leftarrow}+1}, k^{\eta}{}^{\leftarrow}_{>j\leftarrow}, k^{\gamma}{}^{\leftarrow}_{>j\leftarrow}, \Delta^{\leftarrow}_{>j\leftarrow}$ and padding, i.e. a freshly constructed backward header after $j^{\leftarrow}$ and the payload is constructed by decrypting the old payload and then encrypting it with $k^{\Delta}{}^{\leftarrow}_{j^{\leftarrow}+1}$. Exception 2 is adapted to match these new header layers.

$\mathcal{H}_4 \Rightarrow \mathcal{H}_5$*: Security loss:*

$$|\Pr(\mathcal{X}_4) - \Pr(\mathcal{X}_5)| = 0$$

Similarly to $LU^{\leftarrow}, j^{\leftarrow} = 0, \mathcal{H}_6 \Rightarrow \mathcal{H}_7$ we change the point in time in which the keys are chosen. Further, we replace the multiple re-encryption of the payload with a fresh one, which does not change the distribution according to Perfect ReEncryption.

Hybrid 6 As Hybrid 5), but with differences in the following step:

5. The challenger creates the onion as before but *with random payload*:

$$\delta_1 = (\mathsf{UE.Enc}_{k_1^{\Delta}}(R)) \text{ with } R \text{ being a random message}$$

6. The expected plaintext encrypted in the payload of the recognized onion is adapted to be R. Otherwise, the oracle works as before.

$\mathcal{H}_5 \rightarrow \mathcal{H}_6$*:* This is similar to $LU^{\leftarrow}, j^{\leftarrow} = 0, \mathcal{H}_7 \Rightarrow \mathcal{H}_8$, but as we replace *during* the backward path (instead of in the beginning where the backward message is known), we have to retrieve the backward message using the UE decryption as shown below:

Precise steps of $\mathcal{A}_{CCA}$:

1. Pick random router names P_H, P_S and generate corresponding key pairs $(PK_H, SK_H) \leftarrow G(1^{\lambda}, p, P_H)$, $(PK_S, SK_S) \leftarrow G(1^{\lambda}, p, P_S)$. Send P_H, P_S and PK_H, PK_S to $\mathcal{D}$.
2. Oracle access: Use SK_H resp. SK_S to answer the oracle as in the original game.
3. Receive message m, paths $\mathcal{P}^{\rightarrow}$, $\mathcal{P}^{\leftarrow}$ and public keys PK_i for both path directions from $\mathcal{D}$.
4. Check validity of names and set $PK_{n+1} = PK_H$ and $PK^{\leftarrow}_{n^{\leftarrow}+1} = PK_S$.
5. Construct O_1 as before (Note that the (non-repliable) backwards header used for the construction of O_1 is thus constructed with the GenTok and GenKey until $P^{\leftarrow}_j = P_H$ and for $> j^{\leftarrow}$ the keys are already replaced by randomness as in earlier hybrids. Store $\eta^{\leftarrow}_{j^{\leftarrow}}$ and $k^{\Delta}_{j^{\leftarrow}}$ for later use. We thus do not need to involve any UP-IND-RCCA oracles for this construction.)
6. Send O_1 to $\mathcal{D}$
7. Oracle access: Reply are processed normally. Upon receiving $\mathsf{Proc}(P, O = (\eta, \delta))$ from $\mathcal{D}$:
   - check if $\eta = \eta^{\leftarrow}_{j^{\leftarrow}}$ for $P_H = P$:
     - If not, process with knowledge of secret keys and adaptations to the processing as before (if it is not on the η-List and if it not violates exception 2 (is the reply processed at the original sender)).
     - If so: Construct the reply onion carefully using the UP-IND-RCCA oracles and challenge as follows:
       (a) Find out the included message $m^{\leftarrow}$ by $m^{\leftarrow} \leftarrow \mathsf{UE.Dec}(k^{\Delta}_{j^{\leftarrow}}, \delta)$ (Note that this is not an oracle call, but instead uses the key generated during 5.)

(b) Sends $M_0 = m^{\leftarrow}, M_1 = R \leftarrow_{\mathrm{R}} \mathcal{M}_{sp}$ with $|m^{\leftarrow}| = |R|$ to Ch_{CCA} and receives $C^* \leftarrow_{\mathrm{R}} \mathsf{UE.Enc}(k_1^{\Delta}, M_b)$ from Ch_{CCA}. It sets $\delta^{\leftarrow}_{j^{\leftarrow}+1} = C^*$.
(c) Use Next oracles $n^{\leftarrow} - j^{\leftarrow}$ times until epoch $e = n^{\leftarrow} + 1 - j^{\leftarrow}$.
(d) Get tokens $\Delta_1, \ldots, \Delta_{n^{\leftarrow}-j^{\leftarrow}}$ with $\mathsf{Corrupt}(token, 2), \ldots, \mathsf{Corrupt}(token, n^{\leftarrow} + 1 - j^{\leftarrow})$
(e) Pick keys $k_1^{\eta}, \ldots, k_{n^{\leftarrow}-j^{\leftarrow}}^{\eta}$ for the block cipher and $k_1^{\gamma}, \ldots, k_{n^{\leftarrow}-j^{\leftarrow}}^{\gamma}$ for the MAC randomly.
(f) Create the header $\eta^{\leftarrow}_{j^{\leftarrow}+1}$ (for the last part of the backwards path) with the keys $k_1^{\eta}, \ldots, k_{n^{\leftarrow}-j^{\leftarrow}}^{\eta}$, $k_1^{\gamma}, \ldots, k_{n^{\leftarrow}-j^{\leftarrow}}^{\gamma}$, $\Delta_1, \ldots, \Delta_{n^{\leftarrow}-j^{\leftarrow}}$ as the protocol does.
(g) Return the onion $(\eta^{\leftarrow}_{j^{\leftarrow}+1}, \delta^{\leftarrow}_{j^{\leftarrow}+1})$ for $P^{\leftarrow}_{j^{\leftarrow}+1}$.
(h) Adapt exception 2 to work for the newly constructed header (the sender will not process this onion in the oracle).

8. Receive the guess from $\mathcal{D}$ and return it as own guess.

Security loss:

$$|\Pr(\mathcal{X}_5) - \Pr(\mathcal{X}_6)| \leq \epsilon_{\text{UP-IND-RCCA}}$$

with $\epsilon_{\text{UP-IND-RCCA}}$ being the UP-IND-RCCA-advantage of some efficient adversary against our used UE scheme (which is negligible according to our choice).

Remaining hybrids: The remaining hybrids just revert changes done in earlier hybrids and are argued similarly to these.

E.5 Repliable Tail-Indistinguishability: $TI^{\leftrightarrow}$

Table 10. Overview Proof for TI

Hybrid	Description	Reduction
1)	The $TI^{\leftrightarrow}$ game with challenge bit chosen as 0	
2)	We replace the blocks $B_{j+1}^{2N-(j+1)}, \ldots, B_{j+1}^{2N-1}$ in step 5 by random strings $R_{2N-(j+1)}, \ldots, R_{2N-1}$ (and adapt recognizeOnion to the new header).	PRP-CCA
3)	We replace the temporary keys $k^{\eta}{}^{\leftarrow}_{j^{\leftarrow}}, k^{\gamma}{}^{\leftarrow}_{j^{\leftarrow}}, \Delta^{\leftarrow}_{j^{\leftarrow}}$ at the honest relay with $0 \ldots 0$ in their encryption for $E^{\leftarrow}_{j^{\leftarrow}}$ (and adapt recognizeOnion to the new header) as part of the payload of O_{j+1}, but still use the real keys for the processing.	PK-CCA2
4)	We let the oracles in step 7 output a fail, if the challenge $E^{\leftarrow}_{j^{\leftarrow}}$ is recognized, but other parts of the header differ.	SUF-CMA
5)	We replace the block $B^{1}{}^{\leftarrow}_{j^{\leftarrow}}$ with a path-end-block and $B^{2}{}^{\leftarrow}_{j^{\leftarrow}}, \ldots, B^{(n^{\leftarrow}-j^{\leftarrow}+1)}{}^{\leftarrow}_{j^{\leftarrow}}$ with random blocks in the payload part of the challenge onion representing the backward header.	PRP-CCA
6)	We revert the changes of Game 4).	SUF-CMA
7)	We revert the changes of Game 3).	PK-CCA2
8)	The $TI^{\leftrightarrow}$ game with challenge bit chosen as 1	Same behavior as before

Similar to the $LU^{\rightarrow}$ proofs, we now use $\mathcal{X}_i$ as the event that $\mathcal{A}_{TI^{\leftrightarrow}}$ outputs $b' = 1$ in the i-th hybrid game $\mathcal{H}_i$.

Table 11. Overview Hybrids for TI with "bw" denoting parts of the backward onion

	$k_j^\eta, k_j^\gamma, \Delta_j$ in E_j	$B_{j+1}^{2N-(j+1)}, \ldots, B_{j+1}^{2N-1}$	$k^{\eta}_{j^{\leftarrow}}, k^{\gamma}_{j^{\leftarrow}}, \Delta^{\leftarrow}_{j^{\leftarrow}}$ in $E^{\leftarrow}_{j^{\leftarrow}}$ (bw)	$B^{2^{\leftarrow}}_{j^{\leftarrow}}, \ldots, B^{(n^{\leftarrow}-j^{\leftarrow}+1)^{\leftarrow}}_{j^{\leftarrow}}$ (bw)	Oracle (bw)
1)	real	contains real processing	real	complete path	fail, if $b = 0$ reply back at honest relay (exception step 7)
2)		$R^{2N-(j+1)}, \ldots, R^{2N-1}$			
3)			$(0, \ldots, 0)$		(adapt recog.)
4)					fail, if $E^{\leftarrow}_{j^{\leftarrow}} = exp$, η modif.
5)				$\mathsf{PRP.Enc}(\perp, \perp, \perp), R_2, \ldots, R_{n^{\leftarrow}-j^{\leftarrow}+1}$	
6)					proc, if $E^{\leftarrow}_{j^{\leftarrow}} = exp$, η modif.
7)			real		
8)	real	Sender padding	real	shortened path	fail if $b = 1$ reply back at honest relay (exception step 7)

Hybrid 1

1. The adversary receives the router names $P_H, P_H^{\leftarrow}, P_S$ and challenge public keys $PK_S, PK_H, PK_H^{\leftarrow}$, chosen by the challenger by letting $(PK_H, SK_H) \leftarrow G(1^\lambda, p, P_H)$, $(PK_H^{\leftarrow}, SK_H^{\leftarrow}) \leftarrow G(1^\lambda, p, P_H^{\leftarrow})$, $(PK_S, SK_S) \leftarrow G(1^\lambda, p, P_S)$.
2. Oracle access: The adversary may submit any number of Proc and Reply requests for $P_H, P_H^{\leftarrow}$ or P_S to the challenger. For any $\mathsf{Proc}(P_H, O)$, the challenger checks whether η is on the η^H- list. If not, it sends the output of $\mathrm{ProcOnion}(SK_H, O, P_H)$, stores η on the η^H-list and O on the O^H-list. For any $\mathsf{Reply}(P_H, O, m)$ the challenger checks if O is on the O^H- list and if so, the challenger sends $\mathrm{ReplyOnion}(m, O, P_H, SK_H)$ to the adversary. (Similar for requests on $P_H^{\leftarrow}, P_S$).
3. The adversary submits a message m, a path $\mathscr{P}^{\rightarrow} = (P_1, \ldots, P_j, \ldots, P_{n+1})$ with the honest node P_H or $P_H^{\leftarrow}$ at position j, $1 \leq j < n+1$, a path $\mathscr{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{n^{\leftarrow}+1}^{\leftarrow})$ with the honest node $P_H^{\leftarrow}$ at position $1 \leq j^{\leftarrow} \leq n^{\leftarrow} + 1$ and public keys for all nodes PK_i $(1 \leq i \leq n+1$ for the nodes on the path and $n+1 < i$ for the other relays).
4. The challenger checks that the given paths are acyclic, the router names and public keys are valid and that the same key is chosen if the router names are equal, and if so, sets $PK_j = PK_H$ (or $PK_j = PK_H^{\leftarrow}$, if the adversary chose $P_H^{\leftarrow}$ at this position as well) , $PK_{j^{\leftarrow}}^{\leftarrow} = PK_H^{\leftarrow}$,$PK_{n^{\leftarrow}+1}^{\leftarrow} = PK_S$ and sets bit b at random.
5. The challenger creates the onion with the adversary's input choice and honestly chosen randomness $\mathscr{R}$:

$$O_{j+1} \leftarrow \mathrm{FormOnion}(j+1, \mathscr{R}, m, \mathscr{P}^{\rightarrow}, \mathscr{P}^{\leftarrow}, (PK)_{\mathscr{P}^{\rightarrow}}, (PK)_{\mathscr{P}^{\leftarrow}})$$

and a replacement onion with the path from the honest relay P_H to the corrupted receiver $\bar{\mathscr{P}}^{\rightarrow} = (P_{j+1}, \ldots, P_{n+1})$ and the backward path from the corrupted receiver starting at position 0 ending at $j^{\leftarrow}$: $\bar{\mathscr{P}}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{j^{\leftarrow}}^{\leftarrow})$; and another honestly chosen randomness $\bar{\mathscr{R}}$:

$$\bar{O}_1 \leftarrow \mathrm{FormOnion}(1, \bar{\mathscr{R}}, m, \bar{\mathscr{P}}^{\rightarrow}, \bar{\mathscr{P}}^{\leftarrow}, (PK)_{\bar{\mathscr{P}}^{\rightarrow}}, (PK)_{\bar{\mathscr{P}}^{\leftarrow}})$$

6. The challenger sends O_{j+1} to the adversary.
7. Oracle access: the challenger processes all requests as in step 2) except if...

... Proc($P_H^{\leftarrow}, O$) with O being the challenge onion as processed for the honest relay on the backward path, i.e.:
- RecognizeOnion($(n+1)+j^{\leftarrow}, O, \mathcal{R}) = True$

.. then the challenger outputs nothing.

8. The adversary produces guess b'.

Hybrid 2 As Hybrid 1), but with differences in the following step:

5. The challenger creates the onion as before *but the blocks* $B_{j+1}^{n-j+2}, ..., B_{j+1}^{2N-1}$ *are replaced with* $R_{n-j+2}, \ldots R_{2N-1}$ *with* R_i *being randomly chosen blocks to calculate* O_{j+1}:

   $B_{j+1}^{i} = (R_i)$ for $n - j + 2 \leq i \leq 2N - 1$ with R_i being randomly generated

$\mathcal{H}_1 \Rightarrow \mathcal{H}_2$*: Security loss:*

$$|\Pr(\mathcal{X}_1) - \Pr(\mathcal{X}_2)| \leq \epsilon_{\text{PRP-CCA}}$$

with $\epsilon_{\text{PRP-CCA}}$ being the CCA-advantage of some efficient adversary against our used PRP; argued similarly to $LU^{\rightarrow}, j < n + 1, \mathcal{H}_3 \Rightarrow \mathcal{H}_4$. Note that we do not need to replace the keys of the honest relay j before this step (as we do in the $LU^{\rightarrow}$ proof) because the early onion layers $O_{<j}$ are never given to the adversary in $TI^{\leftrightarrow}$.

Hybrid 3 As Hybrid 2), but with differences in the following steps:

5. The challenger creates the onion as before, but encrypts $0 \ldots 0$ instead of $k^{\eta}{}_{j\leftarrow}^{\leftarrow}, k^{\gamma}{}_{j\leftarrow}^{\leftarrow}, \Delta_{j\leftarrow}^{\leftarrow}$ for $E_{j\leftarrow}^{\leftarrow}$ (but still encrypts other blocks of the header with the real $k^{\eta}{}_{j\leftarrow}^{\leftarrow}$, the payload with the real $\Delta_{j\leftarrow}^{\leftarrow}$ and MACs with $k^{\gamma}{}_{j\leftarrow}^{\leftarrow}$):

   $$E_{j\leftarrow}^{\leftarrow} = \mathsf{PK.Enc}_{PK_j}(0, \ldots, 0)$$

   The challenger calculates the new MAC for the blocks. All the later layers $E_{\geq j^{\leftarrow}+1}^{\leftarrow}, B^{i}{}_{\geq j^{\leftarrow}+1}^{\leftarrow}$ are constructed as before but using the replacements for the calculations, i.e. the onion layer $O_{j\leftarrow}^{\leftarrow}$ is wrapped as before.
7. RecognizeOnion now checks for the adapted header (as constructed above) and if $E_{j\leftarrow}^{\leftarrow}$ is reused in a not recognized onion, the original keys $k^{\eta}{}_{j\leftarrow}^{\leftarrow}, k^{\gamma}{}_{j\leftarrow}^{\leftarrow}$, $\Delta_{j\leftarrow}^{\leftarrow}$ are returned as decryption.

$\mathcal{H}_2 \Rightarrow \mathcal{H}_3$*:*

$$|\Pr(\mathcal{X}_2) - \Pr(\mathcal{X}_3)| \leq \epsilon_{\text{PK-CCA2}}$$

with $\epsilon_{\text{PK-CCA2}}$ being the $CCA2$-advantage of some efficient adversary against our PK encryption scheme; argued similarly to $LU^{\rightarrow}, j < n + 1, \mathcal{H}_1 \Rightarrow \mathcal{H}_2$.

Hybrid 4 As Hybrid 3) but with differences in the following step:

7. If an onion is handed to the oracle that reuses $E_{j\leftarrow}^{\leftarrow}$, but changes another part of the header, i.e. is not the recognized as challenge onion processing, processing fails.

$\mathcal{H}_3 \Rightarrow \mathcal{H}_4$*: Security loss:*

$$|\Pr(\mathcal{X}_3) - \Pr(\mathcal{X}_4)| \leq \epsilon_{SUF-CMA}$$

with $\epsilon_{SUF-CMA}$ being the $SUF-CMA$-advantage of some efficient adversary against our used MAC scheme; argued similarly to $LU^{\rightarrow}, j < n+1, \mathcal{H}_2 \Rightarrow \mathcal{H}_3$.

Hybrid 5 As Hybrid 4), but with differences in the following step:

5. The challenger creates the onion as before *but the block* $B^{1\overleftarrow{}}_{j\leftarrow}$ *is replaced with the receiver signal* $\mathsf{PRP.Enc}(\perp,\perp,\perp)$ *the blocks* $B^{2\overleftarrow{}}_{j\leftarrow}, \ldots, B^{(n^{\leftarrow}-j^{\leftarrow}+1)\overleftarrow{}}_{j\leftarrow}$ *are replaced with* $R_2, \ldots R_{n^{\leftarrow}-j^{\leftarrow}+1}$ *with* R_i *being randomly chosen blocks to calculate the backward header embedded in* O_{j+1}*'s payload*:

$$\begin{aligned} E_{j\leftarrow}^{\leftarrow} &= \mathsf{PK.Enc}_{PK_j}(0,\ldots,0) \\ B^{i\overleftarrow{}}_{j\leftarrow} &= \mathsf{PRP.Enc}_{k^{\eta}{}^{\leftarrow}_{j\leftarrow}}(\perp,\perp,\perp) \\ B^{i\overleftarrow{}}_{j\leftarrow} &= (R_i) \text{ for } 2 \leq i \leq n^{\leftarrow} - j^{\leftarrow} + 1 \text{ with } R_i \text{ being randomly generated} \end{aligned}$$

The challenger continues to wrap the onion with the replaced $B_{j\leftarrow}^{\leftarrow}$s to create the backwards header and embed it in O_{j+1}.

7. RecognizeOnion now checks for the adapted header (as constructed above), i.e. $E_{j\leftarrow}^{\leftarrow}$ and $B^{i\overleftarrow{}}_{j\leftarrow}$. In this case, the challenger does not output anything, as usual before.

$\mathcal{H}_4 \Rightarrow \mathcal{H}_5$*: Security loss:*

$$|\Pr(\mathcal{X}_4) - \Pr(\mathcal{X}_5)| \leq 2 \cdot \epsilon_{\text{PRP-CCA}}$$

with $\epsilon_{\text{PRP-CCA}}$ being the CCA-advantage of some efficient adversary against our used PRP; argued similarly to $LU^{\rightarrow}, j < n+1, \mathcal{H}_3 \Rightarrow \mathcal{H}_4$. Note that technically, we need to replace the $B^{1\overleftarrow{}}_{j\leftarrow}$ with randomness first, before we can replace it back to the receiver signal $\mathsf{PRP.Enc}(\perp,\perp,\perp)$, which accounts for the factor of 2 above.

Remaining hybrids: The remaining hybrids just revert changes done in earlier hybrids and are argued similarly to these.

F Security of our SNARG-Based Scheme

In this section, we prove that our SNARG-based scheme also realizes the ideal functionality by showing our properties. We start by describing FormOnion for other layers than the first one and continue to show the proofs. As they are

similar to the ones in the UE-based solution, we sketch them here and only detail the differences.

FormOnion - later layers. FormOnion for $i > 1$ uses the SNARG-trapdoor to create a valid SNARG, encrypts random strings for the ring buffer entries C^j, and creates the other onion parts deterministically as described in the protocol for the current layer. In contrast to the UE-based scheme also the payload is deterministic in the SNARG-based scheme.

F.1 Forwards Layer Unlinkability

Case 1 – Honest Relay ($j < n + 1$). We first replace all SNARG-related parts to unlink them from the SNARG information of other layers and also from the secret information included in them. Then we replace the temporary keys of the honest party included in the header, to be able to change the blocks of the header and the payload corresponding to the $b = 1$ case. For the oracles we further need to ensure, that RecognizeOnion does not mistreat any processing of e.g. modified onions. Therefore, we leverage the SNARG properties for the payload protection and the MAC for the header. An overview of the proof is in Table 12 and an overview of the used hybrids in Table 13 of Appendix F.2.

Proof

Hybrid 1) $LU^{\rightarrow}_{(b=0)}$**.** As Hybrid 1 for the $LU^{\rightarrow}, j < n + 1$-proof for the UE based protocol.

Hybrid 2). As Hybrid 1, but with differences in the following step:

6. The challenger replaces the SNARGs π_i of the challenge O_i's with simulated SNARG π_i^s that were created with the SNARG trapdoor before handing O_1 to the adversary.

Hybrid 1) $\approx_{IND}$ **Hybrid 2).** Assume there exists a distinguisher $\mathcal{D}$ that can distinguish the hybrids with non-negligible advantage. This is a direct contradiction to the simulatability of the SNARG.

Hybrid 3). As Hybrid 2, but with the following differences:

5. The challenger creates the onion as before, but replaces the first ring buffer element C_1^1 as fresh encryptions of `sim` for the special bitstring `sim`.

We note that this change is meaningful, since by our change in Hybrid 2, all SNARGs are simulated, and no witness to the fact that the ring buffer has been generated honestly is required.

Hybrid 2) $\approx_{IND}$ **Hybrid 3).** Recall that the first ring buffer element C_1^1 is a fresh encryption under PK^{M} in Hybrid 2, while in the later layers it is re-randomized for C_k^i. By the IND-CPA security of the used encryption scheme, hence C_1^1 is indistinguishable in both hybrids. Furthermore, the re-randomizability of that scheme also guarantees that the other C_k^i are indistinguishable.

Hybrid 4). As Hybrid 3), but with the following differences:

Table 12. Overview Proof for $LU^{\rightarrow}$, $j < n + 1$

Hybrid	Description	Reduction
1)	The $LU^{\rightarrow}$ game with challenge bit chosen as 0	
2)	We simulate the SNARGs for the challenge onion.	SNARG Simulatability
3)	We replace the first ring buffer entry C_1^1 with a fresh encryption of `sim` for the special bitstring `sim`.	IND-CPA/rerand. of PKM
4)	We replace the ring buffer elements C_{j+1}^i for all i with fresh encryptions of random strings (not `sim`) [as FormOnion does for layers $i > 1$].	IND-CPA/rerand. of PKM
5)	We replace the temporary keys $k_j^\eta, k_j^\gamma, k_j^\delta$ at the honest relay by 0..0 before they are encrypted in E_j (and adapt recognizeOnion to the new header), but still use the real keys for the processing.	PK-CCA2
6)	We let the oracles in step 7 output a fail, if the challenge E_j is recognized, but other parts of the header differ.	SUF-CMA
7) & 8)	We replace the blocks $B_j^1, ..., B_j^{2N-j}$ by $(\texttt{sim}, \texttt{sim}), R_2, \ldots, R_{N-j}$ with R_i being randomly chosen (and adapt recognizeOnion to the new header), but use the real blocks for the processing.	PRP-CCA
9)	We let oracles in step 7 output a fail, if the challenge header η_j is recognized, but the payload differs.	SNARG simulation soundness
10)	We let the oracles in step 7 output the replicated layer $j + 1$:(FormOnion$(j + 1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})$), if the challenge η_j is recognized, the payload matches, and real processing of the given onion would not fail.	Same behavior due to definition of recognition and forming of later layers
11)	We replace the content δ_j by a random string of the same length.	PRP-CCA
12)	We revert the changes made in Game 9).	SNARG simulation soundness
13) & 14)	We replace the blocks $B_j^1, ..., B_j^{2N-j}$ by $(\perp, \perp, \perp), R_2, \ldots, R_{2N-j}$ with R_i being randomly chosen (and adapt recognizeOnion to the new header).	PRP-CCA
15)	We revert the changes made in Game 6).	SUF-CMA
16)	We revert the changes made in Game 5).	PK-CCA2
17)	We revert the changes made in Game 3): The ring buffer entry C_1^1 now includes the sender info as in the $b = 1$ case.	IND-CPA/rerand. of PKM
18)	We use FormOnion with the parameter of the $b = 1$ case to generate the first challenge onion layer.	Same behavior except for new draw of randomness
19)	The $LU^{\rightarrow}$ game with challenge bit chosen as 1	SNARG Simulatability

7. The challenger processes the onion as before to answer Proc(P_H, O) with $\eta = \eta_j$, i.e. RecognizeOnion is true; if η is not on the η^H-list and processing of O would not have failed, *but replaces the* C_{j+1}^i *with encryptions for random strings under* PK^{M}

Hybrid 3) $\approx_{IND}$ **Hybrid 4).** Follows from the IND-CPA security and the rerandomizability of the master encryption scheme similar to the indistinguishability between Hybrid 2) and 3).

Hybrid 5). As Hybrid 4), but with differences in the following step:

5. The challenger creates the onion as before, but encrypts $0 \ldots 0$ instead of $k_j^\eta, k_j^\gamma, k_j^\delta$ for E_j (but still encrypts other blocks of the header with the real

k_j^η, the payload with the real k_j^δ and MACs with k_j^γ):

$$E_j = \mathsf{PK.Enc}_{PK_j}(0, \ldots, 0)$$
$$B_j^1 = \mathsf{PRP.Enc}_{k_j^\eta}(P_{j+1}, E_{j+1}, \gamma_{j+1})$$
$$B_j^i = \mathsf{PRP.Enc}_{k_j^\eta}(B_{j+1}^{i-1}) \text{ for } 2 \leq i \leq 2N-1$$

The challenger calculates the new MAC for the blocks. All the later layers $E_{\geq j+1}, B_{\geq j+1}^i$ are constructed as before but using the replacements for the calculations, i.e. the onion layer O_j is wrapped as before.

6. The challenger gives the final O_1 to the adversary.
7. RecognizeOnion now checks for the adapted header (as constructed above) and if E_j is reused in a not recognized onion, the original keys $k_j^\eta, k_j^\gamma, k_j^\delta$ are returned as decryption.

Hybrid 4) $\approx_{IND}$ Hybrid 5). As $\mathcal{H}_1$ to $\mathcal{H}_2$ in the corresponding proof of the UE-based scheme.

Hybrid 6). As Hybrid 5) but with differences in the following step:

7. If an onion is handed to the oracle that reuses E_j, but changes another part of the header, i.e. is not the recognized as challenge onion processing, processing fails.

Hybrid 5) $\approx_{IND}$ Hybrid 6). As $\mathcal{H}_2$ to $\mathcal{H}_3$ in the corresponding proof of the UE-based scheme.

Hybrid 7). As Hybrid 6), but with differences in the following step:

5. The challenger creates the onion as before *but the blocks* $B_j^1, \ldots, B_j^{2N-j}$ *are replaced with* $R_1, R_2, \ldots R_{2N-j}$ *with* R_i *being randomly chosen blocks to calculate* O_1:

$$E_j = \mathsf{PK.Enc}_{PK_j}(0, \ldots, 0)$$
$$B_j^i = (R_i) \text{ for } 1 \leq i \leq 2N-j \text{ with } R_i \text{ being the randomly generated}$$

The challenger continues to wrap the onion with the replaced B_js to create O_1.

7. The challenger answers oracles as in Hybrid 6), except if the header reuses E_j and B_j^i. In this case, the challenger checks the SNARG, replaces the header with the one first calculated for this position and processes it as usual, except that it skips the SNARG check.

Hybrid 6) $\approx_{IND}$ Hybrid 7). As $\mathcal{H}_3$ to $\mathcal{H}_4$ in the corresponding proof of the UE-based scheme.

Hybrid 8). As Hybrid 7, but with differences in the following step:

5. The challenger creates the onion as before *but the block* B_j^1 *is replaced with* $(\mathtt{sim}, \mathtt{sim}, \mathtt{sim})$ *to calculate* O_1:

$$B_j^1 = \mathsf{PRP.Enc}_{k_j^\eta}((\mathtt{sim}, \mathtt{sim}, \mathtt{sim}))$$

The challenger further wraps the onion to receive O_1 as before.

Hybrid 7) $\approx_{IND}$ **Hybrid 8).** Similarly to the step from Hybrid 6) to 7).

Hybrid 9). As Hybrid 8) but with differences in the following step:

7. The challenger replies with a fail to all $\mathsf{Proc}(P_H, O)$ requests with $\eta = \eta_j$, i.e. RecognizeOnion is true - the header corresponds to the one we created and $\delta \neq \delta_j$, i.e. the payload was modified. Otherwise, the challenger processes the onions for the oracles as before.

Hybrid 8) $\approx_{IND}$ **Hybrid 9).** Note that replying with a fail is also what happens in the original processing of the onion by the protocol if the SNARG check fails. To distinguish the hybrids the distinguisher must thus query the oracle with an onion for which the header is the same as for the challenge, the payload differs and the SNARG is valid.

Assume the distinguisher could find such an onion with non-negligible probability. Then, we construct an adversary $\mathcal{A}_{\text{ext}}$ that succeeds in breaking the simulation-soundness of the used SNARG.

First, $\mathcal{A}_{\text{ext}}$ simulates Hybrid 8, including the LU distinguisher $\mathcal{A}$, and embedding the SNARG CRS from $\mathcal{A}_{\text{ext}}$'s own simulation-soundness game. $\mathcal{A}_{\text{ext}}$ uses its own SNARG simulation oracle to generate simulated proofs $\pi^{1^*}, \ldots, \pi^{n*}$ for the challenge onion O^*. Note that since $\mathcal{A}_{\text{ext}}$ knows all secret keys (except for the SNARG simulation trapdoor), it can answer all Proc oracle requests made by $\mathcal{A}$.

Now if $\mathcal{A}$ manages to submit an onion O as above (with the same header as O^* but different payload), then $\mathcal{A}_{\text{ext}}$ proceeds as follows. First, it decrypts C^1, the first element of the ring buffer, using the ring buffer secret key SK^{M}. We may assume that C^1 decrypts to one of the following:

(1) random coins $\mathcal{R}$ that allow to explain O as the output of FormOnion (such that the corresponding message can be retrieved from O and $\mathcal{R}$), or

(2) a secret key SK and SNARG proofs $\pi^{1'}, \ldots, \pi^{n'}$, such that SK can be used to compute a predecessor onion O' to O (with a ring buffer that is smaller than O's by one ciphertext), and for which $\pi^{1'}, \ldots, \pi^{n-1'}$ are valid SNARG proofs.

(If neither of these conditions hold, $\mathcal{A}_{\text{ext}}$ can output O and π_n as an invalid SNARG statement with forged proof.)

Now case (1) above cannot actually occur: since O has the same header as O^*, at least one B^i $(i \leq N)$ from O will encrypt the message $\mathtt{sim}$ by our change

from Hybrid 8. This means that O cannot be explained as a FormOnion output (since the latter never produces B^i which encrypt `sim`).

In case (2), we can iterate the process above and use SK^{M} to decrypt the first element of the (reduced) ring buffer of O'. Note that by definition of ProcOnion, no sequence of onions $O_1, \ldots, O_{n+1}$ exists such that O_{i+1} is a valid output of ProcOnion(O_i) for all $i = 1, \ldots, n$. Hence, the above process must terminate with a SNARG forgery.

Hybrid 10). As Hybrid 9), but with differences in the following step:

7. The challenger uses P_{j+1} and FormOnion$(j+1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})$ to answer Proc(P_H, O) requests with $\eta = \eta_j$, i.e. RecognizeOnion is true; if η is not on the η^H-list and processing of O would not have failed. Otherwise, the challenger processes the onions for the oracles as before.

Hybrid 9) $\approx_{IND}$ Hybrid 10). Every honest processing is recognized by our definition of RecognizeOnion (the header evolves deterministically) and the layers used to answer only differ in the simulated SNARG and ring buffer C (if the header differs it is not recognized, if the payload differs it will be rejected as in Hybrid 9). Finally, both the SNARG and ring buffer in Hybrid 9) and in Hybrid 10) are simulated in the same way.

Hybrid 11). As Hybrid 10), but with differences in the following step:

5. The challenger creates the onion as before but *with random payload*:

   $\delta_j = R$ with R being the randomly generated replacement for the message

   The challenger wraps the onion until O_1 as before.
7. The expected payload for the recognized onion is adapted to be δ_j. Otherwise, the oracle works as before.

Hybrid 10) $\approx_{IND}$ Hybrid 11). Assume there exists a distinguisher $\mathcal{D}$ that can distinguish Hybrid 10) and 11). We can build an attack $\mathcal{A}_{CCA}$ on the PRP-CCA security of the PRP:

1. $\mathcal{A}_{CCA}$ picks an honest router's name P_j and public key PK as $Ch_{LU^{\rightarrow}}$ would and gives both to $\mathcal{D}$.
2. $\mathcal{A}_{CCA}$ answers the oracle queries from the $\mathcal{D}$ as $Ch_{LU^{\rightarrow}}$ would (including rejecting already seen headers).
3. $\mathcal{A}_{CCA}$ gets the challenge choices from $\mathcal{D}$.
4. $\mathcal{A}_{CCA}$ checks the challenge choices from $\mathcal{D}$ as $Ch_{LU^{\rightarrow}}$ would.
5. $\mathcal{A}_{CCA}$ constructs the onion O_1 as before and sends the block δ_{j+1} to the challenger Ch_{CCA}. The challenger replies with blocks $\tilde{\delta}_j$. $\mathcal{A}_{CCA}$ replaces the calculated content δ_j with the one received from the challenger. $\mathcal{A}_{CCA}$ wraps this new onion to create O_1 and simulates the SNARG and ring buffer as before.
6. $\mathcal{A}_{CCA}$ answers the oracle for $\mathcal{D}$ by calculating the processing, except if it receives the challenge header. In this case, it checks whether the payload is $\tilde{\delta}_1$ and outputs FormOnion$(j+1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})$.

7. $\mathcal{A}_{CCA}$ receives the guess from $\mathcal{D}$ and uses it as its own guess.

Note that $\mathcal{A}_{CCA}$ simulates Hybrid 10) for $b = 0$ and Hybrid 11) for $b = 1$ and thus wins the PRP-CCA game with the same advantage as $\mathcal{D}$ distinguishes the hybrids. As the PRP is secure, there cannot be a successful distinguisher $\mathcal{D}$.

Hybrid 12). As Hybrid 11) but with differences in the following step:

7. The challenger replies with the normal processing to $\mathsf{Proc}(P_H, O)$ requests with $\eta = \eta_j$ and $\delta \neq \delta_j$, again.

Hybrid 11) $\approx_{IND}$ Hybrid 12). Note that replying with a fail (as in Hybrid 11)) is also what happens in the original processing of the onion by the protocol if the SNARG check fails. So, to distinguish the hybrids the distinguisher must query the oracle with an onion for which the header is the same as for the challenge, the payload differs and the SNARG is valid – the same situation as in the indistinguishability between Hybrid 8) and 9) and the same argumentation applies.

Hybrid 13). As Hybrid 12), but with differences in the following step:

5. The challenger creates the onion as before *but the block* B_j^1 *is replaced with a random string to calculate* O_1:

$$B_j^1 = \mathsf{PRP.Enc}_{k_j^\eta}(R), \text{ with } R \text{ being a random string}$$

The challenger wraps the onion to create O_1 as before.

Hybrid 12) $\approx_{IND}$ Hybrid 13). Similarly to the step from Hybrid 6) to 7).

Hybrid 14). As Hybrid 13), but with differences in the following step:

5. The challenger creates the onion as before *but the block* B_j^1 *is replaced with* $(\perp, \perp, \perp)$ *to calculate* O_1:

$$B_j^1 = \mathsf{PRP.Enc}_{k_j^\eta}((\perp, \perp, \perp))$$

The challenger wraps the onion further to create O_1 as before.

Hybrid 13) $\approx_{IND}$ Hybrid 14). Similarly to the step from Hybrid 6) to 7).

Hybrid 15). As Hybrid 14 but with differences in the following step:

7. If an onion is handed to the oracle that reuses E_j, but changes another part of the header, processing is done normally again.

Hybrid 14) $\approx_{IND}$ Hybrid 15). Similarly to the step from Hybrid 5) to 6).

Hybrid 16). As Hybrid 15), but with differences in the following step:

5. The challenger creates the onion O_1 as before, but replaces

$$E_j = \mathsf{PK.Enc}_{PK_j}(k_j^\eta, k_j^\gamma, k_j^\delta)$$

before wrapping further.

Hybrid 15) $\approx_{IND}$ Hybrid 16). Similarly to Hybrid 4) $\approx_{IND}$ Hybrid 5).

Hybrid 17). As Hybrid 16), but with the following differences:

5. The challenger creates the onion as before, but after wrapping it, the challenger *builds the ring buffer element* C_1^1 *as follows from* FormOnion *in the* $b = 1$ *case of* $LU^{\rightarrow}$*, i.e. with the real encrypted sender information.*

Hybrid 16) $\approx_{IND}$ Hybrid 17). Similarly to the step from Hybrid 3) to 4).

Hybrid 18). As Hybrid 17), but with differences in the following step:

5. The challenger constructs the replacement onion with the first part of the forward path $\bar{\mathcal{P}}^{\rightarrow} = (P_1, \ldots, P_j)$, a random message $\bar{m} \in \mathcal{M}$, and an empty backward path $\bar{\mathcal{P}}^{\leftarrow} = ()$, fresh randomness $\bar{\mathcal{R}}$:

$$\bar{O}_1 \leftarrow \text{FormOnion}(1, \bar{\mathcal{R}}, \bar{m}, \bar{\mathcal{P}}^{\rightarrow}, \bar{\mathcal{P}}^{\leftarrow}, (PK)_{\bar{\mathcal{P}}^{\rightarrow}}, (PK)_{\bar{\mathcal{P}}^{\leftarrow}})$$

6. The challenger gives $\bar{O}_1$ to the adversary.
7. The challenger processes all requests as in step 2) except if
   - Request is Proc(P_H, O) with RecognizeOnion$(j, \bar{\mathcal{R}}) = True$, η is not on the η^H-list and ProcOnion$(SK_H, O, P_H) \neq \perp$: The challenger outputs $(P_{j+1}, \text{FormOnion}(j+1, \mathcal{R}, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}))$ and adds η to the η^H-list.

Hybrid 17) $\approx_{IND}$ Hybrid 18). The Hybrids are identical, except that the keys are now chosen with randomness $\bar{\mathcal{R}}$, instead of randomness $\mathcal{R}$, but both are chosen in the same way by the challenger.

Hybrid 19): $LU^{\rightarrow}_{(b=1)}$ The $LU^{\rightarrow}$ game with challenge bit $b = 1$.

Hybrid 18) $\approx_{IND}$ Hybrid 19). Hybrid 18) and 19) are identical, except for the use of the simulation trapdoor/ honest generation of the SNARG (similar to Hybrid 1 $\approx_{IND}$ Hybrid 2).

Case 2 – Honest Receiver ($j = n + 1$): We sketch the proof in Table 14 and 15 of Appendix F.2. The steps are the same as for the first case of $LU^{\rightarrow}$, but in Hybrid 10) we need to treat Reply and Proc requests separately. Note that the earlier restrictions on the oracles work both for Reply and Proc requests.

F.2 Other properties

We sketch the proofs for the other properties in Table 16 – 21.

Backwards Layer Unlinkability. We distinguish the cases that the honest node is the receiver ($j^{\leftarrow} = 0$) and that it is a backward relay ($j^{\leftarrow} > 0$).

Case 1 – Honest receiver ($j^{\leftarrow} = 0$). The steps are similar to the ones for $LU^{\rightarrow}$ Case 1: We replace the SNARG information and temporary keys of honest routers, before we exclude bad events at the oracle and finally set the header and payload parts to correspond to the $b = 1$ case. Note that for $LU^{\leftarrow}$ we can skip the steps related to the modification of the payload (and SNARG properties).

As the forward message is known to the adversary anyways and the backward message (as the final processing) is never given to the adversary, she cannot exploit payload modification at the oracle to break $LU^{\leftarrow}$ in this case.

Case 2 – Honest Relay ($j^{\leftarrow} > 0$). The steps are similar to Case 1 for $LU^{\leftarrow}$.

Repliable Tail-Indistinguishability The steps are similar to Case 2 for $LU^{\leftarrow}$, except that we can skip more steps. For the same reasons as before, we do not need the payload protection in $TI^{\leftrightarrow}$. Further, due to the use of FormOnion (for layers > 1) the ring buffer entries $C_{>j+1}$ do not encrypt any sensitive information, but only random bits and thus do not need to be replaced in the beginning. Finally, the adversary does not obtain any leakage related to k_j^{η} and thus the blocks in the forward header (Step 3)) can be replaced right away.

Table 13. Overview Hybrids for $LU^{\rightarrow}$, $j < n + 1$

	all SNARGs	O_1:	$k_j^{\eta}, k_j^{\gamma}, k_j^{\delta}$ in E_j	$B_j^1, \ldots, B_j^{2N-j}$	δ_j	Oracle
1)	real	param. $b = 0$	real $k_j^{\eta}, k_j^{\gamma}, k_j^{\delta}$	contains path after P_j	contains m	honest proc.
2)	simulated					
3)		C_1^1 fresh `sim`				
4)						encrypts random strings for C_{j+1}^i
5)			$(0, \ldots, 0)$			
6)						fail, if $E_1 = exp$, η modif.
7)				$R_1, R_2, \ldots, R_{2N-j}$		
8)				$(\texttt{sim}, \texttt{sim}, \texttt{sim}), R_2, \ldots, R_{2N-j}$		
9)						fail, if $\eta_1 = exp$, δ modif.
10)						recog+ create
11)					rdm $\bar{m}$	
12)						proc, if $\eta_1 = exp$, δ modif.
13)				$R_1, R_2, \ldots, R_{2N-j}$		
14)				$(\perp, \perp, \perp), R_2, \ldots, R_{2N-j}$		
15)						proc, if $E_1 = exp$, η modif.
16)			real $k_j^{\eta}, k_j^{\gamma}, k_j^{\delta}$			
17)		C_1^1 enc real info				
18)		FormOnion with $b = 1$ param.				
19)	real	param. $b = 1$	(real $k_j^{\eta}, k_j^{\gamma}, k_j^{\delta}$)	(receiver signal and rdm blocks)	(contains $\bar{m}$)	(recog.+ create)

Table 14. Overview Proof for $LU^{\rightarrow}$, j=n+1

Hybrid	Description	Reduction
1)	The $LU^{\rightarrow}$ game with challenge bit chosen as 0	
2)	We simulate the SNARGs for the challenge onion.	SNARG Simulatability
3)	We replace the first ring buffer entry C_1^1 with a fresh encryption of `sim` for the special bitstring `sim`.	IND-CPA/rerand. of PKM
4)	We replace the ring buffer elements C_{j+1}^i for all i with fresh encryptions of random strings (not `sim`) [as FormOnion does for layers $i > 1$].	IND-CPA/rerand. of PKM
5)	We replace the temporary keys $k_j^\eta, k_j^\gamma, k_j^\delta$ at the honest relay by 0..0 before they are encrypted in E_j (and adapt recognizeOnion to the new header), but still use the real keys for the processing.	PK-CCA2
6)	We let the oracle in step 7 output a fail, if the challenge E_j is recognized, but other parts of the header differ.	SUF-CMA
7) & 8)	We replace the blocks $B_j^1, ..., B_j^{2N-j}$ by $(\texttt{sim}, \texttt{sim})$, $R_2, \ldots, R_{2N-j}$ with R_i being randomly chosen (and adapt recognizeOnion to the new header), but use the real blocks for the processing.	PRP-CCA
9)	We let oracle in step 7 output a fail, if the challenge header η_j is recognized, but the payload differs.	SNARG simulation soundness
10)	We let the oracle in step 7 output the replicated layer $j+1$:$(FormOnion(j+1, \mathcal{R}, m^{\leftarrow}, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}}))$ for Reply$(P_H, O, m^{\leftarrow})$ and we output $(\perp, m)$ for Proc(P_H, O), if the challenge η_j is recognized, the payload matches, and real processing of the given onion would not fail.	Same behavior due to definition of recognition and forming of later layers
11)	We replace the content δ_j by a random string of the same length.	PRP-CCA
12)	We revert the changes made in Game 10).	SNARG simulation soundness
13) & 14)	We replace the blocks $B_j^1, ..., B_j^{2N-j}$ by $(\perp, \perp, \perp), R_2, \ldots, R_{2N-j}$ with R_i being randomly chosen (and adapt recognizeOnion to the new header).	PRP-CCA
15)	We revert the changes made in Game 6).	SUF-CMA
16)	We revert the changes made in Game 5).	PK-CCA2
17)	We revert the changes made in Game 3): The ring buffer entry C_1^1 now includes the sender info as in the $b = 1$ case.	IND-CPA/rerand. of PKM
18)	We use FormOnion with the parameter of the $b = 1$ case to generate the first challenge onion layer.	Same behavior except for new draw of randomness
19)	The $LU^{\rightarrow}$ game with challenge bit chosen as 1	SNARG Simulatability

Table 15. Overview Hybrids for $LU^{\rightarrow}$, $j = n + 1$

	all SNARGs	O_1:	$k_j^\eta, k_j^\gamma, k_j^\delta$ in E_j	$B_j^1, \ldots, B_j^{2N-j}$	δ_j	Oracle
1)	real	param. $b = 0$	real $k_j^\eta, k_j^\gamma, k_j^\delta$	contains path after P_j	contains m	honest proc.
2)	simulated					
3)		C_1^1 fresh `sim`				
4)						encrypts random strings for C_{j+1}^i
5)			$(0, \ldots, 0)$			
6)						fail, if $E_1 = exp$, η modif.
7)				$R_1, R_2, \ldots, R_{2N-j}$		
8)				$(\texttt{sim}, \texttt{sim}, \texttt{sim}), R_2, \ldots, R_{2N-j}$		
9)						fail, if $\eta_1 = exp$, δ modif.
10)						recog+ create reply
11)					rdm $\bar{m}$	
12)						proc, if $\eta_1 = exp$, δ modif.
13)				$R_1, R_2, \ldots, R_{2N-j}$		
14)				$(\perp, \perp, \perp), R_2, \ldots, R_{2N-j}$		
15)						proc, if $E_1 = exp$, η modif.
16)			real $k_j^\eta, k_j^\gamma, k_j^\delta$			
17)		C_1^1 enc real info				
18)		FormOnion with $b = 1$ param.				
19)	real	param. $b = 1$	(real $k_j^\eta, k_j^\gamma, k_j^\delta$)	(receiver signal and rdm blocks)	(contains $\bar{m}$)	(recog.+ create)

Table 16. Overview Proof for $LU^{\leftarrow}$, $j^{\leftarrow} = 0$

Hybrid	Description	Reduction
1)	The $LU^{\leftarrow}$ game with challenge bit chosen as 0	
2)	We simulate the SNARGs for the challenge onion.	SNARG Simulatability
3)	We replace the first ring buffer entry C_1^1 with a fresh encryption of `sim` for the special bitstring `sim`.	IND-CPA/rerand. of PKM
4)	We replace the first ring buffer entry $C^{1\overleftarrow{}}_{j^{\leftarrow}+1}$ with a fresh encryption of `sim` for the special bitstring `sim`.	IND-CPA/rerand. of PKM
5)	We replace the temporary keys $k^{\eta}_{n+1}, k^{\gamma}_{n+1}, k^{\delta}_{n+1}$ on the forward path at the honest receiver with $0 \dots 0$ in their encryption for E_{n+1} (and adapt recognizeOnion to the new header), but still use the real keys for the processing.	PK-CCA2
6)	We let the oracle in step 6 output a fail, if the challenge E_{n+1} is recognized, but other parts of the header differ.	SUF-CMA
7)	We replace the block $B^1_{n+1}, \dots, B^{2N-1}_{n+1}$ by a random blocks when forming the challenge onion (and adapt recognizeOnion to the new header), but use the real block for the processing. (In particular, in this way we get rid of all $k^{\eta\overleftarrow{}}_{>j\leftarrow}, k^{\gamma\overleftarrow{}}_{>j\leftarrow}, k^{\delta\overleftarrow{}}_{>j\leftarrow}$)	PRP-CCA
8)	We let the keys $k^{\eta\overleftarrow{}}_{>j\leftarrow}, k^{\gamma\overleftarrow{}}_{>j\leftarrow}, k^{\delta\overleftarrow{}}_{>j\leftarrow}$ used in step 6 be freshly chosen by $P^{\leftarrow}_{j^{\leftarrow}}$.	Games are equivalent
9)	We replace the content $\delta_1^{\leftarrow}$ by a random string of the same length during ReplyOnion.	PRP-CCA
10)	We revert the changes made in Game 7).	PRP-CCA
11)	We revert the changes made in Game 6).	SUF-CMA
12)	We revert the changes made in Game 5).	PK-CCA2
13)	We revert the changes made in Game 4): $C^{1\overleftarrow{}}_{j^{\leftarrow}+1}$ contains now the information of $P^{\leftarrow}_{j^{\leftarrow}}$ as sender.	IND-CPA/rerand. of PKM
14)	We revert the changes made in Game 3).	IND-CPA/rerand. of PKM
15)	The $LU^{\leftarrow}$ game with challenge bit chosen as 1	SNARG Simulatability

G Performance

OR and Mix networks are used for a variety of applications with different performance requirements. For example email services tolerate high latencies, while applications like web browsing, instant messaging or teleconferencing require very low latencies. Further, often high bandwidth is available and networks are built to be scalable.

G.1 UE-Based Scheme - Performance

Onion size In the following, we detail a concrete instantiation of our SNARG-based protocol. All sizes in the following are in bits:

- Kurosawa-Desmedt [27] as the CCA-secure PKE: $|PK| = 512$, $|C| = |M| + 640$.
  - Remark: we count only user-specific parts in pk, the rest can be pushed into global public parameters. pk contains 2 group elements, and C contains 2 group elements and an authenticated encryption of M.
- SHA-3 [2] as hash: $|P| = |\gamma| = 256$ (for HMAC-based MACs with $|k_i^{\gamma}| = 128$)
  - Remark: we count an identity as the size of a hash value (like previous approaches).

Table 17. Overview Hybrids for $LU^{\leftarrow}$, $j^{\leftarrow} = 0$

	all SNARGs	C_1^1	δ_{n+1}	$B_{n+1}^1, \ldots, B_{n+1}^{2N-1}$	$k_{n+1}^{\eta}, k_{n+1}^{\gamma}, k_{n+1}^{\delta}$	$\delta_1^{\leftarrow}$ (bw)	Oracle (Reply)
1)	real	real	real	contains path-end-block, bw path, padding	real	contains $m^{\leftarrow}$ (adv. chosen)	recog.+ honest reply
2)	simulated						
3)		contains `sim`					
4)							in proc result of challenge onion: $C^{1\overleftarrow{\ }}_{j^{\leftarrow}+1} = \mathsf{PKM.Enc}(\mathtt{sim})$
5)					$(0, \ldots, 0)$		(adapt recog.)
6)							fail, if $E_{n+1} = exp$, η modif.
7)				random			(treat B_{n+1}^1 as path-end)
8)							use fresh $k^{\eta\overleftarrow{\ }}_{>j}, k^{\gamma\overleftarrow{\ }}_{>j}, k^{\delta\overleftarrow{\ }}_{>j}$
9)						random	
10)				contains path-end-block, bw path, padding			use actual content of B_{n+1}^1
11)							verify MAC, in case if $E_{n+1} = exp$, η modif.
12)					real		
13)							in proc result of challenge onion: $C^{1\overleftarrow{\ }}_{j^{\leftarrow}+1}$ contains real sender info
14)		real secrets					
15)	real	real	real	contains path-end-block, bw path, padding	real	random	use fresh $k^{\eta\overleftarrow{\ }}_{>j}, k^{\gamma\overleftarrow{\ }}_{>j}, k^{\delta\overleftarrow{\ }}_{>j}$

- AES-128 [1] as symmetric encryption scheme: $|k_i^{\eta}| = 128$
- The NYUAE scheme from [24] for the payload: $|k_i^{\Delta}| = 2560$, $|\Delta_i| = 1536$, $|\delta_i = 37376|$
  - remark: We consider a pairing-based group setting $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with $p = |\mathbb{G}_1| = 256$ and $|\mathbb{G}_2| = 512$. A key consists of 4 $\mathbb{F}_p$, 2 $\mathbb{G}_1$, and 2 $\mathbb{G}_2$ elements, whereas a token consists of 4 $\mathbb{F}_p$, and 2 $\mathbb{G}_1$ elements. As header ciphertexts should not leak whether they contain a key or token, we need to pad to the maximum of both. The payload consists of about 58 $\mathbb{G}_1$, and 44 $\mathbb{G}_2$ elements.

Hence:

- $|E_i| = 2 \cdot 128 + 20 \cdot 128 + 640 = 3456$,
- $|\gamma_i| = 256$,
- $|B_i^j| = 256 + 3456 + 256 = 3968$:
- $|\eta_i| = (2N - 1) \cdot 3968 + 3456 + 256 = (2N - 1) \cdot 3968 + 3712$.
- In total: $|O_i| = |\eta_i| + |\delta_i| = (2N - 1) \cdot 3968 + 3712 + 37376$ bits.

A realistic value for N (maximal path length) can be $N = 3$ or $N = 4$, which leads to an onion size overhead (over $|m|$) of about 7.5 kbytes, resp. 8.5 kbytes.

G.2 SNARG-Based Scheme - Performance

Onion size In the following, we detail a concrete instantiation of our SNARG-based protocol. As before, all sizes in the following are in bits:

- SNARKs of Groth and Maller [21]: $|\pi| = 1024$
  - Remark: this building block operates in a pairing setting with a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where we can assume. In this setting, $\mathbb{G}_1$-, resp. $\mathbb{G}_2$-elements can be set to have 256, resp. 512 bits.
- (Multi-generator-)ElGamal as the rerandomizable PKE: $|C| = |M| + 256$

Table 18. Overview Proof for $LU^{\leftarrow}$, $j^{\leftarrow} > 0$

Hybrid	Description	Reduction
1)	The $LU^{\leftarrow}$ game with challenge bit chosen as 0	
2)	We simulate the SNARGs for the challenge onion.	SNARG Simulatability
3)	We replace the first ring buffer entry C_1^1 with a fresh encryption of `sim` for the special bitstring `sim`.	IND-CPA/rerand. of PKM
4)	We replace the first ring buffer entry $C^{1\overleftarrow{}}_{j^{\leftarrow}+1}$ with a fresh encryption of `sim` for the special bitstring `sim`.	IND-CPA/rerand. of PKM
5)	We replace the temporary keys $k^{\eta\overleftarrow{}}_{j\leftarrow}, k^{\gamma\overleftarrow{}}_{j\leftarrow}, k^{\delta\overleftarrow{}}_{j\leftarrow}$ at the honest relay with $0\ldots0$ in their encryption for $E^{\leftarrow}_{j\leftarrow}$ (and adapt recognizeOnion to the new header) as part of the payload of O_1, but still use the real keys for the processing.	PK-CCA2
6)	We let the oracle in step 6 output a fail, if the challenge $E^{\leftarrow}_{j\leftarrow}$ is recognized, but other parts of the header differ.	SUF-CMA
7)	We replace all $B^{1\overleftarrow{}}_{j\leftarrow}, ..., B^{2N-1\overleftarrow{}}_{j\leftarrow}$ while constructing the header with randomness, but use the real header for $j^{\leftarrow}+1$ as answer to the corresponding Proc oracle request. Note that replacing all $B^{\leftarrow}_{j\leftarrow}$s results in not including any keys for $> j^{\leftarrow}$ in the earlier header.	PRP-CCA
8)	We let the keys $k^{\eta\overleftarrow{}}_{>j\leftarrow}, k^{\gamma\overleftarrow{}}_{>j\leftarrow}, k^{\delta\overleftarrow{}}_{>j\leftarrow}$ used in step 6 be freshly chosen by $P^{\leftarrow}_{j\leftarrow}$.	Games are equivalent
9)	We replace the content $\delta^{\leftarrow}_{j\leftarrow}$ with a random string of the same length during .	PRP-CCA
10)	We revert the changes made in Game 7).	PRP-CCA
11)	We revert the changes made in Game 6).	SUF-CMA
12)	We revert the changes made in Game 5).	PK-CCA2
13)	We revert the changes made in Game 4): $C^{1\overleftarrow{}}_{j^{\leftarrow}+1}$ contains now the information of $P^{\leftarrow}_{j^{\leftarrow}}$ as sender.	IND-CPA/rerand. of PKM
14)	We revert the changes made in Game 3).	IND-CPA/rerand. of PKM
15)	The $LU^{\leftarrow}$ game with challenge bit chosen as 1	SNARG Simulatability

  - Remark: we represent a plaintext string of $\ell \cdot 256$ bits as a vector of ℓ group elements $m_1, \ldots, m_\ell$ (of a suitably-sized group $\mathbb{G}$) and can then set $C = (g^r, h_1^r m_1, \ldots, h_k^r m_k)$ for random r and public key elements $h_1, \ldots, h_k$. Hence $|pk| = \ell \cdot 256$. In our setting, $\ell = 4N + 9$ since we encrypt $(N + 2) \cdot 1024 + 256$ bits (see below). Rerandomization adds (componentwise) an encryption of $(1_{\mathbb{G}})^\ell$.
- Kurosawa-Desmedt [27] as the CCA-secure PKE: $|PK| = 512$, $|C| = |M| + 640$.
  - Remark: as in the UE-based protocol, we count only user-specific parts in PK. Recall that then, the public key PK contains 2 group elements, and the ciphertext C contains 2 group elements and an authenticated encryption of M.
- SHA-3 [2] as hash: $|P| = |\gamma| = 256$ (for HMAC-based MACs with $|k_i^\gamma| = 128$)
  - Remark: we count an identity of a relay as the size of a hash value (like previous OR approaches). Note that identities do not (have to) contain public keys.
- AES-128 [1] as symmetric encryption scheme: $|k_i^\eta| = |k_i^\delta| = 128$

Hence:

- $|E_i| = 3 \cdot 128 + 640 = 1024$,

Table 19. Overview Hybrids for $LU^{\leftarrow}$, $j^{\leftarrow} > 0$

	all SNARGs	C_1^1	$B^{1\overleftarrow{\ }}_{j^{\leftarrow}}, \ldots, B^{2N-1\overleftarrow{\ }}_{j^{\leftarrow}}$	$k^{\eta\overleftarrow{\ }}_{j^{\leftarrow}}, k^{\gamma\overleftarrow{\ }}_{j^{\leftarrow}}, k^{\delta\overleftarrow{\ }}_{j^{\leftarrow}}$ in $E^{\overleftarrow{\ }}_{j^{\leftarrow}}$ (bw)	$B^{(n-j^{\leftarrow}+2)\overleftarrow{\ }}_{j^{\leftarrow}+1}, \ldots, B^{(2N-1)\overleftarrow{\ }}_{j^{\leftarrow}+1}$ (bw)	$\delta_j^{\leftarrow}$ (bw)	Oracle (bw)
1)	real	real secrets	real	real $k^{\eta}_{j^{\leftarrow}}, k^{\gamma}_{j^{\leftarrow}}, k^{\delta}_{j^{\leftarrow}}$	$\mathsf{PRP.Dec}^{-1}(0,\ldots,0)$ (Relay padding)	contains $m^{\leftarrow}$ (adv. chosen)	recog.+ honest proc
2)	simulated						
3)		sim					
4)							in proc result of challenge onion: $C^{1\overleftarrow{\ }}_{j^{\leftarrow}+1} = \mathsf{PKM.Enc(sim)}$
5)				$(0,\ldots,0)$			use $k^{\eta}_{j^{\leftarrow}}, k^{\gamma}_{j^{\leftarrow}}, k^{\delta}_{j^{\leftarrow}}$ if $E^{\overleftarrow{\ }}_{j^{\leftarrow}}$ is recognized
6)							fail, if $E^{\overleftarrow{\ }}_{j^{\leftarrow}} = exp$, η modif.
7)			random		$R_{n-j^{\leftarrow}+2}, \ldots, R_{2N-1}$		use sender's $k^{\eta}_{>j^{\leftarrow}}, k^{\gamma}_{>j^{\leftarrow}}, k^{\delta}_{>j^{\leftarrow}}$ for challenge onion
8)							use fresh $k^{\eta}_{>j^{\leftarrow}}, k^{\gamma}_{>j^{\leftarrow}}, k^{\delta}_{>j^{\leftarrow}}$ for challenge onion
9)						random	
10)			real				
11)							proc, if $E^{\overleftarrow{\ }}_{j^{\leftarrow}} = exp$, η modif.
12)				real $k^{\eta}_{j^{\leftarrow}}, k^{\gamma}_{j^{\leftarrow}}, k^{\delta}_{j^{\leftarrow}}$			
13)							in proc result of challenge onion: $C^{1\overleftarrow{\ }}_{j^{\leftarrow}+1}$ contains real sender info
14)		real secrets					
15)	real	(real secrets)	(real)	(real $k^{\eta}_{j^{\leftarrow}}, k^{\gamma}_{j^{\leftarrow}}, k^{\delta}_{j^{\leftarrow}}$)	(Sender padding)	(contains rdm msg)	(recog.+ $\bar{O}_1$) (new constructed fwd onion)

Table 20. Overview Proof for TI

Hybrid	Description	Reduction
1)	The $TI^{\leftrightarrow}$ game with challenge bit chosen as 0	
2)	We simulate the SNARGs for the challenge onion.	SNARG Simulatability
3)	We replace the blocks $B_{j+1}^{2N-(j+1)}, \ldots, B_{j+1}^{2N-1}$ in step 5 by random strings $R_{n-j+2}, \ldots, R_{N-1}$ (and adapt recognizeOnion to the new header).	PRP-CCA
4)	We replace the temporary keys $k^{\eta\overleftarrow{\ }}_{j^{\leftarrow}}, k^{\gamma\overleftarrow{\ }}_{j^{\leftarrow}}, k^{\delta\overleftarrow{\ }}_{j^{\leftarrow}}$ at the honest relay with $0\ldots0$ in their encryption for $E^{\overleftarrow{\ }}_{j^{\leftarrow}}$ (and adapt recognizeOnion to the new header) as part of the payload of O_{j+1}, but still use the real keys for the processing.	PK-CCA2
5)	We let the oracle in step 7 output a fail, if the challenge $E^{\overleftarrow{\ }}_{j^{\leftarrow}}$ is recognized, but other parts of the header differ.	SUF-CMA
6)	We replace the block $B^{1\overleftarrow{\ }}_{j^{\leftarrow}}$ with a path-end-block and $B^{2\overleftarrow{\ }}_{j^{\leftarrow}}, \ldots, B^{(n^{\leftarrow}+1-j^{\leftarrow})\overleftarrow{\ }}_{j^{\leftarrow}}$ with random blocks in the payload part of the challenge onion representing the backward header.	PRP-CCA
7)	We replace the first ring buffer entry C_{j+1}^1 with the information of P_j as sender.	IND-CPA/rerand. of PKM
8)	We revert the changes of Game 5).	SUF-CMA
9)	We revert the changes of Game 4).	PK-CCA2
10)	The $TI^{\leftrightarrow}$ game with challenge bit chosen as 1	SNARG Simulatability

- $|\gamma_i| = 256$,
- $|B_i^j| = 256 + 1024 + 256 = 1536$:
- $|\eta_i| = (2N-1) \cdot 1536 + 1024 + 256 = (2N-1) \cdot 1536 + 1280$.
- C_i^j: These are ElGamal ciphertexts for messages of size $1024 + N \cdot 1024 + 1024 + 256 = (N+2) \cdot 1024 + 256$ each. This size calculation uses that
  - all C_i^j have the same size, and that
  - C_i^1 has a shorter actual (i.e., in its unpadded form) plaintext than C_i^j $(j > 1)$, since the m encrypted in C_i^1 can be made implicit and reconstructed from R and the encrypted payload in O_1.
  - We do not count the "proc" string at the beginning of each C_i^j $(j > 1)$, since this string can be encoded as a single bit.
- σ_i contains N C_i^j's and N SNARGs: $|\sigma_i| = N \cdot (1024 + N + 2) \cdot 1024 + 256 + 256) = N^2 \cdot 1024 + N \cdot 3584$.

Table 21. Overview Hybrids for TI

	all SNARGs	$k_j^\eta, k_j^\gamma, k_j^\delta$ in E_j	$B^{2N-(j+1)}{}_j, \ldots, B^{2N-1}{}_j$	C^1_{j+1}	$k^{\eta\leftarrow}_{j\leftarrow}, k^{\gamma\leftarrow}_{j\leftarrow}, k^{\delta\leftarrow}_{j\leftarrow}$ in $E^{\leftarrow}_{j\leftarrow}$ (bw)	$B^{2\leftarrow}_{j\leftarrow}, \ldots, B^{(n^\leftarrow - j^\leftarrow+1)\leftarrow}_{j\leftarrow}$ (bw)	Oracle (bw)
1)	real	real	contains real processing	random	real	complete path	fail, if recog. b=0 answer
2)	simulated						
3)			$R^{2N-(j+1)}, \ldots, R^{2N-1}$				
4)					$(0, \ldots, 0)$		(adapt recog.)
5)							fail, if $E^{\leftarrow}_{j\leftarrow} = exp$, η modif.
6)						$(\perp, \perp, \perp), R_2, \ldots, R_{n^\leftarrow - j^\leftarrow+1}$	
7)				real sender info			
8)							proc, if $E^{\leftarrow}_{j\leftarrow} = exp$, η modif.
9)					real		
10)	real	real	Sender padding	real sender info	real	shortened path	fail if recog. b=1 answer

– $|\delta_i| = |m|$ (encrypted in-place)
– In total: $|O_i| = |\eta_i| + |\sigma_i| + |\delta_i| = |m| + N^2 \cdot 1024 + N \cdot 3584 + (2N-1) \cdot 1536 + 1280$ bits.

A realistic value for N (maximal path length) can be $N = 3$ or $N = 4$, which leads to an onion size overhead (over $|m|$) of about 3kbytes, resp. 5kbytes.