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

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 [18]. Mix [8] and Onion Routing (OR) Networks, like Tor [13], 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 [8]; Goldschlag, Reed and Syverson proposed a clever setup procedure together with the same basic ideas, but decided against random delays [16], which later on led to the development of the best known anonymous com-

[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 [1]. We discuss their work, and why we believe that their solution is not sufficient, below.

munication network, Tor [13]. In the following years many solutions applied the same technique [9,10,11,12,26,25]. 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 [4,11]: 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. [23] modeled and analyzed OR. However, this work does not contain any proposal to prove future systems secure. Backes et al.'s ideal functionality [2] 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. [14] on the other hand, oversimplifies the problem and cannot support replies either. Further approaches [6,7,19] 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 [4] 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 [26] of Minx [12], as well as for the security proof of Sphinx [11], 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 [9,10,25] 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. [21] recently recognized and corrected flaws in those definitions that allowed for sincere practical consequences in the form of a *malleability attack*:

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

[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.

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 [21]. 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 [21] was introduced at the example of HORNET [9], an OR network that was proposed as the successor of Tor.

On the work of Ando and Lysyanskaya. In a recent work [1], 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 [1] with randomness. The next honest relays process the onion as usual without noticing this, as [1] 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 [9,10].

While Ando and Lysyanskaya target the same setting, they avoid the challenge imposed by the above malleability attack [21] 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 [9,11,26] 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 [21], 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 [1] and practical proposals [9] 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 [11] and the Shallot scheme of Ando and Lysyanskaya [1]. 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 [5] and plaintext integrity) and re-randomizable keys, as a central primitive. Intuitively, using UE for encrypting the payload message (in both commu-

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

nication 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 [24,3]), 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 [20] 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 kilobytes 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. our extended version [22] 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 [20].

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 [11,1], 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 [17] (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. our extended version [22] 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 [21] while making reply onions indistinguishable from request onions as desired by practical protocols like HORNET [9].

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
$\mathcal{P}$	an onion path
m	a message
P_i	for the i-th router name on the forward path, P_0 (usually $= P^{\leftarrow}_{n^{\leftarrow}+1}$) is the forward sender and $P_{n+1}(= P^{\leftarrow}_0)$ 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 [4] as used in [21] by adding an algorithm to create replies. Our ideal functionality extends the one of [4] as used in [21] and has similarities to [1], 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* $\text{ProcOnion}(SK_i, O_i, P_i)$ *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 [4] as used in [21] with an algorithm to send replies, similar to [1].

Definition 1 (Repliable OR Scheme). *A Repliable OR Scheme is a tuple of PPT algorithms* $(G, \mathrm{FormOnion}, \mathrm{ProcOnion}, \mathrm{ReplyOnion})$ *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. $\mathrm{FormOnion}(i, \mathcal{R}, m, \mathcal{P}^\rightarrow, \mathcal{P}^\leftarrow, (PK)_{\mathcal{P}^\rightarrow}, (PK)_{\mathcal{P}^\leftarrow})$ *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* $\mathcal{P}^\rightarrow = (P_1, \dots, P_{n+1})$*, a backward path* $\mathcal{P}^\leftarrow = (P_1^\leftarrow, \dots, P_{n^\leftarrow+1}^\leftarrow)$*, public keys* $(PK)_{\mathcal{P}^\rightarrow} = (PK_1, \dots, PK_{n+1})$ *of the relays on the forward path, and public keys* $(PK)_{\mathcal{P}^\leftarrow} = (PK'_1, \dots, PK'_{n^\leftarrow+1})$ *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. $\mathrm{ProcOnion}(SK, O, P)$ *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. $\mathrm{ReplyOnion}(m^\leftarrow, O, P, SK)$ *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* $(G, \mathrm{FormOnion}, \mathrm{ProcOnion}, \mathrm{ReplyOnion})$ *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* $\mathcal{P}^\rightarrow = (P_1, \dots, P_{n+1})$ *and* $\mathcal{P}^\leftarrow = (P_1^\leftarrow, \dots, P_{n^\leftarrow+1}^\leftarrow)$*, 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* $Q_1 := P_1$*,* $O_1 \leftarrow \mathrm{FormOnion}(1, \mathcal{R}, m, (P_1, \dots, P_{n+1}), (P_1^\leftarrow, \dots, P_{n^\leftarrow+1}^\leftarrow), (PK_1, \dots, PK_{n+1}), (PK_1^\leftarrow, \dots, PK_{n^\leftarrow+1}^\leftarrow))$*,* $(O_{i+1}, Q_{i+1}) \leftarrow \mathrm{ProcOnion}(SK_i, O_i, Q_i)$*.*

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

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

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

[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 our extended version [22] for 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 [21] 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.

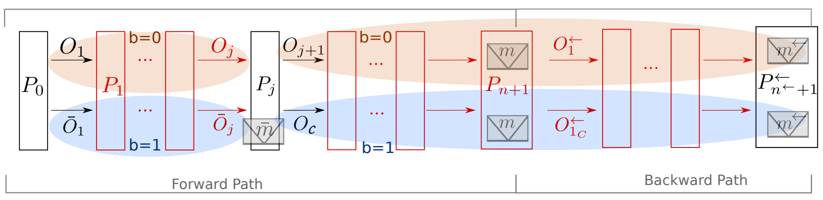


Fig. 1. Forwards Layer-Unlinkability illustrated: Red boxes are corrupted relays, black honest relays, orange ellipses are the $b = 0$ and the blue the $b = 1$ case. $\bar{m}$ is a random message. The main idea is that the adversary cannot distinguish between real and random onions *before* P_j.

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* $\bar{\mathcal{P}}^{\rightarrow} = (P_1, \dots, P_j)$*, 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 $(P^{\leftarrow}_1, O_c)$ *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

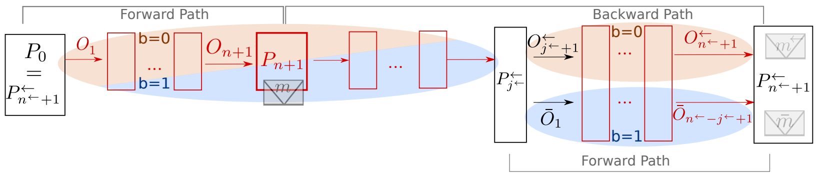

Forward Path
Backward Path
Forward Path
$P_0 = P_{n^\leftarrow+1}^\leftarrow$
O_1
b=0
b=1
O_{n+1}
P_{n+1}
m
$P_{j^\leftarrow}^\leftarrow$
$O_{j^\leftarrow+1}^\leftarrow$
$O_{n^\leftarrow+1}^\leftarrow$
$\bar{O}_1$
$\bar{O}_{n^\leftarrow-j^\leftarrow+1}$
$P_{n^\leftarrow+1}^\leftarrow$
$\bar{m}$

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* $P_{j^\leftarrow}^\leftarrow$.

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 our extended version [22].

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

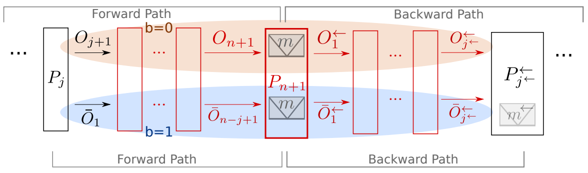


Fig. 3. Repliable Tail-Indistinguishability illustrated: Red boxes are corrupted relays, black honest relays, orange ellipses are the $b = 0$ and the blue the $b = 1$ case. While the adversary can learn the behavior between *between* P_j and $P_{j^\leftarrow}^\leftarrow$ she cannot connect it to anything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

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 [21] 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 our extended version [22].

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 [4,21], we show that these properties imply the ideal functionality in our extended version [22]. 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 our extended version [22] 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 recapitulate the definitions of an UE scheme providing RCCA security and plaintext integrity given in [20].

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 [20]). *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 [20]). UE *is called UP-IND-RCCA secure if for any PPT adversary $\mathcal{A}$ the following advantage is negligible in κ:*

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

```
Experiment $\mathsf{Exp}^{\text{up-ind-rcca}}_{\mathsf{UE},\mathcal{A}}(pp, b)$
  $(sp, k_1, \Delta_0, \mathbf{Q}, \mathbf{K}, \mathbf{T}, \mathbf{C}^*) \leftarrow \mathsf{Init}(pp)$
  $(M_0, M_1, state) \leftarrow_{\text{R}} \mathcal{A}^{\mathsf{Enc},\mathsf{Dec},\mathsf{Next},\mathsf{ReEnc},\mathsf{Corrupt}}(sp)$
  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 $\mathtt{invalid}$.
$\mathsf{ReEnc}(C, i)$: 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 $\mathtt{true}$. Else, return $\mathtt{false}$.

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.

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

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 [20]). *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*

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

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

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 [20]). 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 $\mathsf{Exp}^{\text{up-int-ptxt}}_{\mathsf{UE},\mathcal{A}}(pp)$
  $(sp, k_1, \Delta_0, \mathbf{Q}, \mathbf{K}, \mathbf{T}) \leftarrow \mathsf{Init}(pp)$
  $c^* \leftarrow_R \mathcal{A}^{\mathsf{Enc},\mathsf{Dec},\mathsf{Next},\mathsf{ReEnc},\mathsf{Corrupt}}(sp)$
  return 1 if $\mathsf{UE.Dec}(k_{e_{\mathsf{end}}}, c^*) = m^* \neq \perp$ and $(e_{\mathsf{end}}, m^*) \notin \mathbf{Q}^*$,
     and $\nexists e \in \mathbf{K}$ where $i \in \mathbf{T}$ for $i = e$ to $e_{\mathsf{end}}$; 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 $C \leftarrow_R \mathsf{UE.Enc}(k_e, M)$ 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

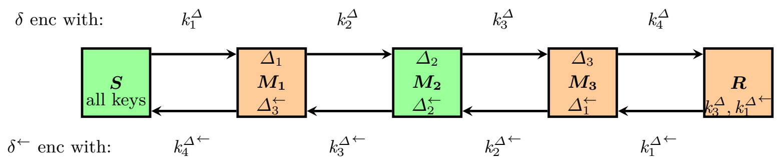

δ enc with:
k_1^{Δ}
k_2^{Δ}
k_3^{Δ}
k_4^{Δ}
$\boldsymbol{S}$ all keys
Δ_1 $\boldsymbol{M_1}$ $\Delta_3^{\leftarrow}$
Δ_2 $\boldsymbol{M_2}$ $\Delta_2^{\leftarrow}$
Δ_3 $\boldsymbol{M_3}$ $\Delta_1^{\leftarrow}$
$\boldsymbol{R}$ $k_3^{\Delta}, k_1^{\Delta\leftarrow}$
$\delta^{\leftarrow}$ enc with:
$k_4^{\Delta\leftarrow}$
$k_3^{\Delta\leftarrow}$
$k_2^{\Delta\leftarrow}$
$k_1^{\Delta\leftarrow}$

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 [11] and the Shallot scheme [1]. 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: $\eta_i = (E_i, B_i^1, B_i^2, \dots, B_i^{2N-1}, \gamma_i)$. We describe a non-repliable header first and later on extend it to be repliable. The first layer's header η_1 contains:

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

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

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

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

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

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

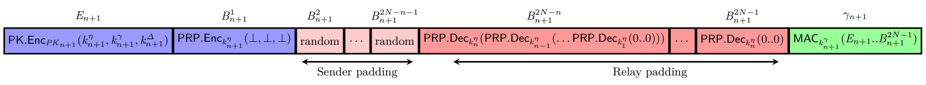
$$\eta_3 = (\dots, \quad B_3^{2N-3} \quad , \quad B_3^{2N-2} \quad , \quad B_3^{2N-1} \quad ,\dots)$$

$$\eta_3 = (\dots, \mathsf{PRP.Enc}_{k_3^{\eta}}(B_4^{2N-4}), \mathsf{PRP.Dec}_{k_2^{\eta}}(\mathsf{PRP.Dec}_{k_1^{\eta}}(\mathbf{0}\dots\mathbf{0})), \mathsf{PRP.Dec}_{k_2^{\eta}}(0\dots0), \dots)$$

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.

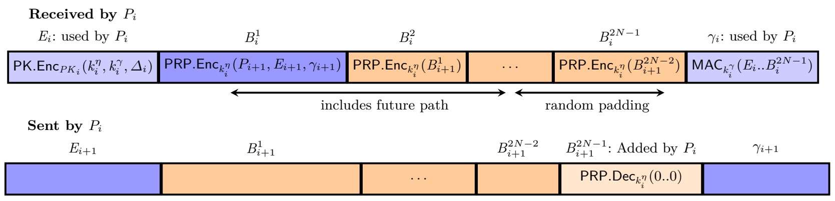
To construct η_1 for a path $\mathcal{P} = (P_1, \dots, P_{n+1}), n+1 \leq N-1$, the sender builds the onion from the center, i.e. calculates the layer for the receiver first:

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



E_{n+1}
B^1_{n+1}
B^2_{n+1}
B^{2N-n-1}_{n+1}
B^{2N-n}_{n+1}
B^{2N-1}_{n+1}
γ_{n+1}
$\mathsf{PK.Enc}_{PK_{n+1}}(k^\eta_{n+1}, k^\gamma_{n+1}, k^\Delta_{n+1})$
$\mathsf{PRP.Enc}_{k^\eta_{n+1}}(\perp,\perp,\perp)$
random
$\dots$
random
$\mathsf{PRP.Dec}_{k^\eta_n}(\mathsf{PRP.Dec}_{k^\eta_{n-1}}(\dots\mathsf{PRP.Dec}_{k^\eta_1}(0..0)))$
$\dots$
$\mathsf{PRP.Dec}_{k^\eta_n}(0..0)$
$\mathsf{MAC}_{k^\gamma_{n+1}}(E_{n+1}..B^{2N-1}_{n+1})$
Sender padding
Relay padding

Fig. 5. Non-repliable receiver header illustrated



Received by P_i
E_i: used by P_i
B^1_i
B^2_i
B^{2N-1}_i
γ_i: used by P_i
$\mathsf{PK.Enc}_{PK_i}(k^\eta_i, k^\gamma_i, \Delta_i)$
$\mathsf{PRP.Enc}_{k^\eta_i}(P_{i+1}, E_{i+1}, \gamma_{i+1})$
$\mathsf{PRP.Enc}_{k^\eta_i}(B^1_{i+1})$
$\dots$
$\mathsf{PRP.Enc}_{k^\eta_i}(B^{2N-2}_{i+1})$
$\mathsf{MAC}_{k^\gamma_i}(E_i..B^{2N-1}_i)$
includes future path
random padding
Sent by P_i
E_{i+1}
B^1_{i+1}
B^{2N-2}_{i+1}
B^{2N-1}_{i+1}: Added by P_i
γ_{i+1}
$\dots$
$\mathsf{PRP.Dec}_{k^\eta_i}(0..0)$

Fig. 6. Processing illustrated

2. Construct η_{n+1}:

$$\begin{aligned}
E_{n+1} &= \mathsf{PK.Enc}_{PK_{n+1}}(k^\eta_{n+1}, k^\gamma_{n+1}, k^\Delta_{n+1})\\
B^1_{n+1} &= \mathsf{PRP.Enc}_{k^\eta_{n+1}}(\perp,\perp,\perp)\\
B^{2N-i}_{n+1} &= \mathsf{PRP.Dec}_{k^\eta_n}(\mathsf{PRP.Dec}_{k^\eta_{n-1}}(\dots\mathsf{PRP.Dec}_{k^\eta_{n+1-i}}(0\dots0)))\\
&\quad \text{for } 1 \le i \le n \text{ (blocks appended by relays)}\\
B^{2N-i}_{n+1} &\leftarrow^R \{0,1\}^{L_1} \text{ for } n+1 \le i \le 2N-2\\
&\quad \text{(blocks as path length padding calculated by sender)}\\
\gamma_{n+1} &= \mathsf{MAC}_{k^\gamma_{n+1}}(E_{n+1}, B^1_{n+1}, B^2_{n+1}, \dots, B^{2N-1}_{n+1})
\end{aligned}$$

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

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

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 $m' = 0\|m$: $\delta_i = \mathsf{UE.ReEnc}_{\Delta_{i-1}}(\dots(\mathsf{UE.ReEnc}_{\Delta_1}(\mathsf{UE.Enc}_{k^\Delta_1}(m'))\dots)$.

Onion Processing. The same processing is used for any forward or backward, repliable or not-repliable onion. If P_i receives an onion $O_i = (\eta_i = (E_i, B^1_i, B^2_i, \dots, B^{2N-1}_i, \gamma_i), \delta_i))$, 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 $P_{i+1} = E_{i+1} = \gamma_{i+1} = \perp$ (P_i is the receiver), skip processing of the header and process the payload (and check for replies as explained below)]

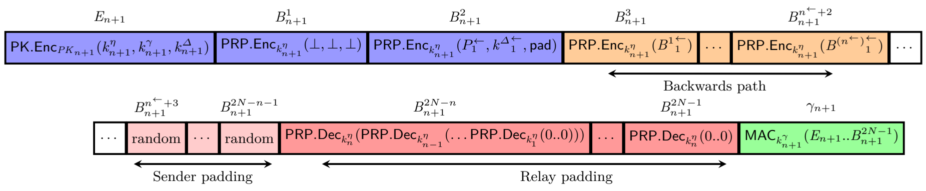

E_{n+1}
B^1_{n+1}
B^2_{n+1}
B^3_{n+1}
$B^{n^{\leftarrow}+2}_{n+1}$
$\mathsf{PK.Enc}_{PK_{n+1}}(k^{\eta}_{n+1}, k^{\gamma}_{n+1}, k^{\Delta}_{n+1})$
$\mathsf{PRP.Enc}_{k^{\eta}_{n+1}}(\perp, \perp, \perp)$
$\mathsf{PRP.Enc}_{k^{\eta}_{n+1}}(P^{\leftarrow}_1, k^{\Delta}{}^{\leftarrow}_1, \mathsf{pad})$
$\mathsf{PRP.Enc}_{k^{\eta}_{n+1}}(B^1{}^{\leftarrow}_1)$
...
$\mathsf{PRP.Enc}_{k^{\eta}_{n+1}}(B^{(n^{\leftarrow})}{}^{\leftarrow}_1)$
...
Backwards path
$B^{n^{\leftarrow}+3}_{n+1}$
B^{2N-n-1}_{n+1}
B^{2N-n}_{n+1}
B^{2N-1}_{n+1}
γ_{n+1}
...
random
...
random
$\mathsf{PRP.Dec}_{k^{\eta}_n}(\mathsf{PRP.Dec}_{k^{\eta}_{n-1}}(\ldots \mathsf{PRP.Dec}_{k^{\eta}_1}(0..0)))$
...
$\mathsf{PRP.Dec}_{k^{\eta}_n}(0..0)$
$\mathsf{MAC}_{k^{\gamma}_{n+1}}(E_{n+1}..B^{2N-1}_{n+1})$
Sender padding
Relay padding

Fig. 7. Repliable receiver header illustrated

4. Decrypt the rest of the header $B^{j-1}_{i+1} = \mathsf{PRP.Dec}_{k^{\eta}_i}(B^j_i)$ for $j \geq 2$
5. Pad the new header $B^{2N-1}_{i+1} = \mathsf{PRP.Dec}_{k^{\eta}_i}(0 \ldots 0)$
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 ($\delta_{i+1} = \mathsf{UE.Dec}_{k^{\Delta}_{n+1}} = 0 \| m$ if no reply)] and abort if this fails
7. Send the new onion $O_{i+1} = ((E_{i+1}, B^1_{i+1}, \ldots, B^{2N-1}_{i+1}, \gamma_{i+1}), \delta_{i+1})$ to the next relay P_{i+1}

Constructing a Repliable Onion. Let m be the message for the receiver, $\mathscr{P}^{\leftarrow} = (P^{\leftarrow}_1, \ldots, P^{\leftarrow}_{n^{\leftarrow}+1})$, $n^{\leftarrow}+1 \leq N-1$ the backward path. To send a repliable onion, the sender performs the following steps:

1. Construct a (non-repliable) header $\eta^{\leftarrow}_1$ with path $\mathscr{P}^{\leftarrow}$. Let the chosen keys be $k^{\eta}{}^{\leftarrow}_1, \ldots, k^{\eta}{}^{\leftarrow}_{n^{\leftarrow}+1}$ and $k^{\Delta}{}^{\leftarrow}_1, \Delta^{\leftarrow}_1, \ldots, \Delta^{\leftarrow}_{n^{\leftarrow}}, k^{\Delta}{}^{\leftarrow}_{n^{\leftarrow}+1}$.
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 $B^2_{n+1} = \mathsf{PRP.Enc}_{k^{\eta}_{n+1}}(P^{\leftarrow}_1, k^{\Delta}{}^{\leftarrow}_1, \mathsf{pad})$.
   – Set $B^i_{n+1} = \mathsf{PRP.Enc}_{k^{\eta}_{n+1}}(B^{(i-2)}{}^{\leftarrow}_1)$ for $3 \leq i \leq n^{\leftarrow} + 2$.
   – Store the key $k^{\Delta}{}^{\leftarrow}_{n^{\leftarrow}+1}$
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 $P^{\leftarrow}_1$ and $k^{\Delta}{}^{\leftarrow}_1$ from B^2_{n+1}. 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 $\delta^{\leftarrow}_1 = \mathsf{UE.Enc}_{k^{\Delta \leftarrow}{}_1}(m^{\leftarrow})$
2. Evolve the header (as before but shifting the header by two blocks):
   – $B^{(j-2)}{}^{\leftarrow}_1 = \mathsf{PRP.Dec}_{k^{\eta}_{n+1}}(B^j_{n+1})$ for $j \geq 3$
   – $B^{(2N-1)}{}^{\leftarrow}_1 = \mathsf{PRP.Dec}_{k^{\eta}_{n+1}}(0 \ldots 0)$ (i.e. receiver padding)
   – $B^{(2N-2)}{}^{\leftarrow}_1 = \mathsf{PRP.Dec}_{k^{\eta}_{n+1}}(1 \ldots 1)$ (i.e. receiver padding)
3. Send the onion $O^{\leftarrow}_1 = (\eta^{\leftarrow}_1, \delta^{\leftarrow}_1)$ to $P^{\leftarrow}_1$

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^{Δ}_{n+1} for her matches a stored ${k^{\Delta}}^{\leftarrow}_{n^{\leftarrow}+1}$s (it indeed is a reply) or not (it is just a new message). She uses the key and decrypts the message: $m = \mathsf{UE.Dec}_{k^{\Delta}_{n+1}}(\delta)$.

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 our extended version [22]. Further, correctness follows from inspection of our scheme.

FormOnion - later layers. FormOnion for $i > 1$ uses the k^{Δ}_i belonging to the corresponding epoch to create the payload $\delta = \mathsf{UE.Enc}_{k^{\Delta}_i}(m)$ 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

[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.

further need to ensure, that RecognizeOnion does not mistreat any processing of e.g. modified onions. Therefore, we leverage the UE properties for the payload protection and the MAC for the header.

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 ($j = n + 1$): 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 our extended version [22].

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 our extended version [22] 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 [15] 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 [3]). 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 $\pi_1, \dots, \pi_N$ and an encrypted ring buffer (that consists of ciphertexts $C_1, \dots, C_N$). 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 $\pi'_1, \dots, \pi'_N$ 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.
- $C_2, \dots, C_N$ are the values $C'_1, \dots, C'_{N-1}$ 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 $C_1, \dots, C_{N-i}$ 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 $C_i = (C_i^1, \dots, C_i^N)$ and the SNARGs $\pi_i = (\pi_i^1, \dots \pi_i^N)$: $\sigma_i = (C_i, \pi_i)$.

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.

$$
\begin{aligned}
C_1^1 &= \mathsf{PKM.Enc}_{PK^{\mathrm{M}}}(\mathcal{I}) \text{ with } \mathcal{I} = (\mathtt{form}, (R, m, \mathcal{P}^{\rightarrow}, \mathcal{P}^{\leftarrow}, (PK)_{\mathcal{P}^{\rightarrow}}, (PK)_{\mathcal{P}^{\leftarrow}})) \\
C_1^j &\leftarrow^R \{0,1\}^{L_3} \setminus \{\mathtt{sim}\} \text{ with } \mathtt{sim} \text{ being a special symbol} \\
&\qquad \text{and } L_3 \text{ the fixed length of ring buffer elements} \\
C_i^1 &= \mathsf{PKM.Enc}_{PK^{\mathrm{M}}}(\mathcal{I}) \text{ with } \mathcal{I} = (\mathtt{proc}, (SK_{i-1}, \pi_{i-1}^1, \dots, \pi_{i-1}^N, E_{i-1}, P_{i-1})) \\
C_i^j &= \mathsf{PKM.ReRand}_{PK^{\mathrm{M}}}(C_{i-1}^{j-1})
\end{aligned}
$$

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 $X = (\eta_i, (C_i^1, \dots, C_i^{N-j}), \delta_i)$ 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 $M = (\texttt{form}, I)$, 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 $M = (\texttt{proc}, (SK_{i-1}, \pi_{i-1}^1, \ldots, \pi_{i-1}^N, E_{i-1}, P_{i-1}, \mathcal{R}))$, 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) $\text{ProcOnion}_{\text{partial}}^j(SK_{i-1}, (\eta_{i-1}, (C_{i-1}^1, \ldots, C_{i-1}^{N-j-1}), \delta_{i-1}), P_{i-1}; \mathcal{R}) = (\eta_i, (C_i^1, \ldots, C_i^{N-j}), \delta_i)$, where $\text{ProcOnion}_{\text{partial}}^j$ 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 $\mathsf{PRP2.Dec}$:

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 $\text{ProcOnion}_{\text{partial}}$ 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: $C_{i+1}^1 = \mathsf{PKM.Enc}_{PK^{\mathrm{M}}}(\mathcal{I})$
6. Build the new SNARG-Sequence π_{i+1}
7. Send the new onion $O_{i+1} = ((E_{i+1}, B_{i+1}^1, \dots, B_{i+1}^{2N-1}, \gamma_{i+1}), (C_{i+1}, \pi_{i+1}), \delta_{i+1})$ 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":
$m = \mathsf{PRP2.Dec}_{k_1^\delta}(\mathsf{PRP2.Enc}_{k_2^\delta}(\dots(\mathsf{PRP2.Enc}_{k_{n^{\leftarrow}}^\delta}(\mathsf{PRP2.Enc}_{k_{n^{\leftarrow}+1}^\delta}(\delta))\dots)))$

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 our extended version [22].

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. M. Ando and A. Lysyanskaya. Cryptographic shallots: A formal treatment of repliable onion encryption. *eprint, https://eprint.iacr.org/2020/215.pdf*, 2020.
2. M. Backes et al. Provably secure and practical onion routing. In *Computer Security Foundations Symposium*, pages 369–385, 2012.
3. N. Bitansky et al. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, 2012.

4. J. Camenisch and A. Lysyanskaya. A formal treatment of onion routing. In *Annual International Cryptology Conference*, 2005.
5. R. Canetti, H. Krawczyk, and J. B. Nielsen. Relaxing chosen-ciphertext security. In *CRYPTO 2003*, pages 565–582, 2003.
6. D. Catalano et al. Fully non-interactive onion routing with forward secrecy. *INT J INF SECUR*, 2013.
7. D. Catalano, D. Fiore, and R. Gennaro. A certificateless approach to onion routing. *INT J INF SECUR*, 2017.
8. D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 1981.
9. 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.
10. C. Chen et al. TARANET: Traffic-Analysis Resistant Anonymity at the NETwork layer. *IEEE EuroS&P*, 2018.
11. G. Danezis and I. Goldberg. Sphinx: A compact and provably secure mix format. In *IEEE S&P*, 2009.
12. G. Danezis and B. Laurie. Minx: A simple and efficient anonymous packet format. In *WPES*, 2004.
13. R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.
14. J. Feigenbaum, A. Johnson, and P. Syverson. Probabilistic analysis of onion routing in a black-box model. *ACM TISSEC*, 2012.
15. G. Fuchsbauer. Subversion-zero-knowledge SNARKs. In *PKC 2018*, pages 315–347, 2018.
16. D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding routing information. In *International workshop on information hiding*, 1996.
17. J. Groth and M. Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In *CRYPTO 2017*, pages 581–612, 2017.
18. M. Hayden. The price of privacy: Re-evaluating the nsa. Johns Hopkins Foreign Affairs Symposium, Apr. 2014.
19. A. Kate, G. M. Zaverucha, and I. Goldberg. Pairing-based onion routing with improved forward secrecy. *ACM TISSEC*, 13, 2010.
20. M. Klooß, A. Lehmann, and A. Rupp. (R)CCA secure updatable encryption with integrity protection. In *EUROCRYPT 2019*, pages 68–99, 2019.
21. C. Kuhn, M. Beck, and T. Strufe. Breaking and (Partially) Fixing Provably Secure Onion Routing. *IEEE S&P*, 2020.
22. C. Kuhn, D. Hofheinz, A. Rupp, and T. Strufe. Onion routing with replies. Cryptology ePrint Archive, Report 2021/1178, 2021. `https://ia.cr/2021/1178`.
23. S. Mauw, J. H. Verschuren, and E. P. de Vink. A formalization of anonymity and onion routing. In *ESORICS*, 2004.
24. S. Micali. CS proofs (extended abstracts). In *35th FOCS*, pages 436–453, 1994.
25. A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis. The loopix anonymity system. In *USENIX*, 2017.
26. E. Shimshock, M. Staats, and N. Hopper. Breaking and provably fixing minx. In *PETS*, 2008.