A Framework for Provably Secure Onion Routing against a Global Adversary

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

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

Thorsten Strufe
KIT Karlsruhe
firstname.lastname@kit.edu

ABSTRACT

Onion routing and mix networks are a central technology to enable anonymous communication on the Internet. As such, a large number of protocols and model variants have been explored in the field, which offer differing levels of privacy, exhibit vulnerabilities, or even supersede each other. These factors make discovering the appropriate formalization for new developments difficult, and some model variants have not been formalized at all.

We address this issue by creating one parametrized framework that encompasses the onion routing and mix network models and functionalities with a global adversary in the related work. In doing so, we create a categorization of the variants of onion routing models in use in the related work and map common OR and mix network protocols to their variants. For each identified variant: Our framework offers i) an ideal functionality in the Universal Composability framework, and ii) game-based properties that imply realization of the ideal functionality when a protocol satisfies them. In effect, our framework both unifies and extends previous formalization efforts in the field.

KEYWORDS

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

1 INTRODUCTION

Onion routing (OR) and mixing are core techniques in the field of privacy enhancing technologies: Both OR networks and mix networks aim to provide an anonymous communication service to their users. To prevent adversaries from linking users to the services they access, any packets a user sends are first routed through a series of relays. Each relay transforms the packet such that the resulting packet cannot be linked to the incoming packet. This is often realized through multiple layers of encryption, one of which is "peeled" at each relay — hence the name *Onion* routing. With enough users and honest relays, each user's packet has a protective anonymity set. Using this approach, OR protocols typically provide protection against local adversaries that control only parts of the network or individual corrupted relays. The most commonly used OR protocol today is TOR [15]. Mix networks like Minx [13, 27] use the same fundamental approach, but additionally apply *mixing*, which shuffles packets at honest relays to prevent an adversary from linking the incoming packets at a relay to the outgoing packets at the cost of more latency [8]. Mix networks typically target global adversaries.

Both OR networks and mix networks require a suitable packet format in order to prevent adversaries from linking the packets that exit an honest relay to those that enter it — if an attacker could link these, the benefit of using these relays would be void. It is common in the field to treat the packet format used for OR and mix protocols separately from the protocols' mechanisms: Since the basic operating principles of OR and mixing stay the same between protocols, one packet format can be reused between protocols [3, 12]. There is a significant body of work in the field that strives to define and create these packet formats in a provably secure context [1, 5, 19, 20, 26]. Many of these works formalize security using both an *ideal functionality* for the protocols as well as game-based security properties. The ideal functionalities are based in the Universal Composability (UC) framework and are effectively an abstract, idealized version of the real OR or mix network protocols. The abstraction makes the security properties of the protocol easier to derive: In many cases, these properties can simply be observed from the idealized protocol. However, showing that a real protocol is equivalent to an ideal functionality can be cumbersome, since it involves proving that any attack on the real protocol can also be simulated on the ideal functionality in an indistinguishable way. The game-based properties solve this issue by offering equivalent security using standard cryptographic games. Proving that a protocol satisfies these can often be done through common reduction techniques.

Authors of OR and mix network schemes use these formalizations to design new packet formats and prove their security [1, 9, 10, 12, 19, 20], but there are significant hurdles here: The existing formalizations are spread over many works that target different variants of OR, with differing network and adversary models, functionalities, and protections. Identifying the right formalization to base one's work upon is difficult, especially since some of these papers build upon each other and correct mistakes in earlier works even when targeting a new variant [20, 26]. In addition, several variants used in existing protocols have not been formalized at all, leaving a gap in the existing works. Adapting the formalizations for new OR variants requires care, since a model change necessitates a complete re-evaluation of the schemes, properties, and proofs used.

Our paper combines and extends previous formalizations by creating a unified theoretical foundation for future work in provably secure packet formats for OR and mix networks with a global adversary. First, we provide a categorization of the related work in provably secure OR, including existing formalizations and protocols. We then classify the variants in use today, creating a unified, parametrized model, our STIR model. Each letter of the acronym

Proceedings on Privacy Enhancing Technologies 2024(2), 141–159

https://doi.org/10.56553/popets-2024-0045

STIR stands for a binary option, with a total of 16 variants being included in our analysis.

Second, we create a complete, unified theoretical foundation for provably secure OR schemes and packet formats in each variant. This formalization encompasses and generalizes the formalizations in the related work while also extending to previously-unformalized OR variants. Multiple existing protocols are based on these variants and can now be analyzed using our new formalization. Our formalization includes an ideal functionality in the UC framework as well as game-based properties that we prove are equivalent to the ideal functionality in each variant.

In addition to our complete, unified and parametrized formal infrastructure, we supply an intuition for the security properties of OR and mix network packet formats.

In summary, our main contributions are:

- a categorization of both protocols and formalizations by the variant of OR they are based in,
- a classification of the OR variants in use today, including an analysis of each variant,
- a complete, parametrized formalization of provably secure OR schemes in each variant we observe, including an ideal functionality, game-based properties, and a proof of their equivalence, while maintaining compatibility with the related work where possible,

Outline. Section 2 introduces OR and mix networks along with the basic principles of the existing formalizations. Section 3 introduces the STIR model and explains each variant along with categorizing the related work into variants. Section 4 explains our notation and enumerates the assumptions used in our model. Section 5 defines OR schemes in STIR variants. Section 6 briefly explains the ideal functionality for OR. Section 7 describes the game-based properties and the proof that they are equivalent to our ideal functionality. Section 8 discusses additional aspects of OR protocol design with our model. Section 9 concludes this paper.

2 BACKGROUND

In this section, we first introduce onion routing (OR) along with mix networks, then explain the various models and capabilities of OR networks and the existing approaches to formally analyzing OR and mix packet formats in the related work.

2.1 Onion Routing and Mix Networks

Onion routing prevents an adversary from linking message senders to their chosen receivers. This is done by routing traffic via multiple intermediate servers known as *relays*. Each relay processes the packets and forwards them to the next relay. If multiple honest senders use the same honest relay for their route (or *path*), an adversary should not be able to tell which of the relay's outgoing packets corresponds to which incoming packet based on an analysis of the packets themselves. To ensure that an incoming packet is not linkable to its outgoing packet, packets have a layer of encryption for each relay on their path. Before sending a packet, the sender adds one encryption layer for each relay on the path. The relays then each remove one layer, with the final relay recovering the plaintext. Packets are thus appropriately referred to as *onions* that are *peeled* at relays [16].

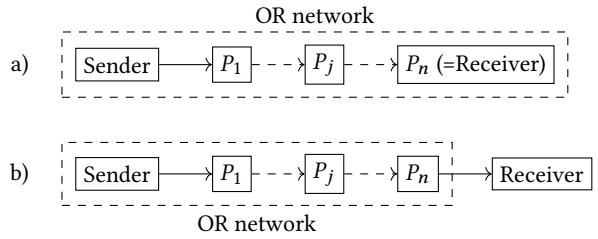

Figure 1: The difference between the integrated-system network model (a) and the service network model (b). P_i denotes a relay. Dashed lines indicate omitted intermediate relays.

Anonymous communication systems designed in this way are typically classified into one of two categories: *Onion Routing* networks and *Mix networks*. The former, (e.g., TOR [15]) mainly protect against local adversaries (i.e., individual corrupted relays or network portions). They do not attempt to protect against global adversaries. The other category is called *Mix networks* [8]. Mix networks like Nym [14] add another step to onion processing at relays: In addition to decrypting and forwarding onions, they delay and reorder onions to combat traffic analysis attacks from global attackers [8].

In this paper, we focus on the construction of packet schemes in the context of a global adversary. Since the structure of packet formats in OR and Mix networks is the same, our work remains applicable to both OR and Mix networks in doing so. Designing a packet format that is resistant to a global adversary ensures that the format can be used both in OR and Mix networks. To remain consistent with the related work in the field, we will stick to the term OR in this work. Naturally, the packet format used in an OR or Mix network has no impact on the protocol's security against attacks involving traffic analysis, dropping packets, or replay or duplicate attacks. Thus, we do not consider these in this work.

2.2 Onion Routing Variants

Multiple different variants of OR have been proposed and used in the field. These differ in their network models, assumptions, the security provided, and their functionality:

- *Network model:* Some OR variants work in the *integrated-system* network model, where message receivers are themselves relays [5, 8, 19], while others are based in the *service* model, where receivers are generic servers that are unaware of the OR network and protocol [9, 10, 12, 15, 16, 26]. In the integrated-system model, the final relay on an onion's path is the onion's receiver. On the other hand, in the service model, the final relay is referred to as the *exit relay* and is responsible for forwarding the message to the receiver outside of the OR network. Figure 1 shows a comparison of the network models.
- *Receiver trust:* Most OR protocols do not assume that relays or receivers are trusted by the sender. However, some protocols assume that honest senders will only choose to send messages to honest or semihonest receivers [1].
- *Payload integrity:* For a given OR packet format, we consider an onion to be composed of two parts: A header that contains

the forwarding information for each hop of the onion and a payload that contains the message and other information for the final relay. A packet format can offer either *hop-by-hop integrity* or *end-to-end integrity* for the payload of the onion. With the former, the payload's integrity is verified at each relay. This allows manipulations to be detected as early as at the next honest relay on an onion's path. Several schemes and protocols instead provide end-to-end integrity, where the payload's integrity is only verified at the final relay [8, 9, 26, 27].

- *Replies:* Many variants support anonymous replies in addition to "forward" messages. In order to accomplish this without receivers learning the identity of the sender, senders typically prepare an anonymous return envelope that the receiver can combine with its reply message to create a *reply onion*. Some formalizations require reply onions to be indistinguishable from forward onions to increase the anonymity set of senders [1, 12, 20].

Our goal in this paper is to create a model that covers all of these different variants in one framework for provable security.

2.3 Provably Secure Onion Routing

There are currently multiple frameworks for analyzing provably secure OR schemes both in the face of a global adversary [1, 5, 19, 20, 26] and against a local adversary [2]. These works are all based on the Universal Composability (UC) framework [6]. We focus on the works that assume a global adversary in this paper. In UC, a protocol is proven secure by showing that it is indistinguishable from an ideal functionality $\mathcal{F}$. $\mathcal{F}$ itself is an abstract version of the protocol that is run by a trusted third party. This approach makes it easy to understand the security properties of a protocol by analyzing the simpler ideal functionality instead of the more complicated real protocol.

2.3.1 Existing Formal Analysis. Analyzing OR using UC was first proposed by Camenisch and Lysyanskaya, who construct an ideal functionality and three game-based properties for the integrated-system model without replies. They also provide an OR scheme for this variant of OR. The properties were constructed such that any OR packet scheme that satisfies the property would securely realize the ideal functionality [5]. Camenisch and Lysyanskaya's work, and especially the game-based properties, are the formal foundation for multiple OR protocols [9, 10, 12].

Later, Kuhn et al. found that the game-based properties do not actually imply the ideal functionality. As a consequence, they also discovered vulnerabilities in the protocols that based their security proofs on the properties. They developed new *onion properties* that they showed implied secure realization of the same ideal functionality [19].

Ando and Lysyanskaya introduce an ideal functionality for replicable OR with end-to-end integrity in the integrated-system model along with another game-based property and an OR scheme [1]. Kuhn et al. similarly extend the integrated-system ideal functionality to include replies with hop-by-hop integrity [20]. The onion properties proposed in their work are the base for the properties we construct in this paper. Finally, [26] adapt [20]'s formalization for the service model with end-to-end integrity with the goal of proving security for the Sphinx protocol. We make use of their work to adapt the framework to the new model in our general formalization. Each of these formalizations covers one specific variant of OR models and functionalities, but does not extend further. In particular, many existing protocols operate in OR variants that have not been formalized.

2.3.2 Overview of Analysis Framework. In this work, we construct a generalization of the OR analysis frameworks introduced by [5] and expanded upon by [1, 19, 20, 26]. In the following, we present the common features of these frameworks, including the basic principle behind the ideal functionalities and the onion properties. The adversary model shared by these frameworks is that of a global adversary with statically corrupted relays and receivers and with control over all of the links between machines on the network. We adopt this adversary model for our framework as well.

To begin, consider a repliable onion sent by an honest relay P_s along a path of relays with the final relay P_n being corrupted. Each relay on the forward and reply paths of the onion is either honest or corrupted. We can split these paths at all of the honest relays. This yields multiple *path segments*. Each segment starts and ends at an honest relay except for the last segment on the forward path and the first segment on the reply path: Since P_n is corrupted, these end (or begin, respectively) with a corrupted relay. The path segments are illustrated in Figure 2. The concept of path segments is crucial for our analysis of provably secure OR.

The first place where path segments play an important role is in the OR ideal functionalities. Since an ideal functionality is run entirely on a trusted party, it does not need to build any actual "real onions". Instead, all of the information on sent onions is handled internally. To model the fact that the adversary can see (and influence) onions being sent over the network and via its corrupted relays, the ideal functionality provides the adversary with random temporary identifiers called tids. Each tid is mapped to a path segment of an onion. Since every tid is chosen randomly, the adversary cannot link multiple path segments of an onion together[1].

Since many OR protocols work in a similar way aside from their packet formats, proving that they realize one of the ideal functionalities can be simplified: The analysis frameworks provide game-based onion properties for the OR packet schemes. These are accompanied by a proof that any scheme that satisfies the properties can be used in an OR protocol that realizes the ideal functionality. We give an overview of the onion properties as proposed by Kuhn et al. [20] here. For details on how the properties are equivalent to realizing the ideal functionality, see Section 7.6.

Kuhn et al. construct four onion properties in total. The first of these is *Correctness*, which demands that an onion processes correctly in the absence of an adversary. The remaining three properties are *Layer Unlinkability* (*LU*), *Backwards Layer Unlinkability* ($LU^{\leftarrow}$), and *Tail Indistinguishability* (*TI*). Each provides protection on a different part of an onion's path.

- *LU* is responsible for path segments on the forward path that start and end with honest relays.
- $LU^{\leftarrow}$ is responsible for path segments on the reply path that start and end with honest relays.

[1]This corresponds to the adversary being able to track an onion along a path segment, but not over an honest relay.

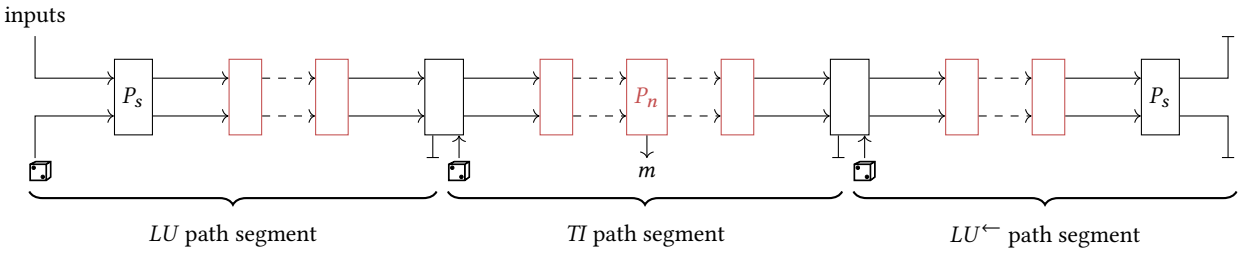

Figure 2: A repliable onion's path can be split into three types of path segments as depicted. Each onion property guarantees security on one type of path segment. In the onion property games, the onion layers on the corresponding path segment (the top arrows) are replaced with random onion layers following the same path segment (the bottom arrows). Honest relays are drawn in black while corrupted relays are drawn in red. Dashed arrows represent omitted relays on the path. P_s is the sender and P_n is the final relay on the onion's forward path. m is the message contained in the onion.

- *TI* is responsible for the last and first path segments of the forward and reply paths respectively if the final relay is corrupted.

The three properties all work analogously: At the beginning of the game, the adversary chooses the parameters for an onion and the boundaries of a path segment for that onion. The challenger builds that onion along with a replacement random onion for that path segment. When the challenger would give the adversary the onion for that chosen path segment, it flips a coin and potentially gives the adversary the random onion instead. Intuitively, if this replacement is indistinguishable for every path segment of an onion, the packets sent on the segments hold no more information than a random identifier *tid* for the adversary. The replacements performed in each property are depicted in Figure 2. Of course, this is exactly what the adversary receives in the ideal functionalities.

2.3.3 Tagging Attacks. In a *tagging* attack, the attacker modifies a part of a packet with the goal of re-identifying the modified packet later. OR packet formats offer different levels of protection against these modification attacks. We distinguish between two levels of integrity protection: The first is *hop-by-hop* integrity, in which modifications are detected immediately at the next honest relay on an onion's path. The second is *end-to-end* integrity, where modifications are only detected at an onion's final relay. A packet format can protect each of a packet's components using either level of integrity. Some formats protect the entire packet with hop-by-hop integrity (e.g., [5, 19, 20]). Other formats only protect the routing information in the packet this way and use end-to-end integrity for their payloads (e.g., [12, 27]).

Packet formats that use end-to-end integrity for their payloads are potentially susceptible to tagging attacks. For example, the attacker can modify the payload of an onion's first layer in flight. The intermediate relays do not notice this modification and forward the onion normally. If the onion's final relay is corrupted, the attacker can observe that the malformed payload fails the integrity check, thus linking the onion's sender to the corrupted final relay. In integrated-system-model variants, the final relay is the receiver, so this attack allows linking honest senders to their chosen receivers if the receivers they contact are corrupted. In service-model variants, the final relay is not the receiver. If the message and receiver address are destroyed in the tagging attack and the sender chose its exit relay at random, then the sender still cannot be linked to its chosen receiver [26].

3 ONION ROUTING VARIANTS

Analyzing the different variants of onion routing settings and requirements used in the related work [1, 5, 12, 19, 20, 26], we identify four main modelling decisions along which the variants differ. Classifying the variants with our model allows us to capture every variant of provably secure onion routing in the related work as well as some currently-unexplored variants.

In the following, we first present each of our dimensions. We then analyze each of the 16 variants on onion routing generated by the different modelling decisions and map the related work to the variants in our model.

3.1 STIR Model

Each letter in the acronym STIR stands for one binary option in constructing the OR variant. When written with a bar (e.g., $\bar{\text{R}}$), a letter stands for the variants "without" that option (e.g., without replies). Without a bar, it stands for the variants "with" the option. The options are as follows:

S: Whether the variant is used as a *service*, i.e., whether the network setting is the *service model* (S) or the *integrated-system model* ($\bar{\text{S}}$). In the integrated-system model, senders and receivers are all themselves relays in the OR network.

T: The *trust* placed in receivers. Some formalizations assume that the communication partners of honest senders are themselves honest and are thus *trusted receivers* [1] (T). Other OR variants explicitly want to protect against *untrusted receivers* [19, 20] ($\bar{\text{T}}$).

I: The *integrity* of onion payloads. Strong formalizations require the integrity of onion payloads to be protected on a *hop-by-hop* basis (I). In some variants, the less costly *end-to-end* integrity ($\bar{\text{I}}$) for payloads is sufficient. We note that the difference in this aspect applies only to the integrity of the payload, not the header of the onion.

Number	S	T	I	R	Used in	Prev. formalized	Notes
V1	$\bar{S}$	$\bar{T}$	I	$\bar{R}$	[5, 19, 22, 23]	Yes [19]	"Baseline" variant
V2	$\bar{S}$	$\bar{T}$	I	R	[20]	Yes [20]	
V3	$\bar{S}$	$\bar{T}$	$\bar{I}$	$\bar{R}$		No	Vulnerable to tagging attacks [19]
V4	$\bar{S}$	$\bar{T}$	$\bar{I}$	R	[8–10, 18, 21, 29][1]	No	Vulnerable to tagging attacks [19]
V5	$\bar{S}$	T	I	$\bar{R}$		No	
V6	$\bar{S}$	T	I	R		No	
V7	$\bar{S}$	T	$\bar{I}$	$\bar{R}$		No	
V8	$\bar{S}$	T	$\bar{I}$	R	[1]	Yes [1]	
V9	S	$\bar{T}$	I	$\bar{R}$	[3][2]	No	
V10	S	$\bar{T}$	I	R	[4, 7, 10, 11, 16, 17, 25][1]	No	
V11	S	$\bar{T}$	$\bar{I}$	$\bar{R}$	[24][3]	No	
V12	S	$\bar{T}$	$\bar{I}$	R	[2, 13, 15, 26–28][1]	Yes [26]	
V13	S	T	I	$\bar{R}$		No	Equivalent to 9
V14	S	T	I	R		No	Equivalent to 10
V15	S	T	$\bar{I}$	$\bar{R}$		No	Equivalent to 11
V16	S	T	$\bar{I}$	R		No	Equivalent to 12

Table 1: Overview of OR variants with references to related work using those variants for OR and mix nets. A variant is considered previously formalized if the related work introduces a formal framework that is generally applicable in the variant.
[1] HORNET and TARANET use Sphinx in the V4 variant in its setup phase and switch to V12 and V10 respectively for data transmission [9, 10].
[2] Beato et al. do not describe their OR model, but their Sphinx variant is suitable for variant V9.
[3] Loopix permits reply messages via add-ons, but does not include them in its core design [24].

R: *Replies.* A formalization may choose to support both forward onions and replies [1, 20] (R) or limit itself to only using forward onions [5, 19] ($\bar{R}$).

We provide an overview of the different OR variants in STIR in Table 1 and give references to the related work where applicable. In the following, we explain the notable features of each OR variants and group those with similar properties into more general categories.

3.1.1 Baseline OR Variants (V1, V9). We choose the integrated-system-model OR variant V1 as our "baseline" variant because it serves to clearly illustrate a basic form of OR. It does not assume that receivers are trusted, does not use replies, and demands hop-by-hop integrity of payloads. The other variants can be interpreted as extensions (through adding replies or making receivers external to the OR network) or relaxations (by assuming that receivers are trusted or that payload integrity need only be verified on an end-to-end basis) of this basic variant. V1 corresponds to the OR variant first proposed by Camenisch and Lysyanskaya [5] and also used by Kuhn et al. [19].

The matching service-model variant V9 behaves very similarly to V1: One can transform an OR scheme in V1 to V9 by simply including an external receiver's address in the onion payloads and changing the final relay's role from that of the receiver to that of the exit relay.

3.1.2 Tagging-Vulnerable Variants (V3, V4). Some variants are vulnerable to tagging attacks by construction. These attacks work in service-model variants with end-to-end integrity protection of payloads, where a modification is not detected until the onion reaches its final relay. The attack works on V3 and V4 as described in Section 2.3.3.

3.1.3 Trusted-Receiver Variants (V5–V8). In these OR variants, we can assume that an honest sender will only communicate with an honest receiver. This assumption allows us to relax some of our security requirements: The final layers of onions (including their decrypted payloads) can leak information about the sender without impacting privacy. Notably, V7 and V8 permit the use of end-to-end integrity without becoming vulnerable to the tagging attack that affects V3 and V4.

3.1.4 Repliable OR Variants (V(2k)). These OR variants extend the functionality available in the baseline variant by adding replies. This addition leads to a more complex formalization since one must ensure privacy for both the forward and the reply paths of an honest sender's onion. In V8, it also causes a vulnerability inherent to the OR variant.

Multiple formalizations for OR with replies exist in the related work, including Kuhn et al.'s work in V2 [20], Ando and Lysyanskaya's formalization for V8 [1], and [26]'s introduction of V12.

3.1.5 Sphinx Variants (V11, V12). V11 and V12 are notable in that they are the service-model counterparts to the vulnerable variants V3 and V4, but do not necessarily suffer from the same vulnerability. This is due to the additional level of indirection introduced by the exit relay. In the integrated-system model, an adversary-modified payload is discovered by the receiver, thereby linking the sender to the chosen receiver. In the service model, the exit relay decrypts the modified payload. If the exit relay is chosen randomly by the sender, this does not expose the link to the receiver as long as the receiver address is destroyed in the modification attack. The onion

properties for these variants are defined in a way that requires the payload to be completely destroyed by such a modification.

We refer to these variants as *Sphinx variants* because V12 was first introduced in [26] to prove the packet format Sphinx secure.

3.1.6 Trusted-Receiver Service-Model Variants (V13–V16). The variants V13 through V16 are the final combinations of the STIR options. They combine the service model with the trusted-receiver assumption. We argue that these variants are equivalent to variants V9–V12 because trusted receivers do not affect the variant in the service model: The adversary is assumed to have full control over the link from the exit relay to the receiver. Since we assume that forward and reply messages are sent in plaintext without integrity protection, the adversary has full access to any receiver communication. As a result, receivers being trustworthy has no benefit for the security of the OR scheme. We omit V13–V16 in the following sections; any results for V9–V12 can be analogously applied to the respective variant.

4 NOTATION AND ASSUMPTIONS

4.1 Notation

In this work, we closely follow the notation introduced by the related work ([19, 20]). In an OR network, we refer to the packets sent between relays as *onion layers* (also shortened to *onions*). An onion layer consists of a header η and a payload δ, where the header contains the forwarding information and the payload holds data including the message m. Onions have *paths* that define which relays process the layers of the onion in which order, denoted as $\mathcal{P} = (P_1, \ldots, P_n)$. In the integrated-system model ($\bar{\text{S}}$), paths end with the *final relay* on the onion's path. In the service model (S), the final relay is also referred to as the *exit relay* and is followed by the external receiver R on the path. In the following sections, differences between variants like the one above are denoted using

X	rounded boxes like this one. The X is one or multiple of the letters in STIR (or their negations) and indicates that the boxes' contents only apply in variants where X holds. Multiple connected boxes are not exclusive.

Public-private keypairs are written as (PK, SK). When discussing layers of an onion O, O_i refers to the i-th layer, which is produced by P_{i-1} and sent to P_i.

R	Parameters for reply onions are marked with a backwards arrow: $\square^{\leftarrow}$.

4.2 Adversary Model

Throughout this paper, we assume that the adversary has global control over all of the links in the network as well as a statically corrupted set of relays and receivers. The adversary may actively manipulate packets. We do not consider traffic analysis attacks in this work since we focus on providing tools for packet formats.

4.3 Assumptions

The assumptions we make in this paper are inherited from the related work, including [19, 20], and [26]. We quote [26]'s phrasings in the following. These assumptions serve to make handling the formal protocols more easily tractable and removing edge cases.

Assumption 1. *A maximum path length (number of relays on the path) of N is used (inclusive upper bound) [5].*

Assumption 2. *Honest senders choose acyclic paths (to increase the chance of picking at least one honest relay) [20].*

Assumption 3. *A sender always knows the public keys PK_i of any relays P_i it uses for its onion's paths [5].*

Assumption 4. *An onion O consists of a header η and a payload δ such that $O = (\eta, \delta)$ [1].*

Assumption 5. *Honest relays inside the OR network communicate with each other via secure channels.*

S	*Relays and external receivers do not communicate via secure channels [26].*

The choice of a secure transport protocol warrants discussion. For instance, practical OR protocols in the service model may choose to secure the channels between relays and receivers (e.g., via TLS). We do not assume so for two reasons: First, we prevent loss of generality this way. Second, securing these channels does not generally improve our privacy or integrity guarantees against a global adversary. Since relays may be corrupted, the adversary could read or manipulate messages at the exit relay even if the channel to the receiver is secured. For a discussion of the use of TLS between senders and receivers, see Section 8.

S	Assumption 6. *Honest external receivers cannot process onions sent to them and thus drop them. Similarly, relays do not expect any onions on channels to external receivers and drop those as well [26].*

Assumption 7. *An onion [...] never has an empty forward path.*

R	*If it is repliable, it does not have an empty [reply] path [26].*

This assumption serves to prevent a useless edge case. In repliable variants, we use an empty reply path as a sentinel value for a non-repliable onion.

R	Assumption 8. *An honest relay will always drop an unsolicited reply (i.e., a reply with a header the relay does not recognize as belonging to the final reply [onion] layer of an onion it sent) [26].* The relay would not be able to map this reply to any communication, so discarding it is the obvious course of action.

5 OR SCHEMES

This section defines OR schemes for each STIR OR variant as tuples of efficient algorithms. These definitions were originally proposed by Camenisch and Lysyanskaya [5] and Ando and Lysyanskaya [1]. We inherit [26]'s presentation of the algorithms and modify them to suit the different OR variants. An OR scheme in a STIR variant is built using the following algorithms:

- G, a key generation algorithm:
$$(PK, SK) \leftarrow G(1^\lambda, p, P)$$
where $P \in \mathcal{N}$ is a relay name and p is the set of public parameters.

- FormOnion, used by senders to build onions:

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

$\bar{\text{S}}$	with $\mathcal{P}^{(\leftarrow)} = (P_1^{(\leftarrow)}, \ldots, P_{n^{(\leftarrow)}}^{(\leftarrow)})$,
S	with $\mathcal{P}^{(\leftarrow)} = (P_1^{(\leftarrow)}, \ldots, P_{n^{(\leftarrow)}}^{(\leftarrow)}, R)$,

$$PK_{\mathcal{P}^{(\leftarrow)}} = (PK_1^{(\leftarrow)}, \ldots, PK_{n^{(\leftarrow)}}^{(\leftarrow)}).$$

i is the desired onion layer index ($i > 1$ is only required in our onion properties, not in practice). If $i > n$, the $(i - n)$-th layer of the reply onion is output instead. $\mathcal{R}$ is the randomness for the algorithm.

R	To build a non-repliable onion, set $\mathcal{P}^{\leftarrow} = ()$.

- ProcOnion, used by relays to process onions:

$$(O', P') \leftarrow \text{ProcOnion}(SK, O, P),$$

where SK is the relay P's secret key and O is the onion layer to process. O' is the next layer and P' is the next address on the onion's path. In case of an error, $(O', P') = (\perp, \perp)$. When processing a final forward onion layer,

$\bar{\text{S}}$	$(m, \perp) \leftarrow \text{ProcOnion}(SK, O_n, P_n)$.
S	$(m, R) \leftarrow \text{ProcOnion}(SK, O_n, P_n)$.
R	$(m^{\leftarrow}, \perp) \leftarrow \text{ProcOnion}(SK, O_{n+n^{\leftarrow}}, P_{n^{\leftarrow}}^{\leftarrow})$

R	• FormReply, used by the final relay to build a reply onion: $(O^{\leftarrow}, P^{\leftarrow}) \leftarrow \text{FormReply}(m^{\leftarrow}, O, P, SK)$, where $m^{\leftarrow}$ is the reply message, O is the final layer of the forward onion, and SK is the final relay P's secret key.

The following algorithms are only required for our onion property definitions and proofs. For these definitions, recall that we consider an onion $O = (\eta, \delta)$ to consist of a header η that contains the forwarding information and a payload δ that holds the message along with any associated data.

- RecognizeOnion (abbrev. ROnion), used in definitions to identify layers of a known onion:

$\bar{\text{R}}$	$b \leftarrow \text{ROnion}(i, O, \mathcal{R}, m, \mathcal{P}, PK_{\mathcal{P}})$.
R	$b \leftarrow \text{ROnion}(i, O, \mathcal{R}, m, \mathcal{P}, PK_{\mathcal{P}}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}^{\leftarrow}})$.

If the header of O matches the header of the i-th onion layer built by FormOnion given these parameters, then $b = \textit{true}$. Otherwise, $b = \textit{false}$. Note that O's payload is deliberately ignored by ROnion [20].

$\bar{\text{I}}$	• TagOnion, used to reproduce the effects of tagging onion payloads: $\overline{O} \leftarrow \text{TagOnion}(O, \delta', O')$, TagOnion is used by the challenger in one of our onion properties when the adversary modifies an onion's payload (thus tagging it) and submits the onion with the correct header, but a tagged payload to the challenger. O is the unmodified onion the challenger "expected" to receive, δ' is the tagged payload of the actually-received onion, and O' is another onion.

TagOnion should modify O''s payload such that the effect of the modification is indistinguishable from the effect of the modification applied to δ'. For example, if the payload is encrypted using a PRP, any modification leads to the payload decrypting to a random bitstring. In this case, TagOnion could simply replace O''s payload with any random bitstring for an indistinguishable effect [26]. This definition is similar to one originally proposed by Ando and Lysyanskaya [1].

Definition 1

An OR scheme is a tuple of PPT algorithms (G, FormOnion, ProcOnion, (R | FormReply), ROnion, ($\bar{\text{I}}$ | TagOnion)).

An OR scheme can be turned into an OR protocol in a straightforward manner: After key generation, a sender chooses the parameters for its onion and builds it with FormOnion. The onion then traverses the network as each relay runs ProcOnion on the onion layer it receives and forwards the result. The final relay recovers the payload contents and uses FormReply to build the reply onion, which is forwarded back to the sender again.

S	The final relay forwards the message to the receiver and gets the reply message in response.

6 IDEAL FUNCTIONALITY

In this section, we describe our generalized ideal functionality $\mathcal{F}$ and the security guarantees it provides. $\mathcal{F}$ adapts to every STIR OR variant and is itself an adaptation and restructuring of the ideal functionalities for OR presented in the related work by Camenisch and Lysyanskaya [5], Kuhn et al. [19], and [26]. The full version of $\mathcal{F}$ is given in Appendix A in pseudocode. Differences between variants are denoted using the STIR letters.

As explained previously, the core concept behind the ideal functionalities for OR in both our work and the related work is the introduction of temporary identifiers (referred to as *tid*s) as an abstract representation of a sequence of onion layers on one of an onion's path segments. While the trusted IF holds all of the information on the onion itself, the only information the adversary (or simulator) receives on the onion as it moves through the network are the random *tid*s associated with each path segment along with other information that an OR protocol in a given OR variant would leak to the adversary.

The exact information provided to the adversary by $\mathcal{F}$ varies depending on the OR variant and determines the level of privacy that can be afforded by an OR protocol in that variant. In the following, we explain the privacy guarantees that are provided by $\mathcal{F}$ in every variant and detail which information is leaked to the adversary.

Fundamentally, $\mathcal{F}$ only provides privacy to its senders if at least two onions from honest senders share the same honest relay on their paths. In that case, the adversary cannot determine which of the two *tid*s "after" the honest relay belongs to which of the *tid*s "before" the honest relay and thus cannot link an honest sender to the message it sent or to its receiver [19].

R	The same logic protects onions on the reply path.

$\mathcal{F}$ does not protect against timing attacks, replay and duplicate attacks, traffic analysis, or attacks that involve dropping all but one of the honest onions that reach an honest relay. Protecting against these attacks is out of scope for an IF and typically handled by the protocol implementation instead of the OR scheme [19].

In every variant, $\mathcal{F}$ leaks *tid*s and the path segments they belong to the adversary. This is unavoidable since an attacker can always link onion layers that it processed itself.

$\bar{\text{S}}$	$\mathcal{F}$ also keeps messages from honest senders secret from the adversary if the receiver is honest.
S	The adversary learns message-receiver combinations for every onion since these are sent in plaintext over the link from the exit relay to the receiver.
R	Received reply messages are not part of the honest parties' output to the environment in $\mathcal{F}$. If they were part of the output, the adversary could learn which honest reply receiver received which message by asking the environment, allowing it to link the reply receiver (the honest sender) to the reply sender (the receiver) [26]. To ensure that $\mathcal{F}$ still provides privacy to senders, we thus do not provide the reply message output. This limitation is instructive for the design of protocols using OR and is discussed in more depth in Section 8.
$\bar{\text{I}}$	In these variants, the attacker can tag a set of *tid*s while they are in flight. When the onions these *tid*s belong to reach their last path segment, the adversary is informed that a tagged onion was received if the last relay on the path is corrupted [26]. In some variants, this allows linking senders and receivers (see Section 3), so $\mathcal{F}$ does not protect senders in these variants.

7 GENERALIZED ONION PROPERTIES

In this section, we present generalized versions of the four onion properties previously introduced by Kuhn et al. [20]. For each OR variant, satisfying a particular subset of these properties is sufficient to securely realize the corresponding ideal functionality $\mathcal{F}$[2]. We first provide a general overview of the properties and explain which of them are required to securely realize $\mathcal{F}$ in one of our variants. Afterwards, we discuss each property in turn.

First, *Correctness* ensures that the scheme operates correctly in the absence of an adversary. Onion path segments that begin and end with an honest relay on the forward path are handled by the next property, called *Layer Unlinkability* (*LU*). Similarly, the segments that begin and end with an honest relay on the reply path are covered by *Backwards Layer Unlinkability* ($LU^{\leftarrow}$). Finally, *Tail Indistinguishability* (*TI*) is responsible for the segments at the end of the forward path and the beginning of the reply path if P_n is corrupted.

7.1 Required Properties

Only some of the OR variants defined in Section 3 require all three of the onion properties. Which property is required in which variants is shown in the following:

All: *Correctness*, *LU*. Every OR variant requires *Correctness* and features forward path segments that begin and end with honest relays.

R: $LU^{\leftarrow}$. Analogously to the previous point, any variant that supports replies has reply path segments that begin and end with honest relays.

$\bar{\text{T}}$ or S: *TI*. If an integrated-system-model OR variant does not trust sender-chosen receivers, *TI* is necessary to protect onions on path segments that end at a corrupted receiver. As described in Section 3.1.6, we assume exit relays in the service model to be untrusted in general. This means that the path segments containing the exit relay require *TI* for protection.

The definitions of the onion properties also change depending on the variant they are used in. These changes are discussed in detail in the following.

7.2 Correctness

Correctness requires an OR scheme to behave as expected in the absence of an adversary. Essentially, this means that processing an onion with ProcOnion leads to the same sequence of relays as in the forward (and reply) paths the onion was created with. Processing the onion at the final relay must result in the correct message and, in the service model, the correct receiver as well. The following definition is adapted from [26]'s version of Correctness.

Definition 2 (*Correctness*)
Correctness is defined as:

Let (G, FormOnion, ProcOnion, (R | FormReply), RecognizeOnion, TagOnion) be an OR scheme with maximum path length N and polynomial $|\mathcal{N}|$ and $|D|$. Then for all $n < N, \lambda \in \mathbb{N}$, all choices of the public parameter p, all choices of randomness $\mathcal{R}$, all messages m, all keypairs $(PK_i^{(\leftarrow)}, SK_i^{(\leftarrow)})$ generated by $G(1^\lambda, p, P_i^{(\leftarrow)})$, and all choices of internal randomness used to run FormOnion and ProcOnion, the following needs to hold:

Correctness of forward path.

$\bar{\text{S}}$	For all choices of $\mathcal{P} = (P_1, \ldots, P_n)$,
S	For all choices of $\mathcal{P} = (P_1, \ldots, P_n, R)$,

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

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

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

Correctness of request reception.

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

R	**Correctness of backward path.** For all choices of $n^{\leftarrow} < N$, $m^{\leftarrow}$, reply path $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_n^{\leftarrow})$, and randomness in FormReply, $Q_i^{\leftarrow} = P_i^{\leftarrow}$ for $1 \le i \le n$ and $(O_1^{\leftarrow}, Q_1^{\leftarrow}) \leftarrow \text{FormReply}(m^{\leftarrow}, O_n, P_n, SK_n)$, $(O_{i+1}^{\leftarrow}, Q_{i+1}^{\leftarrow}) \leftarrow \text{ProcOnion}(SK_i^{\leftarrow}, O_i^{\leftarrow}, Q_i^{\leftarrow})$. **Correctness of reply reception.** $(m^{\leftarrow}, \perp) = \text{ProcOnion}(SK_{n^{\leftarrow}}^{\leftarrow}, O_{n^{\leftarrow}}^{\leftarrow}, P_{n^{\leftarrow}}^{\leftarrow})$.

[2]Note that protocols in the variants vulnerable to tagging attacks remain vulnerable even if the properties are satisfied.

The correctness of RecognizeOnion and TagOnion follows from the following security properties, where they are required.

7.3 Forwards Layer Unlinkability (*LU*)

LU is the onion property that ensures that onion layers on a path segment between two honest relays on the forward path are indistinguishable from random onion layers that only follow that segment. It is used in every OR variant we discuss. We use Kuhn et al.'s formulation of this property [20] and incorporate changes from [26] to accomodate the variants handled in those works. Merging these variants into a single property generates new combinations that we consider in our constructions and proofs.

LU works as follows: The challenger holds two keypairs that correspond to the honest sender relay and another honest relay. The adversary is given oracle access to these honest relays. It then submits parameters to the challenger for the honest sender relay to build an onion. The challenger uses these to make the first layer O_1 of the adversary-chosen onion. The challenger also builds a second "replacement" onion $\overline{O}_1$ that uses only the first path segment of the adversary's chosen path. The challenger then flips a coin and outputs the corresponding onion to the adversary. After this, the adversary gets oracle access again and can submit its challenge onion to the honest relay's oracle. If the onion the adversary received was O_1, it is processed as usual. However, if the adversary received $\overline{O}_1$ and the submitted onion is recognized as the processed replacement onion by the challenger, the oracle responds with O_c. O_c is built with the adversary's original inputs except that the first path segment is cut from the front of the forward path. The adversary must distinguish between the two different scenarios to win the *LU* game.

The *LU* game changes depending on the OR variant. These changes are indicated using rounded boxes in the definition. The letter in the left cell of the box shows which variants the contents of the box apply to. We also summarize the differences here: In service-model variants, paths end in receivers, which are handled accordingly. Repliable variants add reply oracles in addition to the normal processing oracles and feature an additional case if the first path segment consists of the entire forward path. Variants without payload integrity change how the replacement challenge onion is detected in the oracle during the second round of oracle access: If the payload has not been manipulated, processing proceeds as normal. If the payload was tagged, the payload of O_c is also tagged, the payload contents (i.e., the message and, in the service model, the receiver) are replaced with randomness, and O_c becomes non-repliable. The content replacement ensures that any protocol that satisfies *LU* prevents tagging attacks that do not destroy the payload contents.

Definition 3 (*Forwards Layer Unlinkability*)
LU is defined as:

(1) The adversary receives the honest 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** requests for P_H or P_s to the challenger. When asked for **Proc**$(P_H, O = (\eta, \delta))$, the challenger checks whether the header η is on the η_H-list. If it is not on the list, it sends the output of ProcOnion(SK_H, O, P_H) to the adversary, stores η on the η_H-list and O on the O_H-list.

R	The adversary may also submit **Reply** requests. For any **Reply**(P_H, O, m), the challenger checks if O is on the O_H-list and if so, the challenger sends FormReply(m, O, P_H, SK_H) to the adversary.

(Similar for requests to P_s with the η_S and O_S-list).

(3) The adversary submits
- a message m,
- a position j with $1 \le j \le n$,
-

S	a path $\mathcal{P} = (P_1, \ldots, P_n, R)$ with $P_j = P_H$ and a receiver R,
S̄	a path $\mathcal{P} = (P_1, \ldots, P_j, \ldots, P_n)$ with $P_j = P_H$,

-

R	a path $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{n^{\leftarrow}}^{\leftarrow} = P_s)$,

- and public keys for all nodes PK_i ($1 \le i \le n$ for the nodes on the path and $n < i$ for the other relays).

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

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

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

and a replacement onion

S	with the first part of the forward path and a random receiver $\overline{R}$: $\overline{\mathcal{P}} = (P_1, \ldots, P_j, \overline{R})$,
S̄	with the first part of the forward path $\overline{\mathcal{P}} = (P_1, \ldots, P_j)$,

a random message $\overline{m} \in M$, honestly chosen randomness $\overline{\mathcal{R}}$.

$$\overline{O}_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, \overline{m}, \overline{\mathcal{P}}, PK_{\overline{\mathcal{P}}}).$$

R	$O_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, m, \mathcal{P}, PK_{\mathcal{P}}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}^{\leftarrow}})$, $\overline{O}_1 \leftarrow \text{FormOnion}(1, \mathcal{R}, \overline{m}, \overline{\mathcal{P}}, PK_{\overline{\mathcal{P}}}, \overline{\mathcal{P}}^{\leftarrow}, PK_{\overline{\mathcal{P}}^{\leftarrow}})$, with $\overline{\mathcal{P}}^{\leftarrow} = ()$ being an empty reply path.

(6) If $b = 0$, the challenger outputs O_1 to the adversary. Otherwise, the challenger outputs $\overline{O}_1$ 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 those where:
- If $j < n$:
  - **Proc**$(P_H, O = (\eta, \delta))$ with
    ROnion$(j, O, \overline{\mathcal{R}}, \overline{m}, \overline{\mathcal{P}}, PK_{\overline{\mathcal{P}}}) = True$,
    (Ī | and the unmodified payload $\delta = \delta_j$,) η not in η_H and ProcOnion$(SK_H, O, P_H) \neq (\perp, \perp)$:
    The challenger outputs (P_{j+1}, O_c) with
    $O_c \leftarrow \text{FormOnion}(1, \widetilde{\mathcal{R}}, m, \widetilde{\mathcal{P}}, PK_{\widetilde{\mathcal{P}}})$,
    honestly chosen randomness $\widetilde{\mathcal{R}}$, and forward path $\widetilde{\mathcal{P}} = (P_{j+1}, \ldots, P_n)$ and adds η to the η_H-list and O to the O_H-list.

S	$\widetilde{\mathcal{P}} = (P_{j+1}, \ldots, P_n, R)$.
R	Include the reply paths in the algorithms: $\text{RONION}(j, O, \overline{\mathcal{R}}, \overline{m}, \overline{\mathcal{P}}, PK_{\overline{\mathcal{P}}}, \overline{\mathcal{P}}^{\leftarrow}, PK_{\overline{\mathcal{P}}\leftarrow}) = True,$ $O_c \leftarrow \text{FORMONION}(1, \widetilde{\mathcal{R}}, m, \widetilde{\mathcal{P}}, PK_{\widetilde{\mathcal{P}}}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}\leftarrow}).$

Ī	– **Proc**$(P_H, O = (\eta, \delta))$ with $\text{RONION}(j, O, \overline{\mathcal{R}}, \overline{m}, \overline{\mathcal{P}}, PK_{\overline{\mathcal{P}}}) = True,$ *but the incorrect payload* $\delta = \delta'$, η is not on the η_H-list and $\text{PROCONION}(SK_H, O, P_H) \neq (\perp, \perp)$: The challenger outputs $(P_{j+1}, \overline{O}_c)$ with $O_c \leftarrow \text{FORMONION}(1, \widetilde{\mathcal{R}}, \widetilde{m}, \widetilde{\mathcal{P}}, PK_{\widetilde{\mathcal{P}}}),$ honestly chosen randomness $\widetilde{\mathcal{R}}$, $\widetilde{\mathcal{P}} = (P_{j+1}, \ldots, P_n)$, a freshly chosen random message $\widetilde{m}$, and with $\overline{O}_c \leftarrow \text{TAGONION}(\overline{O}_j, \delta', O_c)$. The challenger adds η to the η_H-list and O to the O_H-list.

S	$\widetilde{\mathcal{P}} = (P_{j+1}, \ldots, P_n, \widetilde{R})$ with a random $\widetilde{R}$.
R	Include the reply path $\overline{\mathcal{P}}^{\leftarrow}$ in RONION and an empty reply path in FORMONION: $\text{RONION}(j, O, \overline{\mathcal{R}}, \overline{m}, \overline{\mathcal{P}}, PK_{\overline{\mathcal{P}}}, \overline{\mathcal{P}}^{\leftarrow}, PK_{\overline{\mathcal{P}}\leftarrow}) = True,$ $O_c \leftarrow \text{FORMONION}(1, \widetilde{\mathcal{R}}, \widetilde{m}, \widetilde{\mathcal{P}}, PK_{\widetilde{\mathcal{P}}}, (), ()).$

- If $j = n$:
  - **Proc**$(P_H, O = (\eta, \delta))$ with
    $\text{RONION}(j, O, \overline{\mathcal{R}}, \overline{m}, \overline{\mathcal{P}}, PK_{\overline{\mathcal{P}}}) = True,$
    η is not on the η_H-list and
    $\text{PROCONION}(SK_H, O, P_H) \neq (\perp, \perp)$:
    The challenger outputs $(m, \perp)$ and adds η to the η_H-list and O to the O_H-list.

R	Include the corresponding reply path in the algorithm: $\text{RONION}(j, O, \overline{\mathcal{R}}, \overline{m}, \overline{\mathcal{P}}, PK_{\overline{\mathcal{P}}}, \overline{\mathcal{P}}^{\leftarrow}, PK_{\overline{\mathcal{P}}\leftarrow}) = True,$

R	– **Reply**$(P_H, O, m^{\leftarrow})$ with $\text{RONION}(j, O, \overline{\mathcal{R}}, \overline{m}, \overline{\mathcal{P}}, \overline{\mathcal{P}}^{\leftarrow}, PK_{\overline{\mathcal{P}}}, PK_{\overline{\mathcal{P}}\leftarrow}) = True,$ O is on the O_H-list and has not been replied before and $\text{FORMREPLY}(m^{\leftarrow}, O, P_H, SK_H) \neq (\perp, \perp)$: The challenger outputs $(O_c, P_1^{\leftarrow})$ with $O_c \leftarrow \text{FORMONION}(j+1, \widetilde{\mathcal{R}}, m^{\leftarrow}, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}\leftarrow})$ and honestly chosen randomness $\widetilde{\mathcal{R}}$.

(8) The adversary produces guess b'.

LU is achieved if any PPT adversary $\mathcal{A}$ cannot guess $b' = b$ with a probability non-negligibly better than $\frac{1}{2}$ [20].

7.4 Backwards Layer Unlinkability ($LU^{\leftarrow}$)

$LU^{\leftarrow}$ ensures that onion layers on a path segment between two honest relays on the reply path are indistinguishable from random *forward* onion layers that only follow that segment. Accordingly, it is only used for repliable OR variants. Like with LU, we base our version of $LU^{\leftarrow}$ on Kuhn et al.'s version [20] with changes from [26]. This merger leads to the creation of new combinations, which we discuss in our proofs.

$LU^{\leftarrow}$ works similarly to LU, but the onion replacement takes place on the reply path. After getting oracle access to the honest relays, the adversary submits its parameters for the first forward layer O_1 of the adversary-chosen onion. The challenger then chooses a random bit b. If $b = 0$, O_1 is built and given to the adversary with the correct parameters. If $b = 1$, O_1's reply path is truncated at the end, removing the last path segment between the honest relay and the honest sender. The adversary can now access the oracles again. If the adversary received the onion with the truncated path, the honest relay oracle operates differently when recognizing the challenge onion: A new random onion $\overline{O}_1$ that only uses the last path segment to the honest sender is given to the adversary. If the challenge onion is submitted to the honest sender, it outputs nothing in both scenarios.

The $LU^{\leftarrow}$ game only changes in service-model variants: In those, receivers are included in the paths and handled accordingly. Note that, unlike LU, $LU^{\leftarrow}$ does not require special logic for variants without payload integrity. This is because the honest sender does not output any messages or error symbols for challenge onions, so manipulated payloads are simply discarded.

Definition 4 (*Backwards Layer Unlinkability*)
$LU^{\leftarrow}$ is defined as:

(1) The adversary receives the honest 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**$(P_H, O = (\eta, \delta))$, the challenger checks whether the header η is on the η_H-list. If it is not on the list, it sends $\text{PROCONION}(SK_H, O, P_H)$, stores η on the η_H-list and O on the O_H-list. For any request **Reply**(P_H, O, m), the challenger checks if O is on the O_H-list. If so, the challenger sends $\text{FORMREPLY}(m, O, P_H, SK_H)$ to the adversary. (Similar for requests on P_s with the η_S-list).
(3) The adversary submits
   - a message m,
   - a position $j^{\leftarrow}$ with $0 \leq j^{\leftarrow} \leq n^{\leftarrow}$,
S	a path $\mathcal{P} = (P_1, \ldots, P_n, R)$ with a receiver R and where $P_{n+1} = P_H$, if $j^{\leftarrow} = 0$,
S̄	a path $\mathcal{P} = (P_1, \ldots, P_n)$ where $P_n = P_H$, if $j^{\leftarrow} = 0$,
   - a path $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{j^{\leftarrow}}^{\leftarrow}, \ldots, P_{n^{\leftarrow}}^{\leftarrow} = P_s)$ with the honest node P_H at backward position $j^{\leftarrow}$ if $1 \leq j^{\leftarrow} \leq n^{\leftarrow}$ and the second honest node P_s at position $n^{\leftarrow}$,
   - and public keys for all nodes PK_i ($1 \leq i \leq n$ for the nodes on the path and $n < i$ for the other relays).
(4) The challenger checks that the chosen paths are acyclic, the router names are valid and that the same key is chosen if the router names are equal, and if so, sets $PK_{j^{\leftarrow}}^{\leftarrow} = PK_H$ (resp. PK_{n+1} if $j^{\leftarrow} = 0$), $PK_{n^{\leftarrow}+1}^{\leftarrow} = PK_S$ and sets bit b at random.
(5) The challenger creates the challenge onion:
   - $b = 0$: The challenger creates the onion with the adversary's input choice and honestly chosen randomness $\mathcal{R}$:
     $O_1 \leftarrow \text{FORMONION}(1, \mathcal{R}, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}\leftarrow})$
     and sends O_1 to the adversary.

- $b = 1$: The challenger creates an onion with the adversary's inputs, but a reply path $\mathcal{P}^{\leftarrow} := (P_1^{\leftarrow}, \ldots, P_{j^{\leftarrow}}^{\leftarrow})$ that is truncated after relay $P_{j^{\leftarrow}}^{\leftarrow}$, and honestly chosen randomness $\mathcal{R}$:
  $O_1 \leftarrow \textsc{FormOnion}(1, \mathcal{R}, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$

(6) The adversary gets oracle access as in step 2) except for those where:
  (a) The request is…
  - for $j^{\leftarrow} > 0$: **Proc**$(P_H, O = (\eta, \delta))$ with
    $\textsc{ROnion}(n + j^{\leftarrow}, O, \mathcal{R}, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}) = True$,
    η is not on the η_H-list and
    $\textsc{ProcOnion}(SK_H, O, P_H) \neq (\perp, \perp)$:
    stores η on the η_H and O on the O_H-list and …
  - for $j^{\leftarrow} = 0$: **Reply**$(P_H, O, m^{\leftarrow})$ with
    $\textsc{ROnion}(n, O, \mathcal{R}, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}) = True$,
    O is on the O_H-list and no onion with this η has been replied to before and
    $\textsc{FormReply}(m^{\leftarrow}, O, P_H, SK_H) \neq (\perp, \perp)$:
    …then: The challenger picks the rest of the return path

S	$\overline{\mathcal{P}} = (P_{j^{\leftarrow}+1}^{\leftarrow}, \ldots, P_{n^{\leftarrow}+1}^{\leftarrow}, \overline{R})$ with an honestly chosen random receiver $\overline{R}$,
$\bar{\text{S}}$	$\overline{\mathcal{P}} = (P_{j^{\leftarrow}+1}^{\leftarrow}, \ldots, P_{n^{\leftarrow}+1}^{\leftarrow})$,

    an empty backward path $\overline{\mathcal{P}}^{\leftarrow} = ()$, a random message $\overline{m}$, and honestly chosen randomness $\overline{\mathcal{R}}$ and generates:
    $\overline{O}_1 \leftarrow \textsc{FormOnion}(1, \overline{\mathcal{R}}, \overline{m}, \overline{\mathcal{P}}, \overline{\mathcal{P}}^{\leftarrow}, PK_{\overline{\mathcal{P}}}, PK_{\overline{\mathcal{P}}^{\leftarrow}})$.
  - If $b = 0$, the challenger calculates

$$(O_{j^{\leftarrow}+1}, P_{j^{\leftarrow}+1}^{\leftarrow}) = \begin{cases} \textsc{ProcOnion}(SK_H, O, P_{j^{\leftarrow}}^{\leftarrow}) & \text{for } j^{\leftarrow} > 0, \\ \textsc{FormReply}(m^{\leftarrow}, O, P_{j^{\leftarrow}}^{\leftarrow}, SK_H) & \text{for } j^{\leftarrow} = 0 \end{cases}$$

    and outputs $O_{j^{\leftarrow}+1}$ for $P_{j^{\leftarrow}+1}^{\leftarrow}$ to the adversary.
  - Otherwise, the challenger outputs $\overline{O}_1$ for $P_{j^{\leftarrow}+1}^{\leftarrow}$ to the adversary.
  (b) **Proc**(P_s, O) with O being the challenge onion as processed for the final receiver on the backward path, i.e.:
  - for $b = 0$:
    $\textsc{ROnion}(n + n^{\leftarrow}, O, \mathcal{R}, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}) = True$
  - for $b = 1$:
    $\textsc{ROnion}(n^{\leftarrow} - j^{\leftarrow}, O, \overline{\mathcal{R}}, \overline{m}, \overline{\mathcal{P}}, \overline{\mathcal{P}}^{\leftarrow}, PK_{\overline{\mathcal{P}}}, PK_{\overline{\mathcal{P}}^{\leftarrow}}) = True$
    …then the challenger outputs nothing.

(7) The adversary produces guess b'.

$LU^{\leftarrow}$ is achieved if any PPT adversary $\mathcal{A}$ cannot guess $b' = b$ with a probability non-negligibly better than $\frac{1}{2}$ [20].

7.5 Tail Indistinguishability (*TI*)

TI is the final onion property. It handles onion layers on path segments between an honest relay and the final corrupted relay on the forward path as well as between a corrupted replying relay and the first honest relay on the reply path. For these layers, *TI* ensures that they are indistinguishable from random onion layers that only follow those path segments and have the same message. Thus, *TI* is used for OR variants that do not trust the receivers chosen by honest senders or that are in the service model, where exit relays are generally untrusted.

In repliable variants, *TI* operates as follows: The challenger controls three keypairs. These model the honest sender, an honest relay on the forward path, and an honest relay on the reply path. The adversary is given oracle access to these relays and submits the parameters for its adversary-chosen O_1. The challenger now chooses a random bit b. If $b = 0$, the challenger uses the parameters to build the onion layer that would have been output by the honest relay on the forward path in normal processing and outputs it to the adversary. If $b = 1$, the adversary instead receives a random onion that uses the path segment between the honest relay and the corrupted receiver as its forward path. The random onion also includes the first path segment on O_1's reply path as its reply path. The adversary is given oracle access again and must determine which scenario it is in. The adversary may submit the challenge onion to the oracle for the honest relay on the reply path. In that case, the oracle outputs nothing.

In non-repliable variants, there is no reply path and thus no second honest relay. The random onion is accordingly built without a reply path as well. No special oracle handling is added in non-repliable variants since there is no honest relay left on the path of the challenge onion.

As discussed above, *TI* adds a second honest relay on the reply path for repliable variants. In service-model variants, receivers are added to the paths accordingly and the random onion's receiver is the same as that of the adversary-chosen onion. *TI* does not require modifications in variants without payload integrity since there is no oracle processing that outputs a challenge onion and would thus have to account for it.

Definition 5 (*Tail Indistinguishability*)
TI is defined as:

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

R	The adversary also receives the router name $P_H^{\leftarrow}$ with the public key $PK_H^{\leftarrow}$, where $(PK_H^{\leftarrow}, SK_H^{\leftarrow}) \leftarrow G(1^{\lambda}, p, P_H^{\leftarrow})$.

(2) Oracle access: The adversary may submit any number of **Proc** requests for P_H or P_s to the challenger. When asked for **Proc**$(P_H, O = (\eta, \delta))$, the challenger checks whether the header η is on the η_H-list. If not, it sends the output of $\textsc{ProcOnion}(SK_H, O, P_H)$, stores η on the η_H-list and O on the O_H-list.

R	The adversary may also submit **Reply** requests and $P_H^{\leftarrow}$ is also available as an oracle. For any **Reply**(P_H, O, m), the challenger checks if O is on the O_H-list and if so, the challenger sends $\textsc{FormReply}(m, O, P_H, SK_H)$ to the adversary.

(Similar for requests to $\boxed{\text{R} \mid P_H^{\leftarrow}}, P_s$).

(3) The adversary submits
- a message m,
-

S	a path $\mathcal{P} = (P_1, \ldots, P_j, \ldots, P_n, R)$ with a receiver R,
$\bar{\text{S}}$	a path $\mathcal{P} = (P_1, \ldots, P_j, \ldots, P_n)$

with the honest node P_H (R | or $P_H^{\leftarrow}$) at position $j, 1 \leq j < n$,

R	• a path $\mathcal{P}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{n^{\leftarrow}}^{\leftarrow})$ with the honest node $P_H^{\leftarrow}$ at position $1 \leq j^{\leftarrow} \leq n^{\leftarrow} + 1$

- and public keys for all nodes PK_i ($1 \leq i \leq n$ for the nodes on the path and $n < i$ for the other relays).

(4) The challenger checks that the given paths are acyclic, the router names are valid and that the same key is chosen if the router names are equal, and if so, sets $PK_j = PK_H$

R	(or $PK_j = PK_H^{\leftarrow}$, if the adversary chose $P_H^{\leftarrow}$ at this position as well), $PK_{j^{\leftarrow}}^{\leftarrow} = PK_H^{\leftarrow}$, $PK_{n^{\leftarrow}}^{\leftarrow} = PK_S$

and sets bit b at random.

(5) The challenger creates the onion with the adversary's input choice and honestly chosen randomness $\mathcal{R}$:
$O_{j+1} \leftarrow \textsc{FormOnion}(j+1, \mathcal{R}, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}})$
and a replacement onion with the path from the honest relay P_H to the corrupted final relay $\overline{\mathcal{P}} = (P_{j+1}, \ldots, P_n)$, honestly chosen randomness $\overline{\mathcal{R}}$

R	and the backward path from the corrupted final relay starting at position 0 ending at $j^{\leftarrow}$: $\overline{\mathcal{P}}^{\leftarrow} = (P_1^{\leftarrow}, \ldots, P_{j^{\leftarrow}}^{\leftarrow})$:

$\overline{O}_1 \leftarrow \textsc{FormOnion}(1, \overline{\mathcal{R}}, m, \overline{\mathcal{P}}, PK_{\overline{\mathcal{P}}})$.

S	$\overline{\mathcal{P}} = (P_{j+1}, \ldots, P_n, R)$.
R	Include the corresponding reply path in the algorithms: $\overline{O}_1 \leftarrow \textsc{FormOnion}(1, \overline{\mathcal{R}}, m, \overline{\mathcal{P}}, PK_{\overline{\mathcal{P}}}, \overline{\mathcal{P}}^{\leftarrow}, PK_{\overline{\mathcal{P}}^{\leftarrow}})$.

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

(7) Oracle access: the challenger processes all requests as in step 2)

R	except for those where $\mathbf{Proc}(P_H^{\leftarrow}, O)$ with O being the challenge onion as processed for the honest relay on the backward path, i.e.: • for $b = 0$: $\textsc{ROnion}(n{+}j^{\leftarrow}, O, \mathcal{R}, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, PK_{\mathcal{P}}, PK_{\mathcal{P}^{\leftarrow}}) = True$ or • for $b = 1$: $\textsc{ROnion}((n{-}j){+}j^{\leftarrow}, O, \overline{\mathcal{R}}, m, \overline{\mathcal{P}}, \overline{\mathcal{P}}^{\leftarrow}, PK_{\overline{\mathcal{P}}}, PK_{\overline{\mathcal{P}}^{\leftarrow}}) = True$ …then the challenger outputs nothing.

(8) The adversary produces guess b'.

$TI^{\leftrightarrow}$ is achieved if any PPT adversary $\mathcal{A}$ cannot guess $b' = b$ with a probability non-negligibly better than $\frac{1}{2}$ [20].

7.6 UC Realization

We aim to prove that the onion properties we constructed for each OR variant suffice to securely realize the corresponding IF for that variant. With that, we complete our formalization: When creating an OR scheme, proofs can be written for the OR properties, while the IFs allow for intuitive reasoning about the privacy guarantees of the variants.

Definition 6
A *secure* OR scheme (G, FormOnion, ProcOnion, (R | FormReply), ROnion, TagOnion) in an OR variant is an OR scheme that satisfies that variant's subset of the onion properties *Correctness*, *LU*, $LU^{\leftarrow}$, and *TI*.

Definition 7
A *secure* OR *protocol* is based on a secure OR scheme in an OR variant and behaves as described in Definition 1.

Theorem 1. *A secure OR protocol in an OR variant securely realizes that variant's OR IF $\mathcal{F}$.*

The proof argument for Theorem 1 can be found in Appendix B.

8 DISCUSSION

In this section, we discuss our assumptions as well as additional aspects of OR protocol design to consider when using the STIR model.

Indistinguishability of Forward and Reply Onions. In our ideal functionality and onion properties, we demand that the adversary cannot distinguish forward onion layers between two honest relays from reply onion layers between the same honest relays if the onion's (forward) sender is honest. This requirement is not strictly necessary for secure OR with replies and comes with efficiency and security tradeoffs (see [26] for a detailed discussion). We do so in keeping with the related work on provably secure OR and because it doubles the expected size of the set of onions any given onion could be confused with.

Choice of Relays. A user must choose a path of relays for every onion they send. To provide protection, OR requires that the path includes at least one honest relay. Ideally, paths would thus be acyclic (Assumption 2) and consist of entirely random relays to maximize the chances of choosing an honest relay. In practice, users might instead opt for paths that incorporate topological or geographical concerns. This must be done with care to prevent tracking. In service-model variants that permit tagging, exit relays *must* be chosen randomly. Since tagging allows the global adversaries to link senders to exit relays, any senders that choose exit relays predictably risk deanonymizing themselves.

Reacting to Replies. When an honest sender sends a repliable onion to a corrupted receiver, the adversary controls the reply message. If the adversary can choose a reply message that provokes an (observably distinct) reaction from the sender, the adversary can use that to link the sender to its original forward message. This also applies when the adversary tags a reply onion to destroy its contents. When designing a protocol that uses OR, this potential risk must be taken into account.

End-to-end Encryption. In this work, we assume that senders do not encrypt the messages inside the onions they send to their chosen receivers. However, in practice, senders will often use end-to-end encryption (E2EE) via protocols like TLS inside onions to secure their messages. In the integrated-system model, this makes no difference: the OR protocol itself already ensures end-to-end encryption and integrity from the sender to the final relay, which is also the receiver.

The service model behaves differently due to the addition of the link between the exit relay and the receiver. When using CCA-secure E2EE here, the adversary does not learn plaintext messages or replies on the final link. We considered including the use of E2EE in this manner in our analysis framework, but found that doing so made both the ideal functionality and the onion properties in the service model excessively complex due to the special case on the final link: Since the packet sent on that link is an E2E-encrypted ciphertext instead of an onion, we must essentially duplicate every onion property to also handle the cases where the property's honest relay P_H is the exit relay and it sends or receives a ciphertext instead of an onion. Instead, we choose to discuss the consequences of the inclusion informally here.

The protection offered by adding E2EE to service-model OR is similar to that of adding an additional innermost onion layer to each onion, which is peeled by the receiver. This is not entirely equivalent to the integrated-system model since relays are distinct from receivers (i.e., receivers cannot be chosen as intermediate relays) and encrypted reply messages can be tagged (i.e., destroyed) before they are embedded in their reply onion by the exit relay. Apart from these differences, packets in the service model with E2EE behave like packets in the integrated-system model.

9 CONCLUSION

In this work, we first propose the STIR model to categorize existing OR and mix networks as well as their formalizations. Our STIR model comprises twelve distinct categories distinguished by four binary properties: the Service setting, Trust in receivers, Integrity of payloads, and Replies.

Building upon the identified variants, we propose the STIR framework that covers the privacy requirements of OR packet formats for each of the variants. More precisely, for each variant, we construct an ideal functionality $\mathcal{F}$ and onion properties *Correctness*, LU, $LU^{\leftarrow}$, and TI. We show that an OR packet format that satisfies its variant's properties securely realizes that variant's $\mathcal{F}$.

Using our framework, new OR packet formats can be proven secure using our onion properties. The OR protocols based on these packet formats can then be used as a component in larger protocols with the help of $\mathcal{F}$, which offers an abstract interface to the OR protocol, again making it easier to prove security for the larger protocols.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Megumi Ando and Anna Lysyanskaya. 2021. Cryptographic Shallots: A Formal Treatment of Repliable Onion Encryption. In *Theory of Cryptography*.

[2] Michael Backes, Ian Goldberg, Aniket Kate, and Esfandiar Mohammadi. 2012. Provably Secure and Practical Onion Routing. In *2012 IEEE 25th Computer Security Foundations Symposium*. 369–385. https://doi.org/10.1109/CSF.2012.32

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

[4] Oliver Berthold, Hannes Federrath, and Stefan Köpsell. 2001. *Web MIXes: A System for Anonymous and Unobservable Internet Access.* Springer Berlin Heidelberg, Berlin, Heidelberg, 115–129. https://doi.org/10.1007/3-540-44702-4_7

[5] Jan Camenisch and Anna Lysyanskaya. 2005. A Formal Treatment of Onion Routing. In *CRYPTO*.

[6] R. Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science*.

[7] Dario Catalano, Mario Di Raimondo, Dario Fiore, Rosario Gennaro, and Orazio Puglisi. 2011. Fully Non-interactive Onion Routing with Forward-Secrecy. In *Applied Cryptography and Network Security*, Javier Lopez and Gene Tsudik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 255–273.

[8] David L Chaum. 1981. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM* 24, 2 (1981), 84–90.

[9] Chen Chen, Daniele E. Asoni, David Barrera, George Danezis, and Adrain Perrig. 2015. HORNET: High-Speed Onion Routing at the Network Layer. In *ACM CCS*. 14 pages.

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

[11] G. Danezis, R. Dingledine, and N. Mathewson. 2003. Mixminion: design of a type III anonymous remailer protocol. In *2003 Symposium on Security and Privacy, 2003.* 2–15. https://doi.org/10.1109/SECPRI.2003.1199323

[12] George Danezis and Ian Goldberg. 2009. Sphinx: A Compact and Provably Secure Mix Format. In *IEEE S&P*.

[13] George Danezis and Ben Laurie. 2004. Minx: A Simple and Efficient Anonymous Packet Format. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society* (Washington DC, USA) *(WPES '04)*. Association for Computing Machinery, New York, NY, USA, 59–65. https://doi.org/10.1145/1029179.1029198

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

[15] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Sec*.

[16] David Goldschlag, Michael Reed, and Paul Syverson. 1996. Hiding Routing Information. In *First International Workshop on Information Hiding*.

[17] C. Gulcu and G. Tsudik. 1996. Mixing E-mail with Babel. In *Proceedings of Internet Society Symposium on Network and Distributed Systems Security*. 2–16. https://doi.org/10.1109/NDSS.1996.492350

[18] A. Jerichow, J. Muller, A. Pfitzmann, B. Pfitzmann, and M. Waidner. 1998. Real-time mixes: a bandwidth-efficient anonymity protocol. *IEEE Journal on Selected Areas in Communications* 16, 4 (1998), 495–509. https://doi.org/10.1109/49.668973

[19] Christiane Kuhn, Martin Beck, and Thorsten Strufe. 2020. Breaking and (Partially) Fixing Provably Secure Onion Routing. In *IEEE S&P*.

[20] Christiane Kuhn, Dennis Hofheinz, Andy Rupp, and Thorsten Strufe. 2021. Onion Routing with Replies. In *ASIACRYPT*, Mehdi Tibouchi and Huaxiong Wang (Eds.).

[21] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. 2015. Herd: A Scalable, Traffic Analysis Resistant Anonymity Network for VoIP Systems. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 639–652. https://doi.org/10.1145/2785956.2787491

[22] U Moeller. 2004. *Mixmaster Protocol Version 2.* Internet-Draft draft-sassaman-mixmaster-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-sassaman-mixmaster/03/ Work in Progress.

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

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

[25] M.G. Reed, P.F. Syverson, and D.M. Goldschlag. 1998. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications* 16, 4 (1998), 482–494. https://doi.org/10.1109/49.668972

[26] Philip Scherer, Christiane Weis, and Thorsten Strufe. 2023. Provable Security for the Onion Routing and Mix Network Packet Format Sphinx. (December 2023). https://arxiv.org/abs/2312.08028 Preprint.

[27] Erik Shimshock, Matt Staats, and Nick Hopper. 2008. Breaking and Provably Fixing Minx. In *Privacy Enhancing Technologies*, Nikita Borisov and Ian Goldberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–114.

[28] Paul Syverson, Gene Tsudik, Michael Reed, and Carl Landwehr. 2001. *Towards an Analysis of Onion Routing Security.* Springer Berlin Heidelberg, Berlin, Heidelberg, 96–114. https://doi.org/10.1007/3-540-44702-4_6

[29] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 137–152.

A IDEAL FUNCTIONALITY

In this appendix, we present our generalized ideal functionality $\mathcal{F}$ for OR in any of our OR variants. The functionality is given as pseudocode in three algorithms: Algorithm 1, Algorithm 2, and Algorithm 3. $\mathcal{F}$ consists of several routines that are executed by the trusted third party when triggered. Routines labeled **On message** are triggered by $\mathcal{Z}$ or $\mathcal{S}$ directly via sending the according message with the listed parameters to the trusted party. Routines labeled **procedure** are used internally by $\mathcal{F}$ and are only called from other routines. Outputs are sent to the honest and corrupted parties via the SEND procedure.

In $\mathcal{F}$, onions are a tuple of parameters used for the onion's processing:

- sid: A unique identifier for every onion in the network.
- P_s, m, $\mathcal{P}$, $\mathcal{P}^{\leftarrow}$: The parameters used to construct the onion. These are the same as the parameters to FORMONION. For the sake of readability, we retain the reply path parameter in the pseudocode even in non-repliable variants. In that case, the reply path is always empty.
- i: The index of the most recent honest relay on the onion's path that processed the onion.
- d: The current *direction* of the onion, i.e., whether it is on its forward (f) or reply (b) path.

$\mathcal{F}$ maintains multiple data structures to keep track of onions and temporary identifiers:

- Bad: The static set of corrupted parties in the network.
- L_o: The list of onion layers under the adversary's control. Onion layers are added to this list once per path segment when they are forwarded by their sender or an honest relay.
- B_i: The list of onion layers currently held by the honest relay P_i. A layer is removed from the list when $\mathcal{Z}$ decides to forward the onion.

R	• B_i^r: The list of honest reply onions currently held by the honest relay P_i. These are onions where P_i is the final relay on the forward path and the reply onion has been created, but not yet forwarded by $\mathcal{Z}$. • *Back*: *Back* maps temporary identifiers to onion reply information. An entry in *Back* is created when an onion reaches its final honest relay. The information is used when the reply onion is created due to $\mathcal{S}$ or an honest P_i. • ID_{fwd}: ID_{fwd} maps from the sid of a forward onion to the corresponding sid' of the reply onion. This is used for outputs to $\mathcal{S}$ when processing onions from a corrupted sender.
S	• Rep_i: On an honest exit relay P_i, this list maps the temporary identifiers used in *Back* to *reply IDs*, which keep track of the connection between the exit relay and the receiver.
Ī	• L_{tag}: The list of onions that have been tagged by the adversary.

As a guide for understanding the processing of an onion in $\mathcal{F}$, we provide Figure 3. The figure illustrates the order in which the routines in $\mathcal{F}$ are used to process an onion.

Algorithm 1 Ideal Functionality $\mathcal{F}$ (Creation and Onion Processing)

▷ $P_s \in \mathcal{N}$ creates and sends a new onion
On message PROCESSNEWONION($m, \mathcal{P}, \mathcal{P}^{\leftarrow}$) *from* $\mathcal{Z}$ *or* $\mathcal{S}$ *via* P_s
  $\mathcal{P}^{\leftarrow} \leftarrow ()$ [R̄]
  if $|\mathcal{P}| > N$ **or** $|\mathcal{P}^{\leftarrow}| > N$ **then reject**
  else
    $sid \leftarrow^R$ session ID
    $O \leftarrow (sid, P_s, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, 0, f)$
    OUTPUTCORRUPTSENDER($P_s, sid, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, start, f$)
    PROCESSNEXTSTEP(O)

▷ Give $\mathcal{S}$ all information on an onion if P_s is corrupted
procedure OUTPUTCORRUPTSENDER($P_s, sid, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, tid, d$)
  if $d = f$ **and** $P_s \in$ Bad **then**
    SEND($\mathcal{S}$, "tid is from P_s with sid, m, $\mathcal{P}$, $\mathcal{P}^{\leftarrow}$, d")
  else if $d = b$ **and** $\mathcal{P}^{\leftarrow}_{|\mathcal{P}^{\leftarrow}|-1} \in$ Bad **then** [R]
    SEND($\mathcal{S}$, "tid is reply from P_s with sid, m, $\mathcal{P}$, $\mathcal{P}^{\leftarrow}$, d, replying to onion from $\mathcal{P}^{\leftarrow}_{|\mathcal{P}^{\leftarrow}|-1}$ with $ID_{fwd}(sid)$")

▷ P_{o_i} has processed O, passes it to $P_{o_{i+1}}$
procedure PROCESSNEXTSTEP($O = (sid, P_s, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d)$)
  if $\forall j > i : P_{o_j} \in$ Bad **or** $i = |\mathcal{P}|$ **then**
    ▷ Either all remaining relays including the final relay
    ▷ (or reply receiver) are corrupted, [S | or P_{o_i} is the exit relay]
    if $O \in L_{tag}$ **then** DELIVERTAGGED(O) [Ī]
    else
      if $d = f$ **then** LEAKMESSAGE(O)
      else LEAKREPLYONION(O) [R]
  else PROCESSTORELAY(O)

▷ $\mathcal{S}$ learns the message at the end of O's path
procedure LEAKMESSAGE($O = (sid, P_s, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d)$)
  if $m = \perp$ **then return**
  OUTPUTCORRUPTSENDER($P_s, sid, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, end, d$)
  if $\mathcal{P}^{\leftarrow} \neq ()$ **then** [R]
    CONFIGUREREPLY(O)
  if $i = |\mathcal{P}|$ **then** SEND(P_{o_i}, "Sent message to $P_{o_{|\mathcal{P}|-1}}$") [S]
  else SEND(P_{o_i}, "Sent onion to $P_{o_{i+1}}$")
  SEND($\mathcal{S}$, "P_{o_i} sends onion with message m to $P_{o_{|\mathcal{P}|-1}}$ via $(P_{o_{i+1}}, \ldots, P_{o_{|\mathcal{P}|-2}})$")

▷ Process onion with honest successor relay P_{o_j}
procedure PROCESSTORELAY($O = (sid, P_s, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d)$)
  $P_{o_j} \leftarrow P_{o_k}$ with smallest $k > i$ such that $P_{o_k} \notin$ Bad
  $tid \leftarrow^R$ temporary ID
  SEND($\mathcal{S}$, "P_{o_i} sends tid to P_{o_j} via $(P_{o_{i+1}}, \ldots, P_{o_{j-1}})$")
  SEND(P_{o_i}, "Sent onion to $P_{o_{i+1}}$")
  OUTPUTCORRUPTSENDER($P_s, sid, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, tid, d$)
  if $d = b$ **and** $i = 0$ **then** [R]
    SEND($\mathcal{S}$, "tid belongs to sid")
  Add (tid, O, j) to L_o

▷ The tagged onion O reaches its final relay [Ī]
procedure DELIVERTAGGED($O = (sid, P_s, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d)$)
  OUTPUTCORRUPTSENDER($P_s, sid, m, \mathcal{P}$, *tagged*, d)
  if $i < n$ **then**
    SEND($\mathcal{S}$, "P_{o_i} sends tagged onion via $(P_{o_{i+1}}, \ldots, P_{o_{|\mathcal{P}|-2}})$")
    SEND($\mathcal{Z}$, "P_{o_i} sends onion to $P_{o_{i+1}}$")
  else SEND($\mathcal{Z}$, "Onion at P_{o_i} fails integrity check")

Algorithm 2 Ideal Functionality $\mathcal{F}$ (Delivery and Forwarding)

▷ $\mathcal{S}$ tags the onion tid — $\bar{\text{I}}$
On message Tag(tid) *from* $\mathcal{S}$
  if $(tid, \_, \_) \in L_o$ **then**
    Retrieve $(tid, O, \_)$ from L_o
    Store O in L_{tag}

▷ $\mathcal{S}$ delivers the onion tid to the next relay
On message DeliverOnion(tid) *from* $\mathcal{S}$
  if $(tid, \_, \_) \in L_o$ **then**
    $(tid, O = (sid, P_s, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d), j) \leftarrow L_o$
    $O \leftarrow (sid, P_s, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, j, d)$

  if $d = f$ **and** $j = |\mathcal{P}|$ **then** — $\bar{\text{S}}$
    if $m \neq \perp$ **and** $O \notin L_{tag}$ ($\bar{\text{I}}$) **then**
      if $\mathcal{P}^{\leftarrow} \neq ()$ **then** — R
        ConfigureReply(O)
      Send(P_{o_j}, "Message m received")
      else if $O \in L_{tag}$ **then** — $\bar{\text{I}}$
        Send($\mathcal{Z}$, "Onion at P_{o_j} fails integrity check")

  else if $d = b$ **and** $j = |\mathcal{P}|$ **then** — R
    if $m \neq \perp$ **and** $O \notin L_{tag}$ ($\bar{\text{I}}$) **then**
      Send(P_{o_j}, "Message m received as reply")
      ▷ Not forwarded to $\mathcal{Z}$
  else
    $tid' \leftarrow^R$ temporary ID
    Send(P_{o_j}, "tid' received from $P_{o_{j-1}}$")
    Store (tid', O) in B_{o_j}

▷ P_i (honest or corrupted) is done processing the onion tid'
On message ForwardOnion(tid') *from* $\mathcal{Z}$ *or* $\mathcal{S}$ *via* P_i
  if $(tid', \_) \in B_i$ **then**
    Pop (tid', O) from B_i
    ProcessNextStep(O)

  else if $(tid', \_) \in B_i^r$ **then** — R
    Pop (tid', m, tid) from B_i^r
    SendReplyOnion(m, tid)

▷ $\mathcal{S}$ (masquerading as $P_i \in \mathcal{N}$) delivers a message to a receiver — S
On message DeliverMessage($P_i, m,$ (rid | R), R) *from* $\mathcal{S}$
  Send(R, "Message m received from P_i")
  if $rid \neq \perp$ **then** — R
    Send(R, "Message is repliable with rid")

Algorithm 3 Ideal Functionality $\mathcal{F}$ (Reply Handling)

▷ Send and store information to enable replies to O — R
procedure ConfigureReply($O = (sid, P_s, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d)$)
  $rid, tid \leftarrow^R$ temporary IDs
  Store $(tid, P_s, \mathcal{P}, \mathcal{P}^{\leftarrow}, P_{o_i}, sid)$ in *Back*
  if $i = |\mathcal{P}|$ **then**
    Store (rid, tid) in Rep_{o_i}
    Send($\mathcal{S}$, "Reply to following with reply ID rid") — S
    Send(P_{o_i}, "Reply to following with reply ID rid") — $\bar{\text{S}}$
  else
    $\mathcal{P}_1^{\leftarrow} \leftarrow$ prefix of $\mathcal{P}^{\leftarrow}$ up to (including) the first honest relay
    Send($\mathcal{S}$, "Reply to following with tid, reply path begins with $\mathcal{P}_1^{\leftarrow}$")

▷ $\mathcal{S}$ uses a tid to reply from any adversarial relay
On message SendAdversarialReplyOnion(m, tid) *from* $\mathcal{S}$ *via* P_i
  if $(tid, \ldots) \in Back$ **then**
    SendReplyOnion(m, tid)

▷ P_i replies to an onion with an rid — $\bar{\text{S}}$
On message SendHonestReplyOnion(m, rid) *from* $\mathcal{Z}$ *via* P_i
  if $(rid, \_) \in Rep_i$ **then**
    $(rid, tid) \leftarrow Rep_i$
    $tid' \leftarrow^R$ temporary ID
    Store (tid', m, tid) in B_i^r
    Send(P_i, "Send reply onion with tid'")

▷ $R \in D$ decides to send a reply message to rid via P_i — S
On message SendReplyMessage(P_i, m, rid) *from* $\mathcal{Z}$ *or* $\mathcal{S}$ *via* R
  Send($\mathcal{S}$, "R replies to rid with message m via P_i")

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

▷ Create and send the reply onion for tid
procedure SendReplyOnion(m, tid)
  if $(tid, \ldots) \notin Back$ **then reject**
  else
    $(tid, P_s, \mathcal{P}, \mathcal{P}^{\leftarrow}, sid', \_) \leftarrow Back$
    $sid \leftarrow^R$ session ID
    Store (sid, sid') in ID_{fwd}
    $O \leftarrow (sid, P_i, m, \mathcal{P}^{\leftarrow}, (), 0, b)$
    OutputCorruptSender($P_i, sid, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, start, b$)
    ProcessNextStep(O)

▷ $\mathcal{S}$ learns the reply message as it is delivered to the corrupted sender
procedure LeakReplyOnion($O = (sid, P_s, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, i, d)$)
  Send($\mathcal{S}$, "P_{o_i} sends reply tid with message m to $P_{o_{|\mathcal{P}|-1}}$ via $(P_{o_{i+1}}, \ldots, P_{o_{|\mathcal{P}|-2}})$")
  Send(P_{o_i}, "Sent onion to $P_{o_{i+1}}$")
  OutputCorruptSender($P_s, sid, m, \mathcal{P}, \mathcal{P}^{\leftarrow}, tid, b$)

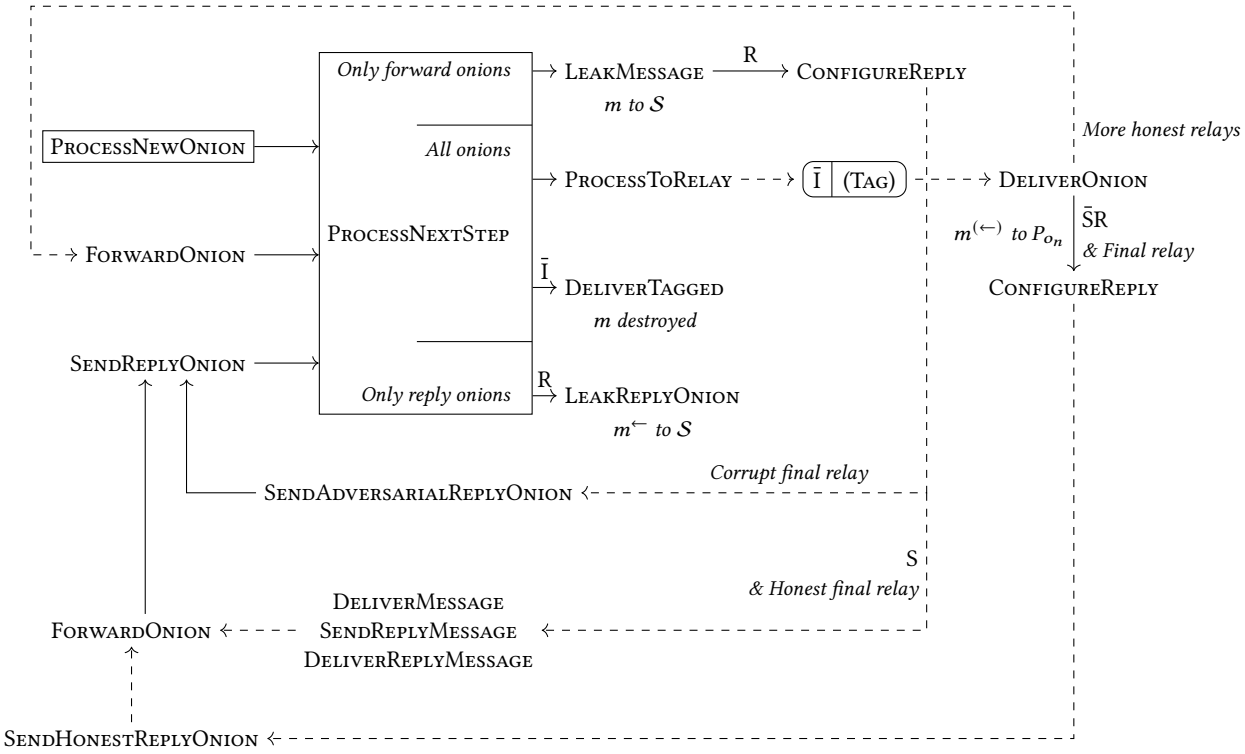

Figure 3: Flowchart illustrating the processing of abstract onions in $\mathcal{F}$. An onion is created using ProcessNewOnion. It then goes through one ProcessToRelay–DeliverOnion–ForwardOnion cycle for every path segment before reaching the end of its path. In repliable variants, repliable onions may go through more cycles via SendReplyOnion. The text below some routine names indicates which party learns the plaintext message in that situation. Solid arrows mean that the transition happens immediately within $\mathcal{F}$, while dashed arrows mean that an input from $\mathcal{Z}$ or $\mathcal{S}$ is required to continue.

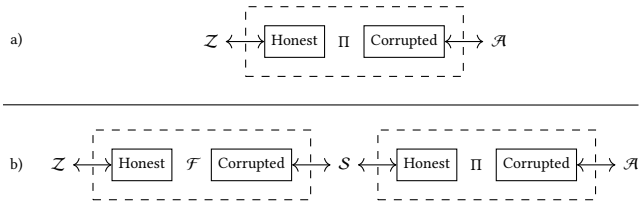

Figure 4: Two scenarios the environment $\mathcal{Z}$ must distinguish in a UC realization proof. In a), $\mathcal{Z}$ interacts with the actual protocol Π in the role of the honest parties, while $\mathcal{A}$ controls the corrupted parties. In b), $\mathcal{Z}$ controls the honest parties in $\mathcal{F}$ and $\mathcal{A}$ commands the corrupted parties in Π. $\mathcal{S}$ simulates their respective other parties.

B UC REALIZATION PROOF ARGUMENT

Theorem 1. *A secure OR protocol in an OR variant securely realizes that variant's OR IF $\mathcal{F}$.*

Proof. The basic structure of our proof follows that of related proofs for existing OR formalizations by Kuhn et al. [19, 20] and [26]. We focus on this structure here since it communicates the most important ideas and intuitions. However, every OR variant requires its own adaptations to the basic structure. We summarize these changes where appropriate. Throughout the proof, we take advantage of our deliberate construction of the onion properties for each variant so they apply cleanly in the proof argument.

To prove that a protocol securely realizes an IF in a UC model, one must show that any given attacker $\mathcal{A}$ on the real protocol can be simulated by a corresponding simulator $\mathcal{S}$ that interacts with the IF such that any environment $\mathcal{Z}$ (which controls the honest parties) cannot distinguish between the two. In particular, the environment and the attacker may communicate and collaborate at will. Our proof thus proceeds in two steps:

(1) Construct a matching simulator $\mathcal{S}$ for every attacker $\mathcal{A}$. Our simulator uses $\mathcal{A}$ internally and translates the $\mathcal{A}$'s messages in the real protocol into messages to $\mathcal{F}$ and vice versa. $\mathcal{S}$ effectively acts as a middle-man between $\mathcal{F}$ and $\mathcal{A}$.
(2) Prove that the interaction between $\mathcal{Z}$, $\mathcal{F}$, and $\mathcal{S}$ is indistinguishable from the interaction between $\mathcal{Z}$, the real protocol, and $\mathcal{A}$ from both $\mathcal{Z}$'s and $\mathcal{A}$'s perspectives. The onion properties are required in this step. Figure 4 illustrates the two scenarios.

Construction of $\mathcal{S}$. $\mathcal{S}$ needs to act as the bridge between the attacker $\mathcal{A}$ and the IF $\mathcal{F}$. This means that the messages sent to $\mathcal{S}$ by $\mathcal{F}$ must be translated into onions or plaintext messages to $\mathcal{A}$, while any real-world onions or messages from $\mathcal{A}$ must be turned into messages to $\mathcal{F}$.

The difficulty in translating the onions from each world into the other is that $\mathcal{S}$ does not have complete information on the onions it needs to translate — it must operate solely based on the information it is given by $\mathcal{F}$ and the onions it receives from $\mathcal{A}$. In the following, we first explain $\mathcal{S}$'s handling of onions from honest senders, then the translation for onions $\mathcal{S}$ receives from $\mathcal{A}$.

Onions from honest senders. The behavior of $\mathcal{S}$ described below is also depicted as a flowchart in Figure 5.

When $\mathcal{Z}$ commands an honest party in the ideal world to send an onion, $\mathcal{S}$ receives information from $\mathcal{F}$ every time the onion is sent along a segment on its path in the ideal world. However, assuming the path segment leads from P_{o_i} to P_{o_j}, the only information $\mathcal{S}$ gets is a random *tid* used to identify that path segment of the onion along with the path segment $(P_{o_i}, \ldots, P_{o_j})$ itself (see ProcessToRelay). $\mathcal{S}$ must use this information to build real-world onions for $\mathcal{A}$. Since $\mathcal{S}$ does not have any information on the contents of the onion or the rest of the path, it can only create an onion with random contents that follows the given path segment. This parallels the behavior of *LU*, which we make use of later.

At this point, the onion is under $\mathcal{A}$'s control until $\mathcal{A}$ eventually decides to process and forward it to $\mathcal{S}$'s real-world P_{o_j}. Once that happens, $\mathcal{S}$ recognizes the onion as being the random onion it sent earlier using RecognizeOnion. $\mathcal{S}$ now uses the *tid* it got from $\mathcal{F}$ to notify $\mathcal{F}$ that the onion has been delivered. This cycle continues until the onion reaches its final *honest* relay.

$\bar{\text{I}}$	While the onion was under $\mathcal{A}$'s control, $\mathcal{A}$ might have decided to tag the payload. $\mathcal{S}$ can detect this using RecognizeOnion and notifies $\mathcal{F}$ of it with the Tag message.

At the final honest relay, two different cases might occur: Either the onion's final honest relay is also the last relay on the path, or there are more corrupted relays following it. In the former case:

S	The final honest relay is the onion's exit relay. $\mathcal{S}$ receives the message (R | and a potential *rid*) of the onion and forwards them to the external receiver via $\mathcal{A}$.
$\bar{\text{S}}$	The final honest relay is the onion's receiver (i.e., the receiver is honest), so the onion's processing ends there. The message (R | and potentially an *rid*) are delivered to the honest receiver.
	$\bar{\text{I}}$ | If the onion was tagged, the payload is destroyed and no message is delivered.

If, on the other hand, there are more corrupted relays on the path, the onion still has one path segment left. In this case, $\mathcal{S}$ gets the message, the receiver, and the remaining path segment of the onion (see LeakMessage). When creating the onion for $\mathcal{A}$, $\mathcal{S}$ can now include the correct contents as well as using the correct path segment. This behavior is comparable to that of *TI*.

$\bar{\text{I}}$	If the adversary tagged the onion before it reached the final honest relay, $\mathcal{S}$ does not get this information (see DeliverTagged) and must instead create a real-world onion with a random message and receiver. Before giving that onion to $\mathcal{A}$, $\mathcal{S}$ tags it. The definition of *LU* ensures that the payload contents are completely destroyed by the tagging attack, so the adversary cannot notice the replacement.
R	For repliable onions, $\mathcal{F}$ also leaks the first path segment on the onion's reply path to $\mathcal{S}$ in LeakMessage. $\mathcal{S}$ includes this path segment as the reply path of its random onion.

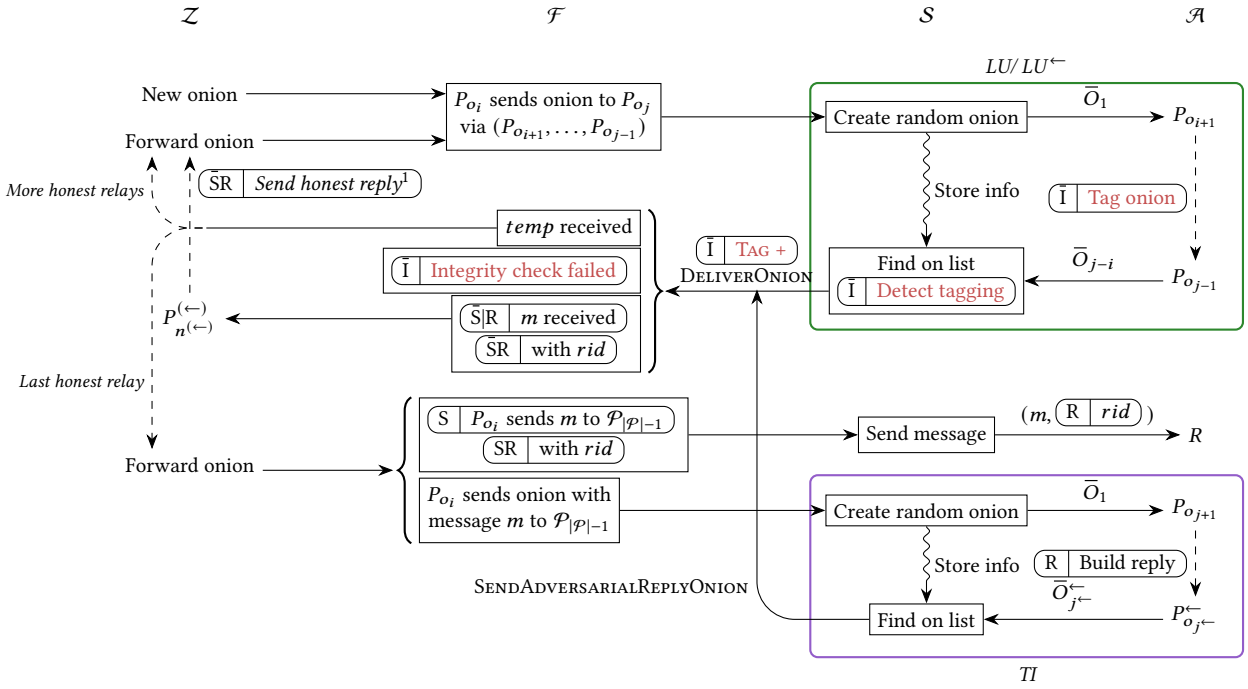

Figure 5: A sequence diagram illustrating the $\mathcal{F}$ simulator's handling of onions by honest senders. The interaction between the environment $\mathcal{Z}$, the IF $\mathcal{F}$, the simulator $\mathcal{S}$, and the adversary $\mathcal{A}$ is shown for an onion's entire lifetime. Red text indicates events that occur if the adversary tags the onion. The colored boxes show where each onion property's security guarantees are used. [1] **The honest reply process is simplified here for the purpose of representation. See Algorithm 3 for details.**

R	*Reply handling.* If the onion is repliable, the receiver may decide to reply. We again distinguish whether the final relay on the path is honest: • The final relay is honest. $\bar{\text{S}}$: The receiver is honest. Since the receiver is a relay, it can build its own reply onion and send it (see SendHonestReplyOnion). S: The exit relay is honest. $\mathcal{A}$ may decide to deliver a reply message with the correct *rid* to the exit relay. $\mathcal{S}$ communicates this to $\mathcal{F}$ with DeliverReplyMessage, from where the reply onion is treated like in the first case. • The final relay is not honest. In this situation, $\mathcal{A}$ has control over the final layer of the onion $\mathcal{S}$ built for it and can build the reply onion itself. If $\mathcal{A}$ chooses to reply, $\mathcal{S}$ will receive its replacement onion from $\mathcal{A}$ at the end of the reply path $\mathcal{S}$ chose. $\mathcal{S}$ can recognize the replacement onion with RecognizeOnion. From it, $\mathcal{S}$ receives the reply message (since it controls the relay that received the replacement onion's reply) and uses SendAdversarialReplyOnion to create the ideal-world reply onion and deliver it to the first honest relay on the reply path.

Onions from $\mathcal{A}$. When $\mathcal{S}$ receives a real-world onion from $\mathcal{A}$ at one of the honest relays it controls, that onion is either $\mathcal{S}$'s replacement for an honest sender's onion (which $\mathcal{S}$ can recognize and process as described above) or an onion that is unknown to $\mathcal{S}$. In the latter case, the onion was sent by an adversarial sender in the real protocol. $\mathcal{S}$ now needs to translate that onion into the ideal world. To do so, $\mathcal{S}$ processes the real-world onion as far as it can using the real-world private keys of the honest relays. This results in $\mathcal{S}$ learning the onion layers on a subpath of the onion's path consisting only of consecutive honest relays. If the subpath is a suffix of the onion's path, $\mathcal{S}$ also learns the contents of the onion. Using this information, $\mathcal{S}$ creates an ideal-world onion using ProcessNewOnion with the appropriate parameters and stores the path and onion layers for later use. $\mathcal{Z}$ now has control over the forwarding of the ideal-world onion. When it chooses to do so, $\mathcal{S}$ is notified and sends the appropriate layer of the real-world onion to $\mathcal{A}$. This layer is returned to $\mathcal{S}$ when $\mathcal{A}$ decides to deliver the onion to the next honest relay. $\mathcal{S}$ calls DeliverOnion to notify $\mathcal{F}$ of the delivery. This process continues until the subpath that $\mathcal{S}$ learned is exhausted, at which point the onion will have been delivered to the last relay on the subpath. This may be the final relay on the real-world onion's path. Alternatively, the real-world onion's next relay could be adversarial. In this case, $\mathcal{S}$ sends the last onion layer it was able to learn to $\mathcal{A}$, concluding the processing of these onion layers. The same onion may be processed further by $\mathcal{A}$ and delivered back to $\mathcal{S}$ at a later time. $\mathcal{S}$ cannot correlate these separate layers of the onion and will treat the new layer as a fresh onion from $\mathcal{A}$.

$\bar{\text{I}}$	$\mathcal{A}$ may choose to deliver a tagged onion to $\mathcal{S}$. If the onion subpath $\mathcal{S}$ can process does not include the final relay, $\mathcal{S}$ will not notice the tagging and will not tag the ideal-world onion. This has no impact on $\mathcal{F}$'s outputs to $\mathcal{Z}$. $\mathcal{A}$ may also tag one of the intermediate onion layers that $\mathcal{S}$ sends to it following $\mathcal{Z}$'s command. $\mathcal{S}$ can notice these tags by simply comparing the onion layer it receives to the onion layer it sent. $\mathcal{S}$ tags the onion in $\mathcal{F}$ accordingly.

R	If the onion is repliable and the subpath of honest relays that $\mathcal{S}$ learns includes the final relay, $\mathcal{S}$ can also build a real-world reply onion and potentially process it to learn a prefix of the real-world onion's reply path. If so, $\mathcal{S}$ includes the reply path prefix in the parameters to ProcessNewOnion. If the ideal-world reply onion ends up being sent by $\mathcal{Z}$, $\mathcal{S}$ can build the corresponding real-world reply onion after learning the reply message via OutputCorruptSender. The remainder of the reply's processing is identical to the forward case.

This concludes our description of the construction of the simulator.

Indistinguishability. Now, all that remains is to show that the interaction between $\mathcal{Z}$ and the real protocol with the attacker $\mathcal{A}$ is indistinguishable from the interaction between $\mathcal{Z}$ and $\mathcal{F}$ with the simulator $\mathcal{S}$. First, we note that adversarial onions are handled indistinguishably from the real protocol by $\mathcal{S}$. $\mathcal{Z}$ receives exactly the same information from $\mathcal{F}$ when handling these onions in the ideal world as it would in the real protocol by construction of $\mathcal{S}$ and $\mathcal{F}$.

That leaves only onions from honest senders. To prove that these are handled correctly by $\mathcal{S}$, we make use of the onion properties in each variant in a hybrid argument beginning with a hybrid machine that controls the honest parties and simply runs the real OR protocol and ending at a hybrid that only sends random onion layers instead of real onion layers for every onion from an honest sender. The behavior of the final hybrid is just like that of $\mathcal{F}$ with $\mathcal{S}$. From $\mathcal{Z}$'s perspective, the honest sender's onions are processed correctly by both the hybrid and $\mathcal{F}$+$\mathcal{S}$. From $\mathcal{A}$'s perspective, both the hybrid and $\mathcal{F}$+$\mathcal{S}$ send the same random onion layers on every path segment. We conclude that completing the hybrid argument proves that the real protocol securely realizes $\mathcal{F}$ in the appropriate OR variant.

To move from the first hybrid machine to the last, we need to apply the onion properties to replace onion layers on individual path segments with random onion layers. We begin with an onion's first path segment: Applying *LU* allows us to replace the onion layers there.

$\bar{\text{I}}$	If an onion layer is tagged on the replaced path segment, the hybrid machine needs to treat it like *LU* would by tagging the re-inserted onion layers after the honest relay.

By repeating this argument, we can replace the onion layers of every honest sender's onion's first path segment in successive hybrids[3].

Following this, the next hybrids replace the onion layers on the other path segments on the forward path for these onions, applying *LU* for every replacement[4]. In OR variants that require *Correctness* and *LU* as their only onion properties, this already concludes the hybrid argument.

R	In variants with replies, we repeat the same sequence of path segment replacements on the reply path. Path segments are replaced from back to front, applying $LU^{\leftarrow}$ for every replacement.
$\bar{\text{T}}$|S	In these variants, *TI* is required because the final relay on the onion's path may not be honest. After the previous sequence of hybrids, any onions with a corrupted final relay will still have one path segment left at the end of their forward path (R | and one segment at the start of the reply path) . The onion layers on these path segment(s) can be replaced with random onion layers by applying *TI*.

With this, we have shown that a secure OR protocol in an OR variant securely realizes $\mathcal{F}$ in that variant as well.

[3] This applies even for polynomially many onions, as can be verified through a standard hybrid reduction.

[4] Applying *LU* for multiple path segments of one onion in this manner is possible because the onion that is re-inserted after the honest relay by *LU* is not the j-th layer of the original adversary-chosen onion, but instead the first layer of an onion with the same parameters and a truncated forward path. Thanks to this, the re-inserted onion fulfills the requirements for applying *LU* on the next path segment.