Cryptographic Shallots: A Formal Treatment of Repliable Onion Encryption

Megumi Ando[1] and Anna Lysyanskaya[2]

[1] MITRE, Bedford, MA 01730 USA
[2] Brown University, Providence, RI 02912 USA

Abstract. Onion routing is a popular, efficient, and scalable method for enabling anonymous communications. To send a message m to Bob via onion routing, Alice picks several intermediaries, wraps m in multiple layers of encryption — a layer per intermediary — and sends the resulting *onion* to the first intermediary. Each intermediary *peels off* a layer of encryption and learns the identity of the next entity on the path and what to send along; finally Bob learns that he is the recipient and recovers the message m.

Despite its wide use in the real world, the foundations of onion routing have not been thoroughly studied. In particular, although two-way communication is needed in most instances, such as anonymous Web browsing or anonymous access to a resource, until now no definitions or provably secure constructions have been given for two-way onion routing. Moreover, the security definitions that existed even for one-way onion routing were found to have significant flaws.

In this paper, we (1) propose an ideal functionality for a *repliable* onion encryption scheme; (2) give a game-based definition for repliable onion encryption and show that it is sufficient to realize our ideal functionality; and finally (3), our main result is a construction of repliable onion encryption that satisfies our definitions.

1 Introduction

Suppose Alice wants to send a message to Bob, anonymously, over a point-to-point network such as the Internet. What cryptographic techniques exist to make this possible? One popular approach is onion routing: Alice sends her message through intermediaries, who mix it with other traffic and forward it on to Bob. To make this approach secure from an adversary eavesdropping on the network, she needs to wrap her message in several layers of encryption, one for each intermediary, giving rise to the term *onion routing.*

As originally proposed by Chaum [10], onion routing meant that Alice just uses regular encryption to derive each subsequent layer of her onion before sending it on to the first intermediary. I.e., if the intermediaries are Carol (public key pk_C), David (public key pk_D), and Evelyn (public key pk_E), then to send message m to Bob (public key pk_B), Alice forms her onion by first encrypting m under pk_B, then encrypting the resulting destination-ciphertext pair (Bob, c_B) under

pk_E, and so forth: $O = \mathsf{Enc}_{\mathsf{pk}_C}((\text{David}, \mathsf{Enc}_{\mathsf{pk}_D}((\text{Evelyn}, \mathsf{Enc}_{\mathsf{pk}_E}((\text{Bob}, \mathsf{Enc}_{\mathsf{pk}_B}(m)))))))$. If we use this approach using regular public-key encryption, then the "peeled" onion O' that Carol will forward to David is going to be a shorter (in bit length) ciphertext than O, because ciphertexts are longer than the messages they encrypt. So even if Carol serves as an intermediary for many onions, an eavesdropping adversary can link O and O' by their lengths, unless Carol happens to also be the first intermediary for another onion.

To ensure that all onions are the same length, no matter which layer an intermediary is responsible for, Camenisch and Lysyanskaya [5] introduced *onion encryption*, a tailor-made public-key encryption scheme where the adversary can't tell how far an intermediary, e.g. Carol, is from an onion's destination, even for adversarial Carol. They gave an ideal functionality [6] for uni-directional onion encryption and a cryptographic scheme that, they argued, UC-realized it. However, their work did not altogether solve the problem of anonymous communication via onion routing. As Kuhn et al. [21] point out, there were significant definitional issues. Also, as, for example, Ando et al. [2, 3] show, onion routing by itself does not guarantee anonymity, as a sufficient number of onions need to be present before any mixing occurs.

Those issues aside, however, Camenisch and Lysyanskaya (CL) left open the problem of "repliable" onions. In other words, once Bob receives Alice's message and wants to respond, what does he do? This is not just an esoteric issue. If one wants to do basic online tasks anonymously — e.g., browse the Web incognito or anonymously fill out a feedback form — a two-way channel between the anonymous original sender (here, Alice) and their interlocutor (here, Bob) needs to be established. Although CL outlined an initial idea for how to reply to an onion, they don't provide any definitions or proofs. Babel [18], Mixminion [14], Minx [16], and Sphinx [15] all provide mechanisms for the recipient to reply to the sender but don't provide any formal definitions or proofs either. This left a gap between proposed ideas for a repliable onion encryption scheme and rigorous examinations of these ideas. For instance, Kuhn et al. [21] pointed out a fatal security flaw in the current state-of-the-art, Sphinx. They also pointed out some definitional issues in the CL paper and proposed fixes for some of these issues but left open the problem of formalizing repliable onion encryption.

The challenge. Let us see why repliable onion encryption is not like other types of encryption. Traditionally, to be able to prove that an encryption scheme satisfies a definition of security along the lines of CCA2 security, we direct honest parties (for example, an intermediary Iris) to check whether a ciphertext (or, for our purposes, an onion) she has received is authentic or has been "mauled;" Iris can then refuse to decrypt a "mauled" ciphertext (correspondingly, process a "mauled" onion). Most constructions of CCA2-secure encryption schemes work along these lines; that way, in the proof of security, the decryption oracle does not need to worry about decrypting ciphertexts that do not pass such a validity check, making it possible to prove security. This approach was made more explicit by Cramer and Shoup [12, 13] who defined encryption with tags, where tags defined the scope of a ciphertext, and a ciphertext would never be decrypted unless it was accompanied by the correct tag.

The CL construction of onion encryption also works this way; it uses CCA2-secure encryption with tags to make it possible for each intermediary to check the integrity of an onion it received. So when constructing an onion, the sender had to construct each layer so that it would pass the integrity check, and in doing so, the sender needed to know what each layer was going to look like. This was not a problem for onion security in the forward direction since the sender knew all the puzzle pieces — the message m and the path (e.g. Carol, David, Evelyn) to the recipient Bob — so the sender could compute each layer and derive the correct tag that would allow the integrity check to pass. But in the reverse direction, the recipient Bob needs to form a reply onion without knowing part of the puzzle pieces. He should not know what any subsequent onion layers will look like: if he did, then an adversarial Bob, together with an adversarial intermediary and the network adversary, will be able to trace the reply onion as it gets back to Alice. So he cannot derive the correct tag for every layer. The sender Alice cannot do so either since she does not know in advance what Bob's reply message is going to be. So it is not clear how a CCA2-style definition can be satisfied at all.

At the same time, it is important to make sure that reply onions are indistinguishable (even to intermediaries who process them) from forward onions. As pointed out in prior work [14], this is crucial because "replies may be very rare relative to forward messages, and thus much easier to trace." Thus, making sure that they are hidden among the more voluminous forward traffic is desirable.

Our first contribution: a definition of secure repliable onion encryption. We define security by describing an ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ in the simplified UC model [7]; from now on we refer to it as the *SUC model*. We chose the SUC model so that our functionality and proof did not have to explicitly worry about network issues and other subtleties of the full-blown UC model [6].

As should be expected of secure onion routing, $\mathcal{F}_{\mathsf{ROES}}$ represents onions originating at honest senders or formed as replies to honest senders, using bit strings that are computed independently on the contents of messages, their destinations, whether the onion is traveling in the forward direction or is a reply, and identities and number of intermediaries that follow or precede an honest intermediary. To process an onion, an honest party P sends it to the functionality $\mathcal{F}_{\mathsf{ROES}}$, which then informs P what its role is — an intermediary, the recipient, or the original sender of this onion. If P is an intermediary, the functionality sends it a string that represents the next layer of the same onion (also formed independently of the input). If P is the recipient, P learns the contents of the message m and whether the onion can be replied to, and can direct the functionality to create a reply onion containing a reply message r. Finally, if P is the sender of the original onion, then he learns r, the reply; he also learns to which one of his previous outgoing onions this one is the response. We describe $\mathcal{F}_{\mathsf{ROES}}$ in Section 3.

It is important to note that our functionality $\mathcal{F}_{\mathsf{ROES}}$ is defined in such a way that it allows for a scheme in which checking that an onion has been "mauled" is not entirely the job of each intermediary. More precisely, we think of the onion as consisting of two pieces. The first piece is the *header* H that, in $\mathcal{F}_{\mathsf{ROES}}$, is a pointer to a data structure that contains the onion's information. The second piece is the payload, the *content* C that can be thought of as a pointer to a data structure inside $\mathcal{F}_{\mathsf{ROES}}$ that contains the message m. The content C does not undergo an integrity check until it gets to its destination. This is how we overcome the challenge (above) of having a definition that enables replies.

Our second contribution: a game-style definition of secure repliable onion encryption. Although UC-style definitions of security are a good way to capture the security properties of a novel cryptographic object such as secure repliable onion encryption, they can be difficult to work with. The SUC model makes the job easier, but it is still cumbersome to prove that a construction SUC-realizes an ideal functionality, especially one as involved as $\mathcal{F}_{\mathsf{ROES}}$. So to make it easier, we provide a game-style definition, called "repliable-onion security" in Section 4.

This definition boils down to a game between an adversary and a challenger. The challenger generates the key pairs for two participants under attack: a sender S and an honest intermediary I. Similar to CCA2-security for public-key encryption, the challenger also responds (before and after the creation of a challenge onion) to the adversary's queries to S and I; i.e. the adversary may send onions to the parties under attack and learn how these onions are peeled. The adversary then requests that a challenge repliable onion be formed by S; the adversary picks the recipient R for this onion, as well as the message m to be routed to R, and the identities and public keys of all the intermediaries on the path from S to R (other than S and I) and the return path from the recipient to the sender. The honest intermediary I must appear somewhere on this path: either (a) I is on the forward path from S to R, or (b) I is the recipient, or (c) I is on the return path from R to S. The challenger then tosses a coin, and depending on the outcome, forms the challenge onion in one of two ways; the adversary's job to win the game is to correctly guess the outcome of the coin toss. If the coin comes up heads, the challenger forms the onion correctly, using the routing path provided by the adversary. If it comes up tails, then the challenger makes a "switch:" he forms two unrelated onions, one from S to I, and the other from I back to S; the details depend on whether this is case (a), (b), or (c). He then patches up the oracles for S and I so as to be able to pretend that the challenge onion was formed correctly. For details, see Section 4.

In Section 5, we show that our game-based definition is sufficient to SUC-realize $\mathcal{F}_{\mathsf{ROES}}$ and that its non-adaptive variant is necessary: any repliable onion encryption scheme SUC-realizing $\mathcal{F}_{\mathsf{ROES}}$ will satisfy it.

Here is how we overcome the definitional challenge of having a CCA2-style definition while enabling replies. When forming a repliable onion, the sender S will generate not just the onion to send on to the first intermediary, but, as a byproduct of forming that onion, will generate all the onion layers — to be precise, the header H_i and the content C_i of the i^{th} onion layer for every i — on the path from himself to the recipient R. However, in the return direction, S is unable to know in advance the content of the onion (otherwise the recipient cannot send a return message); but the sender can still form just the header parts $\{H_i\}$ of those onion layers. So it is the headers that must satisfy CCA2-style non-malleability, while the content accompanying the header can be "mauled" on its way to its destination, be it the recipient R, or, in the case of a reply onion, the original sender S. However, upon arrival to its destination, any "mauled" content should be peeled to $\perp$.

Our main contribution: realizing secure repliable onion encryption. We resolve the problem that CL left open fifteen years ago of constructing provably secure repliable onion encryption. Namely, we give a scheme, which we call *shallot encryption*, for repliable onion encryption. Our scheme is based on a CCA2-secure cryptosystem with tags, a strong PRP (in other words, a block cipher), and a collision-resistant hash function.

In a nutshell, here is how our construction works. As we explained above, we split up the onion into two pieces, the header H and the content C. H contains (in layered encryption form) the routing information and symmetric keys that are needed to process C. C contains the message and, in case this is a forward onion, instructions for forming the reply onion; this part is wrapped in layers of symmetric encryption. This way, the original sender Alice can form the headers for all the layers of the reply onion even though she does not know the contents of the reply in advance; Bob's contribution to the reply onion is just the content C. Each intermediary is responsible for peeling a layer off of H, learning its key k, and applying a strong PRP keyed by k to the contents C. The adaptive security properties guarantee that H cannot be "mauled," but checking the integrity of C is postponed until the onion gets to its destination — recipient Bob or original sender Alice — who check it using a MAC key. This is also why our scheme is called *shallot* encryption: the layered structure of the resulting onion resembles a shallot! (Shallots are a sub-family of onions.) See Section 6 for details.

Related work. Onion routing and mixes were introduced by David Chaum in 1981 [10]. Since then, tremendous interest from applied security researchers resulted in numerous implementations [11, 14, 22, 23].

Tor [14, 17] is the most widely used tool for anonymizing Internet communications; according to statistics shared by the Tor Project (`https://www.torproject.org/`), an estimated two million users use Tor daily. Tor's approach is not, strictly speaking, onion encryption as defined here because no public keys are used for encryption; also, its live connection design is vulnerable to traffic analysis [19, 24, 25].

Despite its practical relevance and widely used implementations, the theoretical foundations of onion routing are somewhat shaky. None of the implementation papers cited above provided definitions or proofs of security. In 2005, Camenisch and Lysyanskaya (CL) provided the first formal definition of secure onion encryption [5]; this was done in Canetti's UC framework [6]. They also gave a game-based definition (onion-security) that they claimed was equivalent to one in the UC model and the first provably secure onion encryption scheme.

CL mentioned the possibility of having a reply option (as did Chaum), but their formal treatment did not extend to it.

In a recent paper, Kuhn et al. [21] found a mistake in CL's game-based definition. In a nutshell, CL's onion-security game proceeded as follows: An adversary attacking an honest participant P is given P's public key and specifies the input to the algorithm for forming an onion. This input includes the identities and public keys of all the intermediaries and the final recipient and the contents of the message m; P is somewhere on the routing path. The challenger either responds with a correctly formed onion or with an onion whose routing path is cut off at P, i.e., for the latter type, P is the recipient of a random unrelated message m'. Kuhn et al. pointed out that, although this property indeed hides where the onion is headed after P, it does not hide where the onion has been before it got to P. Thus, CL's proof that their onion-security definition was sufficient to UC-realize $\mathcal{F}_{\mathsf{onion}}$ had a missing step, which Kuhn et al. found. Kuhn et al. also showed how to use this unfortunate theoretical mistake to attack Sphinx [15]. In addition to pointing out this flaw, Kuhn et al. proposed a new game-based definition that implied the realizability of CL's ideal functionality $\mathcal{F}_{\mathsf{onion}}$. However, they do not tackle repliable onions.

2 Repliable onion encryption: syntax and correctness

Here, we give the formal input/output (I/O) specification for a repliable onion encryption scheme. In contrast to the CL I/O specification for uni-directional onion encryption scheme [5], a repliable onion encryption scheme contains an additional algorithm, $\mathsf{FormReply}$, for forming return onions. This algorithm allows the recipient of a message contained in a repliable onion to respond to the anonymous sender of the message without needing to know who the sender is.

In this paper, an onion O is a pair, consisting of the (encrypted) content C and the header H, i.e., $O = (H, C)$. The maximum length of a path of an onion, be it the forward path or the return path, is N; we assume that N is one of the public parameters pp. The algorithm for forming onions, $\mathsf{FormOnion}$, also takes as one of its parameters, the label ℓ. This is so that when the sender receives a reply message m' along with the label ℓ, the sender can identify to which message m' is responding.

Definition 1 (Repliable onion encryption scheme I/O). *The set* $\Sigma = (G, \mathsf{FormOnion}, \mathsf{ProcOnion}, \mathsf{FormReply})$ *of algorithms satisfies the I/O specification of a* repliable onion encryption scheme *for the label space* $\mathcal{L}(1^\lambda)$*, the message space* $\mathcal{M}(1^\lambda)$*, and a set* $\mathcal{P}$ *of router names if:*

- *G is a probabilistic polynomial-time (p.p.t.) key generation algorithm. On input the security parameter 1^λ (written in unary), the public parameters pp, and the party name P, the algorithm G returns a key pair, i.e.,* $(\mathsf{pk}(P), \mathsf{sk}(P)) \leftarrow G(1^\lambda, \mathsf{pp}, P)$.
- $\mathsf{FormOnion}$ *is a p.p.t. algorithm for forming onions. On input*
  i. a label $\ell \in \mathcal{L}(1^\lambda)$ from the label space,

ii. *a message $m \in \mathcal{M}(1^\lambda)$ from the message space,*
iii. *a forward path $P^{\rightarrow} = (P_1, \ldots, P_d)$ (d stands for destination),*
iv. *the public keys $\mathsf{pk}(P^{\rightarrow})$ associated with the parties in $P^{\rightarrow}$,*
v. *a return path $P^{\leftarrow} = (P_{d+1}, \ldots, P_s)$ (s stands for sender), and*
vi. *the public keys $\mathsf{pk}(P^{\leftarrow})$ associated with the parties in $P^{\leftarrow}$,*

the algorithm FormOnion *returns a sequence $O^{\rightarrow} = (O_1, \ldots, O_d)$ of onions for the forward path, a sequence $H^{\leftarrow} = (H_{d+1}, \ldots, H_s)$ of headers for the return path, and a key κ, i.e., $(O^{\rightarrow}, H^{\leftarrow}, \kappa) \leftarrow$ $\mathsf{FormOnion}(\ell, m, P^{\rightarrow}, \mathsf{pk}(P^{\rightarrow}), P^{\leftarrow}, \mathsf{pk}(P^{\leftarrow}))$. Note: the key κ contains some state information that the sender of the onion might need for future reference; a scheme can still satisfy our definition if $\kappa = \bot$.*

– ProcOnion *is a deterministic polynomial-time (d.p.t.) algorithm for processing onions. On input an onion O, a router name P, and the secret key $\mathsf{sk}(P)$ belonging to P, the algorithm* ProcOnion *returns* (role, output)*, i.e.,* $(\mathsf{role}, \mathsf{output}) \leftarrow \mathsf{ProcOnion}(O, P, \mathsf{sk}(P))$. *When* role = I *(for "intermediary"),* output *is the pair (O', P') consisting of the peeled onion O' and the next destination P' of O'. When* role = R *(for "recipient"),* output *is the message m for recipient P. When* role = S *(for "sender"),* output *is the pair (ℓ, m) consisting of the label ℓ and the reply message m for sender P.*

– FormReply *is a d.p.t. algorithm for replying to an onion. On input a reply message $m \in \mathcal{M}(1^\lambda)$, an onion O, a router name P, and the secret key $\mathsf{sk}(P)$ belonging to P, the algorithm* FormReply *returns the onion O' and the next destination P' of O', i.e.,* $(O', P') \leftarrow \mathsf{FormReply}(m, O, P, \mathsf{sk}(P))$. *Note:* FormReply *may output $(\bot, \bot)$ if P is not the correct recipient of O.*

2.1 Onion evolutions, forward paths, return paths and layerings

Here, we define what it means for a repliable onion encryption scheme to be correct. Before we do this, we first define what onion *evolutions*, *paths*, and *layerings* are; the analogous notions for the unrepliable onion encryption scheme were introduced by Camenisch and Lysyanskaya [5].

Let $\Sigma = (G, \mathsf{FormOnion}, \mathsf{ProcOnion}, \mathsf{FormReply})$ be a repliable onion encryption scheme for the label space $\mathcal{L}(1^\lambda)$, the message space $\mathcal{M}(1^\lambda)$, and the set $\mathcal{P}$ of router names. Let $\mathcal{H} \subseteq \mathcal{P}$ be parties with honestly formed keys. For any $P \notin \mathcal{H}$, let $\mathsf{sk}(P) = \bot$ (i.e., secret keys that were not formed honestly are not well-defined for the purposes of this experiment).

Let $O_1 = (H_1, C_1)$ be an onion received by party $P_1 \in \mathcal{H}$, not necessarily formed using FormOnion.

We define a sequence of onion-location pairs recursively as follows: Let d be the first onion layer of (H_1, C_1) that when peeled, produces either "R" or "S" (if it exists, otherwise $d = \infty$). For all $i \in [d-1]$, let $(\mathsf{role}_{i+1}, ((H_{i+1}, C_{i+1}), P_{i+1})) = \mathsf{ProcOnion}((H_i, C_i), P_i, \mathsf{sk}(P_i))$. Let $s = d$ if peeling (H_d, C_d) produces "S." Otherwise, let $m \in \mathcal{M}(1^\lambda)$ be a reply message from the message space, and let $((H_{d+1}, C_{d+1}), P_{d+1}) = \mathsf{FormReply}(m, (H_d, C_d), P_d, \mathsf{sk}(P_d))$. Let s be the first onion layer of (H_{d+1}, C_{d+1}) that when peeled, produces either "R" or "S" (if it exists, otherwise $s = \infty$). For all $i \in \{d+1, \dots, s-1\}$, let $(\mathsf{role}_{i+1}, ((H_{i+1}, C_{i+1}), P_{i+1})) = \mathsf{ProcOnion}((H_i, C_i), P_i, \mathsf{sk}(P_i))$.

We call the sequence $\mathcal{E}(H_1, C_1, P_1, m) = ((H_1, C_1, P_1), \dots, (H_s, C_s, P_s))$ of onion-location pairs the "*evolution of the onion* (H_1, C_1) *starting at party* P_1 *given* m *as the reply message*." The sequence $\mathcal{P}^{\rightarrow}(H_1, C_1, P_1, m) = (P_1, \dots, P_d)$ is its *forward path*; the sequence $\mathcal{P}^{\leftarrow}(H_1, C_1, P_1, m) = (P_{d+1}, \dots, P_s)$ is its *return path*; and the sequence $\mathcal{L}(H_1, C_1, P_1, m) = (H_1, C_1, \dots, H_d, C_d, H_{d+1}, \dots, H_s)$ is its *layering*.

Definition 2 (Correctness). *Let* G*,* $\mathsf{FormOnion}$*,* $\mathsf{ProcOnion}$*, and* $\mathsf{FormReply}$ *form a repliable onion encryption scheme for the label space* $\mathcal{L}(1^\lambda)$*, the message space* $\mathcal{M}(1^\lambda)$*, and the set* $\mathcal{P}$ *of router names.*

Let N *be the upper bound on the path length (in public parameters* pp*). Let* $P = (P_1, \dots, P_s)$*,* $|P| = s \leq 2N$ *be any list (not containing* $\perp$*) of router names in* $\mathcal{P}$*. Let* $d \in [s]$ *be any index in* $[s]$ *such that* $d \leq N$ *and* $s - d + 1 \leq N$*. Let* $\ell \in \mathcal{L}(1^\lambda)$ *be any label in* $\mathcal{L}(1^\lambda)$*. Let* $m, m' \in \mathcal{M}(1^\lambda)$ *be any two messages in* $\mathcal{M}(1^\lambda)$*.*

For every party P_i *in* P*, let* $(\mathsf{pk}(P_i), \mathsf{sk}(P_i)) \leftarrow G(1^\lambda, \mathsf{pp}, P_i)$ *be* P_i*'s key pair. Let* $P^{\rightarrow} = (P_1, \dots, P_d)$*, and let* $\mathsf{pk}(P^{\rightarrow})$ *be a shorthand for the public keys associated with the parties in* $P^{\rightarrow}$*. Let* $P^{\leftarrow} = (P_{d+1}, \dots, P_s)$*, and let* $\mathsf{pk}(P^{\leftarrow})$ *be a shorthand for the public keys associated with the parties in* $P^{\leftarrow}$*.*

Let $((H_1, C_1), \dots, (H_d, C_d), H_{d+1}, \dots, H_s, \kappa)$ *be the output of* $\mathsf{FormOnion}$ *on input the label* ℓ*, the message* m*, the forward path* $P^{\rightarrow} = (P_1, \dots, P_d)$*, the public keys* $\mathsf{pk}(P^{\rightarrow})$ *associated with the parties in* $P^{\rightarrow}$*, the return path* $P^{\leftarrow} = (P_{d+1}, \dots, P_s)$*, and the public keys* $\mathsf{pk}(P^{\leftarrow})$ *associated with the parties in* $P^{\leftarrow}$*.*

The scheme Σ *is* correct *if with overwhelming probability in the security parameter* λ*,*

i. ***Correct forward path.***
 – $\mathcal{P}^{\rightarrow}(H_1, C_1, P_1, m') = (P_1, \dots, P_d)$.
 – *For every* $i \in [d]$ *and content* C *such that* $|C| = |C_i|$*,* $\mathcal{P}^{\rightarrow}(H_i, C, P_i, m') = (P_i, \dots, P_d)$*.*

ii. ***Correct return path.***
 – $\mathcal{P}^{\leftarrow}(H_1, C_1, P_1, m') = (P_{d+1}, \dots, P_s)$.
 – *For every* $i \in \{d+1, \dots, s\}$*, reply message* m''*, and content* C *such that* $|C| = |C_i|$*,* $\mathcal{P}^{\rightarrow}(H_i, C, P_i, m'') = (P_{d+1}, \dots, P_s)$*.*

iii. ***Correct layering.*** $\mathcal{L}(H_1, C_1, P_1, m') = (H_1, C_1, \dots, H_d, C_d, H_{d+1}, \dots, H_s)$*,*

iv. ***Correct message.*** $\mathsf{ProcOnion}((H_d, C_d), P_d, \mathsf{sk}(P_d)) = (\mathsf{R}, m)$*,*

v. ***Correct reply message.*** $\mathsf{ProcOnion}((H_s, C_s), P_s, \mathsf{sk}(P_s)) = (\mathsf{S}, (\ell, m'))$ *where* (H_s, C_s) *are the header and content of the last onion in the evolution* $\mathcal{E}(H_1, C_1, P_1, m')$*.*

Note that we define onion evolution, (forward and return) paths, and layering so that we can articulate what it means for an onion encryption scheme to be correct. We define correctness to mean that how an onion peels (the evolution, paths, and layerings) exactly reflects the reverse process of how the onion was built up. Thus, for our definition to make sense, both $\mathsf{ProcOnion}$ and $\mathsf{FormReply}$ must be deterministic algorithms.

3 $\mathcal{F}_{\mathsf{ROES}}$: onion routing in the SUC Framework

Here, we provide a formal definition of security for a repliable onion encryption scheme. We chose to define security in the simplified universal composability (SUC) model [7] as opposed to the universal composability (UC) model [6] as this choice greatly simplifies how communication is modeled. Additionally, since SUC-realizability implies UC-realizability [7], we do not lose generality by simplifying the model in this manner. In the SUC model, the environment $\mathcal{Z}$ can communicate directly with each party P by writing inputs into P's input tape and by reading P's output tape. The parties communicate with each other and also with the ideal functionality through an additional party, the router $\mathcal{R}$.

3.1 Ideal functionality $\mathcal{F}_{\mathsf{ROES}}$

In this section, honest parties are capitalized, e.g., P, P_i; and corrupt parties are generally written in lowercase, e.g., p, p_i. An onion formed by an honest party is *honestly formed* and is capitalized, e.g., O, O_i; whereas, an onion formed by a corrupt party is generally written in lowercase, e.g., o, o_i. Recall that an onion O is a pair, consisting of the content C and the header H, i.e., $O = (H, C)$.

How should we define the ideal functionality of a repliable onion encryption scheme? Honestly formed onions in an onion routing protocol should mix at honest nodes. This property is what enables anonymity from the standard adversary who can observe the network traffic on all communication links. Ideally, onions should mix (i) even if the distances from their respective origins or the distances to their respective destinations differ, and (ii) regardless of whether they are forward or return onions. Here, we define the ideal functionality so that a scheme that realizes it necessarily satisfies properties (i) and (ii) above.

Intuitively, onions mix iff onion layers are (computationally) unrelated to each other. Let O' be the onion we get from peeling the onion O. If the values of O and O' are correlated with each other, then O cannot mix with other onions. Conversely, if the values O and O' are unrelated to each other, then O can mix with other onions. However, the adversary necessarily knows how some onions layers are linked together. If the corrupt party p peels onion o, getting peeled onion o', then p knows that o and o' are linked.

Thus, we settle on our idea for an ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ (ROES, for "repliable onion encryption scheme") as follows: Let a *segment* of a routing path be a subpath of the path consisting of a sequence of corrupt parties possibly ending with a single honest party. Note that if there are two consecutive honest parties, (Alice, Bob) on the routing path, then (Bob) is a segment of the path. Each routing path can be uniquely broken up into a sequence $(s_1, \ldots, s_u)$ of non-overlapping segments, such that each segment s_i contains exactly one honest participant, except for the last segment that may end in an adversarial recipient. For $i \neq j$, onion layers corresponding to segment s_i should be computationally unrelated to the layers corresponding to segment s_j.

Thus, the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ forms the onion layers for each segment of a routing path separately and independently from each other. $\mathcal{F}_{\mathsf{ROES}}$ internally keeps tracks of how these layers are linked using two data structures, $\mathsf{OnionDict}$ and $\mathsf{PathDict}$. If $\mathcal{F}_{\mathsf{ROES}}$ forms an onion layer O for Alice (the last party of a segment) that should peel to an onion layer O' for Bob (the first party of the next segment), then it keeps track of this link in $\mathsf{OnionDict}$; the output (O', Bob) is stored under the label (Alice, O). $\mathcal{F}_{\mathsf{ROES}}$ initially forms and stores the onion links only for the forward path and stores the return path in $\mathsf{PathDict}$; onion links for the return path are generated later on when they are needed. To produce the onion layers for a segment, $\mathcal{F}_{\mathsf{ROES}}$ runs the algorithm $\mathsf{SampleOnion}$, which it gets from the ideal adversary $\mathcal{A}$.

Sometimes, the environment $\mathcal{Z}$ instructs an ideal party to process an onion O (or form a reply to an onion O), not stored in either $\mathsf{OnionDict}$ or $\mathsf{PathDict}$. If the header of O is not honestly formed, then $\mathcal{F}_{\mathsf{ROES}}$ processes it according to the algorithm $\mathsf{ProcOnion}$ (or $\mathsf{FormReply}$) supplied by $\mathcal{A}$. Otherwise, if O is the result of "mauling" just the content of an honestly formed onion X that peels to X', then $\mathcal{F}_{\mathsf{ROES}}$ returns the onion O' with the same header as X'. To do this, it runs the algorithm $\mathsf{CompleteOnion}$, also provided by $\mathcal{A}$.

Suppose honest sender Sandy sent an onion to adversarial recipient Robert. Robert responds; eventually an honest intermediary Iris will receive an onion O which contains Robert's response to Sandy. When $\mathcal{F}_{\mathsf{ROES}}$ is called by Iris with onion O, it will be tipped off to the fact that $O = (H, C)$ is a return onion from Robert to Sandy because the header H will be stored in $\mathsf{PathDict}$. At this point, $\mathcal{F}_{\mathsf{ROES}}$ knows what path the onion will have to follow from now on and will be able to create the correct onion layers using $\mathsf{SampleOnion}$ and store them in $\mathsf{OnionDict}$. Once the return onion makes its way to Sandy, Sandy will ask $\mathcal{F}_{\mathsf{ROES}}$ to process it; at this point, $\mathcal{F}_{\mathsf{ROES}}$ will need to know the return message r that Robert sent to Sandy. The algorithm $\mathsf{RecoverReply}$ serves that purpose: from Robert's onion O (received by Iris) it recovers his response r.

So, at setup, the algorithms $\mathsf{ProcOnion}$, $\mathsf{FormReply}$, $\mathsf{SampleOnion}$ $\mathsf{CompleteOnion}$, and $\mathsf{RecoverReply}$ are provided to $\mathcal{F}_{\mathsf{ROES}}$ by $\mathcal{A}$.

Figure 1 gives a summary of the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ for repliable onion encryption. Below, we provide a formal, detailed description of $\mathcal{F}_{\mathsf{ROES}}$.

Setting up. The ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ handles requests from the environment on behalf of the ideal honest parties. During setup, $\mathcal{F}_{\mathsf{ROES}}$ gets the following from the ideal adversary $\mathcal{A}$. For each algorithm in items (iii)-(vi) below, we first describe the input of the algorithm in normal font and then, in italics, provide a brief preview of how the algorithm will be used. $\mathcal{F}_{\mathsf{ROES}}$ only runs for a polynomial number of steps which is specified in the public parameters pp and can time out on running these algorithms from the ideal adversary.

i. The set $\mathcal{P}$ of participants.
ii. The set Bad of corrupt parties in $\mathcal{P}$ (see Remark 4).
iii. The repliable onion encryption scheme's G, $\mathsf{ProcOnion}$, and $\mathsf{FormReply}$ algorithms. *G is used for generating the honest parties' keys.* $\mathsf{ProcOnion}$ *is used for processing onions formed by corrupt parties.* $\mathsf{FormReply}$ *is used for replying to onions formed by corrupt parties.*

```
IdealSetup
1: Get from ideal adversary $\mathcal{A}$:
   $\mathcal{P}$, Bad, $G$, ProcOnion,
   FormReply, SampleOnion,
   CompleteOnion, RecoverReply.
2: Initialize dictionaries OnionDict
   and PathDict.
```

```
IdealFormOnion($\ell, m, P^{\rightarrow}, P^{\leftarrow}$)
1: Break forward path into
   segments.
2: Run SampleOnion on segments
   to generate onion layers.
3: Store onion layers in OnionDict.
4: Store label $\ell$ and (rest of) return
   path in PathDict.
```

```
IdealProcOnion($(H, C), P$)
1: If $(P, H)$ is "familiar," i.e., stored in one of our
   dictionaries
     - If $(P, H, C)$ in OnionDict, return next stored
       onion layer.
     - Else if exists $(P, H, (X \neq C))$ in OnionDict,
       return output of CompleteOnion and stored
       next party (if stored next party exists), or "$\perp$"
       (if next party doesn't exist).
     - Else if $(P, H, \star)$ in PathDict, return output of
       IdealFormOnion on message recovered using
       RecoverReply and label and path stored in
       PathDict.
2: Else if $(P, H)$ is not familiar, return output of
   ProcOnion($(H, C), P, \mathsf{sk}(P)$).
```

```
IdealFormReply($m, (H, C), P$)
1: If $(P, H, C)$ in PathDict, return output of
   IdealFormOnion on $m$ and label and path stored in
   PathDict.
2: Else, return output of
   FormReply($m, (H, C), P, \mathsf{sk}(P)$).
```

Fig. 1: Summary of ideal functionality $\mathcal{F}_{\mathsf{ROES}}$.

iv. The p.p.t. algorithm SampleOnion$(1^\lambda, \mathsf{pp}, p^{\rightarrow}, p^{\leftarrow}, m)$ that takes as input the security parameter 1^λ, the public parameters pp, the forward path $p^{\rightarrow}$, the (possibly empty) return path $p^{\leftarrow}$, and the (possibly empty) message m. The routing path $(p^{\rightarrow}, p^{\leftarrow}) = (p_1, \dots, p_i, P_{i+1})$ is always a sequence $(p_1, \dots, p_i)$ of adversarial parties, possibly ending in an honest party P_{i+1}. $\mathcal{F}_{\mathsf{ROES}}$ fails if SampleOnion ever samples a repeating header or key.

SampleOnion *is used to compute an onion to send to p_1 which will be "peelable" all the way to an onion for P_{i+1}. If the return path $p^{\leftarrow}$ is non-empty and ends in an honest party P_{i+1},* SampleOnion *produces an onion o for the first party p_1 in $p^{\rightarrow}$ and a header H for the last party P_{i+1} in $p^{\leftarrow}$. Else if the return path $p^{\leftarrow}$ is empty, and the forward path $p^{\rightarrow}$ ends in an honest party P_{i+1},* SampleOnion *produces an onion o for the first party p_1 in $p^{\rightarrow}$ and an onion O for the last party P_{i+1} in $p^{\rightarrow}$. Else if the return path $p^{\leftarrow}$ is empty, and the forward path $p^{\rightarrow}$ ends in a corrupt party p_i,* SampleOnion *produces an onion o for the first party p_1 in $p^{\rightarrow}$.*

v. The p.p.t. algorithm CompleteOnion$(1^\lambda, \mathsf{pp}, H', C)$ that takes as input 1^λ, pp, the the party P, the header H', and the content C, and outputs an onion $O = (H', C')$. $\mathcal{F}_{\mathsf{ROES}}$ fails if CompleteOnion ever produces a repeating onion. CompleteOnion *produces an onion (H', C') that resembles the result of peeling an onion with content C.*

vi. The d.p.t. algorithm RecoverReply$(1^\lambda, \mathsf{pp}, O, P)$ that takes as input 1^λ, pp, the onion O, and the party P, and outputs a label ℓ and a reply message m. *This algorithm is used for recovering the label ℓ and reply message m from* *the return onion O that carries the response from a corrupt recipient to an honest sender.*

Let sid denote the session id specific the parameters that the setup, above, creates. $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ denotes the session of $\mathcal{F}_{\mathsf{ROES}}$ with this sid. $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ generates a public key pair $(\mathsf{pk}(P), \mathsf{sk}(P))$ for each honest party $P \in \mathcal{P} \setminus \mathsf{Bad}$ using the key generation algorithm G and sends the public keys to their respective party. (If working within the global PKI framework, each party then relays his/her key to the global bulletin board functionality [9].) $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ also creates the following (initially empty) dictionaries:

- The *onion dictionary* $\mathsf{OnionDict}$ supports:
  - A method $\mathsf{put}((P, H, C), (\mathsf{role}, \mathsf{output}))$ that stores under the label (P, H, C): the role "role" and the output "output." *Should participant P later direct $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ to process onion $O = (H, C)$, it will receive the values* $(\mathsf{role}, \mathsf{output})$ *stored in* $\mathsf{OnionDict}$ *corresponding to (P, H, C).*
  - A method $\mathsf{lookup}(P, H, C)$ that looks up the entry $(\mathsf{role}, \mathsf{output})$ corresponding to the label (P, H, C). *This method will be used when P directs $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ to process onion $O = (H, C)$.*
- The *return path dictionary* $\mathsf{PathDict}$ supports:
  - A method $\mathsf{put}((P, H, C), (P^{\leftarrow}, \ell))$ that stores under the label (P, H, C): the return path $P^{\leftarrow}$ and the label ℓ. *This method is used to store the return path $P^{\leftarrow}$ for the onion corresponding to label ℓ.*
  - A method $\mathsf{lookup}(P, H, C)$ that looks up entry $(P^{\leftarrow}, \ell)$ corresp. to the label (P, H, C). *Should P later direct $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ to either reply to the onion (H, C) or to process an onion with header H, the stored return path $P^{\leftarrow}$ and label ℓ will be used to form the rest of the return onion layers.*

These data structures are stored internally at and accessible only by $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$.

Forming an onion. After setup, the environment $\mathcal{Z}$ can instruct an honest party P to form an onion using the session id sid, the label ℓ, the message m, the forward path $P^{\rightarrow}$, and the return path $P^{\leftarrow}$. To form the onion, P forwards the instruction from $\mathcal{Z}$ to $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ (via the router $\mathcal{R}$).

The goal of the ideal functionality $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ is to create and maintain state information for handling an onion O (the response to the "form onion" request). O should be "peelable" by the parties in the forward path $P^{\rightarrow}$, internally associated with the return path $P^{\leftarrow}$, and for the purpose of realizing this functionality by an onion encryption scheme, each layer of the onion should look "believable" as onions produced from running $\mathsf{FormOnion}$, $\mathsf{ProcOnion}$, or $\mathsf{FormReply}$. Importantly, O and its onion layers should reveal no information to $\mathcal{A}$:

- Each onion routed to an honest party P_i is formed initially with just (P_i) as the routing path and, therefore, reveals only that it is for P_i. When forming the onion, no message is part of the input; this ensures that the onion is information-theoretically independent of any message m.
- For every party p_i or P_i in the forward path, let $\mathsf{next}(i)$ denote the index of the next honest party $P_{\mathsf{next}(i)}$ following p_i. For example, if the forward path is $(P_1, p_2, p_3, P_4, P_5, p_6, p_7)$, then $\mathsf{next}(2) = 4$.

Conceptually, each onion routed to an adversarial party p_i is formed by "wrapping" an onion layer for each corrupt party in $(p_i, \ldots, p_{\mathsf{next}(i)-1})$ (or $(p_{i+1}, \ldots, p_{|P^{\rightarrow}|})$ if no honest party after p_i exists) around an onion formed for an honest party $P_{\mathsf{next}(i)}$ (or a message if no honest party after p_i exists). This reveals at most the sequence $(p_i, \ldots, p_{\mathsf{next}(i)-1}, P_{\mathsf{next}(i)})$ (or the sequence $(p_i, \ldots, p_{|P^{\rightarrow}|})$ and the message m if no honest party after p_i exists). How this wrapping occurs depends on the internals of the $\mathsf{SampleOnion}$ algorithm provided by the ideal adversary.

To ensure these, $\mathcal{F}_{\mathsf{ROES}}$ partitions the forward path $P^{\rightarrow}$ into segments:

Let P_f (f, for first) be the first honest party in the forward path. The first couple of segments are $(p_1, \ldots, p_{f-1}, P_f)$, $(p_{f+1}, \ldots, p_{\mathsf{next}(f)-1}, P_{\mathsf{next}(f)})$, etc.

For each segment $(p_i, \ldots, p_{j-1}, P_j)$, the ideal functionality $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ samples onions (h_i, c_i) and (H_j, C_j) using the algorithm $\mathsf{SampleOnion}$, i.e., $((h_i, c_i), (H_j, C_j)) \leftarrow \mathsf{SampleOnion}(1^{\lambda}, \mathsf{pp}, (p_i, \ldots, p_{j-1}, P_j), (), \perp)$. As we explained when introducing the $\mathsf{SampleOnion}$ input/output structure, (h_i, c_i) is the onion that is intended for the participant $p_i \in \mathsf{Bad}$; once the adversarial participants take turns peeling it, the innermost layer (H_j, C_j) can be processed by the honest participant P_j.

If the recipient P_d is honest, this process will create all the onions in the forward direction. Suppose that the recipient p_d is corrupt. Let P_e (e, for end) be the last honest party in the forward path $P^{\rightarrow}$, and let $P_{\mathsf{next}(d)}$ denote the first honest party in the return path $P^{\leftarrow}$. $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ also runs $\mathsf{SampleOnion}(1^{\lambda}, \mathsf{pp}, (p_{e+1}, \ldots, p_d), (p_{d+1}, \ldots, p_{\mathsf{next}(d)-1}, P_{\mathsf{next}(d)}), m)$; as we explained when introducing the $\mathsf{SampleOnion}$ input/output structure, this produces an onion o_{e+1} and a header $H_{\mathsf{next}(d)}$.

For every honest intermediary party P_i in the forward path, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ stores under the label (P_i, H_i, C_i) in the onion dictionary $\mathsf{OnionDict}$ the role "I," the $(i+1)^{st}$ onion layer (H_{i+1}, C_{i+1}), and destination P_{i+1}. The $(d+1)^{st}$ onion layer doesn't exist for the innermost layer (H_d, C_d) for an honest recipient P_d. In this case, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ stores just the role "R" and the message m.

If the recipient P_d is honest, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ stores the entry $((P_d, H_d, C_d), (P^{\leftarrow}, \ell))$ in the dictionary $\mathsf{PathDict}$. Otherwise if the recipient p_d is corrupt, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ stores the entry $((P_{\mathsf{next}(d)}, H_{\mathsf{next}(d)}, *), (p^{\leftarrow}, \ell))$ in $\mathsf{PathDict}$ where $p^{\leftarrow} = (p_{\mathsf{next}(d)+1}, \ldots, P_s)$. "$*$" is the unique symbol that means "any content."

Example. The recipient P_7 is honest. The forward path is $P^{\rightarrow} = (P_1, p_2, p_3, P_4, P_5, p_6, \boxed{P_7})$, and the return path is $P^{\leftarrow} = (p_8, p_9, P_{10}, p_{11}, P_{12})$. In this case, the first segment is (P_1), and the second segment is (p_2, p_3, P_4) and so on; and

$$\begin{aligned}
(\perp, (H_1, C_1)) &\leftarrow \mathsf{SampleOnion}(1^{\lambda}, \mathsf{pp}, (P_1), (), \perp) \\
((h_2, c_2), (H_4, C_4)) &\leftarrow \mathsf{SampleOnion}(1^{\lambda}, \mathsf{pp}, (p_2, p_3, P_4), (), \perp) \\
(\perp, (H_5, C_5)) &\leftarrow \mathsf{SampleOnion}(1^{\lambda}, \mathsf{pp}, (P_5), (), \perp) \\
((h_6, c_6), (H_7, C_7)) &\leftarrow \mathsf{SampleOnion}(1^{\lambda}, \mathsf{pp}, (p_6, P_7), (), \perp).
\end{aligned}$$

$\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ stores in $\mathsf{OnionDict}$ and $\mathsf{PathDict}$:

$$\begin{aligned}
&\mathsf{OnionDict.put}((P_1, H_1, C_1), (\mathsf{I}, ((h_2, c_2), p_2)))\\
&\mathsf{OnionDict.put}((P_4, H_4, C_4), (\mathsf{I}, ((H_5, C_5), P_5)))\\
&\mathsf{OnionDict.put}((P_5, H_5, C_5), (\mathsf{I}, ((h_6, c_6), p_6)))\\
&\mathsf{OnionDict.put}((P_7, H_7, C_7), (\mathsf{R}, m)),\\
&\mathsf{PathDict.put}((P_7, H_7, C_7), ((p_8, p_9, P_{10}, p_{11}, P_{12}), \ell)).
\end{aligned}$$

After updating $\mathsf{OnionDict}$ and $\mathsf{PathDict}$, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ returns the first onion $O_1 = (H_1, C_1)$ to party P (via the router $\mathcal{R}$). Upon receiving O_1 from $\mathcal{F}$, P outputs the session id sid and O_1.

Processing an onion. After setup, the environment $\mathcal{Z}$ can instruct an honest party P to process an onion $O = (H, C)$ for the session id sid. To process the onion, party P forwards the instruction to the ideal functionality $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ (via the router $\mathcal{R}$).

Case 1: There is an entry $(\mathsf{role}, \mathsf{output})$ under the label (P, H, C) in $\mathsf{OnionDict}$. In this case, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ responds to P (via the router $\mathcal{R}$) with $(\mathsf{role}, \mathsf{output})$.

Case 2: There is no entry under the label (P, H, C) in $\mathsf{OnionDict}$, but there exists $X \neq C$ such that there is an entry $(\mathsf{I}, ((H', X'), P'))$ under the label (P, H, X) in $\mathsf{OnionDict}$. This means that, P has received an onion with a properly formed header, but an improperly formed content. This is where we use the algorithm $\mathsf{CompleteOnion}$ to direct $\mathcal{F}_{\mathsf{ROES}}$ how to peel this "mauled" onion. Recall that $\mathsf{CompleteOnion}$ was provided by the adversary at setup. $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ uses it to sample an onion $(H', C') \leftarrow \mathsf{CompleteOnion}(1^\lambda, \mathsf{pp}, H', C)$. $\mathcal{F}_{\mathsf{ROES}}$ then stores the new entry $(\mathsf{I}, ((H', C'), P'))$ under the label (P, H, C) in $\mathsf{OnionDict}$, and responds to P with $(\mathsf{I}, ((H', C'), P'))$.

Case 3: There is no entry under the label (P, H, C) in $\mathsf{OnionDict}$, but there exists $X \neq C$ such that there is an entry (R, m) under the label (P, H, X) in $\mathsf{OnionDict}$. This means that P is the intended recipient of the onion (H, X) but instead just received the properly formed header H with "mauled" content C. In this case, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ responds to P with $(\mathsf{R}, \perp)$.

Case 4: There is no entry under the label (P, H, C) in $\mathsf{OnionDict}$, but there exists $X \neq C$ such that there is an entry $(\mathsf{S}, (\ell, m))$ under the label (P, H, X) in $\mathsf{OnionDict}$. This means that P was the original sender of an onion, and header H is the correct header for his reply onion; but the content C got "mauled" in transit: the correct reply onion was supposed to have content X (according to in $\mathsf{OnionDict}$). $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ responds to P with $(\mathsf{S}, \perp)$.

Case 5: There is no entry starting with (P, H) in $\mathsf{OnionDict}$, but there is an entry $(P^{\leftarrow}, \ell)$ under the label $(P, H, *)$ in $\mathsf{PathDict}$. This means that P is the first honest intermediary on the return path of an onion whose recipient was adversarial. $\mathcal{F}_{\mathsf{ROES}}$ needs to compute the reply message m' that the adversarial recipient meant to send back to the honest sender. This is the purpose of the $\mathsf{RecoverReply}$ algorithm that the adversary provides to $\mathcal{F}_{\mathsf{ROES}}$ at setup. Let m' be the message obtained by running $\mathsf{RecoverReply}(1^\lambda, \mathsf{pp}, O, P)$.

Next, $\mathcal{F}_{\mathsf{ROES}}$ computes the layers of the reply onion. If $P^{\leftarrow}$ is not empty, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ runs its "form onion" code (Section 3.1) with (ℓ, m') as the "message," $P^{\leftarrow}$ as the forward path, and the empty list "()" as the return path. (The code is run with auxiliary information for correctly labeling the last party in $P^{\leftarrow}$ as the sender.) $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ responds to P with $(\mathsf{I}, ((H', C'), P'))$, where (H', C') is the returned onion, and P' is the first party in $P^{\leftarrow}$.

Otherwise if $P^{\leftarrow}$ is empty, then P is the recipient of the return onion, so $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ responds to P with $(\mathsf{S}, (\ell, m'))$.

Case 6: $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ doesn't know how to peel O (i.e., there is no entry starting with (P, H) in $\mathsf{OnionDict}$ and no entry under $(P, H, *)$ in $\mathsf{PathDict}$). In this case, O does not have an honestly formed header; so, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ responds to P with $(\mathsf{role}, \mathsf{output}) = \mathsf{ProcOnion}(1^\lambda, \mathsf{pp}, O, P, \mathsf{sk}(P))$ (recall that $\mathsf{ProcOnion}$ is an algorithm supplied by the ideal adversary at setup).

The cases above cover all the possibilities. Upon receiving the response $(\mathsf{role}, \mathsf{output})$ from $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$, P outputs the session id sid and $(\mathsf{role}, \mathsf{output})$.

Forming a reply. After setup, the environment $\mathcal{Z}$ can instruct an honest party P to form a reply using the session id sid, the reply message m, and an onion $O = (H, C)$. To form the return onion, P forwards the instruction to the ideal functionality $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ (via the router $\mathcal{R}$).

Case 1: There is an entry $(P^{\leftarrow}, \ell)$ under the label (P, H, C) in $\mathsf{PathDict}$. Then $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ runs its "form onion" code (see Section 3.1) with (ℓ, m) as the "message," $P^{\leftarrow}$ as the forward path, and the empty list "()" as the return path. (The code is run with auxiliary information for correctly labeling the last party in $P^{\leftarrow}$ as the sender.) $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ responds to P (via the router $\mathcal{R}$) with the returned onion O' and the first party P' in $P^{\leftarrow}$.

Case 2: No entry exists for (P, H, C) in $\mathsf{PathDict}$. Then P is replying to an onion formed by an adversarial party, so $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ replies to P with $(O', P') = \mathsf{FormReply}(1^\lambda, \mathsf{pp}, m, O, P, \mathsf{sk}(P))$. Upon receiving the response (O', P') from $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$, P outputs the session id sid and (O', P').

3.2 SUC-realizability of $\mathcal{F}_{\mathsf{ROES}}$

Recall what it means for a cryptographic scheme to SUC-realize $\mathcal{F}_{\mathsf{ROES}}$ [7].

Ideal protocol. In the ideal onion routing protocol, the environment $\mathcal{Z}$ interacts with the participants by writing instructions into the participants' input tapes and reading their output tapes. Each input is an instruction to form an onion, process an onion, or form a return onion. When an honest party P receives an instruction from $\mathcal{Z}$, it forwards the instruction to $\mathcal{F}_{\mathsf{ROES}}$ via the router $\mathcal{R}$. Upon receiving a response from $\mathcal{F}_{\mathsf{ROES}}$ (via $\mathcal{R}$), P outputs the response. Corrupt parties are controlled by the adversary $\mathcal{A}$ and behave according to $\mathcal{A}$. $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ does not interact with $\mathcal{A}$ after the setup phase. At the end of the protocol execution, $\mathcal{Z}$ outputs a bit b. Let $\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{ROES}}, \mathcal{A}, \mathcal{Z}}(1^\lambda, \mathsf{pp})$ denote $\mathcal{Z}$'s output after executing the ideal protocol for security parameter 1^λ and public parameters pp.

Real protocol. Let Σ be a repliable onion encryption scheme. The real onion routing protocol for Σ is the same as the ideal one (described above), except that the honest parties simply run Σ's algorithms to form and process onions. Let $\mathsf{REAL}_{\Sigma,\mathcal{A},\mathcal{Z}}(1^\lambda, \mathsf{pp})$ denote $\mathcal{Z}$'s output after executing the real protocol.

Definition 3 (SUC-realizability of $\mathcal{F}_{\mathsf{ROES}}$). *The repliable onion encryption scheme Σ* SUC-realizes *the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ if for every p.p.t. real-model adversary $\mathcal{A}$, there exists a p.p.t. ideal-model adversary $\mathcal{S}$ s.t. for every polynomial-time environment $\mathcal{Z}$, there exists a negligible function $\nu(\lambda)$ s.t.* $\left|\Pr\left[\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{ROES}},\mathcal{S},\mathcal{Z}}(1^\lambda, \mathsf{pp}) = 1\right] - \Pr\left[\mathsf{REAL}_{\Sigma,\mathcal{A},\mathcal{Z}}(1^\lambda, \mathsf{pp}) = 1\right]\right| \leq \nu(\lambda)$.

Remark 4. The set Bad of corrupted participants is selected non-adaptively, at setup time. Adaptive security is notoriously challenging to realize in the standard model for public-key encryption: As Canetti et al. [8] demonstrated, adaptively secure encryption requires non-committing encryption. A single-layer onion is already a public-key ciphertext, so any reasonable formulation of an onion routing ideal functionality would imply public-key encryption and, thus, would also require non-committing encryption. Non-committing encryption in the standard model requires that public keys can only be used once for a single ciphertext and, thus, is impossible in the standard PKI model. It is possible to realize it in the random-oracle (RO) model, and so in the RO model, adaptively secure onion routing may be possible. We leave this for future work, however.

Remark 5. In describing the ideal functionality, we made an implicit assumption that for every instruction to form an onion, the keys match the parties on the routing path. However, generally speaking, the environment $\mathcal{Z}$ can instruct an honest party to form an onion using the wrong keys for some of the parties on the routing path. Using the dictionary $\mathsf{OnionDict}$, it is easy to extend our ideal functionality to cover this case: the ideal functionality would store in $\mathsf{OnionDict}$, every onion layer for an honest party, starting from the outermost layer, until it reaches a layer with a mismatched key. To keep the exposition clean, we will continue to assume that router names are valid, and keys are as published.

Remark 6. As originally noted by Camenisch and Lysyanskaya [5], the environment is allowed to repeat the same input (e.g., a "process onion" request) in the UC framework (likewise, in the SUC framework). Thus, replay attacks are not only allowed in our model but inherent in the SUC framework. The reason that replay attacks are a concern is that they allow the adversary to observe what happens in the network as a result of repeatedly sending an onion over and over again — which intermediaries are involved, etc — and that potentially allows the adversary to trace this onion. Our functionality does not protect from this attack (and neither did the CL functionality), but a higher-level protocol can address this by directing parties to ignore repeat "process onion" and "form reply" requests. Other avenues to address this (which can be added to our functionality, but we chose not to so as not to complicate it further) may include letting onions time out, so the time frame for repeating them could be limited.

Remark 7. The way that an ideal adversarial participant interacts with $\mathcal{F}_{\mathsf{ROES}}$ to form an onion is by creating an onion in any way it wants and sending it over the network (which, in the SUC model, is controlled by the environment that writes to the participants' input tapes and reads their output tapes) to an ideal honest participant, who then calls $\mathcal{F}_{\mathsf{ROES}}$ to process it. When an ideal honest party is a recipient of such an onion and replies to it with response r, this falls under case (2) of the IdealFormReply interface of $\mathcal{F}_{\mathsf{ROES}}$. The resulting onion is returned to the ideal honest party who then puts it on its output tape, to be read by the environment, who, depending on the algorithm FormReply, immediately learns r without having to route the onion through the network. Thus, $\mathcal{F}_{\mathsf{ROES}}$ itself does not need any additional interfaces to interact with an ideal adversarial participant.

4 Repliable-onion security: a game-based definition

In the previous section, we gave a detailed description of an ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ of repliable onion encryption in the SUC model. However, given the complexity of the description, proving that an onion encryption scheme realizes $\mathcal{F}_{\mathsf{ROES}}$ seems onerous. To address this, we provide an alternative, game-based definition of security that implies realizability of $\mathcal{F}_{\mathsf{ROES}}$. We call this definition, *repliable-onion security.*

Informally, our repliable-onion security requires that the following three properties hold: (a) No adversary can tell (with a non-negligible advantage over random guessing) whether an honest transmitter of an honestly formed onion is the sender of the onion or an intermediary on the forward path. (b) Given an honestly formed onion O received by the recipient, no adversary can tell (with non-negligible advantage) whether the recipient is replying to O or sending an onion unrelated to O. (c) No adversary can tell (with non-negligible advantage) whether an honest transmitter of an honestly formed onion is the sender of the onion or an intermediary on the return path.

We formalize each of these three security properties by defining three corresponding security games. In each game, the adversary is given oracles for processing onions on behalf of the honest parties under attack. The adversary also selects additional inputs of each game, such as the identities of intermediaries, the message conveyed by the onion, etc.

In Figure 2, we give the high-level description of the game ROSecurityGame and its three variants: (a), (b), and (c). The variants differ only in steps 4 and 5.

Formal description of ROSecurityGame variant (a). We now expand on what we described in Figure 2 and provide a formal, detailed description of ROSecurityGame for the first variant, (a).

$\mathsf{ROSecurityGame}(1^\lambda, \Sigma, \mathsf{CompleteOnion}, \mathcal{A})$ is parametrized by the security parameter 1^λ, the repliable onion encryption scheme $\Sigma = (G, \mathsf{FormOnion}, \mathsf{ProcOnion}, \mathsf{FormReply})$, the p.p.t. algorithm CompleteOnion, and the adversary $\mathcal{A}$.

1: $\mathcal{A}$ picks honest parties' router names I and S. I is the honest intermediary router under attack, while S is the honest sender under attack.
2: $\mathcal{C}$ sets keys for honest parties I and S.
3: $\mathcal{A}$ gets access to oracles—$\mathcal{O}.\mathsf{PO_I}$, $\mathcal{O}.\mathsf{FR_I}$, $\mathcal{O}.\mathsf{PO_S}$, and $\mathcal{O}.\mathsf{FR_S}$— for processing onions and replying to them on behalf of I and S.
4: $\mathcal{A}$ provides input for the challenge onion: a label ℓ, a message m, a forward path $P^{\rightarrow} = (P_1, \ldots, P_d)$, a return path $P^{\leftarrow} = (P_{d+1}, \ldots, P_s)$, and keys associated with the routing path $(P^{\rightarrow}, P^{\leftarrow})$. If the return path is non-empty, it ends with S so that $P_s = S$. I appears somewhere on the routing path so that P_j is the first appearance of I on the path. The location of P_j determines which variant of the security game the adversary is playing:
   (a) P_j is an intermediary on the forward path (i.e., $j < d$),
   (b) P_j is the recipient (i.e., $j = d$) or
   (c) P_j is on the return path (i.e., $j > d$).
5: $\mathcal{C}$ flips a coin $b \leftarrow\$ \{0, 1\}$. If $b = 0$, $\mathcal{C}$ forms the onion O as specified by $\mathcal{A}$. If $b = 1$, $\mathcal{C}$ forms the onion O with a "switch" at I and modifies ("rigs") the oracles accordingly.
   (a) To peel the challenge onion O on behalf of forward-path intermediary I, $\mathcal{O}.\mathsf{PO_I}$ will form (in answer to a query from $\mathcal{A}$) a new onion using the remainder of the routing path. To peel an onion $O' \neq O$ with the same header as the challenge onion, $\mathcal{O}.\mathsf{PO_I}$ uses the algorithm $\mathsf{CompleteOnion}$.
   (b) To form a reply to the challenge onion O on behalf of I, $\mathcal{O}.\mathsf{FR_I}$ will form a new onion using the return path as the forward path (and the empty return path).
   (c) To peel the challenge onion O on behalf of the return-path intermediary I, $\mathcal{O}.\mathsf{PO_I}$ will form a new onion using the remainder of the return path as the forward path (and the empty return path).
6: $\mathcal{A}$ once again gets oracle access to $\mathcal{O}.\mathsf{PO_I}$, $\mathcal{O}.\mathsf{FR_I}$, $\mathcal{O}.\mathsf{PO_S}$, and $\mathcal{O}.\mathsf{FR_S}$.
7: $\mathcal{A}$ guesses b' and wins if $b' = b$.

Fig. 2: Summary of the repliable onion security game, $\mathsf{ROSecurityGame}$. The parameters of the game are the security parameter λ, the repliable onion encryption scheme Σ, the p.p.t. algorithm $\mathsf{CompleteOnion}$ and the adversary $\mathcal{A}$.

1. The adversary $\mathcal{A}$ picks two router names $I, S \in \mathcal{P}$ ("I" for intermediary and "S" for sender) and sends them to the challenger $\mathcal{C}$.
2. The challenger $\mathcal{C}$ generates key pairs $(\mathsf{pk}(I), \mathsf{sk}(I))$ and $(\mathsf{pk}(S), \mathsf{sk}(S))$ for I and S using the key generation algorithm G and sends the public keys $(\mathsf{pk}(I), \mathsf{pk}(S))$ to $\mathcal{A}$.
3. $\mathcal{A}$ is given oracle access to (i) $\mathcal{O}.\mathsf{PO_I}(\cdot)$, (ii) $\mathcal{O}.\mathsf{FR_I}(\cdot,\cdot)$, (iii) $\mathcal{O}.\mathsf{PO_S}(\cdot)$, and (iv) $\mathcal{O}.\mathsf{FR_S}(\cdot,\cdot)$ where
   i-ii. $\mathcal{O}.\mathsf{PO_I}(\cdot)$ and $\mathcal{O}.\mathsf{FR_I}(\cdot,\cdot)$ are, respectively, the oracle for answering "process onion" requests made to honest party I and the oracle for answering "form reply" requests made to I.
   iii-iv. $\mathcal{O}.\mathsf{PO_S}(\cdot)$ and $\mathcal{O}.\mathsf{FR_S}(\cdot,\cdot)$ are, respectively, the oracle for answering "process onion" requests made to honest party S and the oracle for answering "form reply" requests made to S

   . Since $\mathsf{ProcOnion}$ and $\mathsf{FormReply}$ are deterministic algorithms, WLOG, the oracles don't respond to repeating queries.
4. $\mathcal{A}$ chooses a label $\ell \in \mathcal{L}(1^\lambda)$ and a message $m \in \mathcal{M}(1^\lambda)$. $\mathcal{A}$ also chooses names of participants on a forward path $P^{\rightarrow} = (P_1, \ldots, P_d)$, and a return path $P^{\leftarrow} = (P_{d+1}, \ldots, P_s)$ such that (i) if $P^{\leftarrow}$ is non-empty, then $P_s = S$,

and (ii) I appears somewhere on $P^{\rightarrow}$ before the recipient. For each $P_i \notin \{S, I\}$, $\mathcal{A}$ also chooses its public key $\mathsf{pk}(P_i)$. $\mathcal{A}$ sends to $\mathcal{C}$ the parameters for the challenge onion: ℓ, m, $P^{\rightarrow}$, the public keys $\mathsf{pk}(P^{\rightarrow})$ of the parties in $P^{\rightarrow}$, $P^{\leftarrow}$ and the public keys $\mathsf{pk}(P^{\leftarrow})$ of the parties in $P^{\leftarrow}$.

5. $\mathcal{C}$ samples a bit $b \leftarrow_{\$} \{0,1\}$.

   If $b = 0$, $\mathcal{C}$ runs $\mathsf{FormOnion}$ on the parameters specified by $\mathcal{A}$, i.e., $((O_1^0, \dots, O_d^0), H^{\leftarrow}, \kappa) \leftarrow \mathsf{FormOnion}(\ell, m, P^{\rightarrow}, \mathsf{pk}(P^{\rightarrow}), P^{\leftarrow}, \mathsf{pk}(P^{\leftarrow}))$.
   In this case, the oracles—$\mathcal{O}.\mathsf{PO}_\mathsf{I}(\cdot)$, $\mathcal{O}.\mathsf{FR}_\mathsf{I}(\cdot,\cdot)$, $\mathcal{O}.\mathsf{PO}_\mathsf{S}(\cdot)$, and $\mathcal{O}.\mathsf{FR}_\mathsf{S}(\cdot,\cdot)$—remain unmodified.

   Otherwise, if $b = 1$, $\mathcal{C}$ performs the "switch" at honest party P_j on the forward path $P^{\rightarrow}$, where P_j is the first appearance of I on the forward path. $\mathcal{C}$ runs $\mathsf{FormOnion}$ twice. First, $\mathcal{C}$ runs it on input a random label $x \leftarrow_{\$} \mathcal{L}(1^\lambda)$, a random message $y \leftarrow_{\$} \mathcal{M}(1^\lambda)$, the "truncated" forward path $p^{\rightarrow} = (P_1, \dots, P_j)$, and the empty return path "(),"
   i.e., $((O_1^1, \dots, O_j^1), (), \kappa) \leftarrow \mathsf{FormOnion}(x, y, p^{\rightarrow}, \mathsf{pk}(p^{\rightarrow}), (), ())$. $\mathcal{C}$ then runs $\mathsf{FormOnion}$ on a random label $x' \leftarrow_{\$} \mathcal{L}(1^\lambda)$, the message m (that had been chosen by $\mathcal{A}$ in step 4), the remainder $q^{\rightarrow} = (P_{j+1}, \dots, P_d)$ of the forward path, and the return path $P^{\leftarrow}$, i.e., $((O_{j+1}^1, \dots, O_d^1), H^{\leftarrow}, \kappa') \leftarrow \mathsf{FormOnion}(x', m, q^{\rightarrow}, \mathsf{pk}(q^{\rightarrow}), P^{\leftarrow}, \mathsf{pk}(P^{\leftarrow}))$,

   We modify the oracles as follows. Let $O_j^1 = (H_j^1, C_j^1)$ and $O_{j+1}^1 = (H_{j+1}^1, C_{j+1}^1)$, and let H_s^1 be the last header in $H^{\leftarrow}$. $\mathcal{O}.\mathsf{PO}_\mathsf{I}$ does the following to "process" an onion $O = (H, C)$:
   i. If $O = O_j^1$ and $\mathsf{ProcOnion}(O, P_j, \mathsf{sk}(P_j)) = (\mathsf{R}, y)$, then return $(\mathsf{I}, (O_{j+1}^1, P_{j+1}))$.
   ii. If $O = O_j^1$ and $\mathsf{ProcOnion}(O, P_j, \mathsf{sk}(P_j)) \neq (\mathsf{R}, y)$, then fail.
   iii. If $O \neq O_j^1$ but $H = H_j^1$ and $\mathsf{ProcOnion}(O, P_j, \mathsf{sk}(P_j)) = (\mathsf{R}, \perp)$, then return $(\mathsf{I}, ((H_{j+1}^1, \mathsf{CompleteOnion}(H_{j+1}^1, C)), P_{j+1}))$.
   iv. If $O \neq O_j^1$ but $H = H_j^1$ and $\mathsf{ProcOnion}(O, P_j, \mathsf{sk}(P_j)) \neq (\mathsf{R}, \perp)$, then fail.
   $\mathcal{O}.\mathsf{PO}_\mathsf{S}$ does the following to "process" an onion O:
   v. If the header of O is H_s^1 and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) = (\mathsf{R}, m')$ for some message $m' \neq \perp$, then return $(\mathsf{S}, (\ell, m'))$.
   vi. If the header of O is H_s^1 and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) = (\mathsf{R}, \perp)$, then return $(\mathsf{S}, \perp)$.
   vii. If the header of O is H_s^1 and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) \neq (\mathsf{R}, m')$ for any message m', then fail.
   All other queries are processed as before.
   $\mathcal{C}$ sends to $\mathcal{A}$, the first onion O_1^b in the output of $\mathsf{FormOnion}$.
6. $\mathcal{A}$ submits a polynomially-bounded number of (adaptively chosen) queries to oracles $\mathcal{O}.\mathsf{PO}_\mathsf{I}(\cdot)$, $\mathcal{O}.\mathsf{FR}_\mathsf{I}(\cdot,\cdot)$, $\mathcal{O}.\mathsf{PO}_\mathsf{S}(\cdot)$, and $\mathcal{O}.\mathsf{FR}_\mathsf{S}(\cdot,\cdot)$.
7. Finally, $\mathcal{A}$ guesses a bit b' and wins if $b' = b$.

Brief formal descriptions of ROSecurityGame variants (b) and (c). Variant (b) differs from variant (a) in steps 4 and 5. In step 4, P_j is the recipient as opposed to an intermediary on the forward path. In step 5, the challenger still samples a random bit $b \leftarrow_{\$} \{0,1\}$ and, if $b = 0$, forms the challenge onion as specified by the adversary. If $b = 1$, the challenger runs $\mathsf{FormOnion}$ on input a random label, a random message, the forward path (provided by the adversary), and the empty path. The oracle for forming a reply on behalf of I is modified so that the oracle replies with the output of $\mathsf{FormOnion}$ on input a random label, a random message, the return path (provided by the adversary), and the empty path "()." For the full description, see Appendix A.

Variant (c) also differs from variant (a) in steps 4 and 5. In step 4, P_j is an intermediary on the return path $(P_{d+1}, \ldots, P_s)$, i.e., $j > d$, as opposed to an intermediary on the forward path $(P_1, \ldots, P_d)$. In step 5, the challenger still samples a random bit $b \leftarrow_{\$} \{0, 1\}$ and, if $b = 0$, forms the challenge onion as specified by the adversary. If $b = 1$, the challenger runs $\mathsf{FormOnion}$ on input a random label, a message (provided by the adversary), the forward path $(P_1, \ldots, P_d)$, and the subpath $(P_{d+1}, \ldots, P_j)$. The oracle for processing an onion on behalf of I is modified so that the oracle replies with the output of $\mathsf{FormOnion}$ on input a random label, a random message, the rest of the return path $(P_{j+1}, \ldots, P_s)$, and the empty path "()." For the full description, see Appendix A.

Definition 8 (Repliable-onion security). *A repliable onion encryption scheme Σ is* repliable-onion secure *if there exist a p.p.t. algorithm* $\mathsf{CompleteOnion}$ *and a negligible function $\nu : \mathbb{N} \mapsto \mathbb{R}$ such that every p.p.t. adversary $\mathcal{A}$ wins the security game* $\mathsf{ROSecurityGame}(1^\lambda, \Sigma, \mathsf{CompleteOnion}, \mathcal{A})$ *with negligible advantage, i.e.,* $\left|\Pr\left[\mathcal{A} \textit{ wins } \mathsf{ROSecurityGame}(1^\lambda, \Sigma, \mathsf{CompleteOnion}, \mathcal{A})\right] - \frac{1}{2}\right| \leq \nu(\lambda)$.

Remark on Definition 8. An onion formed by running a secure onion encryption scheme and received (resp. transmitted) by an honest party P does not reveal how many layers are remaining (resp. came before) since the adversary cannot distinguish between the onion and another onion formed using the same parameters except with the path truncating at the recipient (resp. sender) P.

5 Repliable-onion security $\Rightarrow$ SUC-realizability of $\mathcal{F}_{\mathsf{ROES}}$

Theorem 9. *If the onion encryption scheme Σ is correct (Definition 2) and repliable-onion secure (Definition 8), then it SUC-realizes the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ (Definition 3).*

To do this, we must show that for any static setting (fixed adversary $\mathcal{A}$, set Bad of corrupted parties, and public key infrastructure), there exists a simulator $\mathcal{S}$ such that for all $\mathcal{Z}$, there exists a negligible function $\nu : \mathbb{N} \mapsto \mathbb{R}$ such that $\left|\Pr\left[\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{ROES}},\mathcal{S},\mathcal{Z}}(1^\lambda, \mathsf{pp}) = 1\right] - \Pr\left[\mathsf{REAL}_{\Sigma,\mathcal{A},\mathcal{Z}}(1^\lambda, \mathsf{pp}) = 1\right]\right| \leq \nu(\lambda)$.

We first provide a description of the simulator $\mathcal{S}$:

Recall that during setup, the ideal adversary (i.e., $\mathcal{S}$) sends to the ideal functionality, (i) the set $\mathcal{P}$ of participants, (ii) the set $\mathsf{Bad} \subseteq \mathcal{P}$ of corrupted parties, (iii) the onion encryption scheme's algorithms: G, $\mathsf{ProcOnion}$, and $\mathsf{FormReply}$, (iv) the algorithm $\mathsf{SampleOnion}$, (v) the algorithm $\mathsf{CompleteOnion}$, and (vi) the algorithm $\mathsf{RecoverReply}$. (See Section 3.1 for the syntax of these algorithms.)

In order for our construction to be secure, the simulator $\mathcal{S}$ must provide items (i)-(vi) to $\mathcal{F}_{\mathsf{ROES}}$ such that when the ideal honest parties respond to the environment, one input at a time, the running history of outputs looks like one produced from running the real protocol using the onion encryption scheme.

To complete the description of $\mathcal{S}$, we must provide internal descriptions of how the last three items above – SampleOnion, CompleteOnion, and RecoverReply – work. Since we are in the static setting, we will assume, WLOG, that these algorithms "know" who is honest, who is corrupt, and all relevant keys. See Figure 3 for a summary of the simulator.

Send to $\mathcal{F}_{\mathsf{ROES}}$:

$\mathcal{P}$, Bad, G, ProcOnion, FormReply, SampleOnion, CompleteOnion, RecoverReply.

<u>CompleteOnion(H', C)</u>

Let CO be an algorithm such that no adversary can win ROSecurityGame with non-negligible probability. Such an algorithm must exist since Σ is repliable-onion secure.
CompleteOnion = CO.

<u>SampleOnion$(p^{\rightarrow}, p^{\leftarrow}, m)$</u>

SampleOnion just runs FormOnion on the segments $p^{\rightarrow}$ and $p^{\leftarrow}$ using a random label and, depending on whether the first segment contains the corrupt recipient, either the correct message m (if it does) or a random one (if it doesn't).

<u>RecoverReply(O, P)</u>

Return the message from running ProcOnion$(O, P, \mathsf{sk}(P))$.

Fig. 3: Summary of simulator $\mathcal{S}$

Description of simulator $\mathcal{S}$. We now expand on the summary in Figure 3.

Sampling an onion. Let $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ denote the ideal functionality corresponding to the static setting. When the ideal functionality $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ receives a request from the honest party P to form an onion using the label ℓ, the message m, the forward path $P^{\rightarrow}$, and the return path $P^{\leftarrow}$, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ partitions the routing path $(P^{\rightarrow}, P^{\leftarrow})$ into non-overlapping "segments" where each segment is a sequence of adversarial parties that must end in a single honest party, unless it ends in the adversarial recipient. (See Section 3.1 for a more formal description of these segments.) $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ runs the algorithm SampleOnion independently on each segment of the routing path. Additionally, if the forward path ends in a corrupt party, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ runs SampleOnion on the last segment of the forward path and the first segment of the return path. Using SampleOnion in this way produces onions with the property that onions belonging to different segments are information-theoretically unrelated to each other.

The algorithm SampleOnion takes as input the security parameter 1^{λ}, the public parameters pp, the forward path $p^{\rightarrow}$, and the return path $p^{\leftarrow}$.

Case 0: The routing path $(p^{\rightarrow}, p^{\leftarrow})$ is not a sequence of adversarial parties, possibly ending in an honest party. In this case, the input is invalid, and SampleOnion returns an error.

Case 1: The return path $p^{\leftarrow}$ is non-empty and ends in an honest party P_j. In this case, $\mathsf{SampleOnion}$ first samples a random label $x \leftarrow_{\$} \mathcal{L}(1^\lambda)$ and then runs $\mathsf{FormOnion}$ on the label x, the message m (from the "form onion" request), the forward path $p^{\rightarrow} = (p_1, \ldots, p_i)$, the public keys $\mathsf{pk}(p^{\rightarrow})$ associated with the parties in $p^{\rightarrow}$, the return path $p^{\leftarrow} = (p_{i+1}, \ldots, P_j)$, and the public keys $\mathsf{pk}(p^{\leftarrow})$ associated with the parties in $p^{\leftarrow}$. Finally, $\mathsf{SampleOnion}$ outputs the first onion o_1 and the last header H_j in the output $((o_1, \ldots, o_i), (h_{i+1}, \ldots, H_j), \kappa) \leftarrow \mathsf{FormOnion}(1^\lambda, \mathsf{pp}, x, m, p^{\rightarrow}, \mathsf{pk}(p^{\rightarrow}), p^{\leftarrow}, \mathsf{pk}(p^{\leftarrow}))$.

Case 2: The return path $p^{\leftarrow}$ is empty, and the forward path $p^{\rightarrow}$ ends in an honest party P_i. In this case, $\mathsf{SampleOnion}$ first samples a random label $x \leftarrow_{\$} \mathcal{L}(1^\lambda)$ and a random message $y \leftarrow_{\$} \mathcal{M}(1^\lambda)$ and then runs $\mathsf{FormOnion}$ on the label x, the message y, the forward path $p^{\rightarrow} = (p_1, \ldots, P_i)$, the public keys $\mathsf{pk}(p^{\rightarrow})$ associated with the parties in $p^{\rightarrow}$, the empty return path "()," and the empty sequence "()" of public keys. Finally, $\mathsf{SampleOnion}$ outputs the first onion o_1 and the last onion O_i in the output $((o_1, \ldots, O_i), (), \kappa) \leftarrow \mathsf{FormOnion}(1^\lambda, \mathsf{pp}, x, y, p^{\rightarrow}, \mathsf{pk}(p^{\rightarrow}), (), ())$.

Case 3: The return path $p^{\leftarrow}$ is empty, and the forward path $p^{\rightarrow}$ ends in a corrupt party p_i. In this case, $\mathsf{SampleOnion}$ first samples a random label $x \leftarrow_{\$} \mathcal{L}(1^\lambda)$ and then runs $\mathsf{FormOnion}$ on the label x, the message m (from the "form onion" request), the forward path $p^{\rightarrow} = (p_1, \ldots, p_i)$, the public keys $\mathsf{pk}(p^{\rightarrow})$ associated with the parties in $p^{\rightarrow}$, the empty return path "()," and the empty sequence "()" of public keys. Finally, $\mathsf{SampleOnion}$ outputs the first onion o_1 in the output $((o_1, \ldots, o_i), h^{\leftarrow}, \kappa) \leftarrow \mathsf{FormOnion}(1^\lambda, \mathsf{pp}, x, m, p^{\rightarrow}, \mathsf{pk}(p^{\rightarrow}), (), ())$.

Completing an onion. The environment $\mathcal{Z}$ can modify just the content of an honestly formed onion $O = (H, X)$, leaving the header H intact. When $\mathcal{Z}$ instructs an honest party P to process this kind of onion $O = (H, C)$, the ideal functionality $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ runs the algorithm $\mathsf{CompleteOnion}$ to produce an onion (H', C') that (i) looks like the output of $\mathsf{ProcOnion}$ on (H, C) and (ii) has the same header H' that $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ assigned to the peeled onion (H', X') of (H, X).

Since the onion encryption scheme Σ is repliable-onion secure (Definition 8), by definition, there exist an algorithm CO and a negligible function ν such that no adversary can win $\mathsf{ROSecurityGame}(1^\lambda, \Sigma, \mathsf{CO}, \mathcal{A})$ with probability greater than $\nu(\lambda)$. We shall use this algorithm as the simulator's $\mathsf{CompleteOnion}$ algorithm, i.e., $\mathsf{CompleteOnion} = \mathsf{CO}$.

Recovering a reply message. The environment $\mathcal{Z}$ can instruct an honest party P to process a return onion O formed by a corrupt recipient p_d in response to an onion from an honest sender; P can be an intermediary party on the return path or the original sender. In such a situation, the ideal functionality $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ runs the algorithm $\mathsf{RecoverReply}$ to recover the reply message from O.

The algorithm $\mathsf{RecoverReply}(1^\lambda, \mathsf{pp}, O, P)$ simply runs $\mathsf{ProcOnion}(O, P, \mathsf{sk}(P))$ and returns the message in the output (if it exists). If no message is returned, then $\mathsf{RecoverReply}$ outputs an error.

Proof sketch of Theorem 9. We now show that the view that any environment $\mathcal{Z}$ obtains by running the real protocol is indistinguishable from its view when the honest participants run the ideal protocol $\mathcal{F}_{\mathsf{ROES}}$ with our simulator $\mathcal{S}$.

Proof idea: An onion encryption scheme SUC-realizes $\mathcal{F}_{\mathsf{ROES}}$ if the environment cannot distinguish whether an honest onion's evolution (the sequence of onion layers) comes from a single call to $\mathsf{FormOnion}$ (the real setting), or if it is produced by $\mathcal{F}_{\mathsf{ROES}}$. Recall that, to form an honest onion's evolution, $\mathcal{F}_{\mathsf{ROES}}$ calls $\mathsf{SampleOnion}$ (which, for our simulator, is the same algorithm as $\mathsf{FormOnion}$) multiple times, each call corresponding to a segment of the onion's routing path.

Our game-based definition of repliable-onion security has a very similar requirement: the adversary cannot distinguish whether the evolution of an honestly formed onion comes from a single $\mathsf{FormOnion}$ call or from two computationally unrelated $\mathsf{FormOnion}$ calls. More precisely, if the game picks $b = 0$, then no switch occurs, and the onion layers are formed "honestly," i.e., via a single call to $\mathsf{FormOnion}$. If the game picks $b = 1$, then the onion layers are formed using a "switch:" the path is broken up into two segments, and for each segment of the path, the onion layers are formed using separate calls to $\mathsf{FormOnion}$.

At the heart of our proof is a hybrid argument that shows that onion layers formed using i calls to $\mathsf{FormOnion}$ (so they have $i - 1$ such "switches") are indistinguishable from those formed by $i + 1$ such calls. Thus, we show that onion layers of the real protocol (produced by a single call to $\mathsf{FormOnion}$) are indistinguishable from those in the ideal world (produced by $\mathcal{F}_{\mathsf{ROES}}$ that calls $\mathsf{FormOnion}$ separately for each segment of the routing path). Therefore, we conclude that if an onion encryption scheme is repliable-onion secure, then it SUC-realizes $\mathcal{F}_{\mathsf{ROES}}$. See the full version of this paper for the formal proof [1].

Is repliable-onion security necessary to SUC-realize $\mathcal{F}_{\mathsf{ROES}}$? Let us now address the converse of the theorem. Given an onion encryption scheme Σ that SUC-realizes $\mathcal{F}_{\mathsf{ROES}}$, does it follow that it is correct and repliable-onion secure?

In order to prove that it does, we would have to give a reduction $\mathcal{B}$ that acts as the environment towards honest participants I and S; $\mathcal{B}$'s goal is to determine whether I and S are running Σ or, instead, using $\mathcal{F}_{\mathsf{ROES}}$ with some simulator $\mathcal{S}$. $\mathcal{B}$ would obtain I's and S's public keys from the setup step of the system, and would pass them on to $\mathcal{A}$. Whenever $\mathcal{A}$ issues $\mathsf{ProcOnion}$ queries for I and S, $\mathcal{B}$ acts as the environment that sends these onions to I and S.

Next comes the challenge step, and this is where this proof would run into difficulty. In our repliable-onion security game, it is at this point that $\mathcal{A}$ specifies the names and public keys of the rest of the participants in the system. But our functionality assumed that this setup was done ahead of time; modeling it this way made the functionality more manageable and interacted well with the SUC model. However, we can show that a modified, non-adaptive version of repliable-onion security is, in fact, necessary to SUC-realize $\mathcal{F}_{\mathsf{ROES}}$. Let $\mathsf{NAROSecurityGame}(1^\lambda, \Sigma, \mathsf{CompleteOnion}, \mathcal{A})$ be the $\mathsf{ROSecurityGame}$ security game modified as follows: instead of waiting until the challenge step to specify the names and public keys on the routing path of the challenge onion, $\mathcal{A}$

specifies them at the very beginning. Other than that, we define *non-adaptive repliable-onion security* completely analogously to repliable-onion security:

Definition 10 (Non-adaptive repliable-onion security). *A repliable onion encryption scheme Σ is* non-adaptive repliable-onion secure *if there exist a p.p.t. algorithm* $\mathsf{CompleteOnion}$ *and a negligible function* $\nu : \mathbb{N} \mapsto \mathbb{R}$ *such that every p.p.t. adversary* $\mathcal{A}$ *wins the security game* $\mathsf{NAROSecurityGame}(1^\lambda, \Sigma, \mathsf{CompleteOnion}, \mathcal{A})$ *with negligible advantage, i.e.,* $\left|\Pr\left[\mathcal{A} \text{ wins } \mathsf{NAROSecurityGame}(1^\lambda, \Sigma, \mathsf{CompleteOnion}, \mathcal{A})\right] - \frac{1}{2}\right| \leq \nu(\lambda)$.

Theorem 11 is the closest we can show to the converse of Theorem 9:

Theorem 11. *If an onion encryption scheme Σ SUC-realizes the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ (Definition 3) then it is non-adaptive repliable-onion secure (Definition 10).*

Proof. The proof is by a hybrid argument. Let $\mathsf{Experiment}^0(1^\lambda, \mathcal{A})$ be the adversary's view in the non-adaptive repliable-onion security game when $b = 0$. Let I and S denote the names of the honest parties chosen by $\mathcal{A}$.

Let $\mathsf{Hybrid}^{real_0}(1^\lambda, \mathcal{A})$ be the same as $\mathsf{Experiment}^0$ except in organization. Here, we split up the $\mathsf{NAROSecurityGame}$ challenger into components: one component is responsible for executing Σ on behalf of participant S (i.e., generate S's keys, process and where possible, reply to onions routed to S, and form the challenge onion on behalf of S), another is responsible for executing Σ on behalf of I (i.e. i.e., generate I's keys and deal with onions routed to I), and the third component, $\mathcal{B}$ carries out everything else, including interacting with $\mathcal{A}$. When organized this way, it is easy to see that $\mathcal{B}$ and $\mathcal{A}$ jointly act as the environment (from the SUC model) for the real-world execution of Σ by the honest participants S and I. The environment here directs only one of the participants (S) to ever form an onion: just the one challenge onion. The output of $\mathsf{Hybrid}^{real_0}(1^\lambda, \mathcal{A})$ is the adversary's view.

Let $\mathsf{Hybrid}^{ideal_0}(1^\lambda, \mathcal{A})$ be the same as $\mathsf{Hybrid}^{real_0}(1^\lambda, \mathcal{A})$ except that the real execution of Σ is replaced with executing $\mathcal{F}_{\mathsf{ROES}}$. Hybrid^{real_0} and $\mathsf{Hybrid}^{ideal_0}$ are indistinguishable by the hypothesis. By construction of $\mathcal{F}_{\mathsf{ROES}}$, the layers of the sole onion that's ever created in $\mathsf{Hybrid}^{ideal_0}(1^\lambda, \mathcal{A})$ are computed by splitting the routing path into two segments: one ends in I and the other one in S.

Let us consider another game $\mathsf{Hybrid}^{ideal_1}(1^\lambda, \mathcal{A})$. This game is identical to $\mathsf{Hybrid}^{ideal_0}(1^\lambda, \mathcal{A})$ except in how it is internally organized. Here, acting as the environment responsible for supplying inputs to S, $\mathcal{B}$ will cause two onions to be formed. In case (a), both onions are formed by S: one with I as the recipient, and the second onion is formed using the rest of the routing path; in case (b), S sends a non-repliable onion to I who then replies to S by forming a fresh onion; in case (c), I forms an onion using the first segment of the path, and then a fresh onion with S as the recipient. The parts that are visible to $\mathcal{A}$ are just the onions themselves, and therefore $\mathsf{Hybrid}^{ideal_1}(1^\lambda, \mathcal{A})$ is identical to $\mathsf{Hybrid}^{ideal_0}(1^\lambda, \mathcal{A})$.

Next, define $\mathsf{Hybrid}^{real_1}(1^\lambda, \mathcal{A})$: here, the environment ($\mathcal{B}$ acting jointly with $\mathcal{A}$) interacts with S and I exactly as in $\mathsf{Hybrid}^{ideal_1}(1^\lambda, \mathcal{A})$, but S and I are running Σ instead of $\mathcal{F}_{\mathsf{ROES}}$ with $\mathcal{S}$. By the hypothesis, $\mathsf{Hybrid}^{real_1}(1^\lambda, \mathcal{A})$ is indistinguishable from $\mathsf{Hybrid}^{ideal_1}(1^\lambda, \mathcal{A})$. It is easy to see that $\mathsf{Hybrid}^{real_1}(1^\lambda, \mathcal{A})$ and $\mathsf{Hybrid}^{ideal_1}(1^\lambda, \mathcal{A})$ are identical when I appears only once in the routing path. When I appears more than once in the routing path, the views are indistinguishable due to the realizability of $\mathcal{F}_{\mathsf{ROES}}$.

Finally, let $\mathsf{Experiment}^1(1^\lambda, \mathcal{A})$ be the adversary's view in the non-adaptive repliable-onion security game when $b = 1$. $\mathsf{Hybrid}^{real_1}(1^\lambda, \mathcal{A})$ and $\mathsf{Experiment}^1$ are identical by construction. Therefore, we have shown that $\mathsf{Experiment}^1(1^\lambda, \mathcal{A}) \approx \mathsf{Experiment}^1(1^\lambda, \mathcal{A})$, and therefore, Σ is non-adaptive repliable-onion secure.

6 Shallot Encryption

In this section, we provide our construction of a repliable onion encryption scheme dubbed "*Shallot Encryption Scheme.*" Inspired by the Camenisch and Lysyanskaya (CL) approach [5], our construction forms each onion layer for a party P by encrypting the previous layer under a key k which, in turn, is encrypted under the public key of P and a tag t. Our construction differs from the CL construction in that the tag t is not a function of the layer's content. Instead, authentication of the message happens separately, using a message authentication code. The resulting object is more like a shallot than an onion; it consists of *two* separate layered encryption objects: the header and the content (which may contain a "bud," i.e., another layered encryption object, namely the header for the return onion). We still call these objects "onions" to be consistent with prior work, but the scheme overall merits the name "shallot encryption."

Let λ denote the security parameter. Let $F_{(\cdot)}(\cdot, \cdot)$ be a pseudorandom function family such that, whenever $\mathsf{seed} \in \{0,1\}^k$, F_{seed} takes as input two k-bit strings and outputs a k-bit string. Such a function can be constructed from a regular one-input PRF in a straightforward fashion.

Let $\{f_k(\cdot)\}_{k \in \{0,1\}^*}$ and $\{g_k(\cdot)\}_{k \in \{0,1\}^*}$ be block ciphers, i.e., pseudorandom permutations (PRPs). We use the same key to key both block ciphers: one ($\{f_k(\cdot)\}_{k \in \{0,1\}^*}$) with a "short" blocklength $L_1(\lambda)$ is used for forming headers, and the other ($\{g_k(\cdot)\}_{k \in \{0,1\}^*}$) with a "long" blocklength $L_2(\lambda)$ is used for forming contents. This is standard and can be constructed from regular block ciphers. Following the notational convention introduced by Camenisch and Lysyanskaya [5], let $\{X\}_k$ denote $f_k(X)$ if $|X| = L_1(\lambda)$, or $g_k(X)$ if $|X| = L_2(\lambda)$, and let $\}X\{_k$ correspondingly denote $f_k^{-1}(X)$ or $g_k^{-1}(X)$.

Let $\mathcal{E} = (\mathsf{Gen}_{\mathcal{E}}, \mathsf{Enc}, \mathsf{Dec})$ be a CCA2-secure encryption scheme with tags [12], let $\mathsf{MAC} = (\mathsf{Gen}_{\mathsf{MAC}}, \mathsf{Tag}, \mathsf{Ver})$ be a message authentication code (MAC), and let h be a collision-resistant hash function.

Setting up: Each party P_i forms a public key pair $(\mathsf{pk}(P_i), \mathsf{sk}(P_i))$ using the public key encryption scheme's key generation algorithm $\mathsf{Gen}_{\mathcal{E}}$, i.e., $(\mathsf{pk}(P_i), \mathsf{sk}(P_i)) \leftarrow \mathsf{Gen}_{\mathcal{E}}(1^\lambda, \mathsf{pp}, P_i)$.

Forming a repliable onion. Each onion consists of (1) the header (i.e., the encrypted routing path and associated keys) and (2) the content (i.e., the encrypted message).

Forming the header: In our example, let Alice (denoted P_s) be the sender, and let Bob (denoted P_d, d for destination) be the recipient. To form a *repliable onion*, Alice receives as input a label ℓ, a message m, a *forward* path to Bob: $P^{\rightarrow} = P_1, \ldots, P_{d-1}, P_d$, $|P^{\rightarrow}| = d \leq N$, and a *return* path to herself: $P^{\leftarrow} = P_{d+1}, \ldots, P_{s-1}, P_s$, $|P^{\leftarrow}| = s - d + 1 \leq N$. All other participants P_i are intermediaries.

Let "seed" be a seed stored in $\mathsf{sk}(P_s)$. Alice computes (i) an encryption key $k_i = F_{\mathsf{seed}}(\ell, i)$ for every party P_i on the routing path $(P^{\rightarrow}, P^{\leftarrow})$, (ii) an authentication key K_d for Bob using $\mathsf{Gen}_{\mathsf{MAC}}(1^{\lambda})$ with $F_{\mathsf{seed}}(d, \ell)$ sourcing the randomness for running the key generation algorithm, and (iii) an authentication key K_s for herself using $\mathsf{Gen}_{\mathsf{MAC}}(1^{\lambda})$ with $F_{\mathsf{seed}}(s, \ell)$ sourcing the randomness for running the key generation algorithm.

Remark: We can avoid using a PRF in exchange for requiring state; an alternative to using a PRF is to store keys computed from true randomness locally.

The goal of $\mathsf{FormOnion}$ is to produce an onion O_1 for the first party P_1 on the routing path such that P_1 processing O_1 produces the onion O_2 for the next destination P_2 on the routing path, and so on.

Suppose for the time being that both the forward path and the return path are of the maximum length N, i.e., $d = s - d + 1 = N$.

Let O be an onion that we want party P to "peel." The *header* of O is a sequence $H = (E, B^1, \ldots, B^{N-1})$. E is an encryption under P's public key and the tag $t = h(B^1, \ldots, B^{N-1})$ of the following pieces of information that P needs to correctly process the onion: (i) P's role, i.e., is P an intermediary, or the onion's recipient, or the original sender of the onion whose reply P just received; (ii) in case P is an intermediary or recipient, the encryption key k necessary for making sense of the rest of the onion; (iii) in case P is the original sender, the label ℓ necessary for making sense of the rest of the onion; and (iv) in case P is the recipient, the authentication key K.

If P is an intermediary, it will next process $(B^1, \ldots, B^{N-1})$ by inverting each of them, in turn, using the block cipher's key k, to obtain the values $\}B^1\{_k, \ldots, \}B^{N-1}\{_k$. The value $\}B^1\{_k$ reveals the destination P' and the ciphertext E' of the peeled onion. For each $1 < j < N$, the value $\}B^j\{_k$ is block $(B')^{j-1}$ of the peeled onion, so the header of the peeled onion will begin with $(E', (B')^1, \ldots, (B')^{N-2})$. The final block $(B')^{N-1}$ of the header is formed by computing the inverse of the PRP keyed by k of the all-zero string of length $L_1(\lambda)$, i.e., $(B')^{N-1} = \}0 \ldots 0\{_k$.

Therefore, sender Alice needs to form her onion so that each intermediary applying the procedure described above will peel it correctly. Using the keys $k_1, \ldots, k_d$ and K_d, Alice first forms the header $H_d = (E_d, B_d^1, \ldots, B_d^{N-1})$ for the last onion O_d on the forward path (the one to be processed by Bob): For every $i \in \{1, \ldots, N-1\}$, let $B_d^i = \} \ldots \}0 \ldots 0\{_{k_i} \ldots \{_{k_{d-1}}$. The tag t_d for integrity protection is the hash of these blocks concatenated together, i.e.,

$t_d = h(B_d^1, \ldots, B_d^{N-1})$. The ciphertext E_d is the encryption of (R, k_d, K_d) under the public key $\mathsf{pk}(P_d)$ and the tag t_d, i.e., $E_d \leftarrow \mathsf{Enc}(\mathsf{pk}(P_d), t_d, (\mathsf{R}, k_d, K_d))$. The headers of the remaining onions in the evolution are formed recursively. Let

$$\begin{aligned}
B_{d-1}^1 &= \{P_d, E_d\}_{k_{d-1}}, \\
B_{d-1}^i &= \{B_d^{i-1}\}_{k_{d-1}}, && \forall i \in \{2, \ldots, N-1\}, \\
t_{d-1} &= h(B_{d-1}^1, \ldots, B_{d-1}^{N-1}), \\
E_{d-1} &\leftarrow \mathsf{Enc}(\mathsf{pk}(P_{d-1}), t_{d-1}, (\mathsf{I}, k_{d-1}));
\end{aligned}$$

and so on. (WLOG, we assume that (P_d, E_d) "fits" into a block; i.e., $|P_d, E_d| \leq L_1(\lambda)$. A block cipher with the correct blocklength can be built from a standard one [4, 20].)

Forming the encrypted content: Alice then forms the encrypted content for Bob. First, if the return path $P^{\leftarrow}$ is non-empty, Alice forms the header H_{d+1} for the return onion using the same procedure that she used to form the header H_1 for the forward onion, but using the return path $P^{\leftarrow}$ instead of the forward path $P^{\rightarrow}$ and encrypting (S, ℓ) instead of (R, k_s, K_s). That is, the ciphertext E_s of the "innermost" header H_s is the encryption $\mathsf{Enc}(\mathsf{pk}(P_s), t_s, (\mathsf{S}, \ell))$ rather than $\mathsf{Enc}(\mathsf{pk}(P_s), t_s, (\mathsf{R}, k_s, K_s))$. If the return path is empty, then H_{d+1}, k_s and K_s are the empty string.

When Bob processes the onion, Alice wants him to receive (i) the message m, (ii) the header H_{d+1} for the return onion, (iii) the keys k_s and K_s for forming a reply to the anonymous sender (Alice), and (iv) the first party P_{d+1} on the return path. So, Alice sets the "*meta-message*" M to be the concatenation of m, H_{d+1}, k_s, K_s, and P_{d+1}: $M = (m, H_{d+1}, k_s, K_s, P_{d+1})$.

Alice wants Bob to be able to verify that M is the meta-message, so she also computes the tag $\sigma_d = \mathsf{Tag}(K_d, M)$. (WLOG, (M, σ_d) "fits" exactly into a block; i.e., $|M| \leq L_2(\lambda)$.)

The encrypted content C_i for each onion O_i on the forward path is given by: $C_i = \{\ldots\{M, \sigma_d\}_{k_d}\ldots\}_{k_i}$. See Figure 4 for a pictorial description of the how the repliable onion is formed.

We now explain what happens when $d \neq N$, or $s - d + 1 \neq N$: If either d or $s - d + 1$ exceed the upper bound N, then $\mathsf{FormOnion}$ returns an error. If d is strictly less than N, the header is still "padded" to $N - 1$ blocks by sampling N encryption keys as before. Likewise if $s - d + 1 < N$, the header is padded to $N - 1$ blocks in similar fashion. (Note that the size of each repliable onion is twice the size of a CL non-repliable onion [5] with maximum path length N.)

Processing a repliable onion. Let Carol be an intermediary node on the forward path from Alice to Bob. When Carol receives the onion $O_i = (H_i, C_i)$ consisting of the header $H_i = (E_i, B_i^1, \ldots B_i^{N-1})$ and the content C_i, she processes it as follows:

Carol first computes the tag $t_i = h(B_i^1, \ldots B_i^{N-1})$ for integrity protection and then attempts to decrypt the ciphertext E_i of the header using her secret key $\mathsf{sk}(P_i)$ and the tag t_i to obtain her role and key(s), i.e., $(\mathsf{I}, k_i) =$

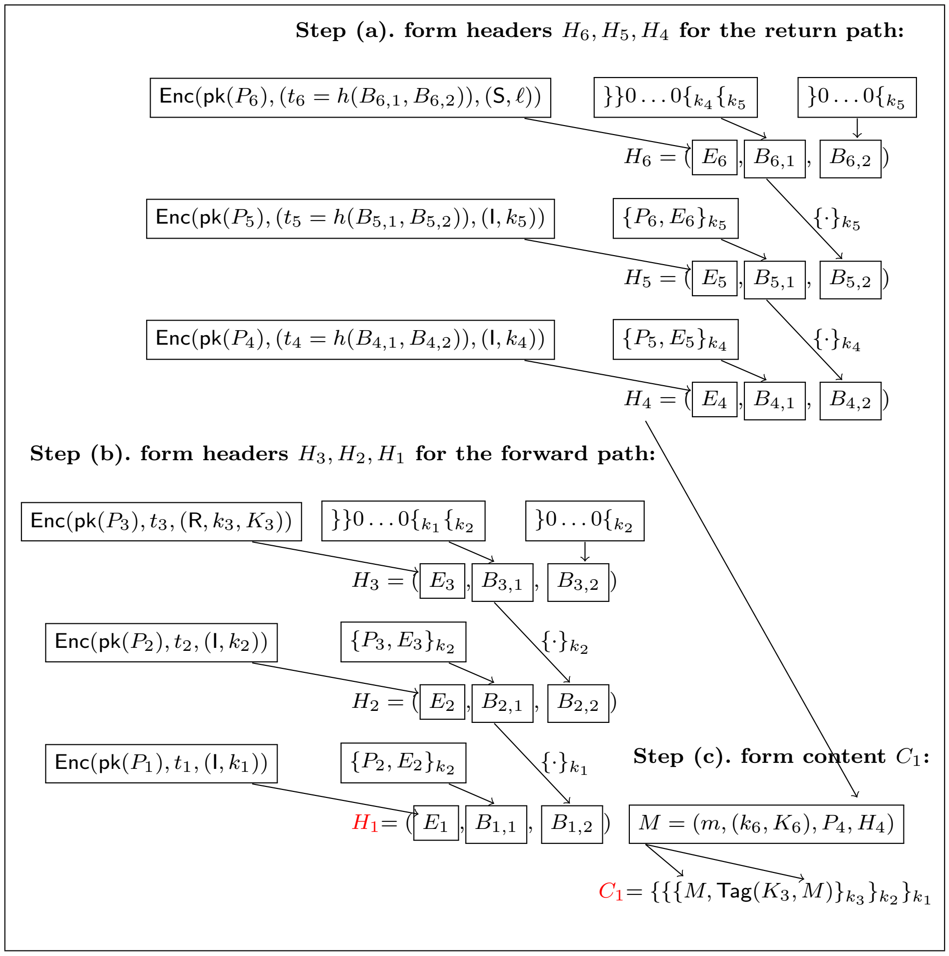
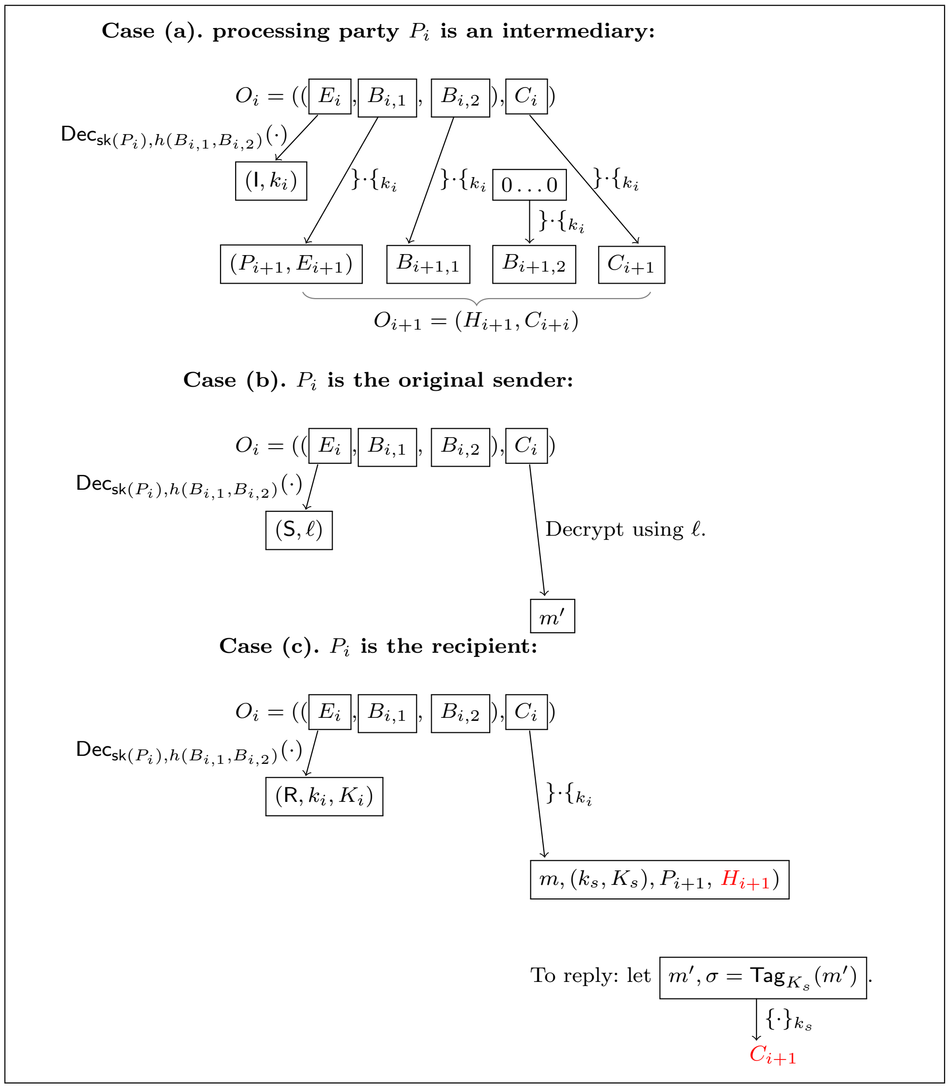

Step (a). form headers H_6, H_5, H_4 for the return path:
$\mathsf{Enc}(\mathsf{pk}(P_6), (t_6 = h(B_{6,1}, B_{6,2})), (\mathsf{S}, \ell))$
$\}\}0\ldots0\{_{k_4}\{_{k_5}$
$\}0\ldots0\{_{k_5}$
$H_6 = (E_6, B_{6,1}, B_{6,2})$
$\{\cdot\}_{k_5}$
$\mathsf{Enc}(\mathsf{pk}(P_5), (t_5 = h(B_{5,1}, B_{5,2})), (\mathsf{I}, k_5))$
$\{P_6, E_6\}_{k_5}$
$H_5 = (E_5, B_{5,1}, B_{5,2})$
$\{\cdot\}_{k_4}$
$\mathsf{Enc}(\mathsf{pk}(P_4), (t_4 = h(B_{4,1}, B_{4,2})), (\mathsf{I}, k_4))$
$\{P_5, E_5\}_{k_4}$
$H_4 = (E_4, B_{4,1}, B_{4,2})$
Step (b). form headers H_3, H_2, H_1 for the forward path:
$\mathsf{Enc}(\mathsf{pk}(P_3), t_3, (\mathsf{R}, k_3, K_3))$
$\}\}0\ldots0\{_{k_1}\{_{k_2}$
$\}0\ldots0\{_{k_2}$
$H_3 = (E_3, B_{3,1}, B_{3,2})$
$\mathsf{Enc}(\mathsf{pk}(P_2), t_2, (\mathsf{I}, k_2))$
$\{P_3, E_3\}_{k_2}$
$\{\cdot\}_{k_2}$
$H_2 = (E_2, B_{2,1}, B_{2,2})$
$\mathsf{Enc}(\mathsf{pk}(P_1), t_1, (\mathsf{I}, k_1))$
$\{P_2, E_2\}_{k_2}$
$\{\cdot\}_{k_1}$
$H_1 = (E_1, B_{1,1}, B_{1,2})$
Step (c). form content C_1:
$M = (m, (k_6, K_6), P_4, H_4)$
$C_1 = \{\{\{M, \mathsf{Tag}(K_3, M)\}_{k_3}\}_{k_2}\}_{k_1}$

Fig. 4: Steps for forming the first shallot onion $O_1 = (H_1, C_1)$ when the forward path is $P^{\rightarrow} = (P_1, P_2, P_3)$, and the return path is $P^{\leftarrow} = (P_4, P_5, P_6)$: (a) steps for forming the headers H_6, H_5, H_4 for the return path, (b) steps for forming the headers H_3, H_2, H_1 for the forward path, and (c) steps for forming the content C_1.

$\mathsf{Dec}(\mathsf{sk}(P_i), t_i, E_i)$. Carol succeeds in decrypting E_i only if the header has not been tampered with. In this case, she gets her role "I" and the key k_i and proceeds with processing the header and content:

Carol first decrypts the first block B_i^1 of the current header to retrieve the next destination P_{i+1} and ciphertext E_{i+1} of the processed header (header of the next onion), i.e., $(P_{i+1}, E_{i+1}) = \}B_i^1\{_{k_i}$. To obtain the first $N-2$ blocks of the processed header, Carol decrypts the last $N-2$ blocks of H: $B_{i+1}^j = \}B_i^{j+1}\{_{k_i}$ for all $j \in [N-2]$. To obtain the last block of the processed header, Carol "decrypts" the all-zero string "$0\ldots0$:" $B_{i+1}^{N-1} = \}0\ldots0\{_{k_i}$. To process the content, Carol simply decrypts the current content C_i: $C_{i+1} = \}C_i\{_{k_i}$.

Let David be an intermediary party on the return path. When David receives the onion O_j, he processes it exactly in the same way that Carol processed the onion O_i in the forward direction. (Critically, David does not know that he is on the return path as opposed to the forward path.) See Figure 5 for a pictorial description of the how the onion is processed.



Case (a). processing party P_i is an intermediary:
$O_i = ((E_i, B_{i,1}, B_{i,2}), C_i)$
$\mathsf{Dec}_{\mathsf{sk}(P_i),h(B_{i,1},B_{i,2})}(\cdot)$
(I, k_i)
$\}\cdot\{_{k_i}$
$0\ldots0$
(P_{i+1}, E_{i+1})
$B_{i+1,1}$
$B_{i+1,2}$
C_{i+1}
$O_{i+1} = (H_{i+1}, C_{i+i})$
Case (b). P_i is the original sender:
(S, ℓ)
Decrypt using ℓ.
m'
Case (c). P_i is the recipient:
(R, k_i, K_i)
$m, (k_s, K_s), P_{i+1}, H_{i+1})$
To reply: let $m', \sigma = \mathsf{Tag}_{K_s}(m')$.
$\{\cdot\}_{k_s}$
C_{i+1}

Fig. 5: Steps for processing a shallot onion $O_i = ((H_i, B_{i,1}, B_{i,2}), C_i)$ when $N = 3$ and when (a) the processing party P_i is an intermediary, (b) P_i is the original sender, and (c) P_i is the recipient. For (c), steps for forming the reply onion $O_{i+1} = (H_{i+1}, C_{i+1})$ for the next destination P_{i+1}.

Replying to the anonymous sender. When Bob receives the onion $O_d = (H_d, C_d)$, he processes it in the same way that the intermediary party Carol does, by running $\mathsf{ProcOnion}$: Bob first decrypts the ciphertext E_d of the header to retrieve his role "R" and the keys k_d and K_d. If O_d hasn't been tampered with, Bob retrieves the meta-message $M = (m, H_{d+1}, k_s, K_s, P_{d+1})$ and the tag σ_d that Alice embedded into the onion by decrypting the content C_d using the key k_d: $((m, H_{d+1}, k_s, K_s, P_{d+1}), \sigma_d) = \}C_d\{_{k_d}$. Bob can verify that the message is untampered by running the MAC's verification algorithm $\mathsf{Ver}(K_d, M, \sigma_d)$.

To respond to the anonymous sender (Alice) with the message m', Bob creates a new encrypted content using the keys k_s and K_s: $C_{d+1} = \{m', \mathsf{Tag}(K_s, m')\}_{k_s}$. Bob sends the reply onion $O_{d+1} = (H_{d+1}, C_{d+1})$ to the next destination P_{d+1}.

Reading the reply. When Alice receives the onion O_s, she retrieves the reply from Bob by first processing the onion, by running $\mathsf{ProcOnion}$:

Alice first decrypts the ciphertext E_s of the header to retrieve her role "S" and the label ℓ. She reconstructs the each encryption key $k_i = F_{\mathsf{seed}}(\ell, i)$ and the authentication key K_s using the pseudo-randomness $F_{\mathsf{seed}}(s, \ell)$. (Alternatively, if she stored the keys locally, she looks up the keys associated with label ℓ in a local data structure). If O_s hasn't been tampered with, Alice retrieves the reply m' that Bob embedded into the onion by decrypting the content C_s using the keys $(k_{d+1}, \ldots, k_s)$: $(m', \sigma_s) = \}\{\ldots\{C_s\}_{k_{s-1}}\ldots\}_{k_{d+1}}\{_{k_s}$. Alice can verify that the message is untampered by running $\mathsf{Ver}(K_s, m', \sigma_s)$.

7 Shallot Encryption Scheme is secure

Theorem 12. *Shallot Encryption Scheme (in Section 6) SUC-realizes the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ (Definition 3).*

By Theorem 9, it suffices to prove that Shallot Encryption Scheme is correct and repliable-onion secure under the assumption that (i) $\{f_k\}_{k\in\{0,1\}^*}$ is a PRP, (ii) $\mathcal{E}$ is a CCA2-secure encryption scheme with tags, (iii) MAC is a message authentication code, and (iv) h is a collision-resistant hash function.

Proof idea: In cases (a) and (c) (in these cases, P_j is an intermediary, not the recipient), we can prove that $\mathcal{A}$'s view when $b = 0$ is indistinguishable from $\mathcal{A}$'s view when $b = 1$ using a hybrid argument. The gist of the argument is as follows: First, P_j's encryption key k_j is protected by CCA2-secure encryption, so it can be swapped out for the all-zero key "$0\ldots0$." Next, blocks $(N - j - 1)$ to $(N - 1)$ of the onion for P_{j+1} look random as they are all "decryptions" under k_j, so they can be swapped out for truly random blocks. Next, blocks 1 to $(N - j - 1)$ and the content of the onion for P_j look random as they are encryptions under k_j, so they can be swapped out for truly random blocks. At this point, the keys for forming O_{j+1} can be independent of the keys for forming O_j, and these onions may be formed via separate $\mathsf{FormOnion}$ calls; see Figure 6.

Experiment^0—game with $b = 0$, same as Hybrid^1
Hybrid^1—make O_{j+1}, then O_1
Hybrid^2—same as Hybrid^1 except swap ℓ for random label
Hybrid^3—same as Hybrid^2 except swap k_j for fake key "0 . . . 0"
Hybrid^4—same as Hybrid^3 except swap $(B_{j+1}^{N-j-1}, \ldots, B_{j+1}^{N-1})$ for truly random blocks
Hybrid^5—same as Hybrid^4 except swap $(B_j^1, \ldots, B_j^{N-j-1})$ and C_j for truly random blocks
Hybrid^6—same as Hybrid^5 except swap onion for intermediary P_j for onion for recipient P_j
Hybrid^7—same as Hybrid^6 except swap truly random blocks and content in O_j for pseudo-random blocks $(B_j^1, \ldots, B_j^{N-j-1}, C_j)$
Hybrid^8—same as Hybrid^7 except swap truly random blocks in H_{j+1} for pseudo-random blocks $(B_{j+1}^{N-j-1}, \ldots, B_{j+1}^{N-1})$
Hybrid^9—same as Hybrid^8 except swap key "0 . . . 0" for real key k_j
Experiment^1—game with $b = 1$, same as Hybrid^9

Fig. 6: Road map of proof of Theorem 12

For case (b) (P_j is the recipient), we can use a simpler hybrid argument since only the content of a forward onion can be computationally related to the keys for the return path. Thus, we can swap out just the content for a truly random string. See the full paper for the full proof [1].

Acknowledgements We thank the reviewers of this paper for their helpful comments. This work was funded in part by Facebook faculty research awards.

References

1. Megumi Ando and Anna Lysyanskaya. Cryptographic shallots: A formal treatment of repliable onion encryption. Cryptology ePrint Archive, Report 2020/215, 2020. `https://eprint.iacr.org/2020/215`.
2. Megumi Ando, Anna Lysyanskaya, and Eli Upfal. Practical and provably secure onion routing. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 144:1–144:14. Schloss Dagstuhl, July 2018.
3. Megumi Ando, Anna Lysyanskaya, and Eli Upfal. On the complexity of anonymous communication through public networks. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
4. Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.2*, 2015.
5. Jan Camenisch and Anna Lysyanskaya. A formal treatment of onion routing. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 169–187. Springer, Heidelberg, August 2005.
6. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
7. Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In Rosario Gennaro

and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 3–22. Springer, Heidelberg, August 2015.

8. Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *28th ACM STOC*, pages 639–648. ACM Press, May 1996.
9. Ran Canetti, Daniel Shahaf, and Margarita Vald. Universally composable authentication and key-exchange with global PKI. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 265–296. Springer, Heidelberg, March 2016.
10. David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.
11. Lance Cottrell. Mixmaster and remailer attacks, 1995.
12. Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 13–25. Springer, Heidelberg, August 1998.
13. Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 45–64. Springer, Heidelberg, April / May 2002.
14. George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *2003 IEEE Symposium on Security and Privacy*, pages 2–15. IEEE Computer Society Press, May 2003.
15. George Danezis and Ian Goldberg. Sphinx: A compact and provably secure mix format. In *2009 IEEE Symposium on Security and Privacy*, pages 269–282. IEEE Computer Society Press, May 2009.
16. George Danezis and Ben Laurie. Minx: A simple and efficient anonymous packet format. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 59–65, 2004.
17. Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: the second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 303–320, 2004.
18. Ceki Gulcu and Gene Tsudik. Mixing e-mail with babel. In *Proceedings of Internet Society Symposium on Network and Distributed Systems Security*, pages 2–16. IEEE, 1996.
19. Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul F. Syverson. Users get routed: traffic correlation on tor by realistic adversaries. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 337–348. ACM Press, November 2013.
20. Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. Chapman and Hall/CRC, 2014.
21. Christiane Kuhn, Martin Beck, and Thorsten Strufe. Breaking and (partially) fixing provably secure onion routing. In *2020 IEEE Symposium on Security and Privacy*, pages 168–185. IEEE Computer Society Press, May 2020.
22. Ulf Möller and Lance Cottrell. Mixmaster protocol—v2. unfinished draft, Jan 2000.
23. Sameer Parekh. Prospects for remailers. *First Monday*, 1(2), 1996.
24. Yixin Sun, Anne Edmundson, Nick Feamster, Mung Chiang, and Prateek Mittal. Counter-RAPTOR: Safeguarding tor against active routing attacks. In *2017 IEEE Symposium on Security and Privacy*, pages 977–992. IEEE Computer Society Press, May 2017.

25. Ryan Wails, Yixin Sun, Aaron Johnson, Mung Chiang, and Prateek Mittal. Tempest: Temporal dynamics in anonymity systems. *PoPETs*, 2018(3):22–42, 2018.

A Security game for variants (b) and (c)

Variant (b) Below, we provide a description of steps 4 and 5 of the repliable-onion security game, $\mathsf{ROSecurityGame}$, in Section 4 for case (b).

4. $\mathcal{A}$ chooses a label $\ell \in \mathcal{L}(1^\lambda)$ and a message $m \in \mathcal{M}(1^\lambda)$. $\mathcal{A}$ also chooses a forward path $P^\rightarrow = (P_1, \ldots, P_d)$ and a return path $P^\leftarrow = (P_{d+1}, \ldots, P_s)$ such that (i) if $P^\leftarrow$ is non-empty, then it ends with S, (ii) I appears in the routing path, and (iii) the first time it appears in the path is at the recipient P_d. $\mathcal{A}$ sends to $\mathcal{C}$ the parameters for the challenge onion: ℓ, m, $P^\rightarrow$, the public keys $\mathsf{pk}(P^\rightarrow)$ of the parties in $P^\rightarrow$, $P^\leftarrow$, and the public keys $\mathsf{pk}(P^\leftarrow)$ of the parties in $P^\leftarrow$.
5. $\mathcal{C}$ samples a bit $b \leftarrow\!\!\$\, \{0,1\}$.
   If $b = 0$, $\mathcal{C}$ runs $\mathsf{FormOnion}$ on the parameters specified by $\mathcal{A}$, i.e., $((O_1^0, \ldots, O_d^0), H^\leftarrow, \kappa) \leftarrow \mathsf{FormOnion}(\ell, m, P^\rightarrow, \mathsf{pk}(P^\rightarrow), P^\leftarrow, \mathsf{pk}(P^\leftarrow))$.
   In this case, the oracles—$\mathcal{O}.\mathsf{PO_I}(\cdot)$, $\mathcal{O}.\mathsf{FR_I}(\cdot,\cdot)$, $\mathcal{O}.\mathsf{PO_S}(\cdot)$, and $\mathcal{O}.\mathsf{FR_S}(\cdot,\cdot)$—remain unmodified.
   Otherwise, if $b = 1$, $\mathcal{C}$ performs the "switch" at honest recipient P_d. $\mathcal{C}$ runs $\mathsf{FormOnion}$ on input a random label $x \leftarrow\!\!\$\, \mathcal{L}(1^\lambda)$, a random message $y \leftarrow\!\!\$\, \mathcal{M}(1^\lambda)$, the forward path $P^\rightarrow$, and the empty return path "()", i.e., $((O_1^1, \ldots, O_d^1), (), \kappa) \leftarrow \mathsf{FormOnion}(x, y, P^\rightarrow, \mathsf{pk}(P^\rightarrow), (), ())$.
   We modify the oracles as follows. $\mathcal{O}.\mathsf{FR_I}$ does the following to "form a reply" using message m' and onion $O = O_d^1$: $\mathcal{O}.\mathsf{FR_I}$ runs $\mathsf{FormOnion}$ on a random label x', a random message y', the return path $P^\leftarrow$ as the forward path, and the empty return path "()", i.e., $((O_{j+1}^{m'}, \ldots, O_s^{m'}), (), \kappa^{m'}) \leftarrow \mathsf{FormOnion}(x', y', P^\leftarrow, \mathsf{pk}(P^\leftarrow), (), ())$, stores the pair $(O_s^{m'}, m')$ (such that the pair is accessible by $\mathcal{O}.\mathsf{PO_S}$), and returns $(O_{j+1}^{m'}, P_{j+1})$. $\mathcal{O}.\mathsf{PO_S}$ does the following to "process" an onion O:
   i. If $O = O'$ for some stored pair (O', m') and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) = (\mathsf{R}, m')$, then return $(\mathsf{S}, (\ell, m'))$.
   ii. If $O = O'$ for some stored pair (O', m') and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) \neq (\mathsf{R}, m')$, then fail.
   iii. If $O \neq O'$ for any stored pair (O', m') but $O = (H', C)$ for some stored pair $((H', C'), m')$ and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) = (\mathsf{R}, \perp)$, then return $(\mathsf{S}, \perp)$.
   iv. If $O \neq O'$ for any stored pair (O', m') but $O = (H', C)$ for some stored pair $((H', C'), m')$ and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) \neq (\mathsf{R}, \perp)$, then fail.

   All other queries are processed as before.

Variant (c) Below, we provide a description of steps 4 and 5 of the repliable-onion security game, $\mathsf{ROSecurityGame}$, in Section 4 for case (c).

4. $\mathcal{A}$ chooses a label $\ell \in \mathcal{L}(1^\lambda)$ and a message $m \in \mathcal{M}(1^\lambda)$. $\mathcal{A}$ also chooses a forward path $P^\rightarrow = (P_1, \ldots, P_d)$ and a return path $P^\leftarrow = (P_{d+1}, \ldots, P_s)$

such that (i) if $P^{\leftarrow}$ is non-empty, then it ends with S, (ii) I doesn't appear on the $P^{\rightarrow}$, and (iii) I appears somewhere on $P^{\leftarrow}$. $\mathcal{A}$ sends to $\mathcal{C}$ the parameters for the challenge onion: ℓ, m, $P^{\rightarrow}$, the public keys $\mathsf{pk}(P^{\rightarrow})$ of the parties in $P^{\rightarrow}$, $P^{\leftarrow}$, and the public keys $\mathsf{pk}(P^{\leftarrow})$ of the parties in $P^{\leftarrow}$.

5. $\mathcal{C}$ samples a bit $b \leftarrow_{\$} \{0,1\}$.

   If $b = 0$, $\mathcal{C}$ runs $\mathsf{FormOnion}$ on the parameters specified by $\mathcal{A}$, i.e., $((O_1^0, \ldots, O_d^0), H^{\leftarrow}, \kappa) \leftarrow \mathsf{FormOnion}(\ell, m, P^{\rightarrow}, \mathsf{pk}(P^{\rightarrow}), P^{\leftarrow}, \mathsf{pk}(P^{\leftarrow}))$.
   In this case, the oracles—$\mathcal{O}.\mathsf{PO}_\mathsf{I}(\cdot)$, $\mathcal{O}.\mathsf{FR}_\mathsf{I}(\cdot,\cdot)$, $\mathcal{O}.\mathsf{PO}_\mathsf{S}(\cdot)$, and $\mathcal{O}.\mathsf{FR}_\mathsf{S}(\cdot,\cdot)$—remain unmodified.

   Otherwise, if $b = 1$, $\mathcal{C}$ performs the "switch" at honest party P_j on the return path $P^{\leftarrow}$, where P_j is the first appearance of I on the routing path. $\mathcal{C}$ runs $\mathsf{FormOnion}$ on input a random label $x \leftarrow_{\$} \mathcal{L}(1^\lambda)$, the message m (that had been chosen by $\mathcal{A}$ in step 4), the forward path $P^{\rightarrow}$, and the "truncated" return path $p^{\leftarrow} = (P_{d+1}, \ldots, P_j)$, i.e., $(O^{\rightarrow}, (H_{d+1}^1, \ldots, H_j^1), \kappa) \leftarrow \mathsf{FormOnion}(x, m, P^{\rightarrow}, \mathsf{pk}(P^{\rightarrow}), p^{\leftarrow}, \mathsf{pk}(p^{\leftarrow}))$.

   We modify the oracles as follows. $\mathcal{O}.\mathsf{PO}_\mathsf{I}$ does the following to "process" an onion O:

   i. If $O = (H_j^1, C)$ for some content C and $\mathsf{ProcOnion}(O, P_j, \mathsf{sk}(P_j)) = (\mathsf{R}, m')$ for some message m' (possibly equal to "$\perp$"), then runs $\mathsf{FormOnion}$ on a random label x', a random message y', the remainder the return path $q^{\leftarrow} = (P_{j+1}, \ldots, P_s)$ as the forward path, and the empty return path "()", i.e., $((O_{j+1}^{m'}, \ldots, O_s^{m'}), (), \kappa^{m'}) \leftarrow \mathsf{FormOnion}(x', y', q^{\leftarrow}, \mathsf{pk}(q^{\leftarrow}), (), ())$, stores the pair $(O_s^{m'}, m')$ (such that the pair is accessible by $\mathcal{O}.\mathsf{PO}_\mathsf{S}$), and returns $(O_{j+1}^{m'}, P_{j+1})$.

   ii. If $O = (H_j^1, C)$ for some content C and $\mathsf{ProcOnion}(O, P_j, \mathsf{sk}(P_j)) \neq (\mathsf{R}, m')$ for some message m', then fails.

   $\mathcal{O}.\mathsf{PO}_\mathsf{S}$ does the following to "process" an onion O:

   iii. If $O = O'$ for some stored pair (O', m') and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) = (\mathsf{R}, m')$, then return $(\mathsf{S}, (\ell, m'))$.

   iv. If $O = O'$ for some stored pair (O', m') and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) \neq (\mathsf{R}, m')$, then fail.

   v. If $O \neq O'$ for any stored pair (O', m') but $O = (H', C)$ for some stored pair $((H', C'), m')$ and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) = (\mathsf{R}, \perp)$, then return $(\mathsf{S}, \perp)$.

   vi. If $O \neq O'$ for any stored pair (O', m') but $O = (H', C)$ for some stored pair $((H', C'), m')$ and $\mathsf{ProcOnion}(O, P_s, \mathsf{sk}(P_s)) \neq (\mathsf{R}, \perp)$, then fail.

   All other queries are processed as before.