Cryptographic Shallots: A Formal Treatment of Repliable Onion Encryption

Megumi Ando and Anna Lysyanskaya

Computer Science Department, 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 — one per intermediary — and sends the resulting "onion" to the first intermediary. Each intermediary "peels" 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 (e.g., Tor, Mixminion), 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.

In this paper, we propose an ideal functionality for a *repliable* onion encryption scheme and provide a construction that UC-realizes it.

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. In order 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*.

Onion routing is at the heart of Tor [13, 16], which is a tool used by millions of people to obscure their communication and browsing patterns. Although the security properties guaranteed by the Tor protocol are a subject of study and debate, it appears clear that, because of its scalable nature — the more "onions" are sent over the network, the better the protocol hides the origin of each onion, — as well as its fault tolerance — if some of the intermediaries fail or are malicious, an onion won't get it its destination, but another one can be sent on instead, and other onions are not affected, — onion routing is the favored method for achieving anonymity over the Internet [1].

As originally proposed by Chaum [8], 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 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 [4] introduced *onion encryption*, a tailor-made public-key encryption scheme where each onion layer looks the same and have the same length and you can't tell how far an intermediary, e.g. Carol, is from an onion's destination, even if you are Carol. They gave an ideal functionality [5] for 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. [19] point out, there were significant definitional issues. Also, as, for example, Ando et al. [1, 2] 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, for example, browse the Web anonymously, or anonymously download and fill out a feedback form, or carry out most Internet tasks anonymously, a two-way channel between Alice and Bob needs to be established. Although CL point out that their construction can be modified to potentially allow two-way communication, this is nothing more than a suggestion, since they don't provide any definitions or proofs.

Babel [17], Mixminion [13], Minx [15] and Sphinx [14] all provide mechanisms for the recipient to reply to the sender but don't provide any formal definitions or proofs either. In fact, Kuhn et al. [19] pointed out a fatal security flaw in the state-of-the-art, Sphinx.

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/onion

she has received is authentic or has been "mauled;" Iris can then refuse to decrypt/process a "mauled" ciphertext. Most constructions of CCA2 secure encryption scheme work along these lines; that way, in the proof of security, the decryption oracle does not need to worry about decrypting/processing onions that do not pass such a validity check, making it possible to prove security. This approach was made more explicit by Cramer and Shoup [11, 12] 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 in order 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 *any* 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.

Another difficult technical issue to address is how to make sure that reply onions are indistinguishable (even to intermediaries who process them) from forward onions. As pointed out in prior work [13], this is crucial because "replies may be very rare relative to forward messages, and thus much easier to trace."

Our contribution. In this paper, we define and realize repliable onion encryption.

We define security by describing an ideal functionality in the simplified UC model [6]. We chose the simplified UC 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.

As should be expected of secure onion routing, our functionality 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 proceed an honest intermediary.

To process an onion, an honest party P sends it to the functionality, 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 the next bit representation of the same onion (again, formed independently of the input). If P is the recipient, it 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. The details of this functionality are relatively involved; we describe it in Section 3.

Our functionality 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. The part of the onion that potentially carries the message, or the response to an original message, does not undergo an integrity check until it gets to its destination. This is the way in which we overcome the challenge (described above) of satisfying a CCA2-style definition and yet enabling replies.

Next, in Section 4, 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 block cipher and a collision-resitant hash function. We then show that it resolves, in the affirmative, the problem that CL left open fifteen years ago of constructing provably secure repliable onion encryption.

To make CCA2-style security possible even while allowing replies, we split up the onion into two pieces. The first piece is the *header* H that contains (in layered encryption form) the routing information and symmetric keys that are needed to process the second piece. The second piece is the content C that contains the message and, in case this is a forward onion, instructions for forming the reply onion; this part wrapped is 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 PRP keyed by k to the contents C. In a nutshell, 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.)

As a byproduct of our proof, given in Section 5, we provide a cryptographic, game-based definition of security for repliable onion encryption. This definition is interesting because we show that a scheme satisfying it will SUC-realize our ideal functionality. We also show that our scheme satisfies it.

Related work. Recently, Kuhn, Beck and Strufe [19] showed an attack on several onion-routing systems [9, 10, 14, 20], and traced it to a flaw in the CL paper. Namely, they found a mistake in the CL proof of security, and showed that CL's formulation of onion-security was insufficient to UC-realize their ideal functionality. The broken systems, although lacking a formal proof of security, relied on the CL definition of onion-security to justify their architecture. Kuhn, Beck and Strufe (KBS) showed how to strengthen onion-security in such a way that a resulting onion encryption scheme UC-realizes the CL ideal functionality. However, they do not tackle repliable onions, nor does their paper contain any onion

encryption scheme of their own (other than proposed fixes, without security proofs, that would eliminate their attacks on the broken systems).

2 Repliable onion encryption: syntax and correctness

In this paper, an onion O is a pair, consisting of the (encrypted) content C and the header H, i.e., $O = (H, C)$.

We assume that the upper bound N on the length of the forward or return path is one of the public parameters pp.

Here, we give the formal input/output (I/O) specification for a repliable onion encryption scheme. In contrast to the I/O specification for an (unrepliable) onion encryption scheme given by Camenisch and Lysyanskaya [4], 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.

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 which message m' is responding to.

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 $\mathsf{FormOnion}$ *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.,*

$$(O^{\rightarrow}, H^{\leftarrow}, \kappa) \leftarrow \mathsf{FormOnion}(\ell, m, P^{\rightarrow}, \mathsf{pk}(P^{\rightarrow}), P^{\leftarrow}, \mathsf{pk}(P^{\leftarrow}))$$

- $\mathsf{ProcOnion}$ *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 $\mathsf{ProcOnion}$ *returns* $(\mathsf{role}, \mathsf{output})$*, i.e.,*

$$(\mathsf{role}, \mathsf{output}) \leftarrow \mathsf{ProcOnion}(O, P, \mathsf{sk}(P)).$$

When $\mathsf{role} = \mathsf{I}$ *(for "intermediary"),* output *is the pair (O', P') consisting of the peeled onion O' and the next destination P' of O'. When* $\mathsf{role} = \mathsf{R}$*,* output *is the message m for recipient P. When* $\mathsf{role} = \mathsf{S}$*,* output *is the pair (ℓ, m) consisting of the label ℓ and the reply message m for sender P.*

– $\mathsf{FormReply}$ *is a d.p.t. algorithm for forming return onions. 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* $\mathsf{FormReply}$ *returns the peeled onion O' and the next destination P' of O', i.e.,*

$$(O', P') \leftarrow \mathsf{FormReply}(m, O, P, \mathsf{sk}(P)).$$

$\mathsf{FormReply}$ *outputs $(\perp, \perp)$ if O is "not repliable".*

2.1 Onion evolutions, forward paths, return paths and layerings

Now that we have defined the I/O specification for a repliable onion encryption scheme, we can 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 [4].

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 there be honest parties with honestly formed keys.

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

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

$$((H_{d+1}, C_{d+1}), P_{i+1}) = \mathsf{FormReply}(m, (H_d, C_d), P_d, \mathsf{sk}(P_d)).$$

Let $s = d$ if (H_d, C_d) is "not repliable", i.e., $((H_{d+1}, C_{d+1}), P_{i+1}) = (\perp, \perp)$. Otherwise, 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 $i \in \{d+1, \ldots, s-1\}$, 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 $\mathcal{E}(H_1, C_1, P_1, m) = ((H_1, C_1, P_1), \ldots, (H_s, C_s, P_s))$ 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 $\mathcal{P}^{\rightarrow}(H_1, C_1, P_1, m) = (P_1, \ldots, P_d)$ is its *forward path*; the sequence $\mathcal{P}^{\leftarrow}(H_1, C_1, P_1, m) = (P_{d+1}, \ldots, P_s)$ is its *return path*; and the sequence $\mathcal{L}(H_1, C_1, P_1, m) = (H_1, C_1, \ldots, H_d, C_d, H_{d+1}, \ldots, H_s)$ is its *layering*.

We define correctness as follows:

Definition 2 (Correctness). *Let* G, FormOnion, ProcOnion *and* 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* pp*). Let* $P = (P_1, \ldots, P_s)$, $|P| = s \leq 2N$ *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* $(\mathsf{pk}(P_i), \mathsf{sk}(P_i)) \leftarrow G(1^\lambda, \mathsf{pp}, P_i)$.

Let $P^{\rightarrow} = (P_1, \ldots, P_d)$, *and let* $\mathsf{pk}(P^{\rightarrow})$ *be a shorthand for the public keys associated with the parties in* $P^{\rightarrow}$. *Let* $P^{\leftarrow} = (P_{d+1}, \ldots, P_s)$, *and let* $\mathsf{pk}(P^{\leftarrow})$ *be a shorthand for the public keys associated with the parties in* $P^{\leftarrow}$.

Let $((H_1, C_1), \ldots, (H_d, C_d), H_{d+1}, \ldots, H_s, \kappa)$ *be the output of* FormOnion *on input the label* ℓ, *the message* m, *the forward path* $P^{\rightarrow} = (P_1, \ldots, P_d)$, *the public keys* $\mathsf{pk}(P^{\rightarrow})$ *associated with the parties in* $P^{\rightarrow}$, *the return path* $P^{\leftarrow} = (P_{d+1}, \ldots, P_s)$ *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.***
   - $\mathcal{P}^{\rightarrow}(H_1, C_1, P_1, m') = (P_1, \ldots, P_d)$.
   - *For every* $i \in [d]$ *and content* C *such that* $|C| = |C_i|$, $\mathcal{P}^{\rightarrow}(H_i, C, P_i, m') = (P_i, \ldots, P_d)$.

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

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

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* $\mathcal{E}(H_1, C_1, P_1, m')$.

Remark. 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 ProcOnion and FormReply must be deterministic processes given the secret keys.

3 Onion Routing in the SUC Framework

In this section, we provide a formal definition of security for repliable onion encryption schemes. We chose to define security in the simplified universal composability (SUC) model [6] as opposed to the universal composability (UC) model [5] as this choice greatly simplifies how communication is modeled, in turn, allowing for a more easily digestible description of the ideal functionality. Additionally, since SUC-realizability implies UC-realizability [6], we do not lose generality by simplifying the model in this manner.

Communication model. 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 $\mathcal{R}$.

We first describe the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ (ROES, for "repliable onion encryption scheme") for the repliable onion encryption scheme. See Figure 1 for a summary of $\mathcal{F}_{\mathsf{ROES}}$.

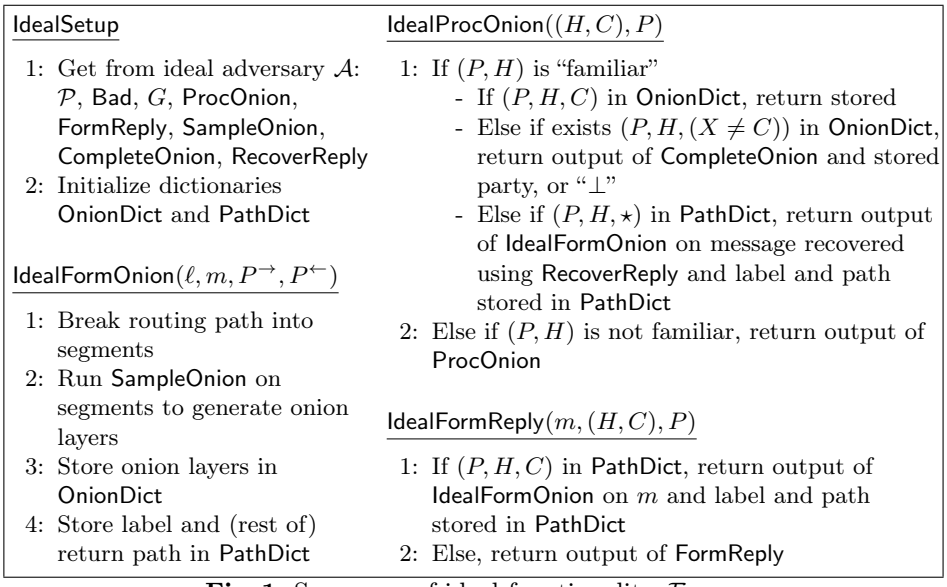

Fig. 1: Summary of ideal functionality $\mathcal{F}_{\mathsf{ROES}}$.

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

Notation. 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 (encrypted) content C and the header H, i.e., $O = (H, C)$.

The ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ handles requests from the environment (to form an onion, process an onion or form a return onion) on behalf of the ideal honest parties.

Setting up. Each static setting for a fixed set of participants and a fixed public key infrastructure requires a separate setup. During setup, $\mathcal{F}_{\mathsf{ROES}}$ gets the following from the ideal adversary $\mathcal{A}$. (For each algorithm in items (iv)-(vi), 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 $\mathcal{P}$
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.*
iv. the p.p.t. algorithm $\mathsf{SampleOnion}(1^\lambda, \mathsf{pp}, p^\rightarrow, p^\leftarrow, m)$ 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 $(p^\rightarrow, p^\leftarrow) = (p_1, \ldots, p_i, P_{i+1})$ is always a sequence $(p_1, \ldots, p_i)$ of adversarial parties, possibly ending in an honest party P_{i+1}. $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ fails if $\mathsf{SampleOnion}$ ever samples a repeating header or key.

   $\mathsf{SampleOnion}$ *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},* $\mathsf{SampleOnion}$ *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},* $\mathsf{SampleOnion}$ *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,* $\mathsf{SampleOnion}$ *produces an onion o for the first party p_1 in $p^\rightarrow$.*
v. the p.p.t. algorithm $\mathsf{CompleteOnion}(1^\lambda, \mathsf{pp}, H', C)$ that takes as input the security parameter 1^λ, the public parameters pp, the identity of the party P, the header H' and the content C, and outputs an onion $O = (H', C')$. $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ fails if $\mathsf{CompleteOnion}$ ever produces a repeating onion.

   $\mathsf{CompleteOnion}$ *produces an onion (H', C') that resembles the result of peeling an onion with content C.*
vi. the d.p.t. algorithm $\mathsf{RecoverReply}(1^\lambda, \mathsf{pp}, O, P)$ that takes as input the security parameter 1^λ, the public parameters 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.

$\mathcal{F}$ creates a copy $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ of itself for handling instructions pertaining to session-id sid and sends items (i)-(vi) to $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$; these items pertain to the same static public key infrastructure setting.

$\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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 [7].)

$\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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"
  - a method $\mathsf{lookup}(P, H, C)$ that looks up the entry $(\mathsf{role}, \mathsf{output})$ corresponding to the label (P, 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
  - a method $\mathsf{lookup}(P, H, C)$ that looks up the entry $(P^{\leftarrow}, \ell)$ corresponding to the label (P, H, C)

These data structures are stored internally at and are 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 $\mathcal{A}$, by which we mean the following:

- 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 properties, the ideal functionality partitions the forward path $P^{\rightarrow}$ into segments, where each segment starts with a sequence of corrupt parties and ends with a single honest party:

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 $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ samples onions (h_i, c_i) and (H_j, C_j) using the algorithm $\mathsf{SampleOnion}$, i.e., $((h_i, c_i), (H_j, C_j)) \leftarrow \mathsf{SampleOnion}(1^\lambda, \mathsf{pp}, (p_i, \ldots, p_{j-1}, P_j), (), \bot)$.

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 $P^{\leftarrow}$. If the recipient p_d is corrupt, then $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ also runs $\mathsf{SampleOnion}(1^\lambda, \mathsf{pp}, (p_{e+1}, \ldots, p_d), (p_{d+1}, \ldots, p_{\mathsf{next}(d)-1}, P_{\mathsf{next}(d)}), m)$; 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, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ stores just the role "R" and the message m.

If the recipient P_d is honest, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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".)

See Appendix A.1 for the pseudocode for $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$'s "onion forming" algorithm.

Example 1. 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

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

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

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

and stores in $\mathsf{PathDict}$:

$$\mathsf{PathDict.put}((P_7, H_7, C_7), ((p_8, p_9, P_{10}, p_{11}, P_{12}), \ell)).$$

Example 2. The recipient p_7 is corrupt. 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,

$$\begin{aligned}
(\bot, (H_1, C_1)) &\leftarrow \mathsf{SampleOnion}(1^\lambda, \mathsf{pp}, (P_1), (), \bot) \\
((h_2, c_2), (H_4, C_4)) &\leftarrow \mathsf{SampleOnion}(1^\lambda, \mathsf{pp}, (p_2, p_3, P_4), (), \bot) \\
(\bot, (H_5, C_5)) &\leftarrow \mathsf{SampleOnion}(1^\lambda, \mathsf{pp}, (P_5), (), \bot) \\
(o_6, H_{10}) &\leftarrow \mathsf{SampleOnion}(1^\lambda, \mathsf{pp}, (p_5, p_6, p_7), (p_8, p_9, P_{10}), m).
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{OnionDict.put}((P_1, H_1, C_1), (\mathsf{I}, ((h_2, c_2), p_2))) \\
&\mathsf{OnionDict.put}((P_4, H_4, C_4), (\mathsf{I}, ((H_5, C_5), P_5))) \\
&\mathsf{OnionDict.put}((P_5, H_5, C_5), (\mathsf{I}, ((h_6, c_6), p_6))),
\end{aligned}$$

and stores in $\mathsf{PathDict}$:

$$\mathsf{PathDict.put}((P_{10}, H_{10}, *), ((p_{11}, P_{12}), \ell)).$$

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 $\mathcal{R}$).

The ideal functionality first checks if there is an entry under (P, H, C) in the dictionary $\mathsf{OnionDict}$.

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 $(\mathsf{I}, ((H', X'), P'), k)$ under the label (P, H, X) in $\mathsf{OnionDict}$. In this case, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ samples an onion $(H', C') \leftarrow \mathsf{CompleteOnion}(1^\lambda, \mathsf{pp}, H', C)$, stores the new entry $(\mathsf{I}, ((H', C'), P'))$ under the label (P, H, C) in $\mathsf{OnionDict}$ and responds to P with $(\mathsf{I}, ((H', X'), P'))$.

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 $\mathsf{OnionDict}$. In this case, $\mathcal{F}^{\mathsf{sid}}_{\mathsf{ROES}}$ responds to P with $(\mathsf{R}, \bot)$.

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 $\mathsf{OnionDict}$. In this case, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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 $\mathsf{PathDict}$. Let m' be the message obtained from running $\mathsf{RecoverReply}(1^\lambda, \mathsf{pp}, O, P)$.

If $P^{\leftarrow}$ is not empty, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ runs its "form onion" code ($\mathsf{IdealFormOnion}$ in Figure 7, Appendix A.1) with (ℓ, m') as the "message", $P^{\leftarrow}$ as the forward path and the empty list "()" as the return path. (The "form onion" code is run with auxiliary information for correctly labeling the last party in $P^{\leftarrow}$ as the sender.) In this case, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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 $(\mathsf{role}, \mathsf{output}) = \mathsf{ProcOnion}(1^\lambda, \mathsf{pp}, O, P, \mathsf{sk}(P))$.

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 $\mathcal{R}$).

Upon receiving the forwarded request, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ looks up (P, H, C) in $\mathsf{PathDict}$.

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 ($\mathsf{IdealFormOnion}$ in Figure 7, Appendix A.1) with (ℓ, m) as the "message", $P^{\leftarrow}$ as the forward path and the empty list "()" as the return path. (The "form onion" 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 $\mathcal{F}_{\mathsf{ROES}}$

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 the participants' 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 the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$ 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 routing protocol (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 $\mathcal{S}$ such that for every polynomial-time balanced environment $\mathcal{Z}$, there exists a negligible function $\nu(\lambda)$ such that*

$$\left|\Pr\left[\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{ROES}},\mathcal{A},\mathcal{Z}}(1^\lambda, \mathsf{pp}) = 1\right] - \Pr\left[\mathsf{REAL}_{\Sigma,\mathcal{A},\mathcal{Z}}(1^\lambda, \mathsf{pp}) = 1\right]\right| \leq \nu(\lambda).$$

3.3 Remarks

On the assumption that keys are consistent with PKI. 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 inputs are well-behaved, i.e., router names are valid, and keys are as published.

On replayed onions. As originally noted by Camenisch and Lysyanskaya [4], 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.

4 Our repliable onion encryption scheme

In this section, we provide our construction of a repliable onion encryption scheme dubbed "*Shallot Encryption Scheme*". Like in Camenisch and Lysyanskaya's construction [4], each onion layer for a party P is encrypted under a key k which, in turn, is encrypted under the public key of P and a tag t. Our construction differs from the original 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 (MAC). The resulting object is more like a shallot than an onion; it consists of *two* 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".

Notation. Let λ denote the security parameter. Let $\{F_{\mathsf{seed}(\cdot,\cdot)}\}_{\mathsf{seed}\in\{0,1\}^*}$ be a pseudo-random function (PRF) of two inputs. 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, 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 [4], let $\{X\}_k$ denote $f_k(X)$ (or $g_k(X)$) , and let $\}X\{_k$ 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 [11], let $\mathsf{MAC} = (\mathsf{Gen}_{\mathsf{MAC}}, \mathsf{Tag}, \mathsf{Ver})$ be a MAC, and let h be a collision-resistant hash function.

4.1 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., $(\mathsf{pk}(P_i), \mathsf{sk}(P_i)) \leftarrow \mathsf{Gen}_{\mathcal{E}}(1^{\lambda}, \mathsf{pp}, P_i)$.

4.2 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:

$$P^{\rightarrow} = P_1, \ldots, P_{d-1}, P_d, \qquad |P^{\rightarrow}| = d \leq N,$$

and a *return* path to herself:

$$P^{\leftarrow} = P_{d+1}, \ldots, P_{s-1}, P_s, \qquad |P^{\leftarrow}| = s - d + 1 \leq N.$$

In other words, Bob is P_d, and Alice is P_s. 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, e.g., in a dictionary.

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 $d = s - d + 1 = N$.

Let O be an onion peelable by party P. The *header* of O is a sequence $H = (E, B^1, \ldots, B^{N-1})$. E is an encryption under the tag $t = h(B^1, \ldots, B^{N-1})$ of (i) P's role, (ii) P's encryption key k (or label ℓ) and (iii) authentication key K (if it exists). The decryption $\}B^1\{_k$ reveals the destination P' and the ciphertext E' of the peeled onion. For each $1 < j < N$, the decryption $\}B^j\{_k$ 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 "decrypting" the all-zero string of length $L_1(\lambda)$, i.e., $(B')^{N-1} = \}0 \ldots 0\{_k$.

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): Let $B_d^1 = \}\bot, \bot\{_{k_d}$. For every $i \in \{2, \ldots, N-1\}$, 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 [3, 18].) See FormHeader in Figure 2.

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 FormContent in Figure 2.

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 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.

4.3 Processing a repliable onion (in the forward direction)

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.,

$$(\mathsf{I}, k_i) = \mathsf{Dec}(\mathsf{sk}(P_i), t_i, E_i).$$

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:

$$B_{i+1}^j = \}B_i^{j+1}\{_{k_i} \qquad \forall j \in [N-2].$$

To obtain the last block of the processed header, Carol "decrypts" the all-zero string "$(0 \ldots 0)$":

$$B_{i+1}^{N-1} = \}0 \ldots 0\{_{k_i}.$$

To process the content, Carol simply decrypts the current content C_i:

$$C_{i+1} = \}C_i\{_{k_i}.$$

See ProcOnion in Figure 3.

4.4 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 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}. See ProcOnion and FormReply in Figures 3 and 4.

$\mathsf{SES.FormOnion}(\ell, m, (P_1, \ldots, P_d), P^{\leftarrow})$

$(H_1, \ldots, H_d, k_1, \ldots, k_d, K_d) \leftarrow \mathsf{FormHeader}(\rightarrow, \ell, (P_1, \ldots, P_d))$
$(C_1, \ldots, C_d), H^{\leftarrow}, k^{\leftarrow}) \leftarrow \mathsf{FormContent}(\ell, m, P^{\leftarrow}, k_1, \ldots, k_d, K_d)$
return $(((H_1, C_1), \ldots, (H_d, C_d)), H^{\leftarrow}, ((k_1, \ldots, k_d, K_d), k^{\leftarrow}))$

$\mathsf{SES.FormHeader}(\mathsf{direction}, \ell, (P_1, \ldots, P_N))$

$k_j = F_{\mathsf{seed}}(\ell, j)\,, \forall j \in [N]$
$K_N = \mathsf{Gen}_{\mathsf{MAC}}(1^{\lambda}, F_{\mathsf{seed}}(N, \ell))$
$B_N^1 = \{\perp, \perp\}_{k_N}$
$B_N^i = \}\ldots\}0\ldots0\{_{k_i}\ldots\{_{k_{N-1}}\,, \forall i \in \{2, \ldots, N-1\}$
if direction $=\rightarrow$
  $E_N \leftarrow \mathsf{Enc}(\mathsf{pk}(P_N), h(B_N^1, \ldots, B_N^{N-1}), (\mathsf{R}, k_N, K_N))$
else (if direction $=\leftarrow$)
  $E_N \leftarrow \mathsf{Enc}(\mathsf{pk}(P_N), h(B_N^1, \ldots, B_N^{N-1}), (\mathsf{S}, \ell))$
$H_N = (E_N, B_N^1, \ldots, B_N^{N-1})$
for all j from $N-1$ to 1
  $B_j^1 = \{P_N, E_N\}_{k_j}$
  $B_j^i = \{B_{j+1}^{i-1}\}_{k_j}\,, \forall i \in \{2, \ldots, N-1\}$
  $E_j \leftarrow \mathsf{Enc}(\mathsf{pk}(P_j), h(B_j^1, \ldots, B_j^{N-1}), (\mathsf{I}, k_j))$
  $H_j = (E_j, B_j^1, \ldots, B_j^{N-1})$
return $(H_1, \ldots, H_N, k_1, \ldots, k_N, K_N)$

$\mathsf{SES.FormContent}(\ell, m, (P_{d+1}, \ldots, P_s), k_1, \ldots, k_d, K_d)$

$(H_{d+1}, \ldots, H_s, k_{d+1}, \ldots, k_s, K_s) \leftarrow \mathsf{FormHeader}(\leftarrow, \ell, (P_{d+1}, \ldots, P_s))$
$M = (H_{d+1}, m, k_s, K_s, P_{d+1})$
$\sigma_d = \mathsf{Tag}(K_d, M)$
$C_d = \{M, \sigma_d\}_{k_d}$
for all j from $d-1$ to 1
  $C_j = \{C_{j+1}\}_{k_j}$
return $((C_1, \ldots, C_d), (H_{d+1}, \ldots, H_s), (k_1, \ldots, k_d, K_d))$

Fig. 2: Pseudocode for FormOnion. FormOnion takes as input the label ℓ, the message m, the forward path $P^{\rightarrow}$ and the return path $P^{\leftarrow}$ (and the public keys associated with the routing path), and outputs onions $O^{\rightarrow}$, headers $H^{\leftarrow}$ and associated keys κ.

SES.ProcOnion$(((E, B^1, \ldots, B^{N-1}), C), P)$

1 : $(\mathsf{role}, \mathsf{key}, H', P') \leftarrow \mathsf{ProcHeader}((E, B^1, \ldots, B^{N-1}), P)$
2 : $C' = \mathsf{ProcContent}(\mathsf{role}, \mathsf{key}, C)$
3 : **if** $\mathsf{role} = \mathsf{I}$, **return** $(\mathsf{I}, ((H', C'), P'))$
4 : **else**, **return** (role, C')

SES.ProcHeader$((E, B^1, \ldots, B^{N-1}), P)$

1 : $t = h(B^1, \ldots, B^{N-1})$
2 : $(\mathsf{role}, \mathsf{key}) = \mathsf{Dec}(\mathsf{sk}(P), t, E)$
3 : **if** $\mathsf{role} = \mathsf{I}$
4 : $\quad (P', E') = \}B^1\{_{\mathsf{key}}$
5 : $\quad (B')^j = \}B^{j+1}\{_{\mathsf{key}}, \forall j \in [N-2]$
6 : $\quad (B')^{N-1} = \}0 \ldots 0\{_{\mathsf{key}}$
7 : $\quad$ **return** $(\mathsf{I}, \mathsf{key}, (E', (B')^1, \ldots, (B')^{N-1}), P')$
8 : **else**
9 : $\quad$ **return** $(\mathsf{role}, \mathsf{key}, \perp, \perp)$

SES.ProcContent$(\mathsf{role}, \mathsf{key}, C)$

1 : **if** $\mathsf{role} = \mathsf{I}$
2 : $\quad$ **return** $\}C\{_{\mathsf{key}}$
3 : **if** $\mathsf{role} = \mathsf{R}$
4 : $\quad$ parse $\mathsf{key} = (k_d, K_d)$
5 : $\quad ((H_{d+1}, m, k_s, K_s, P_{d+1}), \sigma_d) = \}C\{_{k_d}$
6 : $\quad$ **if** $\mathsf{Ver}(K_d, (H_{d+1}, m, k_s, K_s, P_{d+1}), \sigma_d) = 1$, **return** m
7 : $\quad$ **else**, **return** $\perp$
8 : **else**
9 : $\quad$ reconstruct keys $k_{d+1}, \ldots, k_s, K_s$ from key
10 : $\quad (m, \sigma_s) = \}\{\ldots\{C\}_{k_{s-1}} \ldots\}_{k_{d+1}}\{_{k_s}$
11 : $\quad$ **if** $\mathsf{Ver}(K_s, m, \sigma_s) = 1$, **return** (ℓ, m)
12 : $\quad$ **else**, **return** $\perp$

Fig. 3: Pseudocode for ProcOnion. ProcOnion takes as input the onion $((E, B^1, \ldots, B^{N-1}), C)$ and the party P (and the secret key of P), and returns a role (either I, R or S) and an output (either an onion and next destination $((H', C'), P')$ or a decrypted content C').

4.5 Processing a repliable onion (in the return direction)

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; he also runs ProcOnion in Figure 3.

SES.FormReply$(m', (H, C), P)$

1 : $(\mathsf{role}, \mathsf{key}, H', P') \leftarrow \mathsf{ProcHeader}(H, P)$
2 : **if** $\mathsf{role} = \mathsf{R}$
3 : parse $\mathsf{key} = (k_d, K_d)$
4 : $((H_{d+1}, m, k_s, K_s, P_{d+1}), \sigma_d) = \}C\{_{k_d}$
5 : **if** $\mathsf{Ver}(K_d, (H_{d+1}, m, k_s, K_s, P_{d+1}), \sigma_d) = 1$
6 : $\sigma_s = \mathsf{Tag}(K_s, m')$
7 : **return** $((H_{d+1}, \{m', \sigma_s\}_{k_s}), P_{d+1})$
8 : **else**, **return** $(\perp, \perp)$
9 : **else**, **return** $(\perp, \perp)$

Fig. 4: Pseudocode for FormReply. FormReply takes as input the reply message m', the onion (H, C) and the party P (and the secret key of P), and returns a return onion $(H_{d+1}, \{m', \sigma_s\}_{k_s})$ and next destination P_{d+1}.

4.6 Reading the reply

When Alice receives the onion O_s, she retrieves the reply from Bob by first processing the onion, by running 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 $\mathsf{Ver}(K_s, m', \sigma_s)$. See ProcOnion in Figure 3.

5 Shallot Encryption Scheme SUC-realizes $\mathcal{F}_{\mathsf{ROES}}$

In this section, we prove that our construction (in Section 4) SUC-realizes the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$.

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

$$\left|\Pr\left[\mathsf{IDEAL}_{\mathcal{F}_{\mathsf{ROES}}, \mathcal{A}, \mathcal{Z}}(1^\lambda, \mathsf{pp}) = 1\right] - \Pr\left[\mathsf{REAL}_{\mathsf{SES}, \mathcal{A}, \mathcal{Z}}(1^\lambda, \mathsf{pp}) = 1\right]\right| \leq \nu(\lambda).$$

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, ProcOnion and FormReply, (iv) the algorithm SampleOnion, (v) the algorithm CompleteOnion and (vi) the algorithm 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 5 for a summary of the simulator.

Send to $\mathcal{F}_{\mathsf{ROES}}$:	$\underline{\mathsf{SampleOnion}(p^{\rightarrow}, p^{\leftarrow}, m)}$
$\mathcal{P}$, Bad, G, ProcOnion, FormReply, SampleOnion, CompleteOnion, RecoverReply $\underline{\mathsf{CompleteOnion}(H', C)}$	SampleOnion just runs FormOnion on the segments $p^{\rightarrow}$ and $p^{\leftarrow}$ using the all-zero 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).
Return the header H' along with a randomly chosen string C'	$\underline{\mathsf{RecoverReply}(O, P)}$ Return output of ProcOnion

Fig. 5: Summary of simulator $\mathcal{S}$

5.1 Description of simulator $\mathcal{S}$

Sampling an onion. Let $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ denote the ideal functionality corresponding to the static setting. When the ideal functionality $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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 $P^{\leftarrow}$, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ partitions the routing path $(P^{\rightarrow}, P^{\leftarrow})$ into "segments" where each segment is a sequence of adversarial parties that may end in a single honest party. (See Section 3.1 for a more formal description of these segments.) $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ runs the algorithm SampleOnion independently on each segment of the routing path. Additionally, if the forward path ends in a corrupt party, $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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 $\mathsf{SampleOnion}$ 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 $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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 $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ assigned to the peeled onion (H', X') of (H, X).

To do this, the algorithm $\mathsf{CompleteOnion}(1^\lambda, \mathsf{pp}, H', C)$ samples a random string $C' \leftarrow\!\!\$\, \{0,1\}^{L_2(\lambda)}$, where $\{0,1\}^{L_2(\lambda)}$ corresponds to the blocklength for the PRP (in the construction), and outputs (H', C').

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 $\mathcal{F}_{\mathsf{ROES}}^{\mathsf{sid}}$ 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.

5.2 A cryptographic definition of security

To prove that our onion encryption scheme SUC-realizes the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$, it will be useful to first show that the scheme satisfies *repliable-onion security*. Informally, an onion encryption scheme is repliable-onion secure if the adversary cannot tell (i) whether an honest receiver of an honestly formed onion is an intermediary for the onion or the recipient, (ii) whether an honest transmitter of an honestly formed onion is an intermediary for the onion or the sender and (iii) how far an honestly formed onion is from its origin and destination.

Formally, we define repliable-onion security using the game, $\mathsf{ROSecurityGame}$. See Figure 6 for a summary of the game.

1: $\mathcal{A}$ picks honest parties' router names Q_1 and Q_2
2: $\mathcal{C}$ sets keys for honest parties
3: $\mathcal{A}$ gets oracle access to oracles—$\mathcal{O}.\mathsf{PO}_1$, $\mathcal{O}.\mathsf{FR}_1$, $\mathcal{O}.\mathsf{PO}_2$ and $\mathcal{O}.\mathsf{FR}_2$— for processing onions and replying to them on behalf of Q_1 and Q_2
4: $\mathcal{A}$ provides input for challenge onion
5: $\mathcal{C}$ flips a coin $b \leftarrow_{\$} \{0,1\}$
6: If $b = 0$, $\mathcal{C}$ forms onion specified by $\mathcal{A}$
7: If $b = 1$, $\mathcal{C}$ forms onion with "switch" at Q_1 and modifies oracles accordingly.
   (a) If Q_1 is on the forward path, to peel the onion on behalf of Q_1, $\mathcal{O}.\mathsf{PO}_1$ forms a new onion using the remainder of the routing path
   (b) If Q_1 is the recipient, to form a reply on behalf of Q_1, $\mathcal{O}.\mathsf{FR}_1$ forms a new onion using the return path as the forward path (and the empty return path)
   (c) If Q_1 is on the return path, to peel the onion of behalf of Q_1, $\mathcal{O}.\mathsf{PO}_1$ forms a new onion using the remainder of the return path as the forward path (and the empty return path)
8: $\mathcal{A}$ gets oracle access to $\mathcal{O}.\mathsf{PO}_1$, $\mathcal{O}.\mathsf{FR}_1$, $\mathcal{O}.\mathsf{PO}_2$ and $\mathcal{O}.\mathsf{FR}_2$
9: $\mathcal{A}$ guesses b' and wins if $b' = b$

Fig. 6: Summary of $\mathsf{ROSecurityGame}$

$\mathsf{ROSecurityGame}(1^\lambda, \Sigma, \mathsf{CompleteOnion}, \mathcal{A})$ 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 $\mathsf{CompleteOnion}$ and the adversary $\mathcal{A}$.

1. The adversary $\mathcal{A}$ picks two router names $Q_1, Q_2 \in \mathcal{P}$ and sends them to the challenger $\mathcal{C}$.
2. The challenger $\mathcal{C}$ generates key pairs $(\mathsf{pk}(Q_1), \mathsf{sk}(Q_1))$ and $(\mathsf{pk}(Q_2), \mathsf{sk}(Q_2))$ for Q_1 and Q_2 using the key generation algorithm G and sends the public keys $(\mathsf{pk}(Q_1), \mathsf{pk}(Q_2))$ to $\mathcal{A}$.
3. $\mathcal{A}$ is given oracle access to (i) $\mathcal{O}.\mathsf{PO}_1(\cdot)$, (ii) $\mathcal{O}.\mathsf{FR}_1(\cdot,\cdot)$, (iii) $\mathcal{O}.\mathsf{PO}_2(\cdot)$ and (iv) $\mathcal{O}.\mathsf{FR}_2(\cdot,\cdot)$ where
   i-ii. $\mathcal{O}.\mathsf{PO}_1(\cdot)$ and $\mathcal{O}.\mathsf{FR}_1(\cdot,\cdot)$ are, respectively, the oracle for answering "process onion" requests made to honest party Q_1 and the oracle for answering "form reply" requests made to Q_1.

iii-iv. $\mathcal{O}.\mathsf{PO}_2(\cdot)$ and $\mathcal{O}.\mathsf{FR}_2(\cdot,\cdot)$ are, respectively, the oracle for answering "process onion" requests made to honest party Q_2 and the oracle for answering "form reply" requests made to Q_2, i.e.,

$$\begin{aligned}\mathcal{O}.\mathsf{PO}_1(O) &= \mathsf{ProcOnion}(O, Q_1, \mathsf{sk}(Q_1))\\ \mathcal{O}.\mathsf{FR}_1(m', O) &= \mathsf{FormReply}(m', O, Q_1, \mathsf{sk}(Q_1))\\ \mathcal{O}.\mathsf{PO}_2(O) &= \mathsf{ProcOnion}(O, Q_2, \mathsf{sk}(Q_2))\\ \mathcal{O}.\mathsf{FR}_2(m', O) &= \mathsf{FormReply}(m', O, Q_2, \mathsf{sk}(Q_2))\end{aligned}$$

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 a forward path $P^{\rightarrow} = (P_1, \ldots, P_d)$ and a return path $P^{\leftarrow} = (P_{d+1}, \ldots, P_s)$ such that exactly one party P_j in the routing path $(P^{\rightarrow}, P^{\leftarrow})$ is equal to Q_1, and only the last party P_s is equal to Q_2. $\mathcal{A}$ 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.,

   $$((O_1^0, \ldots, O_d^0), H^{\leftarrow}, \kappa) \leftarrow \mathsf{FormOnion}(\ell, m, P^{\rightarrow}, \mathsf{pk}(P^{\rightarrow}), P^{\leftarrow}, \mathsf{pk}(P^{\leftarrow})).$$

   The oracles—$\mathcal{O}.\mathsf{PO}_1(\cdot)$, $\mathcal{O}.\mathsf{FR}_1(\cdot,\cdot)$, $\mathcal{O}.\mathsf{PO}_2(\cdot)$ and $\mathcal{O}.\mathsf{FR}_2(\cdot,\cdot)$—remain unmodified.
   Otherwise, if $b = 1$,
   (a) If $j < d$, $\mathcal{C}$ performs the "switch" at honest party $P_j = Q_1$ on the forward path $P^{\rightarrow}$. $\mathcal{C}$ 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 $p^{\rightarrow} = (P_1, \ldots, P_j)$ and the empty return path "()", i.e.,

   $$((O_1^1, \ldots, O_j^1), (), \kappa) \leftarrow \mathsf{FormOnion}(x, y, p^{\rightarrow}, \mathsf{pk}(p^{\rightarrow}), (), ()).$$

   $\mathcal{C}$ 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 $q^{\rightarrow} = (P_{j+1}, \ldots, P_d)$ of the forward path and the return path $P^{\leftarrow}$, i.e.,

   $$((O_{j+1}^1, \ldots, O_d^1), H^{\leftarrow}, \kappa') \leftarrow \mathsf{FormOnion}(x', m, q^{\rightarrow}, \mathsf{pk}(q^{\rightarrow}), P^{\leftarrow}, \mathsf{pk}(P^{\leftarrow})),$$

   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 $H^{\leftarrow}$. $\mathcal{O}.\mathsf{PO}_1$ 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.

$\mathcal{O}.\mathsf{PO}_2$ 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.

(b) If $j = d$, $\mathcal{C}$ performs the "switch" at honest recipient P_j. $\mathcal{C}$ 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.,

$$((O_1^1, \ldots, O_j^1), (), \kappa) \leftarrow \mathsf{FormOnion}(x, y, P^{\rightarrow}, \mathsf{pk}(P^{\rightarrow}), (), ()).$$

We modify the oracles as follows. $\mathcal{O}.\mathsf{FR}_1$ does the following to "form a reply" using message m' and onion $O = O_j^1$: $\mathcal{O}.\mathsf{FR}_1$ runs $\mathsf{FormOnion}$ on a random label x', the reply message m', the return path $P^{\leftarrow}$ as the forward path and the empty return path "()", i.e.,

$$((O_{j+1}^{m'}, \ldots, O_s^{m'}), (), \kappa^{m'}) \leftarrow \mathsf{FormOnion}(x', m', P^{\leftarrow}, \mathsf{pk}(P^{\leftarrow}), (), ()),$$

stores the pair $(O_s^{m'}, m')$ (such that the pair is accessible by $\mathcal{O}.\mathsf{PO}_2$) and returns $(O_{j+1}^{m'}, P_{j+1})$.

$\mathcal{O}.\mathsf{PO}_2$ 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.

(c) If $j > d$, $\mathcal{C}$ performs the "switch" at honest party P_j on the return path $P^{\leftarrow}$. $\mathcal{C}$ 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. $\mathcal{O}.\mathsf{PO}_1$ 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', the reply message m', the remainder the return path $q^{\leftarrow} = (P_{j+1}, \ldots, P_s)$ as the forward path and the empty return path "()", i.e.,

$$((O_{j+1}^{m'}, \ldots, O_s^{m'}), (), \kappa^{m'}) \leftarrow \mathsf{FormOnion}(x', m', q^{\leftarrow}, \mathsf{pk}(q^{\leftarrow}), (), ()),$$

stores the pair $(O_s^{m'}, m')$ (such that the pair is accessible by $\mathcal{O}.\mathsf{PO}_2$) 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.

$\mathcal{O}.\mathsf{PO}_2$ 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.

$\mathcal{C}$ 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 $\mathcal{O}.\mathsf{PO}_1(\cdot)$, $\mathcal{O}.\mathsf{FR}_1(\cdot,\cdot)$, $\mathcal{O}.\mathsf{PO}_2(\cdot)$ and $\mathcal{O}.\mathsf{FR}_2(\cdot,\cdot)$.
7. Finally, $\mathcal{A}$ guesses a bit b' and wins if $b' = b$.

We define *repliable-onion security* as follows.

Definition 4 (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 ν 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.,*

$$\left|\Pr\left[\mathcal{A} \textit{ wins } \mathsf{ROSecurityGame}(1^\lambda, \Sigma, \mathsf{CompleteOnion}, \mathcal{A})\right] - \frac{1}{2}\right| \leq \nu(\lambda).$$

Remarks on Defintion 4. 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 recipient (resp. sender) P.

In the security game, the adversary chooses a routing path with only one honest party (besides the necessarily honest sender). Restricting the adversary's

choice for a routing path in this way simplifies the definition of security without restricting the usefulness of the definition since each segment of an onion formed using a secure scheme (the way we defined it) must be computationally unrelated to any other segment.

5.3 Security results

We first argue that our construction satisfies repliable-onion security.

Lemma 5. *Shallot Encryption Scheme (in Section 4) is correct (Definition 2) and repliable-onion secure (Definition 4) 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 (Sketch). The onion encryption scheme is correct by inspection.

We sketch the proof of security for cases (a) when $P_j = Q_1$ is an intermediary on the forward path and (c) when P_j is an intermediary on the return path. The proof for case (b) (when P_j is the recipient) is similar.

In cases (a) and (c), 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 CCA-secure encryption, so it can be swapped out for the all-zero key "$0 \dots 0$". 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. For case (b), 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.

For the full proof, see Appendix A.2. □

We now show that the history of outputs that $\mathcal{S}$ produces is indistinguishable from one produced by running the real protocol to any environment $\mathcal{Z}$.

Theorem 6. *Shallot Encryption Scheme (in Section 4) SUC-realizes the ideal functionality* $\mathcal{F}_{\mathsf{ROES}}$ *(Definition 3).*

Proof. From Lemma 5, it suffices to prove that if a repliable-onion encryption scheme Σ is repliable-onion secure, then it also SUC-realizes the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$.

Our proof is via a hybrid argument.

Let $\mathsf{Experiment}^1$ be the ideal onion routing protocol in which the ideal honest parties query $\mathcal{F}_{\mathsf{ROES}}$ to form onions, process onions and form return onions.

Let $\mathsf{Experiment}^0$ be the real onion routing protocol in which the honest participants run Σ's algorithms to form onions, process onions and form return onions.

Let Hybrid^0 be the experiment with the same set up at $\mathsf{Experiment}^0$ except that the challenger controls all honest parties and the ideal functionality $\mathcal{F}_{\mathsf{ROES}}$, and the environment is the adversary.

We define the remaining hybrid experiments as follows:

Let numFO be the upper bound on the number of honest "form onion" queries from the environment.

For all $i \in [\mathsf{numFO}]$, let Hybrid^i be the experiment in which the first i "form onion" queries are "simulated", and all remaining queries are "real". The challenger runs $\mathcal{F}_{\mathsf{ROES}}$ for the first i "form onion" queries and runs $\mathsf{FormOnion}$ for the remaining ones. The challenger always runs $\mathcal{F}_{\mathsf{ROES}}$ to process an onion or to form a reply. This works because $\mathcal{F}_{\mathsf{ROES}}$ just runs $\mathsf{ProcOnion}$ or $\mathsf{FormReply}$ in cases where $\mathcal{F}_{\mathsf{ROES}}$ doesn't recognize the query onion's header.

Recall that N is the upper bound on the length of the forward (or return) path which is also an upper bound on the number of segments per "form onion" query.

Next, we describe the hybrids between Hybrid^{i-1} and Hybrid^i. For all $j \in [N]$, let $\mathsf{Hybrid}^{i,j}$ be the experiment in which the first $i-1$ "form onion" queries and the first j segments of the i-th "form onion" query are "simulated", and all remaining segments are "real". (i) The first $i-1$ "form onion" queries are processed via $\mathcal{F}_{\mathsf{ROES}}$. (ii) All queries after the i-th query are processed by running $\mathsf{ProcOnion}$. (iii) The i-th query is processed specially as follows.

Let $(P^{\rightarrow}, P^{\leftarrow})$ denote the routing path of the i-th query. First, the challenger partitions $(P^{\rightarrow}, P^{\leftarrow})$ into (at most) $j+1$ subpaths $(p_1^{\rightarrow}, \ldots, p_j^{\rightarrow}, q^{\rightarrow})$, consisting of the first j segments $(p_1^{\rightarrow}, \ldots, p_j^{\rightarrow})$ of $(P^{\rightarrow}, P^{\leftarrow})$ (or as many as they are) and the remaining subpath $q^{\rightarrow}$ of $(P^{\rightarrow}, P^{\leftarrow})$ not covered by the segments (if it exists).

To process the i-th "form onion" request, the challenger essentially runs the same code as $\mathcal{F}_{\mathsf{ROES}}$ except with the subpaths as the "segments" of $(P^{\rightarrow}, P^{\leftarrow})$. (Onion layers and return paths are stored in the same dictionaries, $\mathsf{OnionDict}$ and $\mathsf{PathDict}$, used by the unmodified $\mathcal{F}_{\mathsf{ROES}}$ code.) As before, the challenger always runs $\mathcal{F}_{\mathsf{ROES}}$ to process an onion or to form a reply.

By construction, (i) $\mathsf{Experiment}^0$ and Hybrid^0 produce identical results, (ii) for all $i \in [\mathsf{numFO}]$, $\mathsf{Hybrid}^{i-1,N}$ and Hybrid^i produce identical results, and (iii) $\mathsf{Hybrid}^{\mathsf{numFO},N}$ and $\mathsf{Experiment}^1$ produce identical results. For any $i \in [\mathsf{numFO}]$ and $j \in [N-1]$, the repliable-onion security of Σ guarantees that the environment cannot distinguish between running $\mathsf{Hybrid}^{i,j}$ and $\mathsf{Hybrid}^{i,j+1}$. Since the total number of segments is polynomially bounded in the security parameter, it follows the environment cannot distinguish between running $\mathsf{Experiment}^0$ and running $\mathsf{Experiment}^1$. In other words, Σ SUC-realizes $\mathcal{F}_{\mathsf{ROES}}$.

□

References

1. 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.
2. Megumi Ando, Anna Lysyanskaya, and Eli Upfal. On the complexity of anonymous communication through public networks. *arXiv preprint arXiv:1902.06306*, 2019.
3. Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.2*, 2015.
4. 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.
5. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
6. 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.
7. 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.
8. David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.
9. Chen Chen, Daniele E Asoni, Adrian Perrig, David Barrera, George Danezis, and Carmela Troncoso. Taranet: Traffic-analysis resistant anonymity at the network layer. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 137–152. IEEE, 2018.
10. Chen Chen, Daniele Enrico Asoni, David Barrera, George Danezis, and Adrian Perrig. HORNET: High-speed onion routing at the network layer. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 1441–1454. ACM Press, October 2015.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.

16. 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.
17. 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.
18. Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography.* Chapman and Hall/CRC, 2014.
19. Christiane Kuhn, Martin Beck, and Thorsten Strufe. Breaking and (partially) fixing provably secure onion routing. *arXiv preprint arXiv:1910.13772*, 2019.
20. Erik Shimshock, Matt Staats, and Nick Hopper. Breaking and provably fixing minx. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 99–114. Springer, 2008.

A Supplementary material

A.1 Pseudocode for ideal functionality $\mathcal{F}_{\mathsf{ROES}}$'s onion forming algorithm

$\mathsf{IdealFormOnion}(\ell, m, (P_1, \ldots, P_d), (P_{d+1}, \ldots, P_s))$

1 : $\mathsf{cur} = 1$
2 : $\mathsf{next} = \mathsf{next}(\mathsf{cur})$
3 : **while** $\mathsf{next} \leq d$
4 : $\quad p^{\rightarrow} = (p_{\mathsf{cur}+1}, \ldots, p_{\mathsf{next}-1}, P_{\mathsf{next}})$
5 : $\quad (o_{\mathsf{cur}+1}, O_{\mathsf{next}}, \kappa_{\mathsf{next}}) \leftarrow \mathsf{SampleOnion}(1^{\lambda}, \mathsf{pp}, p^{\rightarrow}, (), \perp)$
6 : $\quad \mathsf{cur} = \mathsf{next}$
7 : $\quad \mathsf{next} = \mathsf{next}(\mathsf{cur})$
8 : **if** the recipient p_d is corrupt
9 : $\quad p^{\rightarrow} = (p_{\mathsf{cur}+1}, \ldots, p_d)$
10 : $\quad p^{\leftarrow} = (p_{d+1}, \ldots, p_{\mathsf{next}(d)-1}, P_{\mathsf{next}(d)})$
11 : $\quad (o_{\mathsf{cur}+1}, H_{\mathsf{next}(d)}, \kappa_{\mathsf{next}(d)}) \leftarrow \mathsf{SampleOnion}(1^{\lambda}, \mathsf{pp}, p^{\rightarrow}, p^{\leftarrow}, m)$
12 : $\mathsf{cur} = \mathsf{next}(1)$
13 : **while** $\mathsf{cur} < d$
14 : $\quad \mathsf{OnionDict.put}((P_{\mathsf{cur}}, H_{\mathsf{cur}}, C_{\mathsf{cur}}), (\mathsf{I}, ((H_{\mathsf{cur}+1}, C_{\mathsf{cur}+1}), P_{\mathsf{cur}+1}), \kappa_{\mathsf{cur}+1})$
15 : **if** the recipient P_d is honest
16 : $\quad \mathsf{OnionDict.put}((P_d, H_d, C_d), (\mathsf{R}, m), \perp)$
17 : $\quad \mathsf{PathDict.put}((P_d, H_d, C_d), (P^{\leftarrow}, \ell))$
18 : **else if** $\mathsf{next}(d) \leq s$
19 : $\quad ((P_{\mathsf{next}(d)}, H_{\mathsf{next}(d)}, *), ((p_{\mathsf{next}(d)+1}, \ldots, P_s, \ell))$
20 : **return** (H_1, C_1)

Fig. 7: Pseudocode for the ideal functionality's "onion forming" algorithm $\mathsf{IdealFormOnion}$. $\mathsf{IdealFormOnion}$ takes as input label ℓ, message m, forward path $(P_1, \ldots, P_d)$ and return path $(P_{d+1}, \ldots, P_s)$, and outputs an onion (H_1, C_1). When $\mathsf{IdealFormOnion}$ forms onions for the return path, it outputs string "S" in place of "R" (in line 19).

A.2 Proof of Lemma 5

Proof. The onion encryption scheme is correct by inspection.

We present the proof of security for case (a) when the switch occurs at intermediary $P_j = Q_1$ on the forward path. The proofs for cases (b) and (c) are similar.

For the analysis of the scheme's repliable-onion security, we will make the simplifying assumption that labels are truly random as opposed to generated using a PRF. We can make this assumption "without loss in rigor" since a proof that relies on this assumption implies one without making the assumption.

To prove the lemma, we need to prove that $\mathcal{A}$ cannot distinguish between running $\mathsf{Experiment}^0$ (game with $b = 0$) and $\mathsf{Experiment}^1$ (game with $b = 1$). To do this, we define hybrids Hybrid^1 through Hybrid^9 and prove that (i) running Hybrid^1 produces the same result that running $\mathsf{Experiment}^0$ produces, (ii) $\mathcal{A}$ cannot distinguish between running any two consecutive hybrids, and (iii) running Hybrid^9 produces the same result that running $\mathsf{Experiment}^1$ produces. See Figure 8 for the road map of the proof.

$\mathsf{Experiment}^0$—game with $b = 0$, same as Hybrid^1
Hybrid^1—make H_{d+1}, then 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 "$0 \dots 0$"
Hybrid^4—same as Hybrid^3 except swap $(B_{j+1}^{N-j-1}, \dots, B_{j+1}^{N-1})$ for truly random blocks
Hybrid^5—same as Hybrid^4 except swap $(B_j^1, \dots, B_j^{N-j-1})$ and content C_j for truly random strings
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 $(B_j^1, \dots, B_j^{N-j-1}, C_j)$
Hybrid^8—same as Hybrid^7 except swap truly random blocks in H_{j+1} for pseudo-random blocks $(B_{j+1}^{N-j-1}, \dots, B_{j+1}^{N-1})$
Hybrid^9—same as Hybrid^8 except swap key "$0 \dots 0$" for for real key k_j
$\mathsf{Experiment}^1$—game with $b = 1$, same as Hybrid^9

Fig. 8: Road map of proof of Lemma 5

Security game with $b = 1$. Let $\mathsf{Experiment}^1$ be the challenger's algorithm in the security game when $b = 1$. In $\mathsf{Experiment}^1$, the challenger does the following:

1: get from $\mathcal{A}$ router names Q_1 and Q_2 and sends (Q_1, Q_2)
2: generate keys for Q_1 and Q_2 and sends public keys $(\mathsf{pk}(Q_1), \mathsf{pk}(Q_2))$ to $\mathcal{A}$
3: give $\mathcal{A}$ oracle access to $\mathcal{O}.\mathsf{PO}_1$, $\mathcal{O}.\mathsf{FR}_1$, $\mathcal{O}.\mathsf{PO}_2$ and $\mathcal{O}.\mathsf{FR}_2$
4: get from $\mathcal{A}$ parameters for challenge onion: label ℓ, message m, forward path $P^{\rightarrow} = (P_1, \dots, P_d)$ and return path $P^{\leftarrow} = (P_{j+1}, \dots, P_s)$ such that $P_j = Q_1$ and $P_s = Q_2$, and the public keys of the adversarial parties in the routing path
5: pick a random message $y \leftarrow_{\$} \mathcal{M}(1^\lambda)$;
let $p^{\rightarrow} = (P_1, \dots, P_j)$ and $q^{\rightarrow} = (P_{j+1}, \dots, P_d)$;
run $\mathsf{FormOnion}((0 \dots 0), y, p^{\leftarrow}, \mathsf{pk}(p^{\leftarrow}), (), ()) \rightarrow ((O_1, \dots, O_j), (), \kappa)$;
run $\mathsf{FormOnion}((0 \dots 0), m, q^{\leftarrow}, \mathsf{pk}(q^{\leftarrow}), P^{\leftarrow}, \mathsf{pk}(P^{\leftarrow})) \rightarrow ((O_{j+1}, \dots, O_d), H^{\leftarrow}, \kappa')$

6: modify $\mathcal{O}.\mathsf{PO}_1$: to "process" an onion $O = (H_j, C)$ with the same header H_j as O_j, $\mathcal{O}.\mathsf{PO}_1$ returns $(\mathsf{I}, (\mathsf{CompleteOnion}(1^\lambda, \mathsf{pp}, H_{j+1}, C), P_{j+1})$;
modify $\mathcal{O}.\mathsf{PO}_2$: to "process" an onion O with the header H_s (i.e., last header in $H^{\leftarrow}$), $\mathcal{O}.\mathsf{PO}_2$ extracts the message $m' = \mathsf{RecoverReply}(\lambda, \mathsf{pp}, O, P_j, \mathsf{sk}(P_j))$ and returns $(\mathsf{S}, (\ell, m'))$
7: send to $\mathcal{A}$ the first onion O_1
8: give $\mathcal{A}$ oracle access to $\mathcal{O}.\mathsf{PO}_1$, $\mathcal{O}.\mathsf{FR}_1$, $\mathcal{O}.\mathsf{PO}_2$ and $\mathcal{O}.\mathsf{FR}_2$

Security game with $b = 0$. Let $\mathsf{Experiment}^0$ be the challenger's algorithm in the security game when $b = 0$. In $\mathsf{Experiment}^0$, the challenger does the following:

1: get from $\mathcal{A}$ router names Q_1 and Q_2 and sends (Q_1, Q_2)
2: generate keys for Q_1 and Q_2 and sends public keys $(\mathsf{pk}(Q_1), \mathsf{pk}(Q_2))$ to $\mathcal{A}$
3: give $\mathcal{A}$ oracle access to $\mathcal{O}.\mathsf{PO}_1$, $\mathcal{O}.\mathsf{FR}_1$, $\mathcal{O}.\mathsf{PO}_2$ and $\mathcal{O}.\mathsf{FR}_2$
4: get from $\mathcal{A}$ parameters for challenge onion: label ℓ, message m, forward path $P^{\rightarrow} = (P_1, \ldots, P_d)$ and return path $P^{\leftarrow} = (P_{j+1}, \ldots, P_s)$ such that $P_j = Q_1$ and $P_s = Q_2$, and the public keys of the adversarial parties in the routing path
5: run $\mathsf{FormOnion}(\ell, m, P^{\rightarrow}, \mathsf{pk}(P^{\rightarrow}), P^{\leftarrow}, \mathsf{pk}(P^{\leftarrow})) \rightarrow ((O_1, \ldots, O_d), H^{\leftarrow}, \kappa)$
6: keep the oracles unmodified
7: send to $\mathcal{A}$ the first onion O_1
8: give $\mathcal{A}$ oracle access to $\mathcal{O}.\mathsf{PO}_1$, $\mathcal{O}.\mathsf{FR}_1$, $\mathcal{O}.\mathsf{PO}_2$ and $\mathcal{O}.\mathsf{FR}_2$

Hybrid^1—*make H_{d+1}, then O_{j+1}, then O_1.* Let Hybrid^1 be the same procedure as $\mathsf{Experiment}^0$ except for step 5.

In step 5, rather than using $\mathsf{FormOnion}$ as a black box to obtain O_1, the challenger forms onion O_{j+1} by running $\mathsf{FormHeader}(\rightarrow, \ell, (P_{j+1}, \ldots, P_d))$ (to get the header H_{j+1}) and $\mathsf{FormContent}(\ell, m, P^{\leftarrow}, k_{j+1}, \ldots, k_d, K_d)$ (to get the content C_{j+1}); and finally produces onion O_1 by wrapping onion layers around $O_{j+1} = (H_{j+1}, C_{j+1})$:

5: form onion O_{j+1}:

$$(H_{j+1}, \ldots, H_d, \kappa) \leftarrow \mathsf{FormHeader}(\rightarrow, \ell, (P_{j+1}, \ldots, P_d))$$
$$((C_{j+1}, \ldots, C_d), H^{\leftarrow}, \kappa') \leftarrow \mathsf{FormContent}(\ell, m, P^{\leftarrow}, \kappa);$$

form onion O_1 by wrapping layers around $O_{j+1} = ((E_{j+1}, B^1_{j+1}, \ldots, B^{N-1}_{j+1}), C_{j+1})$ using the keys in κ; for all u from j to one, recursively obtain O_u from O_{u+1} as follows:

$$\begin{aligned} & B^1_u = \{P_{u+1}, E_{u+1}\}_{k_u} \\ \forall i \in \{2, \ldots, N-1\},\ & B^i_u = \{B^{i-1}_{u+1}\}_{k_u} \\ & E_u \leftarrow \mathsf{Enc}(\mathsf{pk}(P_u), h(B^1_u, \ldots, B^{N-1}_u), k_u) \\ & C_u = \{C_{u+1}\}_{k_u} \end{aligned}$$

6: keep the oracles unmodified

Hybrid^1 is the same procedure as $\mathsf{Experiment}^0$ "under the hood".

Hybrid^2—*swap ℓ for random label.* Let Hybrid^2 be the same procedure as Hybrid^1 except in steps 5-6, the challenger swaps out the label ℓ (from $\mathcal{A}$) for a random label x and modifies oracle $\mathcal{O}.\mathsf{PO}_2$ accordingly:

5: form onion O_{j+1}:

$$\begin{aligned} x &\leftarrow \mathcal{L}(1^\lambda) \\ (H_{j+1}, \ldots, H_d, \kappa) &\leftarrow \mathsf{FormHeader}(\rightarrow, x, (P_{j+1}, \ldots, P_d)) \\ ((C_{j+1}, \ldots, C_d), H^{\leftarrow}, \kappa') &\leftarrow \mathsf{FormContent}(x, m, P^{\leftarrow}, \kappa); \end{aligned}$$

form onion O_1 by wrapping layers around O_{j+1} (using keys produced from forming O_{j+1})

6: modify $\mathcal{O}.\mathsf{PO}_2$: to "process" an onion O with the header H_s (i.e., last header in $H^{\leftarrow}$), $\mathcal{O}.\mathsf{PO}_2$ extracts the message $m' = \mathsf{RecoverReply}(\lambda, \mathsf{pp}, O, P_j, \mathsf{sk}(P_j))$ and returns $(\mathsf{S}, (\ell, m'))$

Here, we prove that $\mathcal{A}$ cannot distinguish between running Hybrid^1 and running Hybrid^2. For the sake of reaching a contradiction, suppose that $\mathcal{A}$ can distinguish between running Hybrid^1 (i.e., $b = 0$) and running Hybrid^2 (i.e., $b = 1$), then we can construct a reduction $\mathcal{B}$ that can break the CCA2-security of the underlying encryption scheme as follows:

1: $\mathcal{B}$ receives the router names Q_1, Q_2 from $\mathcal{A}$.

2: $\mathcal{B}$ generates keys $(\mathsf{pk}(Q_1), \mathsf{sk}(Q_1))$ for Q_1 using the key generation algorithm G but gets the public key $\mathsf{pk}(Q_2)$ of Q_2 from its challenger. $\mathcal{B}$ sends the public keys $(\mathsf{pk}(Q_1), \mathsf{pk}(Q_2))$ to $\mathcal{A}$.

3: $\mathcal{B}$ gives oracle access to $\mathcal{A}$; whenever $\mathcal{B}$ needs to process an onion $O = ((E, B^1, \ldots, B^{N-1}), C)$ for Q_2, $\mathcal{B}$ uses the decryption oracle $\mathcal{O}.\mathsf{Dec}$ to decrypt the ciphertext portion E of O. For all other "process onion" requests, $\mathcal{B}$ simply runs $\mathsf{ProcOnion}$.

4: $\mathcal{B}$ gets from $\mathcal{A}$ the challenge onion parameters: label ℓ, message m, forward path $P^{\rightarrow}$ and return path $P^{\leftarrow}$ and the public keys of the adversarial parties in the routing path. $\mathcal{B}$ sends the challenge messages $m^0 = \ell$ and $m^1 \leftarrow_\$ \mathcal{L}(1^\lambda)$ to the challenger, and the challenger responds with the encryption E_s^b of one of the messages.

5: Let $|k|(\lambda)$ be the length of the encryption keys. $\mathcal{B}$ uses E_s^b to form header H'_s:

$$\begin{aligned} k_1, \ldots, k_N &\leftarrow_\$ \{0,1\}^{|k|(\lambda)} \\ B_s^1 &= \{\bot, \bot\}_{k_N} \\ \forall i \{2, \ldots, N-1\},\ B_s^i &= \}\ldots\}0\ldots0\{_{k_i}\ldots\{_{k_{N-1}} \\ H'_s &= (E_s^b, B_s^1, \ldots, B_s^{N-1}) \end{aligned}$$

and forms header H'_{d+1} by "wrapping" H'_s (using the return path $P^{\leftarrow}$ and the keys $k_1, \ldots, k_{N-1}$). $\mathcal{B}$ then forms onion O_{j+1} by running $\mathsf{FormHeader}(\rightarrow, x, (P_{j+1}, \ldots, P_d))$ and $\mathsf{FormContent}(x, m, P^{\leftarrow}, k_{j+1}, \ldots, k_d, K_d)$ but replacing the internally created H_{d+1} with H'_{d+1}. Finally, $\mathcal{B}$ forms onion O_1 by wrapping onion layers around O_{j+1}.

6: $\mathcal{B}$ (possibly) modifies $\mathcal{O}.\mathsf{PO}_2$ so that if running $\mathsf{ProcOnion}$ on an onion with header H_s outputs $(\mathsf{S}, (x, m'))$ for some label x and message m, $\mathcal{O}.\mathsf{PO}_2$ outputs $(\mathsf{S}, (\ell, m'))$ instead (all other "process onion" requests are handled by running $\mathsf{ProcOnion}$).
7: $\mathcal{B}$ sends O_1 to $\mathcal{A}$.
8: $\mathcal{B}$ gives oracle access to $\mathcal{A}$ (again using $\mathcal{O}.\mathsf{Dec}$ to decrypt ciphertexts for Q_2).

Finally, $\mathcal{B}$ guesses the bit b' that $\mathcal{A}$ outputs.

The reduction works since $\mathcal{B}$ wins if $\mathcal{A}$ wins; otherwise, $\mathcal{A}$ would be able to break the collision-resistance of the hash function. Clearly, the reduction runs in polynomial-time.

Hybrid^3—*swap k_j for fake key "$0 \ldots 0$".* Let Hybrid^3 be the same procedure as Hybrid^2 except in step 5, the challenger obtains ciphertext E_j by encrypting the all-zero key "$0 \ldots 0$" instead of key k_j:

5: form onion O_{j+1} (like in Hybrid^2);
form onion O_j by wrapping a layer around O_{j+1} (using key k_j produced from forming O_{j+1} but don't encrypt the key under $\mathsf{sk}(P_j)$):

$$\begin{aligned} & B_j^1 = \{P_{j+1}, E_{j+1}\}_{k_j} \\ \forall i \in \{2, \ldots, N-1\},\ & B_j^i = \{B_{j+1}^{i-1}\}_{k_j} \\ & E_j \leftarrow \mathsf{Enc}(\mathsf{pk}(P_j), h(B_j^1, \ldots, B_j^{N-1}), (0 \ldots 0)) \\ & C_j = \{C_{j+1}\}_{k_j}; \end{aligned}$$

form onion O_1 by wrapping layers around O_j (using keys produced from forming O_{j+1})

Here, we prove that $\mathcal{A}$ cannot distinguish between running Hybrid^2 and running Hybrid^3. For the sake of reaching a contradiction, suppose that $\mathcal{A}$ can distinguish between running Hybrid^2 (i.e., $b = 0$) and running Hybrid^3 (i.e., $b = 1$), then we can construct a reduction $\mathcal{B}$ that can break the CCA2-security of the underlying encryption scheme as follows:

1: $\mathcal{B}$ receives the router names Q_1, Q_2 from $\mathcal{A}$.
2: $\mathcal{B}$ generates keys $(\mathsf{pk}(Q_2), \mathsf{sk}(Q_2))$ for Q_2 using the key generation algorithm G but gets the public key $\mathsf{pk}(Q_1)$ of Q_1 from its challenger. $\mathcal{B}$ sends the public keys $(\mathsf{pk}(Q_1), \mathsf{pk}(Q_2))$ to $\mathcal{A}$.
3: $\mathcal{B}$ gives oracle access to $\mathcal{A}$; whenever $\mathcal{B}$ needs to process an onion $O = ((E, B^1, \ldots, B^{N-1}), C)$ for Q_1, $\mathcal{B}$ uses the decryption oracle $\mathcal{O}.\mathsf{Dec}$ to decrypt the ciphertext portion E of O. For all other "process onion" requests, $\mathcal{B}$ simply runs $\mathsf{ProcOnion}$.
4: $\mathcal{B}$ gets from $\mathcal{A}$ the challenge onion parameters: label ℓ, message m, forward path $P^{\rightarrow}$ and return path $P^{\leftarrow}$ and the public keys of the adversarial parties in the routing path. $\mathcal{B}$ sends the challenge messages $m^0 = k_j$ and $m^1 = (0 \ldots 0)$ to the challenger, and the challenger responds with the encryption E_j^b of one of the messages.

5: $\mathcal{B}$ forms onion O_{j+1} (like in Hybrid^1) and sets onion O_j to be $((E_j^b, B_j^1, \ldots, B_j^{N-1}), C_j)$ where

$$\begin{aligned} B_j^1 &= \{P_{j+1}, E_{j+1}\}_{k_j} \\ \forall i \in \{2, \ldots, N-1\},\ B_j^i &= \{B_{j+1}^{i-1}\}_{k_j} \\ C_j &= \{C_{j+1}\}_{k_j}. \end{aligned}$$

Finally, $\mathcal{B}$ forms onion O_1 by wrapping onion layers around O_j.

6: $\mathcal{B}$ modifies oracle $\mathcal{O}.\mathsf{PO}_2$ so that if running $\mathsf{ProcOnion}$ on an onion with header H_s outputs $(\mathsf{S}, (x, m'))$ for some label x and message m, $\mathcal{O}.\mathsf{PO}_2$ outputs $(\mathsf{S}, (\ell, m'))$ instead (all other "process onion" requests are handled by running $\mathsf{ProcOnion}$).

7: $\mathcal{B}$ sends O_1 to $\mathcal{A}$.

8: $\mathcal{B}$ gives oracle access to $\mathcal{A}$ (again using $\mathcal{O}.\mathsf{Dec}$ to decrypt ciphertexts for Q_1).

Finally, $\mathcal{B}$ guesses the bit b' that $\mathcal{A}$ outputs.

The reduction works since $\mathcal{B}$ wins if $\mathcal{A}$ wins; otherwise, $\mathcal{A}$ would be able to break the collision-resistance of the hash function. Clearly, the reduction runs in polynomial-time.

Hybrid^4—*swap* $(B_{j+1}^{N-j-1}, \ldots, B_{j+1}^{N-1})$ *for truly random blocks.* Let Hybrid^4 be the same procedure as Hybrid^3 except in step 5, blocks $(N-j-1)$ to $(N-1)$ in O_{j+1} are formed using a truly random permutation function F rather than the PRP keyed with k_j:

5: form $\hat{O}_j = (\hat{H}_j, \hat{C}_j)$:

$$\begin{aligned} x &\leftarrow \mathcal{L}(1^\lambda) \\ (\hat{H}_j, \ldots, \hat{H}_d, k_j, \ldots, k_d, K_d) &\leftarrow \mathsf{FormHeader}(\rightarrow, x, (P_j, \ldots, P_d)) \\ ((\hat{C}_j, \ldots, \hat{C}_d), k_j, \ldots, k_d) &\leftarrow \mathsf{FormContent}(x, m, P^{\leftarrow}, k_j, \ldots, k_d, K_d); \end{aligned}$$

form $O_{j+1} = ((E_{j+1}, B_{j+1}^1, \ldots, B_{j+1}^{N-j-2}, R_{j+1}^{N-j-1}, \ldots, R_{j+1}^{N-1}), C_{j+1})$ from $\hat{O}_j = ((\hat{E}_j, \hat{B}_j^1, \ldots, \hat{B}_j^{N-1}), \hat{C}_j)$:

$$\begin{aligned} (P_{j+1}, E_{j+1}) &= \}\hat{B}_j^1\{_{k_j} \\ \forall i \in \{1, \ldots, N-j-2\},\ B_{j+1}^i &= \}\hat{B}_j^{i+1}\{_{k_j} \\ \forall i \in \{N-j-1, \ldots, N-2\},\ R_{j+1}^i &= F(\hat{B}_j^{i+1}) \\ C_{j+1} &= \}\hat{C}_j\{_{k_j}; \end{aligned}$$

form O_1 by wrapping layers around O_{j+1}

Here, we prove that $\mathcal{A}$ cannot distinguish between running Hybrid^3 and running Hybrid^4. For the sake of reaching a contradiction, suppose that $\mathcal{A}$ can distinguish between running Hybrid^3 (i.e., $b = 0$) and running Hybrid^4 (i.e., $b = 1$), then we can construct a reduction $\mathcal{B}$ that can break the underlying PRP-CCA security of the PRP as follows:

1: $\mathcal{B}$ receives the router names Q_1, Q_2 from $\mathcal{A}$.
2: $\mathcal{B}$ generates keys $(\mathsf{pk}(Q_1), \mathsf{sk}(Q_1))$ for Q_1 and keys $(\mathsf{pk}(Q_2), \mathsf{sk}(Q_2))$ for Q_2 using the key generation algorithm G and sends the public keys $(\mathsf{pk}(Q_1, \mathsf{pk}(Q_2))$ to $\mathcal{A}$.
3: $\mathcal{B}$ gives oracle access to $\mathcal{A}$.
4: $\mathcal{B}$ gets from $\mathcal{A}$ the challenge onion parameters: label ℓ, message m, forward path $P^{\rightarrow}$ and return path $P^{\leftarrow}$ and the public keys of the adversarial parties in the routing path.
5: $\mathcal{B}$ forms onion $O_j = ((E_j, B_j^1, \ldots, B_j^{N-1}), C_j)$ by running $\mathsf{FormHeader}(\rightarrow, x, (P_j, \ldots, P_d))$ and $\mathsf{FormContent}(x, m, k_j, \ldots, k_d, K_d)$. $\mathcal{B}$ queries the challenger for the pseudo-random permutations of $(B_j^2, \ldots, B_j^{N-j-1}$, and the challenger responds with $(B_{j+1}^1, \ldots, B_{j+1}^{N-j-2})$. $\mathcal{B}$ sets O'_{j+1} to be $((E_{j+1}, B_{j+1}^1, \ldots, B_{j+1}^{N-j-2}, B_{j+1}^{b,N-j-1}, \ldots, B_{j+1}^{b,N-1}), C_{j+1})$, where $B_{j+1}^i = \}B_j^{i+1}\{_{k_j}$ for all $i \in [N-j-2]$, and $C_{j+1} = \}C_j\{_{k_j}$. $\mathcal{B}$ forms onion O_1 by wrapping onion layers around O_{j+1}.
6: $\mathcal{B}$ modifies oracle $\mathcal{O}.\mathsf{PO}_2$ so that if running $\mathsf{ProcOnion}$ on an onion with header H_s outputs $(\mathsf{S}, (x, m'))$ for some label x and message m, $\mathcal{O}.\mathsf{PO}_2$ outputs $(\mathsf{S}, (\ell, m'))$ instead (all other "process onion" requests are handled by running $\mathsf{ProcOnion}$).
7: $\mathcal{B}$ sends O_1 to $\mathcal{A}$.
8: $\mathcal{B}$ gives oracle access to $\mathcal{A}$.

Finally, $\mathcal{B}$ guesses the bit b' that $\mathcal{A}$ outputs.

The reduction works since the distribution of the input to $\mathcal{A}$ in steps 5-6 is exactly what is expected "in the wild". Clearly, the reduction runs in polynomial-time.

Hybrid^5—*swap* $(B_j^1, \ldots, B_j^{N-j-1})$ *and content* C_j *for truly random strings.* Let Hybrid^5 be the same procedure as Hybrid^4 except in step 5, the first $N-j$ blocks and the content of onion O_j are outputs of a truly random permutation function F rather than outputs of the PRP keyed with k_j:

5: form onion O_{j+1} (with some truly random blocks);
form onion $O_j = ((E_j, R_j^1, \ldots, R_j^{N-j-1}, B_j^{N-j}, \ldots, B_j^{N-1}), R_j)$ by wrapping a layer around O_{j+1} (using key k_j produced from forming O_{j+1}):

$$
\begin{aligned}
& & R_j^1 &= F(P_{j+1}, E_{j+1}) \\
\forall i \in \{2, \ldots, N-j-1\}, & & R_j^i &= F(B_{j+1}^{i-1}) \\
\forall i \in \{N-j, \ldots, N-1\}, & & B_j^i &= \{B_{j+1}^{i-1}\}_{k_j} \\
& & E_j &\leftarrow \mathsf{Enc}(\mathsf{pk}(P_j), h(B_j^1, \ldots, B_j^{N-1}), (0 \ldots 0)) \\
& & R_j &= F(C_{j+1});
\end{aligned}
$$

form onion O_1 by wrapping layers around O_j (using keys produced from forming O_{j+1})

Here, we prove that $\mathcal{A}$ cannot distinguish between running Hybrid^4 and running Hybrid^5. For the sake of reaching a contradiction, suppose that $\mathcal{A}$ can dis-

tinguish between running Hybrid^4 (i.e., $b = 0$) and running Hybrid^5 (i.e., $b = 1$), then we can construct a reduction $\mathcal{B}$ that can break the underlying pseudo-randomness of the PRP as follows:

1: $\mathcal{B}$ receives the router names Q_1, Q_2 from $\mathcal{A}$.
2: $\mathcal{B}$ generates keys $(\mathsf{pk}(Q_1), \mathsf{sk}(Q_1))$ for Q_1 and keys $(\mathsf{pk}(Q_2), \mathsf{sk}(Q_2))$ for Q_2 using the key generation algorithm G and sends the public keys $(\mathsf{pk}(Q_1, \mathsf{pk}(Q_2))$ to $\mathcal{A}$.
3: $\mathcal{B}$ gives oracle access to $\mathcal{A}$.
4: $\mathcal{B}$ gets from $\mathcal{A}$ the challenge onion parameters: label ℓ, message m, forward path $P^{\rightarrow}$ and return path $P^{\leftarrow}$ and the public keys of the adversarial parties in the routing path.
5: $\mathcal{B}$ forms onion O_{j+1} like in Hybrid^4 (with some truly random blocks). $\mathcal{B}$ sends to the challenger the sequence $((P_{j+1}, E_{j+1}), B_{j+1}^1, \ldots, B_{j+1}^{N-j-2}, C_{j+1})$. The challenger responds with $(E_j^b, B_j^{b,1}, \ldots, B_j^{b,N-j-1}, C_j^b)$ which are either pseudo-random permutations (if $b = 0$) or truly random permutations (if $b = 1$). $\mathcal{B}$ sets O_j to be $((E_j, B_j^{b,1}, \ldots, B_j^{b,N-j-1}, B_j^{N-j}, \ldots, B_j^{N-1}), C_j^b)$ where

$$\forall i \in \{N-j, \ldots, N-1\},\ B_j^i = \{B_{j+1}^{i-1}\}_{k_j}$$
$$E_j \leftarrow \mathsf{Enc}(\mathsf{pk}(P_j), t_j, (0 \ldots 0)),$$

and where $t_j = h(B_j^{b,1}, \ldots, B_j^{b,N-j-1}, B_j^{N-j}, \ldots, B_j^{N-1})$. Finally, $\mathcal{B}$ forms onion O_1 by wrapping onion layers around O_j.
6: $\mathcal{B}$ modifies oracle $\mathcal{O}.\mathsf{PO}_2$ so that if running $\mathsf{ProcOnion}$ on an onion with header H_s outputs $(\mathsf{S}, (x, m'))$ for some label x and message m, $\mathcal{O}.\mathsf{PO}_2$ outputs $(\mathsf{S}, (\ell, m'))$ instead (all other "process onion" requests are handled by running $\mathsf{ProcOnion}$).
7: $\mathcal{B}$ sends O_1 to $\mathcal{A}$.
8: $\mathcal{B}$ gives oracle access to $\mathcal{A}$.

Finally, $\mathcal{B}$ guesses the bit b' that $\mathcal{A}$ outputs.

The reduction works since the distribution of the input to $\mathcal{A}$ in steps 5-6 is exactly what is expected "in the wild". Clearly, the reduction runs in polynomial-time.

Hybrid^6—*swap onion for intermediary P_j for onion for recipient P_j.* Let Hybrid^6 be the same procedure as Hybrid^5 except in steps 5-6, the challenger wraps layers around a bogus onion O_j for recipient P_j to obtain the output O_1 and modifies oracle $\mathcal{O}.\mathsf{PO}_1$ accordingly:

5: form onion O_{j+1} (with some truly random blocks);

form bogus onion $O_j = ((E_j, R_j^1, \ldots, R_j^{N-j-1}, B_j^{N-j}, \ldots, B_j^{N-1}), R_j)$ for recipient P_j:

$$\begin{aligned}
\forall i \in [N-j-1],\ R_j^i &\leftarrow_{\$} \{0,1\}^{L_1(\lambda)} \\
k_1, \ldots, k_N &\leftarrow_{\$} \{0,1\}^{|k|(\lambda)} \\
\forall i \in \{N-j, \ldots, N-1\},\ B_j^i &= \}\ldots\}0\ldots0\{_{k_{N-i-1}}\{_{k_{N-j}} \\
E_j &\leftarrow \mathsf{Enc}(\mathsf{pk}(P_j), h(B_j^1, \ldots, B_j^{N-1}), (0\ldots0)) \\
R_j &\leftarrow_{\$} \{0,1\}^{L_1(\lambda)};
\end{aligned}$$

form onion O_1 by wrapping layers around O'_j (using keys $k_1, \ldots, k_{N-1}$)

6: modify $\mathcal{O}.\mathsf{PO}_1$: to "process" an onion $O = (H_j, C)$ with the same header H_j as O_j, $\mathcal{O}.\mathsf{PO}_1$ returns $(\mathsf{I}, (\mathsf{CompleteOnion}(1^\lambda, \mathsf{pp}, H_{j+1}, C), P_{j+1})$;
modify $\mathcal{O}.\mathsf{PO}_2$: to "process" an onion O with the header H_s (i.e., last header in $H^{\leftarrow}$), $\mathcal{O}.\mathsf{PO}_2$ extracts the message $m' = \mathsf{RecoverReply}(\lambda, \mathsf{pp}, O, P_j, \mathsf{sk}(P_j))$ and returns $(\mathsf{S}, (\ell, m'))$

The adversary can query the oracle $\mathcal{O}.\mathsf{PO}_1$ to process an onion with the correct challenge header but with "mangled" content. In this case, the peeled onion in Hybrid^5 looks like the peeled onion in Hybrid^6 because the former has a truly random header and content whereas the latter has a truly random header and pseudo-random content. For all other queries, the responses in the hybrids are statistically the same. Thus, using a straightforward hybrid argument, we can show that the adversary cannot distinguish between running Hybrid^5 and running Hybrid^6.

Hybrid^7—*swap truly random blocks and content in O_j for pseudo-random blocks* $(B_j^1, \ldots, B_j^{N-j-1}, C_j)$. Let Hybrid^7 be the same procedure as Hybrid^6 except in step 5, the challenger wraps layers around the onion O_j with real blocks and real content to obtain the output O_1:

5: form onion O_{j+1} (with some truly random blocks);
form onion O_j for recipient P_j:

$$\begin{aligned}
y &\leftarrow_{\$} \mathcal{M}(\lambda) \\
((((\hat{E}_j, B_j^1, \ldots, B_j^{N-1}), C_j)), (), \kappa) &\leftarrow \mathsf{FormOnion}((0\ldots0), y, (P_j), \mathsf{pk}(P_j), (), ()) \\
E_j &\leftarrow \mathsf{Enc}(\mathsf{pk}(P_j), h(B_j^1, \ldots, B_j^{N-1}), (0\ldots0));
\end{aligned}$$

form onion O_1 by wrapping layers around O_j (using keys from $\mathsf{FormOnion}$)

$\mathcal{A}$ cannot distinguish between running Hybrid^6 and running Hybrid^7. Otherwise, we could construct a reduction (very similar to the reduction used for proving that $\mathsf{Hybrid}^4 \approx \mathsf{Hybrid}^5$) that can break the underlying pseudo-randomness of the PRP.

Hybrid^8—*swap truly random blocks in H_{j+1} for pseudo-random blocks* $(B_{j+1}^{N-j-1}, \ldots, B_{j+1}^{N-1})$. Let Hybrid^8 be the same procedure as Hybrid^7 except in step 5, the challenger wraps layers around a real onion O_j to obtain the output O_1:

5: form onion O_{j+1} (with all pseudo-random blocks);
form onion O_j for recipient P_j;

$$y \leftarrow_{\$} \mathcal{M}(\lambda)$$
$$((((\hat{E}_j, B_j^1, \ldots, B_j^{N-1}), C_j)), (), \kappa) \leftarrow \mathsf{FormOnion}((0 \ldots 0), y, (P_j), \mathsf{pk}(P_j), (), ())$$
$$E_j \leftarrow \mathsf{Enc}(\mathsf{pk}(P_j), h(B_j^1, \ldots, B_j^{N-1}), (0 \ldots 0));$$

form onion O_1 by wrapping layers around O_j

$\mathcal{A}$ cannot distinguish between running Hybrid^7 and running Hybrid^8. Otherwise, we could construct a reduction (very similar to the reduction used for proving that $\mathsf{Hybrid}^3 \approx \mathsf{Hybrid}^4$) that can break the underlying PRP-CCA2 security of the PRP.

Hybrid^9 —*swap key "*$0 \ldots 0$*" for for real key* k_j*.* Let Hybrid^9 be the same procedure as Hybrid^8 except in step 5, the challenger wraps layers around a real onion O_j to obtain the output O_1:

5: form onion O_{j+1};
form onion O_j for recipient P_j;
form onion O_1 by wrapping layers around O_j

$\mathsf{Hybrid}^8 \approx \mathsf{Hybrid}^9$. $\mathcal{A}$ cannot distinguish between running Hybrid^8 and running Hybrid^9. Otherwise, we could construct a reduction (very similar to the reduction used for proving that $\mathsf{Hybrid}^2 \approx \mathsf{Hybrid}^3$) that can break the underlying CCA2-security of the encryption scheme.

Finally, Hybrid^9 and $\mathsf{Experiment}^1$ produce the same result. In both procedures, onion O_1 is formed using the all-zero label "$0 \ldots 0$", a random message y, the truncated path $(P_1, \ldots, P_j)$ as the forward path and the empty return path "()", and the oracle $\mathcal{O}.\mathsf{PO}_1$ ensures that O_j "peels" to the separately formed O_{j+1}.

This concludes our proof for case (a). The proofs for cases (b) and (c) are similar. □