Full Anonymity in the Asynchronous Setting from Peony Onion Encryption

Megumi Ando [*] Miranda Christ [†] Kashvi Gupta [‡] Tal Malkin [§]
Dane Smith [¶]

February 19, 2026

Abstract

Onion routing is a popular and practical approach to anonymous communication, and the subject of a growing body of work on designing efficient schemes with provable anonymity, the strongest notion being full anonymity. However, all prior schemes achieving full anonymity assume a synchronous communication setting, which is unrealistic since real networks suffer from message loss and timing attacks. Recently, Ando, Lysyanskaya, and Upfal (TCC'24) made progress toward the asynchronous setting by introducing bruisable onion encryption and constructing an efficient protocol with the weaker guarantee of differential privacy.

In this paper, we construct the first efficient fully anonymous onion routing protocol in the asynchronous setting. To do so, we address two main challenges: (1) we develop the first bruisable onion encryption scheme that does not leak information about the onion's position on the routing path, and (2) we design an onion routing protocol using this primitive to achieve full anonymity, rather than only differential privacy. Along the way, we also give a new fully anonymous protocol in the synchronous setting, improving the state-of-the-art in both communication and round complexity.

Both protocols are secure against an active adversary that corrupts a constant fraction of nodes (up to < 1 in the synchronous case, and $< 1/2$ in the asynchronous case), and rely on standard cryptographic assumptions (CCA-secure public-key encryption and collision-resistant hash functions).

[*]Computer Science Department, Tufts University, mando@cs.tufts.edu
[†]Computer Science Department, Columbia University, mchrist@cs.columbia.edu
[‡]Computer Science Department, Tufts University, kashvi.gupta@tufts.edu
[§]Computer Science Department, Columbia University, tal@cs.columbia.edu
[¶]The MITRE Corporation, danesmith@mitre.org

Contents

1 Introduction **1**
- 1.1 Prior Work . . . 1
- 1.2 Our Contributions . . . 2
- 1.3 Technical Challenges . . . 3

2 Problem Setting **3**

3 Bruisable Onion Encryption: Definitions **4**
- 3.1 Correctness . . . 5
- 3.2 Security . . . 5

4 Our Onion Encryption Scheme: Peony **8**
- 4.1 A Description of the Sepal Matrix . . . 10
- 4.2 Complete Description of Peonies . . . 12
- 4.3 Proof of Security . . . 15

5 Our Fully Anonymous Onion Routing Protocols **17**
- 5.1 Tools from Prior Work: Checkpoint Onions and Merging Onions . . . 18
- 5.2 Anonymity in Synchronous Setting: $\Pi_{\bullet}$ . . . 19
  - 5.2.1 Forming Onions. . . . 19
  - 5.2.2 Routing Onions. . . . 20
  - 5.2.3 Analysis of $\Pi_{\bullet}$. . . . 22
- 5.3 Anonymity in Asynchronous Setting: $\Pi_{\check{o}}$ . . . 23
  - 5.3.1 Changes from $\Pi_{\bullet}$. . . . 23
  - 5.3.2 Forming Onions. . . . 23
  - 5.3.3 Routing Onions. . . . 24
  - 5.3.4 Analysis of $\Pi_{\check{o}}$. . . . 24

6 Conclusion and Future Work **25**

A Ideal Functionality for Onion Encryption Schemes **28**
- A.1 Ideal Functionality and $\mathcal{F}_{OES}$ . . . 28
  - A.1.1 Setting up. . . . 28
  - A.1.2 Forming an onion. . . . 29
  - A.1.3 Processing an onion. . . . 30
- A.2 Game-based Security Definition for OES . . . 30
- A.3 Onion Security Implies SUC-realizability of $\mathcal{F}_{OES}$ . . . 31
  - A.3.1 Description of the simulator $\mathcal{S}$. . . . 31
  - A.3.2 Proof of Theorem 4. . . . 32
- A.4 Extending to POES . . . 36

B Peonies: Proof of Security **36**

C Peonies Example Diagrams **39**

D $\Pi_\bullet$: Proof of Anonymity **42**
D.1 Sampling with Replacement 44
D.2 Sampling without Replacement 45

1 Introduction

Privacy is recognized by the U.N. as a fundamental human right.[1] Yet, even when message content is encrypted, communication patterns often reveal sensitive information. Designing *anonymous peer-to-peer (P2P) channels*, systems that allow parties to communicate while concealing their identities, is therefore a critical goal. Such channels are key components in many privacy-preserving applications, from anonymous messaging to secure voting systems.

A widely adopted approach to achieve anonymity is *onion routing*. Introduced by Chaum [Cha81], the idea is simple: if Alice wants to send a message to Bob anonymously, she selects a random path of intermediate parties $(P_0, \ldots, P_{\ell-2}, P_{\ell-1} = \mathsf{Bob})$ and constructs a layered encryption structure called an *onion*. Each party P_i along the path removes one encryption layer, revealing the next hop, until Bob receives the innermost layer and can read the message.

This concept underlies practical systems such as Tor [DMS04], operated by thousands of volunteer nodes and serving millions of users daily [LMD10].[2] However, Tor and other widely used tools (such as Freenet [CSWH01] and Crowds [RR98]) remain vulnerable to *traffic analysis attacks* [MD05, Rop21, SEV+15].

1.1 Prior Work

Synchronous setting. Most provably anonymous protocols assume a synchronous network, where time proceeds in rounds and messages are delivered instantly. In this model, simple protocols like Π_p [ALU18] achieve full anonymity against *passive adversaries* who can observe network traffic at all links and corrupted nodes, while $\Pi_\triangle$ and $\Pi_\bowtie$ [ALU21] achieve full anonymity against *active adversaries* who can additionally control corrupted parties. They accomplish this through mechanisms such as batch processing, shuffling, and *merging onions*, which stabilize message counts despite adversarial interference [ALU18, ALU21].

Other schemes, Vuvuzela [vdHLZZ15], Stadium [TGL+17], Atom [KCGDF17], and XRD [KLD20], also achieve provable anonymity but lack fault tolerance: a single dropped message may prevent delivery entirely. Moreover, all synchronous protocols are vulnerable to timing attacks [SEV+15], highlighting the need to consider the more realistic asynchronous setting. Table 1 compares key properties of existing provably secure, fault-tolerant routing protocols.

Protocol	Setting	Encryption	Server load	# of Rounds	Security
Π_p [ALU18]	sync	standard	$\mathsf{polylog}(\lambda)$	$\mathsf{polylog}(\lambda)$	full anonymity$^-$
Π_a [ALU18]	sync	standard	$\mathsf{polylog}(\lambda)$	$\mathsf{polylog}(\lambda)$	differential privacy
$\Pi_\triangle$ [ALU21]	sync	standard	$\Omega(\sqrt{N})$	$\mathsf{polylog}(\lambda)$	full anonymity
$\Pi_\bowtie$ [ALU21]	sync	standard	$\mathsf{polylog}(\lambda)$	$\omega(\mathsf{polylog}(\lambda))$	full anonymity
$\Pi_\bullet$ (this work)	sync	standard	$\mathsf{polylog}(\lambda)$	$\mathsf{polylog}(\lambda)$	full anonymity
Π_t [ALU24]	async	bruisable	$\mathsf{polylog}(\lambda)$	$\mathsf{polylog}(\lambda)$	differential privacy
$\Pi_{\tilde{o}}$ (this work)	async	bruisable$^+$	$\mathsf{polylog}(\lambda)$	$\mathsf{polylog}(\lambda)$	full anonymity

Table 1: A table comparing properties of provably secure and fault-tolerant onion routing protocols. By server load, we mean the number of onions per intermediary per round. λ denotes the security parameter, and N, the number of participants. Full anonymity$^-$ denotes security against the passive adversary, and bruisable$^+$ denotes our enhanced bruisable encryption.

[1] https://www.un.org/sites/un2.un.org/files/2021/03/udhr.pdf

[2] Data retrieved from https://metrics.torproject.org/ on February 19, 2026.

Asynchronous setting. The first step toward asynchronous anonymous communication was made by Ando, Lysyanskaya, and Upfal [ALU24], who introduced *bruisable onion encryption* and constructed a protocol Π_t that achieves differential privacy. In this scheme, each onion carries a "bruise count" which is incremented when a message arrives late. If the count exceeds a threshold, the onion loses access to the routing information needed to reach its recipient.

This motivates our central question:

> *Can we construct a practical onion routing protocol that achieves full anonymity in the asynchronous setting?*

1.2 Our Contributions

We answer this affirmatively. Our contributions are:

1. **Bruisable Onion Encryption Definitions.** We formalize bruisable onion encryption that hides the distance to the recipient, strengthening the security guarantees of prior work [ALU24].
2. **Peony Onion Encryption.** We construct the first bruisable onion that hides the distance between the onion's current position and the recipient. This scheme relies only on standard cryptographic assumptions: CCA-secure public-key encryption, pseudorandom permutation, and collision-resistant hash functions.
3. **Pi-SEED.** We introduce the Synchronous Environment with Efficient Diagnostics (Pi-SEED, $\Pi_\bullet$), which achieves full anonymity in the synchronous model against active adversaries controlling any constant fraction < 1 of the parties. $\Pi_\bullet$ improves asymptotically on server load and round complexity compared to previous protocols (see Table 1).
4. **Pi-APPLE.** We present the Asynchronous Protocol with Poly-Logarithmic Epochs (Pi-APPLE, $\Pi_{\tilde{o}}$), the first fully anonymous onion routing protocol in the asynchronous setting. $\Pi_{\tilde{o}}$ uses the stronger bruisable onion primitive and achieves polylogarithmic server load and round complexity. It is secure against active adversaries controlling a constant fraction $< \frac{1}{2}$ of the parties and is near-optimal in round complexity, matching a known lower bound [Chr20].

Comparison of our security results against prior work. Technically, $\Pi_\bullet$ and $\Pi_{\tilde{o}}$ achieve full anonymity only under a restriction on parties' inputs and outputs, the so-called *simple I/O setting*, in which each party sends exactly one message and receives exactly one message. This restriction also appears in previously known fully anonymous (synchronous) P2P protocols, including $\Pi_{\triangle}$ and $\Pi_{\bowtie}$.

The simple I/O setting is standard and partly unavoidable. Every participant must send and receive something; otherwise, non-conforming parties are immediately identifiable. Without placing some limit on the number of messages a party can receive, indistinguishability between scenarios such as "everyone sends to Bob" and "everyone sends to different recipients" would require prohibitive overhead. Protocols therefore typically assume that everyone sends and receives exactly one message. Atom avoids this restriction because it is an anonymous posting protocol in which all parties send messages to a bulletin board.

Parties running Vuvuzela and Stadium likewise send and receive a single message, though this constraint arises from protocol design rather than assumption.

Vuvuzela, Stadium, Π_a, and Π_t provide only differential privacy. Their security guarantees require that an adversary cannot distinguish the real execution from a *neighboring* execution that differs only in the *honest* parties' inputs and outputs. Consequently, while these protocols do not impose the simple I/O restriction, they also do not provide sender anonymity (i.e., anonymity of the sender from a corrupted recipient).

1.3 Technical Challenges

Below, we outline the technical challenges we tackled.

Detecting targeted attacks. Simply instantiating a previously known synchronous protocol with bruisable onions does not yield anonymity in the asynchronous setting. Most of these protocols either lack fault tolerance [vdHLZZ15, TGL+17, KCGDF17, KLD20] or are too inefficient ($\Pi_{\triangle}$ in [ALU21]). $\Pi_{\bowtie}$ [ALU21] is the best candidate, but with bruisable onions, it leaks information: an adversary can delay Alice's onions so that they are all dropped upfront, which reveals Alice's recipient.

Our challenge was to detect such targeted delays efficiently. We solve this using *looping onions* in the synchronous case (which return to the sender) and new *signaling onions* in the asynchronous case (a variant of checkpoint onions). These mechanisms ensure that if Alice's onions are delayed or dropped, the attack is detected by either Alice or the network.

Position-hiding bruisable onion encryption. Bruisable onions allow state changes along the path (e.g., incrementing a bruise counter), but this flexibility risks adversarial tampering or tracking. To prevent this, we restrict onions to a small set of "valid forms," enforcing integrity while keeping onion size compact.

Prior constructions [ALU24] reduced onion size linearly with depth, but at the cost of revealing distance to the recipient. Our goal is to allow controlled state changes (e.g., incrementing bruises) without allowing forbidden ones (e.g., decrementing), while hiding the onion's distance from its destination. Obtaining this compact, position-hiding design is a key technical challenge.

2 Problem Setting

We consider the scenario with n participants, labeled $P_0, P_1, \ldots, P_{n-1}$. The communication between them is peer-to-peer (P2P). That is, any party P_i can send a message directly to any other party P_j.

Let $(\mathsf{KeyGen}, \mathsf{FormOnion}, \mathsf{PeelOnion})$ be a secure onion encryption scheme where KeyGen is the algorithm for generating keys, $\mathsf{FormOnion}$ is the algorithm for forming onions, and $\mathsf{PeelOnion}$ is the algorithm for processing onions.

Each honest party P_i generates their key pair, denoted $(\mathsf{pk}(P_i), \mathsf{sk}(P_i))$, by running the key generation algorithm KeyGen, i.e., $(\mathsf{pk}(P_i), \mathsf{sk}(P_i)) \leftarrow \mathsf{KeyGen}(1^\lambda)$. We assume the existence of a Public Key Infrastructure (PKI) that recognizes the participants involved; specifically, the public key $\mathsf{pk}(P_i)$ of each honest party P_i is made public through this PKI.

Inputs. Each party P_i receives an input, denoted as σ_i, representing a set of message-recipient pairs. We can think of each message-recipient pair (m_i, R_i) as instructions for P_i to send the message m_i to the recipient R_i.

Like in prior works, e.g., [vdHLZZ15,KCDF17,ALU21], we operate under the *simple I/O setting*: Each party P_i sends exactly one message to a recipient R_i and is slated to receive exactly one message from another sender P_j. This implies there exists a permutation function $\pi : [n] \mapsto [n]$ such that $R_i = P_{\pi(i)} = P_j$. The input vector $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$ comprises the inputs from all parties.

Let Bad be the set of corrupted parties. Two input vectors $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$ and $\sigma' = (\sigma'_0, \sigma'_1, \ldots, \sigma'_{n-1})$ are considered equivalent with respect to Bad, denoted $\sigma \equiv_{\mathsf{Bad}} \sigma'$, if every corrupted party holds the same input in both σ and σ' and is set to receive the same set of messages from honest parties in both input vectors.

Adversary model and anonymity definition. We assume that the adversary can observe all traffic on all communication links. Additionally, they can corrupt up to a constant fraction of the participants. These corruptions are *non-adaptive*, meaning the choices for the corrupted parties are made independently of the protocol run. Thus, the adversarial view consists of traffic on all links and the states of computations of all corrupted parties throughout the protocol run. Given a protocol Π, an adversary $\mathcal{A}$, and an input vector σ, we denote the adversary's view in a run of Π on input σ as $\mathsf{View}^{\Pi,\mathcal{A}}(\sigma)$.

A communication protocol is considered anonymous if what the adversary sees is essentially the same, regardless of the input vector (which indicates who is communicating with whom). More formally,

Definition 1 (Anonymity [ALU21])**.** *A communications protocol* Π *is computationally (resp. statistically) anonymous if for any adversary* $\mathcal{A}$*; any possible set of corruptions* Bad*; and any pair of input vectors* σ, σ' *that are equivalent w.r.t.* Bad*,*[3] *the adversary's view when the input is* σ *is computationally close (resp. statistically close) to the view when the input vector is* σ' *instead, i.e.,*

$$\mathsf{View}^{\Pi,\mathcal{A}}(\sigma) \approx \mathsf{View}^{\Pi,\mathcal{A}}(\sigma').$$

3 Bruisable Onion Encryption: Definitions

In this section, we provide our definitions for bruisable onion encryption. We improve the definitions presented in the initial bruisable onion paper [ALU24] to be more in line with those of standard onion encryption [CL05].

A bruisable onion encryption scheme consists of algorithms KeyGen, $\mathsf{FormOnion}$, $\mathsf{PeelOnion}$, and $\mathsf{BruiseOnion}$. We describe these below.

KeyGen: takes the security parameter, 1^λ, and outputs the public key pair $(\mathsf{pk}(P), \mathsf{sk}(P))$, i.e., $(\mathsf{pk}(P), \mathsf{sk}(P)) \leftarrow \mathsf{KeyGen}(1^\lambda)$.

FormOnion: takes a fixed-length message m; a routing path consisting of l_1 "mixers" and l_2 "gatekeepers," $\vec{P} = (M_0, ..., M_{l_1-1}, G_0, ..., G_{l_2-1}, R)$; the public keys of parties in $\vec{P}$, $\mathsf{pk}(\vec{P})$; a sequence of metadata, $\vec{y} = (y_0, ..., y_{l-1})$, where $l = l_1 + l_2$; an upper bound N on the number of mixers; and an upper bound N_t on the total number of intermediaries (mixers and gatekeepers).

Note: The routing path $\vec{P}$ is composed of two kinds of intermediaries: mixers (who can bruise) and gatekeepers (who cannot). The gatekeepers function to prevent a final adversarial mixer from manipulating the bruise count to try to determine the recipient.

$$\vec{O} = (\vec{O}_0, ..., \vec{O}_l) \leftarrow \mathsf{FormOnion}(m, \vec{P}, \mathsf{pk}(\vec{P}), \vec{y}, N, N_t)$$

The output $\vec{O}$ is a vector of vectors of onions where each sub-vector, $\vec{O}_i$, lists all the valid ways that an onion can be received by the ith intermediary on the path (assuming it is always processed correctly).

For example, if $\vec{P} = (M_0, M_1, G_0, G_1, R)$ where the M_i's are mixers, the G_i's are gatekeepers, and the recipient is R, then $\vec{O} = (\vec{O}_0, \vec{O}_1, \vec{O}_2, \vec{O}_3, \vec{O}_4) = ((O_{0,0}), (O_{1,0}, O_{1,1}), (O_{2,0}, O_{2,1}, O_{2,2}), (O_{3,0}, O_{3,1}, O_{3,2}), (O_{4,0}, O_{4,1}, O_{4,2}))$. $O_{0,0}$ is the onion for mixer M_0 (there's only one valid form). M_0 will process and send the resulting onion to M_1. If

[3] The adversary must choose σ and σ' so that they are equivalent w.r.t. Bad since the adversary can always observe the corrupted parties' inputs and the sets of messages they receive.

it peels the onion without bruising it, M_1 will receive the onion $O_{1,0}$; whereas if M_0 bruised the onion, M_1 will receive the onion $O_{1,1}$, and so on.

Note: As (N, N_t) are typically system-wide parameters, we usually omit to give them as input to this algorithm.

PeelOnion: takes a secret key $\mathsf{sk}(P)$ and an onion O. It returns the sequence (y, O', P'), where y is the metadata for the processing party, O' is the peeled onion or message, and P' is the next location for the peeled onion, i.e., $(y, O', P') \leftarrow \mathsf{PeelOnion}(\mathsf{sk}(P), O)$.

BruiseOnion: takes a secret key and an onion O, and returns the bruised onion O' to send to the next destination, i.e., $O' \leftarrow \mathsf{BruiseOnion}(\mathsf{sk}(P), O)$.

Note: Since onions are indistinguishable, onions necessarily need to be peeled before a party can determine whether or not to bruise the onion going forward.

3.1 Correctness

A bruisable onion encryption scheme $(\mathsf{KeyGen}, \mathsf{FormOnion}, \mathsf{PeelOnion}, \mathsf{BruiseOnion})$ is correct if all onions resulting from a call to $\mathsf{FormOnion}$ with

- a message from the message space, m;
- a routing path (consisting of an appropriate number of mixers and an appropriate number of gatekeepers), $\vec{P} = (M_0, \ldots, M_{l_1-1}, G_0, \ldots, G_{l_2-1}, R)$;
- the public keys (generated correctly by running KeyGen) of the parties on the routing path;
- and appropriately-sized metadata, $\vec{y} = (y_0, \ldots, y_{l-1})$;

process correctly. E.g., let $\vec{O} = ((O_{0,0}), (O_{1,0}, O_{1,1}), \ldots, (O_{l,0}, \ldots, O_{l,l_1-1}))$ be the output of $\mathsf{FormOnion}$.

For $0 \leq i < l$, peeling the onion $O_{i,j}$ without bruising it (i.e., running $\mathsf{PeelOnion}(\mathsf{sk}(P_i), O_{i,j})$) produces the triple $(y_i, O_{i+1,j}, P_{i+1})$. (In the special case, $i = l$, running $\mathsf{PeelOnion}$ produces the message m.) Additionally, for $0 \leq i < l_1$, bruising the onion (running $\mathsf{BruiseOnion}(\mathsf{sk}(P_i), O_{i,j})$) produces the onion $O_{i+1,j+1}$ instead of $O_{i+1,j}$.

3.2 Security

A bruisable onion encryption scheme is secure if an adversary can learn no information about an honestly generated onion after it has been processed by an honest party. Our security game captures this intuition by having the adversary specify the onion parameters to the challenger. The challenger either forms the onion correctly, or forms the onion with modified parameters. The onion encryption scheme is secure if the adversary cannot guess which scenario it is in with more than negligible advantage.

Our definition modifies the original notion of bruisable onion security [ALU24] in two ways. First, we simplify the security game by requiring the adversary to pick only an honest mixer and an honest gatekeeper, without also picking an honest recipient. (An honest party, either a mixer or gatekeeper, is the recipient when the routing path terminates at the party.) Despite this simplification, the revised game still implies what we want: namely, the realizability of an ideal onion functionality (see Appendix A), so no generality is lost. Second, the new definition is *stronger*: in our formulation, the position of a processing party P_j on the routing path must remain hidden, even from P_j itself.

Summary of the security game. Our definition follows the standard notion of CCA2 security. In the challenge phase, the adversary specifies a message m, a path $\vec{P}$ containing one honest mixer H_m and one honest gatekeeper H_g, and associated metadata $\vec{y}$. The challenger samples a bit $b \leftarrow\$ \{0,1\}$.

If $b = 0$, the challenger returns $\mathsf{FormOnion}(m, \vec{P}, \vec{y})$. The adversary may peel layers corresponding to corrupted parties, but not those of honest parties. Upon reaching an honest party's onion layer, the adversary must forward the onion to the challenger for processing during the second query phase. The challenger then returns the output of $\mathsf{PeelOnion}$ (or $\mathsf{BruiseOnion}$).

If $b = 1$, the challenger instead returns $\mathsf{FormOnion}(m, \vec{Q}_{\to h_0}, \vec{z}_{\to h_0})$, where $\vec{Q}_{\to h_0}$ is the prefix of $\vec{P}$ ending at the honest mixer H_m, and $\vec{z}_{\to h_0}$ is the corresponding metadata. When the adversary reaches H_m's layer, it forwards the onion to the challenger in the second query phase, who responds with a fresh onion computed as $\mathsf{FormOnion}(\perp, \vec{Q}_{h_0+1\to h_1}, \vec{z}_{h_0+1\to h_1})$, where $\vec{Q}_{h_0+1\to h_1}$ is the subsequent segment of $\vec{P}$ beginning after H_m and ending at the honest gatekeeper H_g. This onion contains no message, no accumulated bruises, and only the truncated path and metadata. Finally, when the adversary forwards the onion to the honest gatekeeper H_g in the second query phase, the challenger forms a new onion. If the accumulated bruise count exceeds the threshold, it returns $\mathsf{FormOnion}(\perp, \vec{Q}_{h_1+1\to}, \vec{z}_{h_1+1\to})$, where $\vec{Q}_{h_1+1\to}$ ends at a dummy recipient $R' = \perp$. Otherwise, it returns $\mathsf{FormOnion}(m, \vec{Q}_{h_1+1\to}, \vec{z}_{h_1+1\to})$, where $\vec{Q}_{h_1+1\to}$ ends at $R' = R$.

Security requires that no adversary can distinguish between the two cases with more than negligible advantage. Intuitively, the adversary should not be able to tell whether the onion processed by the honest parties encodes the remaining adversary-chosen parameters (message, remaining path and metadata, and bruise count) or instead contains no such information; see Fig. 1 for a visual representation of this summary.

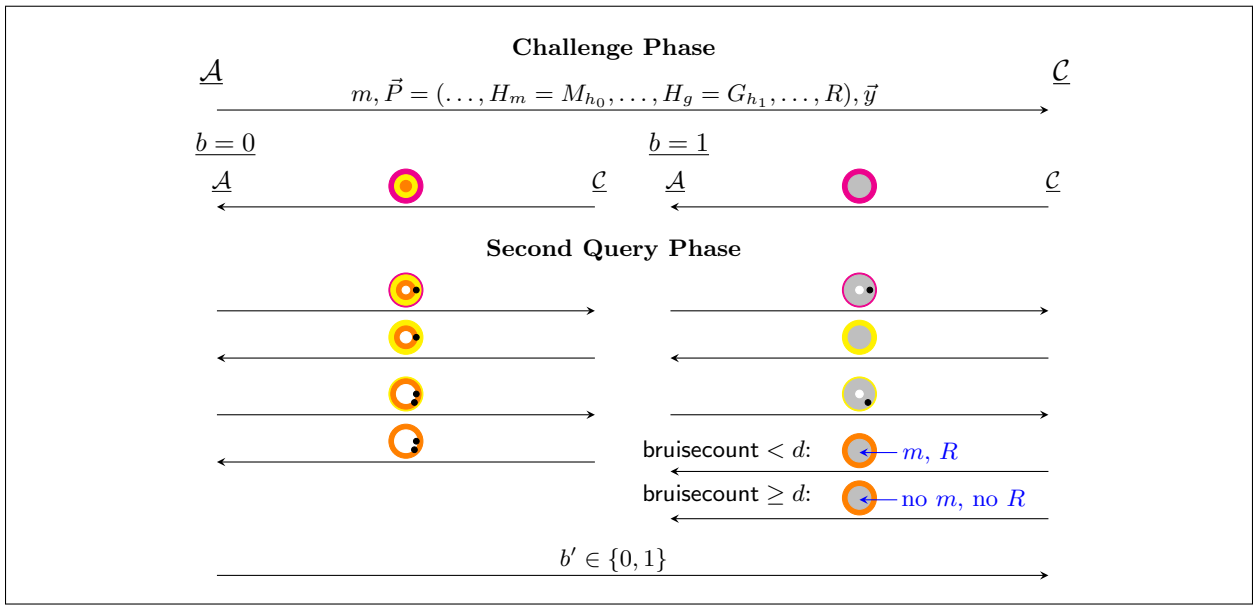

Figure 1: Visual description of the challenge and second query phase. The adversary submits parameters to the challenger, who either forms the full onion ($b = 0$) or only an initial prefix ($b = 1$). The magenta layers end at the honest mixer H_m, the yellow layers span from H_m to the honest gatekeeper H_g, and the orange layers span from H_g to the recipient R. For $b = 0$, accumulated bruises persist under honest processing, whereas for $b = 1$, newly formed onions contain no bruises. Security means that the adversary cannot tell what is encapsulated within the honest parties' layers.

Formal security definition.

We now present the formal definition of bruisable onion security through the security game, BrOnSHH2 (Bruisable Onion Security with an Honest mixer and Honest gatekeeper, version 2), below.

Security game. BrOnSHH2 is parameterized by the security parameter 1^λ, the adversary $\mathcal{A}$, the bruisable encryption scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{FormOnion}, \mathsf{PeelOnion}, \mathsf{BruiseOnion})$, and the system parameter d (which controls the amount of bruising that can be tolerated).

The challenger controls an honest mixer H_m and an honest gatekeeper H_g. (We will refer to these parties by where they are located on the path: $H_m = M_{h_0}$, the mixer at the index h_0, and $H_g = G_{h_1}$, the gatekeeper at the index h_1. *Note:* An honest party can be the recipient. This is the case when the routing path ends with either M_{h_0} or G_{h_1}.) The adversary controls all other intermediaries, as well as the recipient R (as stated under the Adversary Model, the corruptions are modeled as non-adaptive).

- **Setup:** The adversary $\mathcal{A}$ and the challenger $\mathcal{C}$ set up the parties' keys.
  1. For each party P, $\mathcal{C}$ generates a key pair $(\mathsf{pk}(P), \mathsf{sk}(P)) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and sends the public keys to $\mathcal{A}$.
  2. The adversary $\mathcal{A}$ sends the names of the honest mixer H_m; the honest gatekeeper H_g; the adversarial parties Bad and their new public keys $\mathsf{pk}(\mathsf{Bad})$ to the challenger $\mathcal{C}$.
- **First query phase:** $\mathcal{A}$ is allowed to submit queries to have an onion peeled or bruised by an honest party.
  3. $\mathcal{A}$ can direct an honest party to peel or bruise an arbitrary onion by submitting queries to peel (or bruise) onion O on behalf of an honest party $P \in \{H_m, H_g\}$. In this case, $\mathcal{C}$ will respond with the output of $\mathsf{PeelOnion}(\mathsf{sk}(P), O)$ (or $\mathsf{BruiseOnion}(\mathsf{sk}(P), O)$). $\mathcal{A}$ can submit a polynomial number of such queries.
- **Challenge phase:** $\mathcal{A}$ picks the parameters of the challenge onion and $\mathcal{C}$ responds with the challenge onion.
  4. $\mathcal{A}$ sends to $\mathcal{C}$: the message m; the routing path $\vec{P}$, where the honest mixer $H_m = M_{h_0}$ has $0 \leq h_0 < l_1$ and the honest gatekeeper $H_g = G_{h_1}$ has $0 \leq h_1 < l_2$; and the sequence of metadata $\vec{y} = (y_0, ..., y_{l-1})$.
  5. $\mathcal{C}$ samples bit $b \leftarrow\!\!\$\ \{0, 1\}$.
     - If $b = 0$, $\mathcal{C}$ runs $\mathsf{FormOnion}(m, \vec{P}, \mathsf{pk}(\vec{P}), \vec{y})$
     - If $b = 1$, $\mathcal{C}$ runs $\mathsf{FormOnion}(m, \vec{Q}_{\to h_0}, \mathsf{pk}(\vec{Q}_{\to h_0}), \vec{z}_{\to h_0})$, where $\vec{Q}_{\to h_0} = (M_0, ..., M_{h_0})$ and $\vec{z}_{\to h_0} = (y_0, ..., y_{h_0})$.
  6. $\mathcal{C}$ returns the first onion $O_{0,0}$ in the output from running FormOnion, (ex. in the case $b = 0$, $((O_{0,0}), \vec{O}_1, ..., \vec{O}_l) = \mathsf{FormOnion}(m, \vec{P}, \mathsf{pk}(\vec{P}), \vec{y})$)
- **Second query phase:** $\mathcal{A}$ is again allowed to submit queries to have an onion peeled or bruised by an honest party P.
  7. If $b = 0$; or the request *is not*
     - to peel or bruise an onion in $\vec{O}_{h_0}$ as the honest mixer M_{h_0} (query type 1)
     - to peel an onion in $\vec{O}_{h_1}$ as the honest gatekeeper G_{h_1} (type 2)

     the challenger processes the request by running the scheme's algorithm.
  8. If $(b = 1)$ and *it is the first request of its type*:
     i. **Query type 1:** the query is to the honest mixer M_{h_0} to peel or bruise an onion $O_{h_0,j} \in \vec{O}_{h_0}$. $\mathcal{C}$ runs FormOnionon the dummy message $\perp$ and the path after M_{h_0} to

G_{h_1}, i.e.,

$$\begin{aligned}\vec{Q}_{h_0+1\to h_1} &= (M_{h_0+1}, ..., M_{l_1-1}, G_0, ..., G_{h_1})\\ \vec{z}_{h_0+1\to h_1} &= (y_{h_0+1}, ..., y_{l_1+h_1})\\ \vec{O}_{h_0+1\to l_1+h_1} &\leftarrow \mathsf{FormOnion}(\perp, \vec{Q}_{h_0+1\to h_1}, \mathsf{pk}(\vec{Q}_{h_0+1\to h_1}), \vec{z}_{h_0+1\to h_1})\end{aligned}$$

$\mathcal{C}$ records $\mathsf{bruisecount} = j$, or $\mathsf{bruisecount} = j + 1$ if requested to bruise. (Tracking this information is necessary to ensure that when the challenger receives a query of type 2, the innermost onion is only recoverable if it is not too bruised.) $\mathcal{C}$ then sends $\mathcal{A}$ the output of $\mathsf{FormOnion}$, returning the tuple $(y_{h_0}, O_{h_0+1,0}, M_{h_0+1})$. Effectively, the challenger returns a new onion, formed on a *truncated path*, with *zero bruises*, and *no message*.

ii. **Query type 2:** the query is to the gatekeeper G_{h_1} to peel an onion $O_{h_1+1,j} \in \vec{O}_{h_1}$. If $\mathsf{bruisecount} + j < d$, let $m' = m$ and $R' = R$; else let $m' = \perp$ and $R' = \perp$. $\mathcal{C}$ runs $\mathsf{FormOnion}$on the message m' and the routing path consisting of the gatekeepers after G_{h_1} to the recipient R', i.e.,

$$\begin{aligned}\vec{Q}_{h_1+1\to} &= (G_{h_1+1}, ..., G_{l_2-1}, R')\\ \vec{z}_{h_1+1\to} &= (y_{l_1+h_1+1}, ..., y_{l-1})\\ \vec{O}_{l_1+h_1+1\to} &\leftarrow \mathsf{FormOnion}(m', \vec{Q}_{h_1+1\to}, \mathsf{pk}(\vec{Q}_{h_1+1\to}), \vec{z}_{h_1+1\to})\end{aligned}$$

$\mathcal{C}$ returns $(y_{h_1}, O_{l_1+h_1+1,0}, G_{h_1+1})$ to $\mathcal{A}$ where $O_{l_1+h_1+1,0}$ is the first onion in the output $\vec{O}_{l_1+h_1+1} = (O_{l_1+h_1+1,0}, O_{l_1+h_1+1,1}, ..., O_{l_1+h_1+1,l_1+1})$ of $\mathsf{FormOnion}$. Once again, the challenger forms an onion on a *truncated path* with *zero bruises*, simply having modified the message and recipient to ensure that the new onion cannot be delivered unless the bruising was below the threshold.

9. If $b = 1$ and it is *not* the first request of its type, the challenger responds with an error message.

- At the end, $\mathcal{A}$ outputs a guess b' for the bit b and wins if $b' = b$. We define bruisable-onion security as follows.

Definition 2. *A bruisable onion encryption scheme Σ is bruisable-onion secure for parameter d if there exists a negligible function $v : \mathbb{N} \mapsto \mathbb{N}$ such that every probabilistic polynomial-time adversary $\mathcal{A}$ wins $\mathsf{BrOnSHH2}(1^\lambda, \mathcal{A}, \Sigma, d)$ with advantage at most $v(\lambda)$, i.e., $\left|\Pr\left[\mathcal{A} \text{ wins } \mathsf{BrOnSHH2}(1^\lambda, \mathcal{A}, \Sigma, d)\right] - \frac{1}{2}\right| \leq v(\lambda)$.*

4 Our Onion Encryption Scheme: Peony

In many onion encryption schemes, the onion $O_i = (X_i, C_i)$ is comprised of a header and a content component. The header X_i encodes the encrypted path, while the content C_i contains the encrypted payload. We briefly summarize the essence of some foundational prior work before describing our own scheme.

Notation: In all the following, the subscript of a vector or matrix will refer to the onion layer that it corresponds to. Similarly, the index notation refers to a specific block in the set. So the $A_3[0, 0]$ refers to the matrix A intended for the third party on the path, and the block located at $[0, 0]$ of that matrix.

CL05 construction for standard onion encryption. In the CL05 construction, the header consists of blocks $X_i = (X_i[0], X_i[1], \ldots, X_i[l])$. To peel the onion, the processing party P_i first uses their secret key $\mathsf{sk}(P_i)$ to decrypt $X_i[0]$, obtaining the layer key (a symmetric key) k_i and the next party P_{i+1}. The remaining header blocks are decrypted with k_i, resulting in $(X_{i+1}[0], X_{i+1}[1], \ldots, X_{i+1}[l-1])$. To maintain header size, a final "dummy" block $X_{i+1}[l]$ is created by decrypting an all-zero string and appended to the end of the list.

The content C_i is just a single block; it is the message layer encrypted with all remaining layer keys $(k_i, k_{i+1}, \ldots, k_l)$. Decrypting this with the layer key k_i yields C_{i+1}, the content of the peeled onion.

ALU24 construction for bruisable onion encryption. The ALU24 bruisable onion $O_i = (X_i, C_i, A_i)$ includes an additional component, the sepal $A_i = (A_i[0], \ldots, A_i[l_1])$, where l_1 is the number of mixers on the path. The purpose of the sepal is to reveal the *master key* K for peeling the penultimate onion layer (revealing the recipient and corresponding onion) to the last gatekeeper, but only if the onion is not too bruised.

When the sepals A_0 are first formed by the sender, the first d sepal blocks $A_0[0], \ldots, A_0[d-1]$ in O_0 (the onion for the first mixer M_0) contain layered encryptions of K. The remaining sepal blocks $A_0[d+1], \ldots, A_0[l_1]$ are dummy blocks and do not carry any key material.

To process the onion, each mixer $M_{i<l_1}$ first obtains their layer key k_i, then derives the peeled header and content (like the CL05 construction), as well as the peeled sepal blocks. Next, they drop one sepal block:

- *To peel without bruising*: drop the rightmost sepal block (likely a dummy).
- *To bruise*: drop the leftmost sepal block (likely contains a copy of the key).

After passing through the mixers, the onion has only one sepal block remaining. This process ensures that the last remaining sepal block contains K (and can be used to obtain the final onion for the recipient) if and only if the number of bruises is less than the threshold d.

To process the onion, each gatekeeper $G_{i<l_2-1}$ (excepting the final gatekeeper G_{l_2-1}) first obtains its layer key k_i by decrypting the first header block in the onion. It then decrypts the remaining header, content, and (singular) sepal block to produce the peeled onion.

In contrast, the last gatekeeper has a special role. G_{l_2-1} obtains its key k_{l-1} but can only use it to process the sepal block. If peeling the sepal block reveals the master key K, G_{l_2-1} uses K to peel the remaining header and content blocks and obtain the onion for the recipient R (at this point, G_{l_2-1} finally learns R). Otherwise, if the master key is lost, G_{l_2-1} cannot produce the onion for R.

The ALU24 construction is conceptually simple and fairly efficient. Regardless of which processing parties chose to bruise, the sepal blocks that a mixer M_i receives depend only on the bruise count so far. For example, if M_3 receives an onion with two bruises, this looks the same whether the bruises were applied by (M_0, M_2) or by (M_1, M_2). Thus, the number of "valid" forms the onion can take is "small:" for each processing party, it equals the number of possible bruise counts for that party. This enables efficient integrity checks on the sepal blocks by verifying that their hash matches one of the options stored in the header block $X_i[0]$, which keeps the onion size relatively small.

The downside is that the ALU24 construction clearly leaks the mixer's position on the path. Anyone observing the onion can determine how many hops remain from the onion size, and onions with different numbers of layers no longer mix together. We use the analysis above to highlight our construction goals.

Construction goals. First, to ensure that the number of valid forms remains small, *bruising must not depend on who did the bruising.* Second, as in the ALU24 construction, bruising must be irreversible, and the total number of bruises on the onion should remain hidden from all parties along the path. Finally, the construction *must not leak the current position of the onion on the routing path to the processing party.*

Below, we present Peony Onion Encryption Scheme (POES). Unlike its predecessor, POES satisfies a stronger notion of bruisable onion security (Definition 2). As in ALU24, our onions consist of a header, content, and sepal. Since the header and content function essentially the same way as in prior work, we focus on providing a detailed description of the sepal. In Section 4.1, we first build intuition for how the sepal operates, and in Section 4.2, we give a complete description of our scheme. The security proof appears in Section 4.3.

4.1 A Description of the Sepal Matrix

In addition to the symmetric keys $k_0, k_1, \ldots, k_l$, including the master key K, our construction uses a set of *sepal keys* $(s_{i,j})_{i\in[l_1],0\le j\le i}$ to enforce correct processing of the sepal matrix; these are sampled by the sender.

Matrix state. Each mixer M_i can receive an onion in one of i states of bruising. We can track these possible states in a matrix, where the rows correspond to bruises and the columns correspond to mixers. Each block (i, j) corresponds to the mixer M_i, with j bruises; for example, $(1, 0)$, corresponds to the mixer M_1, with zero bruises (the light pink circle in Figure 2).

From any state (i, j), we can only progress to one of two states: $(i + 1, j)$ without bruising or $(i + 1, j + 1)$ with bruising. Taking our earlier example, from the cell $(1, 0)$, we mandate that the onion progresses to state $(2, 0)$ or $(2, 1)$ (corresponding to the next mixer M_2 with the same number of bruises or one additional bruise); see Figure 2 below. Note that we do not access the states representing prior mixers or fewer bruises.

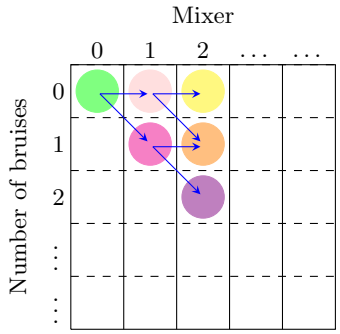

Figure 2: The matrix of possible states includes the possible combinations of mixers and the number of bruises. Starting from mixer M_0 with 0 bruises (shown in green), we can transition to the state for the next mixer M_1 with either 0 bruises (shown in the pink) or with 1 bruise (shown in magenta). Assuming that M_0 doesn't bruise, we are in state $(1, 0)$ (pink) and can move to either $(2, 0)$ (yellow) or $(2, 1)$ (orange). If M_0 does bruise, we are in state $(1, 1)$ (magenta) and can move to either $(2, 1)$ (orange) or $(2, 2)$ (purple).

Matrix rotation. Let the *active cell* be the cell (i, j) corresponding to the current state of the matrix. Knowledge of the active cell reveals both the bruise count and the path position, so we must hide the current location of the active cell from the processing party.

We accomplish this by requiring that the active cell is always positioned at the index $[0, 0]$. If a router wants to process the onion without bruising it, it deletes the first column and inserts a

dummy column at the end of the matrix. In the example shown in Figure 3 (top row), we can see that after this operation (where M_0 peels without bruising), the active cell shown in orange $(1, 0)$, which represents mixer M_1 with zero bruises, is now located at the index $[0, 0]$.

To "bruise" the onion, the first row of the matrix is *moved* to the bottom, preserving its contents. Then, as with processing, the first column is deleted and a dummy column inserted at the end of the matrix. In Figure 3 (bottom row), the matrix begins with the orange active cell $(1, 1)$ at index $[1, 1]$. After the rotations are performed, the active cell $(1, 1)$, is now positioned at index $[0, 0]$.

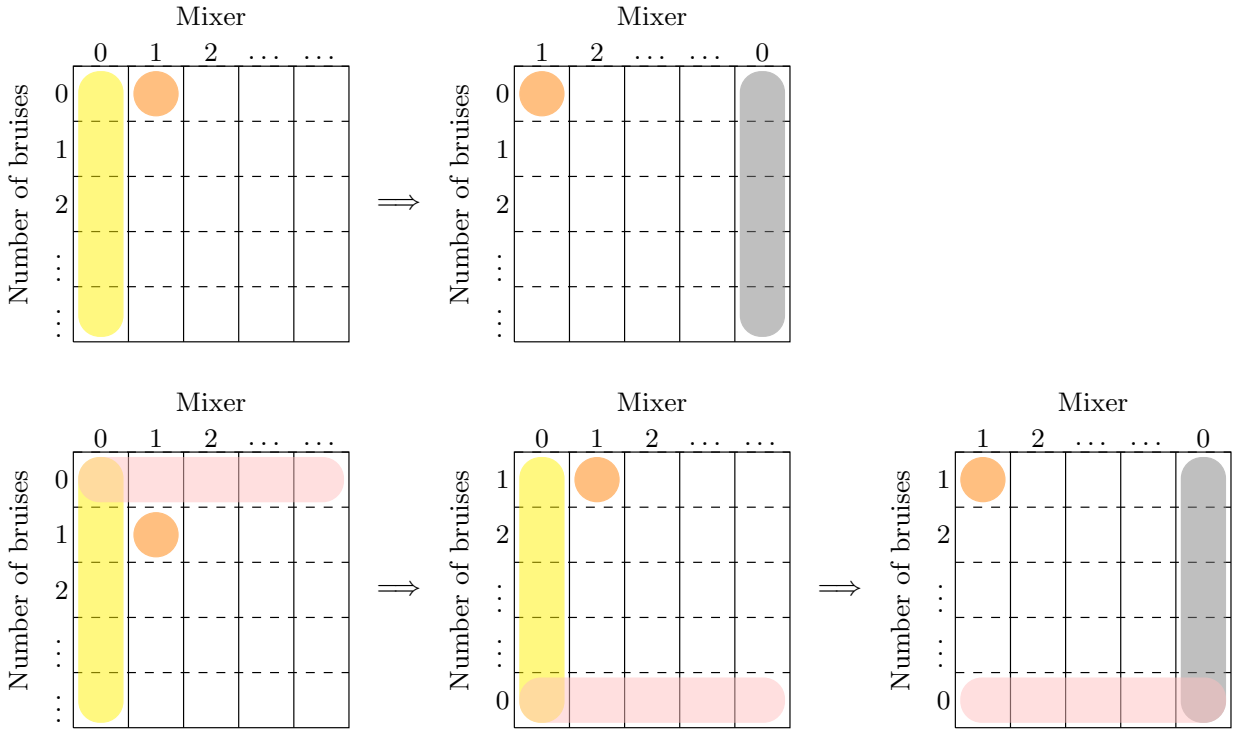

Figure 3: A visual representation of matrix rotations with the next active cell highlighted in orange. First, M_0 removes its column and replaces it with a dummy on the right side of the matrix. Next, M_1 bruises the onion, which involves a row rotation followed by a column replacement (shown in the bottom row).

Sepal matrix. We need to ensure that each sepal block (i, j) can only be used to access blocks $(i + 1, j)$ and $(i + 1, j + 1)$. We do this by embedding two keys into block (i, j): $s_{i+1,j}$, $s_{i+1,j+1}$. The first decrypts block $(i + 1, j)$, the second, block $(i + 1, j + 1)$. For party M_i to process the sepal matrix without bruising, it first uses sepal key $s_{i+1,j}$ to decrypt block $(i + 1, j)$ and then manipulates the columns. To bruise, it uses the sepal key $s_{i+1,j+1}$ to decrypt block $(i+1, j+1)$ and then manipulates the rows and columns. In the end, the block located at $[0, 0]$ will be encrypted only under the layer key for the next party k_{i+1} and is therefore the only block the next party will be able to access. All other blocks are locked with a combination of sepal and layer keys that make them inaccessible.

To ensure that the onion for M_{i+1} appears the same, regardless of which parties contributed to the j (or $j + 1$) prior bruises, each mixer must deterministically generate a dummy column. The mixer generates encryptions of $0, 1, 2, ...$ under its layer key. It then creates a dummy column using the ciphertexts by placing the encryption of 0, next to the smallest numerical value in the prior column, the encryption of 1 next to the second smallest numerical value, etc.

Putting it all together, each mixer proceeds with the following instructions for processing the

sepal matrix:

1. Remove a layer of encryption off of the whole matrix using layer key k_i, to get access to the active cell located at $[0, 0]$.
2. Using sepal key $s_{i+1,j}$ (not bruise) or $s_{i+1,j+1}$ (to bruise), remove an encryption off of the corresponding block in the column $i + 1$.
3. If bruising, move the first row to the bottom.
4. Delete the first column and insert the dummy column at the end of the matrix.

Gatekeepers and master key. Finally, we embed d copies of the master key K into the first d cells of the final column, filling the remaining cells with $\emptyset$. If the onion is bruised less than d times, the cell in $[0, 0]$ for the final gatekeeper will contain a copy of K. Otherwise it will be $\emptyset$ and the onion will be undecryptable.

Note: Gatekeepers don't do any matrix manipulation. Instead, they *encrypt* the first column with the layer key, and *decrypt* all the other cells in the matrix, along with the active cell. This ensures that no information about the bruise count leaks to the final gatekeeper.

Integrity checks. If the onion was processed incorrectly by a prior party (such as if a dishonest mixer used both sepal keys to unlock two separate blocks), then this should be detected by an honest processor. To this end, the header block for each party stores a set of verification hashes/tags $t_i, \vec{T_i}$, which guarantee onion integrity. The tag t_i corresponds to a hash of the header and content, which does not change based on the amount of bruising. The tags $\vec{T_i}$ are the set of valid states of the sepal matrix, where each of the valid matrix states is hashed and stored in the header. When P_i receives the onion, they will check to ensure the hash of the sepal matrix matches a value in $\vec{T_i}$ and the hash of the headers and content match t_i. If it does not, an honest party will realize the onion has been tampered with and drop it.

4.2 Complete Description of Peonies

Here, we provide a complete description of our bruisable onion encryption scheme $(\mathsf{KeyGen}, \mathsf{FormOnion}, \mathsf{PeelOnion}, \mathsf{BruiseOnion})$. We build on a CCA-secure public-key encryption scheme $(G, \mathsf{Enc}, \mathsf{Dec})$, a collision-resistant hash function h, and a pseudorandom permutation F defined over the space $(\mathcal{K}, \mathcal{X})$. Following prior work [CL05], we adopt the convention that $\{X\}_k = P(k, X)$ denotes encryption of a block $X \in \mathcal{X}$ under the symmetric key $k \in \mathcal{K}$, while $\}X\{_k = P^{-1}(k, X)$ denotes decryption of X under k.

Key generation. The key generation algorithm is inherited directly from the underlying encryption scheme. The public key pair of each party P is generated by running KeyGen on the security parameter, i.e., $(\mathsf{pk}(P), \mathsf{sk}(P)) \leftarrow G(1^\lambda)$.

Onion processing. To process the onion $O_{i,j} = \big(X_i = (X_i[0], \ldots, X_i[l]), C_i, A_{i,j}\big)$, as the processing party P_i, first decrypt the header block $X_i[0]$ using the secret key $\mathsf{sk}(P_i)$. If this decryption succeeds (i.e., the integrity checks $h((X_i[0], \ldots, X_i[l]), C_i) = t_i$ and $h(A_{i,j}) \in \vec{T_i}$ passes), then recover the role (Mixer, $\mathsf{Gatekeeper}$, $\mathsf{LastGatekeeper}$, or $\mathsf{Recipient}$), the symmetric key k_i, and the metadata y_i (if present).

Next, remove the remaining encryption from $X_i[0]$ to learn the location of the next processing party P_{i+1}, using the layer key k_i; additionally use it to peel a layer of encryption from the remaining header blocks $(X_i[1], \ldots, X_i[l])$, the content block C_i, and the sepal matrix $A_{i,j}$. (In the special case of the $\mathsf{LastGatekeeper}$, the header and content are peeled with the master key K.)

Peeling the content C_i produces the content C_{i+1}. (In the special case of the Recipient, this produces the message m.)

The decrypted header blocks of X_i (excluding the first block), become the first $l-1$ blocks for X_{i+1}. Append a dummy l-th block the end to preserve header size.

If the role is Mixer, the sepal blocks are processed as follows:

- *Peeling without bruising:* use the first of the two candidate sepal keys $(s_{i+1,j}, s_{i+1,j+1})$ to unlock the cell $A_{i,j}[1,0]$.
- *Bruising:* use the second key $s_{i+1,j+1}$ to unlock the cell $A_{i,j}[1,1]$, and then rotate the rows of the sepal matrix.

In both cases, drop the first column of the sepal matrix and append a dummy column at the end, ensuring that the matrix maintains its size and shape across rounds.

If the role is Gatekeeper, the sepal matrix $A_{i+1,j}$ is $A_{i,j}$, where all the cells except for the first column are decrypted with the layer key. The cells in the first column are *encrypted* with k_i, except for the active cell, which is decrypted like all the others.

In the case of the LastGatekeeper, if processing the sepal block retrieves the master key K, then use K to decrypt the remaining header blocks and the content block. This yields both the onion for the recipient $O_{i+1} = O_l$ and the recipient's identity R. Otherwise, the onion dies as it cannot be forwarded. See Appendix C for an example (with diagrams) of sepal processing.

Onion forming. We now explain how to form an onion given the following parameters: the message m, the metadata $\vec{y} = (y_0, y_1, \ldots, y_{l-1})$, the routing path $\vec{P} = (M_0, \ldots, M_{l_1-1}, G_0, \ldots, G_{l_2-1}, R)$, and the associated public keys $\mathsf{pk}(\vec{P}) = \{\mathsf{pk}(P) \mid P \in \vec{P}\}$. Here, l_1 is the number of mixers, l_2 is the number of gatekeepers, and $l = l_1 + l_2$, where $l+1$ is the total number of parties on the path (including the recipient).

First, we sample the symmetric keys: the layer keys $k_0, k_1, \ldots, k_l$, the master key K, and sepal keys $(s_{i,j})_{i\in[l_1],0\le j\le i}$.

Recall that the output of FormOnion is a sequence of lists of onions $(\vec{O}_0, \ldots, \vec{O}_{l-1})$, where each $\vec{O}_i = (O_{i,j})_{0\le j\le i}$ consists of the onions for the ith party on the path, with one onion for each possible valid bruising variant. We begin by describing how to construct the sepal matrices for each onion $O_{i,j}$, and then, how to assemble the onions themselves starting from the innermost layers.

Intermediary on the path

Number of bruises	M_0	M_1	M_2	G
0	$\{\{s_{11}\}_{k_0}\}^{s_{10}}$	$\{\{s_{21}\}_{k_0}\}^{s_{20}}_{s_{10}, k_1}$	$\{\{s_{31}\}_{k_0}\}^{s_{30}}_{k_1, s_{20}, k_2}$	$\{\{K\}_{k_0, k_1, k_2, s_{30}, k_3, k_4}\}$
1	$\{r_{0,1}\}$	$\{\{s_{22}\}_{k_0}\}^{s_{21}}_{s_{11}, k_1}$	$\{\{s_{32}\}_{k_0}\}^{s_{31}}_{k_1, s_{21}, k_2}$	$\{\{K\}_{k_0, k_1, k_2, s_{31}, k_3, k_4}\}$
2	$\{r_{0,2}\}$	$\{r_{1,2}\}$	$\{\{s_{33}\}_{k_0}\}^{s_{32}}_{k_1, s_{31}, k_2}$	$\{\{\emptyset\}_{k_0, k_1, k_2, s_{32}, k_3, k_4}\}$
3	$\{r_{0,3}\}$	$\{r_{1,3}\}$	$\{r_{2,3}\}$	$\{\{\emptyset\}_{k_0, k_1, k_2, s_{33}, k_3, k_4}\}$

Figure 4: Sepal matrix as generated by the sender, with routing path $\vec{P} = \{M_0, M_1, M_2, G_0, G_1, R\}$, and bruising tolerance of 1.

The sepal matrix $A_{0,0}$. As a starting point, consider the sepal matrix $A_{0,0}$ for onion $O_{0,0}$ (corresponding to the first mixer with no bruises). Each cell $A_{0,0}[x,y]$ that corresponds to a *valid state* (mixer M_x with y bruises) contains a pair of sepal keys $(s_{x+1,y}, s_{x+1,y+1})$, where $s_{x+1,y}$ unlocks

the cell $[x+1, y]$ (peeling without bruising), and $s_{x+1,y+1}$ unlocks the cell $[x+1, y+1]$ (peeling with bruising). A cell should be accessible only if the onion is for mixer M_x and contains exactly y bruises; this would be the cell $A_{x,y}[0,0]$. To enforce this, each block $A_{0,0}[x, y]$ is encrypted under the layer key k_x, the sepal key $s_{x,y}$, and then sequentially under the layer keys $k_{x-1}, \ldots, k_0$. We denote this by

$$A_{0,0}[x, y] = \{\, s_{x+1,y}, s_{x+1,y+1} \,\}_{k_x, s_{x,y}, k_{x-1}, \ldots, k_0}, \quad x \neq l_1$$

The final column of the matrix contains d copies of the master key K, the remaining blocks contain null values. Each block $A_{0,0}[l_1, y]$ is encrypted under the gatekeeper's layer keys $k_{l-1}, \ldots, k_{l_1}$, the corresponding sepal key $s_{l_1,y}$, and the remaining layer keys $k_{l_1-1}, \ldots, k_0$. We denote this by

$$A_{0,0}[l_1, y] = \{\, K \,\}_{k_{l-1}, \ldots, k_{l_1}, s_{l_1,y}, k_{l_1-1}, \ldots, k_0}, \quad y < d$$
$$A_{0,0}[l_1, y] = \{\, \emptyset \,\}_{k_{l-1}, \ldots, k_{l_1}, s_{l_1,y}, k_{l_1-1}, \ldots, k_0}, \quad y \geq d$$

All other cells in the sepal matrix are dummy entries. (See Fig. 4 for an example of sepal matrix $A_{0,0}$ as generated by the sender.)

All other sepal matrix variants $A_{i,j}$. To obtain the sepal matrix $A_{i,j}$ for mixer M_i with j bruises, we start from the base sepal matrix $A_{0,0}$ and apply the appropriate sequence of peel and bruise operations. For example, to construct $A_{3,2}$: peel and bruise $A_{0,0}$ to obtain $A_{1,1}$ (the sepal matrix that M_1 would receive when running PeelOnion); peel and bruise $A_{1,1}$ to obtain $A_{2,2}$; peel (without bruising) $A_{2,2}$ to obtain $A_{3,2}$. After deriving each sepal matrix $A_{i,j}$, we compute its hash $T_{i,j} = h(A_{i,j})$. The collection of these values forms the sepal integrity set $\vec{T}_i$ for M_i, namely $\vec{T}_i = \{\, T_{i,j} \mid j \in \text{valid bruise counts for } M_i \,\}$.

Forming the onion layers. The onion layers are formed as in prior work. The header and content are formed recursively, starting from the onion intended for the recipient, and working outwards. At each layer, the tag $t_i = h((X_i[1], \ldots, X_i[l]), C_i)$ is also included in the header (along with sepal tags $\vec{T}_i$) to ensure onion integrity. See Figure 5 for a summary of Peony Onion Encryption Scheme.

FormOnion$(m, \vec{P}, \mathsf{pk}(\vec{P}), \vec{y})$

1: Sample symmetric keys: layer keys $k_0, \ldots, k_l$, master key K, and sepal keys $(s_{i,j})_{i \in [l_1], 0 \leq j \leq i}$.
2: Construct base sepal matrix $A_{0,0}$ for onion $O_{0,0}$:
  i: For mixer M_x with y bruises, set $A_{0,0}[x, y] = \{s_{x+1,y}, s_{x+1,y+1}\}_{k_x, s_{x,y}, \ldots, k_0}$
  ii: For the first d cells of the final column, set $A_{0,0}[l_1, y] = \{K\}_{k_{l-1}, \ldots, k_{l_1}, s_{l_1,y}, k_{l_1-1}, \ldots, k_0}$. When $y \geq d$, set $A_{0,0}[l_1, y] = \{\emptyset\}_{k_{l-1}, \ldots, k_{l_1}, s_{l_1,y}, k_{l_1-1}, \ldots, k_0}$.
  iii: Fill all remaining cells with dummy entries.
3: Derive sepal matrices $A_{i,j}$ from $A_{0,0}$, capture sepal tags for each layer $\vec{T}_i$.
4: Assemble onions recursively from recipient to first layer:
  i: Form header blocks X_i, which include the tags $t_i, \vec{T}_i$, and layer key k_i encrypting P_i's role, the next processing party P_{i+1}, and metadata y_i.
  ii: Form content block C_i.

PeelOnion$(\mathsf{sk}(P_i), O_{i,j})$

1: Decrypt header block $X_i[0]$ using $\mathsf{sk}(P_i)$:
  i: Verify integrity: $h((X_i[0], \ldots, X_i[l]), C_i) = t_i$ and $h(A_{i,j}) \in \vec{T}_i$.
  ii: Recover layer key k_i, and use it to retrieve P_i's role, metadata y_i, and next processing party P_{i+1}
2: Peel remaining header blocks $X_i[1], \ldots, X_i[l]$, content C_i, and sepal matrix $A_{i,j}$ using k_i (or K for the last gatekeeper):
  i: Mixer: unlock appropriate active cell. If bruising, rotate rows. Regardless, drop the first column and append a dummy column to preserve matrix size.
  ii: Gatekeeper: decrypt non-first columns and active cell; re-encrypt first column except for active cell.
  iii: LastGatekeeper: decrypting sepal matrix with k_i reveals the master key K. (If doing so reveals $\emptyset$, stop processing the onion.) Use K to obtain recipient onion and recipient R.
  iv: Recipient: peeling the content yields message m.

Figure 5: Summary of the Peony Onion Encryption Scheme.

4.3 Proof of Security

Here, we summarize our proof that our construction satisfies the definition of security provided in Section 3.2.

Theorem 1. *Peony Onion Encryption Scheme is bruisable-onion secure (Definition 2), assuming there exist a CCA2-secure public key encryption scheme, a pseudorandom permutation, and a collision-resistant hash function.*

Proof. We describe a sequence of hybrid experiments $\mathsf{Hybrid}_0, \ldots, \mathsf{Hybrid}_{15}$, and provide a brief explanation of why each pair of consecutive experiments consists of indistinguishable scenarios, for the case where the challenge onion is too bruised to recover from the innermost onion. Detailed proofs of indistinguishability for experiment pairs are supplied in Appendix B.

Recall that in the security game $\mathsf{BrOnSHH2}$, the honest mixer is $H_m = M_{h_0}$, located in the position $0 \leq h_0 < l_1$ and the honest gatekeeper $H_g = G_{h_1}$ is sitting in the position $l_1 \leq l_1 + h_1 < l$. When the challenger returns the onion O_0, intended for the first intermediary M_0, the subscript indicates the *layer* of the onion (the party it is intended for).

Hybrid_0: the challenge onion O_0 is formed correctly. (This is the same as the game when $b = 0$.)

$\updownarrow$ Indistinguishable from CCA2-secure public key encryption.

Hybrid_1: same as Hybrid_0 except that ciphertext E_{h_0}, where $(E_{h_0}, I_{h_0}) \in X_{h_0}^{h_0}$, is an encryption under $\mathsf{pk}(M_{h_0})$ of the dummy key 0. (The challenger still samples for the layer key k_{h_0} and uses it to form the h_0^{th} onion layers $\vec{O}_{h_0}$.)

$\updownarrow$ Indistinguishable from collision resistance of the hash function.

Hybrid_2: same as Hybrid_1 except that if, in the second query phase, the challenger receives an onion $O = (\vec{X}, C, \vec{A}) \notin \vec{O}_{h_0}$ such that the public portion of the first header block $E = E_{h_0}$, the challenger responds with $\perp$ rather than processing O.

$\updownarrow$ Indistinguishable from PRP security.

Hybrid_3: same as Hybrid_2 except that the challenger forms the h_0^{th} onion layers $\vec{O}_{h_0}$ using a truly random permutation rather than a PRP keyed with k_{h_0}.

$\updownarrow$ Identically distributed since $\vec{O}_{h_0-1}, ..., \vec{O}_0$ are wrapped around truly random blocks.

Hybrid_4: same as Hybrid_3 except that the challenger uses the message m and the truncated path $M_0, \ldots, M_{h_0}$ with the associated metadata sequence $(y_0, \ldots, y_{h_0})$ to form O (instead of the full path, and full sequence of metadata).

$\updownarrow$ Identically distributed since $\vec{O}_l, ..., \vec{O}_{h_0+1}$ are independent of the path up to M_{h_0}.

Hybrid_5: same as Hybrid_4 except that the first query to peel or bruise an onion $O_{h_0} = O_{h_0,b} \in \vec{O}_{h_0}$ on behalf of M_{h_0} peels to a new onion formed using the message m, partial routing path $(M_{h_0+1}, \ldots, R)$, and the associated sequence of metadata $(y_{h_0+1}, \ldots, y_{l-1})$. (The newly formed onion O_{h_0+1} has same number of bruises b, as the initial onion that was queried.)

$\updownarrow$ Indistinguishable from PRP security.

Hybrid_6: same as Hybrid_5 except that the challenger forms the h_0^{th} onion layers $\vec{O}_{h_0}$ using the PRP keyed with k_{h_0}, instead of a truly random permutation.

$\updownarrow$ Indistinguishable from CCA2-security.

Hybrid_7: same as Hybrid_6 except that the ciphertext E_{h_0} is an encryption of the real key k_{h_0}, rather than the dummy key 0. (At this stage, the challenge onion O_0 is the same as in the game when $b = 1$, but the onions returned by M_{h_0} and G_{h_1} are not quite the same as when $b = 1$. The challenger forms the O_0 using the message m, the routing path $(M_0, \ldots, M_{h_0})$, and the metadata $(y_0, \ldots, y_{h_0})$. The onion O_{h_0+1} returned by the challenger on behalf of the mixer M_{h_0} is a new onion with the correct number bruises, formed by running $\mathsf{FormOnion}(m, (M_{h_0+1}, \ldots, R), (y_{h_0+1}, \ldots, y_{l-1}))$.)

$\updownarrow$ Indistinguishable from CCA2-secure public key encryption.

Hybrid_8: same as Hybrid_7 except that the ciphertext E_{h_1} is an encryption under $\mathsf{pk}(G_{h_1})$ of the dummy key 0. (The challenger still samples for the layer key k_{h_1} and uses it to form the h_1^{th} onion layers $\vec{O}_{h_1}$.)

$\updownarrow$ Indistinguishable from the collision resistance of the hash function.

Hybrid_9: Same as Hybrid_8 except that if, in the second query phase, the challenger recieves an onion $O = (\vec{X}, C, \vec{A}) \notin \vec{O}_{h_1}$ such that the public portion of the first header block $E = E_{h_1}$, the challenger responds with $\perp$ (rather than processing O).

$\updownarrow$ Indistinguishable from PRP security.

Hybrid_{10}: same as Hybrid_9 except that the challenger forms the h_1^{th} onion layers $\vec{O}_{h_1}$ using a truly random permutation, rather than a PRP keyed with k_{h_1}.

$\updownarrow$ Identically distributed since $\vec{O}_{h_1-1}, \ldots, \vec{O}_{h_0+1}$ are wrapped around truly random blocks.

Hybrid_{11}: same as Hybrid_{10} except that the challenger uses the dummy message $\perp$, truncated path $(M_{h_0}, \ldots, G_{h_1})$ and associated sequence of metadata (instead of the real message, full path, and full sequence of metadata) to form O_{h_0+1}.

$\updownarrow$ Identically distributed since the inner layers $\vec{O}_l, \ldots, \vec{O}_{l_1+h_1+1}$ are independent of the path up to G_{h_1}.

Hybrid_{12}: same as Hybrid_{11} except that the first query to peel or bruise an onion $O_{l_1+h_1} = O_{l_1+h_1,b} \in \vec{O}_{l_1+h_1}$ on behalf of G_{h_1} peels to a new onion formed using the message m, the routing path $(G_{h_1+1}, \ldots, R)$, and the associated metadata sequence. (The newly formed onion $O_{l_1+h_1+1}$ has the correct number of b bruises.)

$\updownarrow$ Indistinguishable from PRP security.

Hybrid_{13}: same as Hybrid_{12} except that the challenger forms the $l_1 + h_1{}^{th}$ onion layers $\vec{O}_{l_1+h_1}$ using the PRP keyed with k_{h_1} instead of a truly random permutation.

$\updownarrow$ Indistinguishable from CCA2-security.

Hybrid_{14}: same as Hybrid_{13} except that the ciphertext $E_{l_1+h_1}$ is an encryption of the real key $k_{l_1+h_1}$ rather than the dummy key 0.

$\updownarrow$ Identically distributed since the sepal matrix is composed of truly random blocks wrapped in layers of encryption, and the bruise count of the onion is hidden.

Hybrid_{15}: same as Hybrid_{14} except for how bruised the onion O_{h_0+1} is. The onion O_{h_0+1} that M_{h_0} returns will be completely unbruised. The challenger remembers how bruised O_{h_0} was however, and forms the onion $O_{l_1+h_1+1}$ accordingly; thus $O_{l_1+h_1+1}$ is formed identically as in Hybrid_{11}. Note that Hybrid_{15} is indistinguishable to the case when $b = 1$.

□

Remark 1. *Following [ALU24], we give a game-based definition of bruisable onion security which implies an ideal functionality in the UC model. The corresponding ideal functionality (formally defined and analyzed in Appendix A) builds and manages onions on behalf of honest parties, mirroring the challenger's behavior in our game. Onions are formed piecewise, with path segments delimited by honest parties along the path. Each segment is encrypted independently, so that layers across segments are information-theoretically unrelated. For each* FormOnion *query, the functionality records the layers and the bruise count. Security requires that an adversary cannot distinguish honestly formed onions from onions generated segment by segment by a simulator. We show that our game-based definition implies realizability of this functionality in Appendix A.*

Remark 2. *In both the real world and our game, the adversary may replay an onion with varying bruise levels (e.g., $O_{i,j}$ vs. $O_{i,j+1}$). Allowing both versions to be processed would make the bruise count malleable, breaking security. Thus, in our game, the challenger never processes the same*

onion twice. Protocols using bruisable onion encryption must do the same: in our construction, all versions of an onion share a key k_i, which honest parties can store to detect and reject replays.

Remark 3. *It is straightforward to extend Peony Onion Encryption to a "resettable" form. Instead of maintaining a single bruise counter along the entire path, we introduce reset points $Q_1, Q_2, \ldots, Q_\ell$. An onion accumulates bruises during each epoch, defined as the subpath between consecutive reset points. If the count exceeds the threshold before Q_i, the onion is dropped; otherwise, the counter is reset to zero and a new epoch begins.*

In this setting, the single sepal matrix is replaced with a set of matrices. At each reset point Q_i, provided the onion survives the epoch, the leftmost matrix is discarded and a dummy matrix appended to the right. The next epoch then proceeds using the new leftmost matrix, unlocked by the master key of the discarded one. See Figure 6 below.

We omit to include this property in the security definition for readability, it follows an identical structure to other indistinguishability properties. In this case, when $b = 0$ is sampled, the onion will be processed over multiple epochs and be reset, while in $b = 1$, it will be replaced by a new onion beginning at the first epoch. However, the asynchronous routing protocol relies on the ability to reset these sepal matrices.

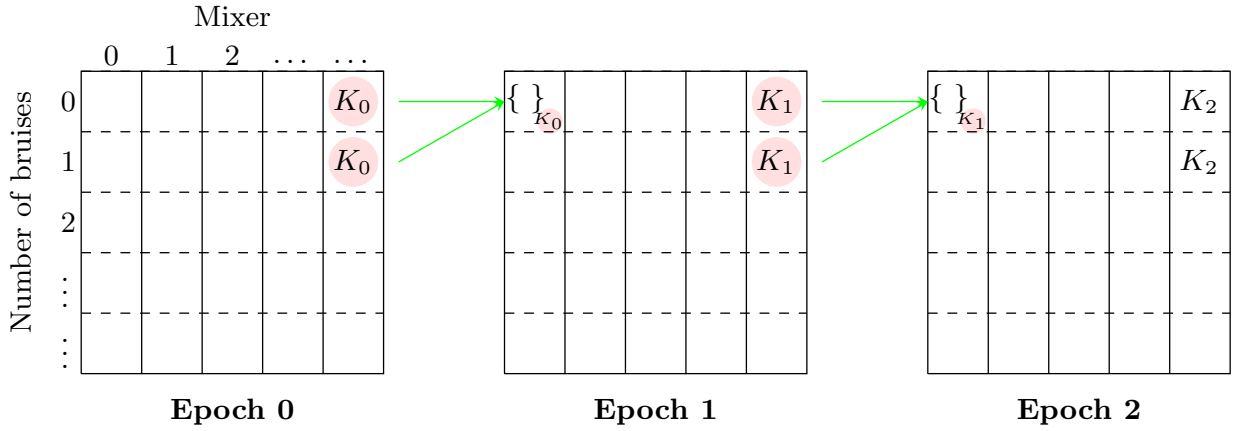

Figure 6: Here, we describe a series of sepal matrices, corresponding to each epoch in our routing protocol. The key K_0 acts as the master key for the first sepal matrix. If the onion is not too bruised, obtaining this key will allow us to unlock the active cell of the next sepal matrix for epoch 1. If the onion obtains too many bruises in a given epoch, it will not be able to retrieve the master key to unlock the next sepal matrix, and will die at the end of the epoch.

5 Our Fully Anonymous Onion Routing Protocols

All prior fully anonymous schemes [vdHLZZ15, TGL+17, KCGDF17, ALU21] suffer from at least one fundamental limitation: (1) poor scalability, with round complexity linear in the number of parties; (2) weak fault tolerance, failing upon the loss of even a single message packet; or (3) an inability to support the asynchronous setting using bruisable onions. Each of these shortcomings is highly undesirable.

The primary technical challenge we address is the adversary's ability to drop or delay Alice's onions at the outset of the protocol. By selectively suppressing these onions, the adversary can wait until protocol termination, identify which parties did not receive an onion, and thereby infer the recipient's identity.

Our main contribution is the design of two new protocols that eliminate this attack and achieve

full anonymity. First, we present $\Pi_\bullet$,[4] a synchronous protocol that employs *looping onions*, an idea inspired by Loopix [PHE+17] and Trellis [LSSD23]. Compared to existing fully anonymous, fault-tolerant synchronous schemes, $\Pi_\bullet$ is conceptually simpler and achieves strictly better asymptotic efficiency.

We then introduce $\Pi_{\check{o}}$,[5] the first fully anonymous protocol in the asynchronous setting. $\Pi_{\check{o}}$ is obtained from $\Pi_\bullet$ via two key modifications. First, we incorporate a new onion encryption scheme, POES. Second, we replace looping onions with *signaling onions*, which enable cascading aborts that cannot be delayed by the adversary.

We conclude this section with a brief overview of the technical building blocks required to instantiate these protocols.

5.1 Tools from Prior Work: Checkpoint Onions and Merging Onions

As seen in Section 4, the FormOnion algorithm takes an array of metadata $(y_0, y_1, \ldots, y_{l-1})$ as part of its input. During processing with PeelOnion, the party P_i is given access to its corresponding metadata, y_i. This metadata can be used to embed a "short message" from the sender to each processing party. These short messages can be used to create *checkpoint onions* and *merging onions*.

Checkpoint onions. Checkpoint onions in aggregate across the network serve as a diagnostic tool to determine if the adversary has been manipulating the network traffic by dropping or delaying onions.

If a party expects to encounter "a lot" (at least polylogarithmic in the security parameter) of checkpoint onions, the fraction of checkpoints they receive is nearly always very close to the overall fraction of onions in circulation. Relevant here, the proportion of adversarially dropped checkpoint onions reflects the overall proportion of onions dropped by the adversary. These follow from well-known probability concentration bounds.

An individual *checkpoint onion* [ALU18] is a dummy onion formed by a party P_i for a partner P_j in a round r (necessarily, P_j will be creating a corresponding onion for P_i in the same round). To determine whether to form a checkpoint onion, we compute a pseudorandom function over two inputs, $F(\cdot,\cdot)$. The first input is the shared secret between the parties $\mathsf{sk}(P_i, P_j)$. The second input is the concatenation of the round number and zero, $r||0$. The output of the pseudorandom function is used as an input to a binary predicate b; if the result is 1, a checkpoint onion should be formed. The metadata y_r used in the rth layer of the checkpoint onion is also based on the output of the pseudorandom function, this time computed on an input of $r||1$. Since P_j also follows this protocol, P_j expects to see the metadata y_r when it processes its onions in round r; if this value is not present, it indicates that the adversary has dropped the onion.

Merging onions. Merging onions [ALU21] are a way to ensure that the recipient receives a message, even if the adversary drops many of the sender's onions upfront.

To create a pair of merging onions, we start by constructing two partially overlapping routing paths. The paths start off independently for the first T intermediaries, and then converge at the intermediary I_{T+1}. E.g., let $\vec{I^0} = (I_1^0, I_2^0, ..., I_T^0, I_{T+1}, \ldots, I_l)$ be the path for the first merging onion, and let $\vec{I^1} = (I_1^1, I_2^1, ...I_T^1, I_{T+1}, \ldots, I_l)$ be the path for the second merging onion; we can see that from I_{T+1} to the recipient, the onions have identical paths.

[4] $\Pi_\bullet$ is pronounced "Pi-SEED."
[5] $\Pi_{\check{o}}$ is pronounced "Pi(NE)-APPLE."

Using the partially overlapping routing paths, we construct the onions by calling FormOnion twice; each time we provide the message m, the public keys for the associated parties, and the metadata $(\perp, \ldots, y_{T+1}, \ldots, \perp)$. The difference is that for the first onion, O^0, we use the path I^0; for the second onion, we use path I^1. The resulting pair of onions, O^0 and O^1 will take different routes to the intermediary I_{T+1} where they will meet. That intermediary can then recognize the two onions are identical after peeling both and can therefore *merge* them (i.e., discard one).

Merging onions help stabilize the number of onions sent by the sender to the recipient. For example, if the adversary drops either onion O^0 or O^1 before it reaches the intermediary point I_{T+1}, the surviving onion will still be able to proceed. Whether the adversary drops neither or just one of them, the recipient R ultimately receives a single onion.

Of course, the adversary can drop both onions. To ensure that R still receives an onion, we need sufficiently many (some 2^h) merging onions. This can be done by organizing their routing paths in the structure of a binary tree (with height h) where nodes represent intermediary parties where onions merge and the edges represent independently random paths through the network (see Figure 7).

5.2 Anonymity in Synchronous Setting: $\Pi_\bullet$

We first present a stepping stone construction that is anonymous in the synchronous setting, i.e. time progresses in rounds and every message sent in a round is delivered instantaneously without loss.

Our protocol, $\Pi_\bullet$ (pronounced "Pi-SEED" and meaning Protocol for the Synchronous Environment with Efficient Diagnostics) builds on a secure onion encryption scheme $\mathsf{OES} = (\mathsf{KeyGen}, \mathsf{FormOnion}, \mathsf{PeelOnion})$, such as the construction by Camenisch and Lysyanskaya [CL05]. The execution of the protocol consists of h epochs, where each epoch contains T rounds.[6] This results in a total number of rounds given by $l = hT$. Each epoch concludes with a merging round, during which pairs of merging onions converge. The set of merging rounds is $\mathcal{E} = \{T, 2T, \ldots, hT\}$.

Additionally, during the first epoch, each party sends a number of looping onions, to ensure the adversary is not dropping all their onions upfront.

Similar to its predecessors, $\Pi_\bullet$ has two phases: the *onion-forming phase*, during which all the onions are created, and the *execution phase*, during which the formed onions follow their designated routing paths.

5.2.1 Forming Onions.

During the onion-forming phase, each participant P forms three kinds of onions: x looping onions, (expected) y checkpoint onions, and z message-bearing merging onions. We explain what these onions are and how they are formed below.

Looping onions. Looping onions are onions sent by a party P to itself, and they arrive at the end of the first epoch. They are used as a diagnostic tool to ensure the party's onions were not dropped upfront. Since a single dropped onion does not necessarily indicate adversarial activity, each party P will create x looping onions; if more than ρ_x fraction of these onions are missing, P aborts the protocol (see Section 5.2.2).

Looping onions help a single party detect if their onions were dropped upfront, unlike checkpoint onions, which only reveal information about the network in aggregate. Importantly, adding looping

[6] The parameter h will turn out to be the logarithm of the number of merging onions plus one, while T is polylog in the security parameter.

onions does not have much effect on the protocol's onion cost or round complexity, as those are dominated by message-bearing onions.

Structurally, a looping onion is an onion that follows a random route and ends back at the sender after T rounds. To form x looping onions, P first picks a random message m and random intermediary locations $I_1^1, I_2^1, \ldots, I_{T-1}^1, I_1^2, I_2^2, \ldots, I_{T-1}^2, \ldots, I_1^x, I_2^x, \ldots, I_{T-1}^x$, and generates the jth looping onion by running $\mathsf{FormOnion}$ on message loop, the routing path $(I_1^j, I_2^j, \ldots, I_{T-1}^j, P)$, the public keys associated with the parties on this path, and empty metadata.

Diagnostic onions. Recall from Section 5.1 that *checkpoint onions* are exchanged between two parties P_i and P_j in round r depending on the output of a pseudorandom function $F(\cdot, \cdot)$. As we want each party to be expecting y checkpoint onions over all the rounds, F needs to be chosen so that the probability that P_i forms a checkpoint onion for a random pair (P_j, r) is $\frac{y}{n \cdot hT}$.

To form the checkpoint onions, P_i computes $f = b(F(\mathsf{sk}(P_i, P_j), r||0))$ for every other party P_j and round r, (where b is a binary predicate function and $\mathsf{sk}(P_i, P_j)$ is the shared secret key[7] for parties P_i and P_j). If the result is 1, then P_i forms a checkpoint onion for P_j (and vice versa) containing the nonce $w = F(\mathsf{sk}(P_i, P_j), r||1)$. They do this by calling $\mathsf{FormOnion}$ on the empty message, a randomly chosen routing path with P_j in the rth position, and a sequence of metadata that is empty except for $y_r = w$.

As P_i knows how many checkpoint onions to expect in a given round, it can determine if more than the fraction ρ_y is missing. Should too many checkpoints be missing, they abort the protocol by ceasing to process any onions other than abort messages.

Message-bearing onions. Let $\sigma = (m, R)$ be the input for party P; that is, m is the message that P aims to deliver to their intended recipient, who is R.

P forms a set of $z = 2^a$ merging onions for some integer $a > 0$, each containing the message m for the recipient R. The routing paths for the set of merging onions are organized like a binary tree, where each edge represents a subpath of length T in the routing path of an onion and the nodes represent the meeting points where the onions meet and merge.

See Figure 7 for a visual representation of the different onions' routing paths.

5.2.2 Routing Onions.

At the start of the protocol, party P initializes its round counter t to 0 and releases its sets of looping, diagnostic, and message-bearing onions. Then, P waits to receive onions from other parties.

At the end of each round t, P peels each onion received during that round by calling the processing algorithm $\mathsf{PeelOnion}$. If the layer key has been seen before (indicating a replayed onion), P drops that onion. If P is the intended recipient of the onion, it receives the embedded message. P then counts how many of the expected checkpoint onions are missing. If the fraction of missing checkpoints exceeds ρ_y, P_i ceases processing (without sending an abort message). Otherwise, P adds the processed onion to its outbox.

If the current round marks the end of the first epoch (when $t = T$), each party should receive its looping onions, which it will identify through the message. P_i counts the number of missing looping onions. If the fraction of missing looping onions exceeds ρ_x, P_i aborts the protocol and alerts the other parties.

[7]The secret key $\mathsf{sk}(P_i, P_j)$ is the Diffie-Hellman value derived from the public keys in the PKI; no exchange is needed during the protocol. This value is not used for encryption but only as a seed to a PRF that determines whether parties P_i and P_j should exchange diagnostic onions.

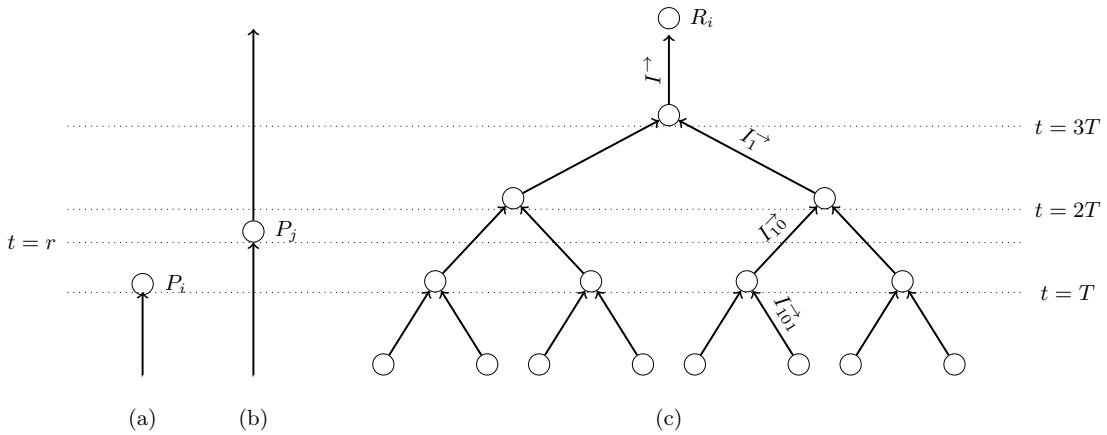

Figure 7: (a) The routing path of a looping onion spans only the first epoch and ends at P_i, whereas (b) the routing path of a diagnostic onion spans the entire execution and has P_j for the rth hop. (c) The routing paths for the message-bearing onions are structured like a binary tree, where each path, e.g., $(I^{\rightarrow}_{101}, I^{\rightarrow}_{10}, I^{\rightarrow}_{1}, I^{\rightarrow})$, ends with R_i and corresponds to an onion. In the example with $z = 2^3$ message-bearing onions, this results in 4 epochs.

During merging rounds (when $t \in \mathcal{E}$), it is expected that some onions will be paired with their counterparts and merge. If two onions arrive with the same nonce, P_i drops one of the onions and adds the other to the outbox.

Assuming no attacks are detected, the party releases its outbox (in random order), increments the round counter t, and waits to receive onions again.

Importantly, if at any point P receives an abort message from another party, or wishes to abort itself, it identifies a polylogarithmic number of random peers. It notifies that set and then ceases to process onions and begins sending only abort messages. See Figure 8 for a summary of Protocol $\Pi_\bullet$.

Forming Onions (P_i, m_i, R_i)

1: Initialize $(\mathcal{O}_x, \mathcal{O}_y, \mathcal{O}_z) \leftarrow \emptyset$.
2: **Looping.** For $j \in [x]$:
  i: Sample $\vec{I}^j = (I^j_1, \ldots, I^j_{T-1})$.
  ii: $(O^j_1, \ldots) \leftarrow \mathsf{FO}(\mathsf{loop}, (\vec{I}^j, P_i), \ldots)$.
  iii: $\mathcal{O}_x \leftarrow \mathcal{O}_x \cup \{O^j_1\}$.
3: **Diagnostic.** For $j \in [n], r \in [l]$:
  i: If $b(F(\mathsf{sk}(P_i, P_j), r \| 0)) \neq 1$, break.
  ii: Sample $\vec{Q}^{j,r}$ s.t. $Q^{j,r}_r = P_j$.
  iii: Set $\vec{y}^{j,r}[r] \leftarrow F(\mathsf{sk}(P_i, P_j), r \| 1)$.
  iv: $(O^{j,r}_1, \ldots) \leftarrow \mathsf{FO}(\perp, \vec{Q}^{j,r}, \vec{y}^{j,r}, \ldots)$.
  v: $\mathcal{O}_y \leftarrow \mathcal{O}_y \cup \{O^{j,r}_1\}$.
4: **Messages.**
  i: Build tree (V, E) with 2^a leaves.
  ii: For $e \in E$, sample $(Q^e_1, \ldots, Q^e_{T-1})$.
  iii: For $v \in V$, sample (P^v, w^v).
  iv: For each leaf ℓ, set $\vec{Q}^\ell, \vec{y}^\ell$ to parties/nonces on path $(\ell \rightarrow \text{root})$.
  v: $(O^\ell_1, \ldots) \leftarrow \mathsf{FO}(m_i, \vec{Q}^\ell, \vec{y}^\ell, \ldots)$.
  vi: $\mathcal{O}_z \leftarrow \mathcal{O}_z \cup \{O^\ell_1\}$.

Routing Onions $(\mathsf{sk}(P_i))$

1: Set $t \leftarrow 0$; release $\mathcal{O}_x, \mathcal{O}_y, \mathcal{O}_z$.
2: **Per round t.** For each received onion O:
  i: $(O', \ldots) \leftarrow \mathsf{PO}(\mathsf{sk}(P_i), O)$.
  ii: If nonce k was seen before, drop O.
  iii: If recipient, output $O' = m$.
  iv: If missing proportion of checkpoints $> \rho_y$, halt.
  v: Else add O' to buffer $\mathcal{B}$.
3: **Epoch end.** If $t = T$ and missing proportion of looping onions $> \rho_x$, abort.
4: **Merging.** If $t \in \mathcal{E}$, then for equal nonces, drop one onion.
5: **Abort.** If abort is sent or received, notify peers and cease processing.
6: **Advance.** If no abort: shuffle and release $\mathcal{B}$; reset $\mathcal{B} \leftarrow \emptyset$; increment t.

Figure 8: Summary of onion formation and routing executed by party P_i in $\Pi_\bullet$. FO and PO denote onion formation and processing.

Remark 4. *Note that in* $\Pi_\bullet$ *we can still achieve anonymity while forming diagnostic onions only to be verified at the end of each epoch. This would reduce the number of checkpoints onions by a factor of* T*. While this would work in the synchronous setting, we rely on the checkpoint onions to also handle timing in the asynchronous protocol in Section 5.3. Therefore, we chose to describe* $\Pi_\bullet$ *with diagnostic onions being verified in every round.*

5.2.3 Analysis of $\Pi_\bullet$.

An efficient implementation of $\Pi_\bullet$ with a polylogarithmic number (in the security parameter) of looping onions, diagnostic onions, and message-bearing onions, and a polylogarithmic (in the security parameter) round complexity is fully anonymous. More specifically, we can show that:

Theorem 2. *Protocol* $\Pi_\bullet$ *is anonymous from an active adversary who corrupts up to a constant fraction* $\kappa < 1$ *of the parties in the synchronous communication model whenever* $\rho_x = \frac{\rho_y}{(1-\kappa)^2} = \frac{1}{3}$*, and* $x = \frac{y}{hT} = \sqrt{z} = T = \Theta(\log^\epsilon \lambda)$ *for any constant* $\epsilon > 1$*, where* λ *denotes the security parameter.*

We give only a proof sketch here; the full proof appears in Appendix D. The argument is organized around two main lemmas:

1. **First lemma.** If the adversary drops too many of any party's onions early in the execution (e.g., more than one-third before the end of the first epoch), then the honest parties detect this behavior and abort the protocol with overwhelming probability. Consequently, conditioned on the protocol not aborting, with overwhelming probability at least one half of each honest party's onions survive to the second epoch (formalized in Lemma 1 below).
2. **Second lemma.** Assuming that at least one half of each honest party's onions survive the first epoch, the protocol achieves anonymity. This follows from sufficient mixing of onions and the "equalization" effect of merging onions by the end of the protocol. We refer the reader to Appendix D and the original proof in [ALU21] for the full argument.

Protocol $\Pi_\bullet$ can be viewed as a modification of $\Pi_{\bowtie}$ in which *looping onions*, rather than a butterfly overlay, are used to enforce the survivability guarantee of the first lemma. Accordingly, the anonymity proof for $\Pi_\bullet$ follows that of the prior work $\Pi_{\bowtie}$, with the sole difference being the proof of Lemma 1, which we provide below.

Lemma 1. *Let* P_j *be an honest party. If the fraction of looping onions missing for* P_j *is at most* $\rho_x = \frac{1}{3}$*, then with overwhelming probability* $\alpha_j \leq \frac{1}{2}$*.*

of Lemma 1. The lemma follows from a standard Azuma-Hoeffding argument.

Let Z_i denote the expected number of dropped looping onions conditioned on the first i onions dropped by the adversary during the first epoch. Then $|Z_{i+1} - Z_i| \leq 1$, since dropping one additional onion can increase the total number of dropped looping onions by at most one. By the Azuma-Hoeffding inequality, the actual number of dropped looping onions is tightly concentrated around its expectation; see Appendix D.2.

Therefore, if at most a $\rho_x = \frac{1}{3}$ fraction of looping onions are missing in expectation, then with overwhelming probability no more than half of the message-bearing onions are lost, i.e., $\alpha_j \leq \frac{1}{2}$. □

Remark 5. *Recall that* λ *denotes the security parameter.* $\Pi_\bullet$ *achieves anonymity with* $\log^\epsilon \lambda$ *rounds per epoch and* $h = \log(z) = \Theta(\mathsf{loglog}\,\lambda)$ *epochs; that is with round complexity* $\Theta(\log^\epsilon \lambda \cdot \mathsf{loglog}\,\lambda)$*. The onion cost per party (i.e., the number of message packets transmitted per message in the input vector) is the total number of onions formed,* $x + y + z = \Theta\big(\log^{2\epsilon} \lambda\big)$*, times the round complexity; thus, it is* $\Theta\big(\log^{3\epsilon} \lambda \cdot \mathsf{loglog}\,\lambda\big)$*. Compared with previous protocols,* $\Pi_\bullet$ *requires significantly fewer onions than* Π_Δ*, which has an onion cost per party of* $\omega(\sqrt{n})$*, where* n *represents the number of participants, and is faster than* $\Pi_{\bowtie}$ *by an additive factor of* $\omega(\log n \cdot \log^{2\epsilon} \lambda)$ *rounds.*

5.3 Anonymity in Asynchronous Setting: $\Pi_{\check{o}}$

In this section of the paper, we introduce our onion routing protocol, which we call the Asynchronous Protocol with Poly-Logarithmic Epochs (Pi-APPLE, pronounced "Pi(NE)-APPLE" and denoted as $\Pi_{\check{o}}$). This protocol guarantees anonymity in the standard asynchronous communication setting, where every transmitted onion can be delayed by the adversary who has complete control over the message schedule, in addition to corrupting a constant fraction ($< \frac{1}{2}$) of the nodes.

Anonymity protocols in asynchronous settings face two main challenges: there is no shared "global clock," and an adversary can drop or arbitrarily delay onions. Without a global clock, parties cannot know when a round ends or when to start processing, and they do not know in advance how many onions will arrive, some of which may be delayed by the network or adversary.

However, each party knows the expected number of checkpoint onions per round. Since this number is polylogarithmic in the security parameter, receiving a sufficient fraction of them indicates that a party has likely received enough onions to proceed. A practical strategy is to wait for a constant fraction of diagnostic onions, process them as "on time," and treat later arrivals as "late."

5.3.1 Changes from $\Pi_{\bullet}$.

In the synchronous setting, looping onions allow each sender to know if their onions were under attack during the first epoch (before mixing with the other onions). If the adversary drops too many of Alice's onions upfront, then Alice won't receive enough looping onions and would be able to alert the other parties to terminate the protocol via the abort messages.

In the asynchronous setting, this strategy doesn't work since the adversary can delay abort messages until all other messages go through. While looping onions still enable senders to know if they are under attack, abort messages are no longer a reliable method for signaling to the other parties that something went wrong.

To address this, in $\Pi_{\check{o}}$, we replace looping onions with a different set of onions called *signaling onions*. Like diagnostic onions, these are also checkpoint onions. Since we can't reliably deliver an abort message, we will instead have parties *expect* to receive these signaling onions and stop processing if more than ρ_x fraction of onions from any of its partners are missing. No more signaling onions will be released by this processing party, causing a cascade effect across subsequent epochs. We explain how these are formed in Section 5.3.2 and how they're routed in Section 5.3.3.

Another difference between $\Pi_{\bullet}$ and $\Pi_{\check{o}}$ is the switch from standard onion encryption to our new Peony Onion Encryption Scheme (POES) from Section 4. For $\Pi_{\check{o}}$, we need l_1 mixers and l_2 gatekeepers such that $l_1 + l_2 = T$ where T is the number of rounds in the epoch. We additionally want to check the bruising at the end of each epoch and discard any onions that are too bruised before they can merge. This ensures that every onion that survives the epoch was sufficiently mixed. POES has a mechanism for resetting the bruise count which allows each sufficiently mixed onion to begin the next epoch with no existing bruises. Finally, since POES hides the length of the successive path, each of the onions we use in our protocol is indistinguishable from the other.

5.3.2 Forming Onions.

In $\Pi_{\check{o}}$, each party forms three types of onions using POES, with bruising threshold $d = \theta l_1$ where $\theta < 1$ is any constant strictly less than one: x signaling checkpoint onions (rather than looping onions), y diagnostic checkpoint onions, and z message-bearing merging onions. The processes for the diagnostic checkpoint onions and the message-bearing merging onions are exactly as in $\Pi_{\bullet}$. We explain how to form the signaling checkpoint onions below.

Each party P_i is associated with a set of polylog number of *signaling partners*, $\mathcal{P}_{i,t}$, per epoch t. The sets of signaling partners can be determined using a PRF, similar to the approach used for diagnostic onion partners.

For the merging round of epoch t, denoted $r_t \in \mathcal{E}$ and every $P_j \in \mathcal{P}_{i,t}$, the party P_i forms a polylogarithmic number of signaling onions, expected by P_j in the final round of the epoch.

To form these onions, P_i computes $f_2 = b(F_2(\mathsf{sk}(P_i, P_j), t||0))$ (where b is a binary predicate function, F_2 is a PRF, and $\mathsf{sk}(P_i, P_j)$ is the shared secret key for parties P_i and P_j), for every party P_j in the protocol, for each epoch t. If the output of f_2 is 1, then $P_j \in \mathcal{P}_{i,t}$; P_i forms a polylogarithmic number of checkpoint onions for each $P_j \in \mathcal{P}_{i,t}$ containing the nonce $w = F_2(\mathsf{sk}(P_i, P_j), t||1)$ in the Tth position, where T is the number of rounds in an epoch. They do this by calling $\mathsf{FormOnion}$ on the empty message, a randomly chosen routing path with P_j in the Tth position, and a sequence of metadata that is empty except for the Tth metadata set to w.

Note: Signaling onions only last one epoch (T rounds), with the next set being released at the start of the next epoch. I.e., the signaling partners in $\mathcal{P}_{i,t}$ will be expecting to see these onions at the last round of the epoch but those onions will not continue routing.

5.3.3 Routing Onions.

At the beginning of the protocol, each party P_i initializes its round counter to 0 and releases its set of signaling onions for the first epoch, along with its diagnostic and message-bearing onions. Then P_i waits to receive onions from other parties.

Each party P_i peels onions as they are received using the function $\mathsf{PeelOnion}$. If the layer key (the symmetric key embedded in the header) has been seen before, P_i drops the onion. If P_i is the intended recipient of the onion, the embedded message is returned. If the received onion is for the current round (or future round), the processed onion is added to the outbox for that round. If the received onion is for a prior round, P_i bruises it with a call to $\mathsf{BruiseOnion}$ and releases it immediately. Once (1-ρ_y) fraction of the diagnostic onions are received for the current round, the party advances their round counter and releases their outbox for the round.

The last round of every epoch ($r_t \in \mathcal{E}$) has several functions. Firstly, merging onions converge at designated intermediate nodes to merge; when two onions arrive with the same nonce, P_i drops one of the onions and adds the other to the outbox. Since bruise count is reset at the end of each epoch, it doesn't matter which onion is forwarded. A chronically late, unmixed onion would have been dropped before it met its partner at the end of the epoch.

These rounds are also where P_i receives its signaling onions. If a sufficient number of the signaling onions was received from each partner for that epoch, P_i will release its set of signaling onions for the next epoch and continue processing onions. Otherwise, it will stop processing all onions and will fail to release the next set of signaling onions, triggering an abort cascade.

Finally, the bruise counter is also reset at the last round of each epoch, ensuring that sufficiently mixed onions can begin the next epoch with a clean slate.

5.3.4 Analysis of $\Pi_{\check{o}}$.

In the asynchronous setting, an adversary may not only drop an onion but also delay it. However, as long as the onion shuffles a constant fraction of the time during an epoch, it will mix with other onions in the same epoch. Onions that do not mix sufficiently are dropped. Therefore, the analysis for the protocol $\Pi_{\check{o}}$ remains essentially the same as that for $\Pi_{\bullet}$.

The only structural difference between the two protocols is that $\Pi_{\bullet}$ uses looping onions, whereas $\Pi_{\check{o}}$ uses signaling onions. If the adversary drops too many of a sender's onions upfront, then, w.o.p.,

everyone will be alerted by the midway point in either protocol. In $\Pi_\bullet$, this happens because the number of parties receiving abort messages rapidly increases every round [ALU21]. Similar analysis shows that the number of parties that stop receiving signaling onions would also increase in the same manner.

Thus, we conclude:

Theorem 3. *Protocol Π_δ is anonymous from an active adversary who corrupts up to a constant fraction $\kappa < \frac{1}{2}$ of the parties in the asynchronous communication model whenever $\rho_x = \frac{\rho_y}{(1-\kappa)^2} = \frac{1}{3}$, and $x = \frac{y}{hT} = \sqrt{z} = T = \Theta(\log^\epsilon \lambda)$ for any constant $\epsilon > 1$, where λ denotes the security parameter.*

6 Conclusion and Future Work

We present the first efficient onion routing protocol achieving full anonymity in asynchronous networks. Our contributions are twofold: (1) a bruisable onion construction that hides position on the path, and (2) a protocol that leverages it to attain full anonymity, beyond differential privacy. We also give a new synchronous protocol improving prior work in communication and round complexity.

Acknowledgments

This work was partially supported by NSF grants CCF-2312243, CCF-2312242, CCF-2212233, and CCF-2107187, MITRE's Independent Research and Development Program, an Amazon Research Award, a Google CyberNYC grant, and Google's Cyber NYC Institutional Research Program through Columbia's School of Engineering and Applied Sciences.

The authors thank Anna Lysyanskaya, Gabby Raymond, and Chris Niessen for helpful comments.

License

Approved for Public Release under CC-BY-NC-SA. Distribution unlimited. Case: 25-1521.

References

[AL21] Megumi Ando and Anna Lysyanskaya. Cryptographic shallots: A formal treatment of repliable onion encryption. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 188–221. Springer, Cham, November 2021.

[ALU18] 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.

[ALU21] Megumi Ando, Anna Lysyanskaya, and Eli Upfal. On the complexity of anonymous communication through public networks. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[ALU24] Megumi Ando, Anna Lysyanskaya, and Eli Upfal. Bruisable onions: Anonymous communication in the asynchronous model. In *Theory of Cryptography Conference*, pages 476–507. Springer, 2024.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[Cha81] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[Chr20] Miranda Christ. New lower bounds on the complexity of provably anonymous onion routing. *Undergraduate honors thesis, Brown University, Providence, RI*, 2912, 2020.

[CL05] 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, Berlin, Heidelberg, August 2005.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing privacy enhancing technologies: international workshop on design issues in anonymity and unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*, pages 46–66. Springer, 2001.

[DMS04] 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.

[KCDF17] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 406–422. ACM, 2017. URL `https://doi.org/10.1145/3132747.3132755`.

[KCGDF17] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 406–422, 2017.

[KHRS21] Christiane Kuhn, Dennis Hofheinz, Andy Rupp, and Thorsten Strufe. Onion routing with replies. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part II*, volume 13091 of *LNCS*, pages 573–604. Springer, Cham, December 2021.

[KLD20] Albert Kwon, David Lu, and Srinivas Devadas. {XRD}: Scalable messaging system with cryptographic privacy. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 759–776, 2020.

[LMD10] Karsten Loesing, Steven J. Murdoch, and Roger Dingledine. A case study on measuring statistical data in the tor anonymity network. In Radu Sion, Reza Curtmola, Sven Dietrich, Aggelos Kiayias, Josep M. Miret, Kazue Sako, and Francesc Sebé, editors, *FC 2010 Workshops*, volume 6054 of *LNCS*, pages 203–215. Springer, Berlin, Heidelberg, January 2010.

[LSSD23] Simon Langowski, Sacha Servan-Schreiber, and Srinivas Devadas. Trellis: Robust and scalable metadata-private anonymous broadcast. In *NDSS 2023*. The Internet Society, February 2023.

[MD05] Steven J. Murdoch and George Danezis. Low-cost traffic analysis of tor. In *2005 IEEE Symposium on Security and Privacy*, pages 183–195. IEEE Computer Society Press, May 2005.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge university press, 2005.

[PHE+17] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *26th usenix security symposium (usenix security 17)*, pages 1199–1216, 2017.

[Rop21] Lucas Ropek. Someone is running hundreds of malicious servers on the Tor network and might be de-anonymizing users. https://tinyurl.com/2p999e8e, December 2021.

[RR98] Michael K Reiter and Aviel D Rubin. Crowds: Anonymity for web transactions. *ACM transactions on information and system security (TISSEC)*, 1(1):66–92, 1998.

[SEV+15] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. RAPTOR: Routing Attacks on Privacy in Tor. In *USENIX Security Symposium*, pages 271–286, 2015.

[TGL+17] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440, 2017.

[vdHLZZ15] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 137–152. ACM, 2015. URL https://doi.org/10.1145/2815400.2815417.

A Ideal Functionality for Onion Encryption Schemes

There are a number of works that describe ideal functionality for onion encryption schemes, beginning with [CL05]. This was followed by "Onion Routing with Replies" [KHRS21] and "Cryptographic Shallots: A Formal Treatment of Repliable Onion Encryption" [AL21]. For ease of understanding, we start by providing a full definition and security proof in the [AL21] style for the standard onion encryption scheme given by [CL05]. We then explain how to modify it to develop the ideal functionality for our bruisable encryption scheme $POES$.

We begin by highlighting the key changes from [AL21] in brief before proceeding to the full description.

1. The definitions are much simplified because we omit repliable onions. This allows us to condense $\mathcal{F}_{OES}$ and our description of the simulator $\mathcal{S}$ to fewer and simpler algorithms.
2. We modify the nature of IdealFormOnion and IdealProcOnion in $\mathcal{F}_{OES}$ to be iterative. In [AL21], upon receiving a "form onion" request, $\mathcal{F}_{ROES}$ uses IdealFormOnion to segment the routing path, form individual onions for each segment, and link them in OnionDict. This is all done upfront. Instead of this, our ideal functionality $\mathcal{F}_{OES}$ simply forms an onion for the first segment of the routing path, and stores the remaining routing path and message in the data structure PathLookup. Any subsequent call to process the onion looks up the remaining path, forms an onion on the next segment, and saves the resulting onion and remaining path to PathLookup.

A.1 Ideal Functionality and $\mathcal{F}_{OES}$

What properties do we wish to capture in our ideal functionality? Honestly formed onions mix at honest nodes, regardless of distance to their respective recipients. From this follows that onions can only mix if their layers are unrelated to each other.

We develop our ideal functionality $\mathcal{F}_{OES}$ accordingly. Let a routing path $\vec{P}$ be composed of segments $\vec{s} = (s_0, s_1, ..., s_n)$ where each segment is a non-overlapping series of adversarial parties, usually ending in an honest party. (This is true except for the final segment s_n, which ends with the (possibly dishonest) recipient.) For example, the path $\vec{P} = (p_0, P_1, P_2, p_3, r)$ is composed of segments $\vec{P} = ((p_0\, P_1), (P_2), (p_3, r))$. (We denote honest parties in capital P_i and dishonest parties in lowercase p_i.) Note that if there are two consecutive honest parties P_1, P_2 in the path, (P_2) forms its own segment. Onion layers formed from any two segments of the routing path should be computationally unrelated to each other.

The ideal functionality uses segmentation as follows: when $\mathcal{F}_{OES}$ is asked to form an onion, it forms an onion *only* on the first segment of the path. It stores the remaining path in the data structure PathLookup, under the onion and party (P, O) from which it expects to receive a processing request. When the next honest party asks $\mathcal{F}_{OES}$ to process their honestly formed onion, the ideal functionality obtains the remaining path from PathLookup and forms an onion on the next segment, again saving the resulting path under a new tuple (P', O'). (If the request is not stored in PathLookup, then $\mathcal{F}_{OES}$ simply runs Σ's original algorithm ProcOnion, which it obtains from the adversary $\mathcal{A}$.)

A.1.1 Setting up.

$\mathcal{F}_{OES}$ handles requests made by the environment $\mathcal{Z}$ on behalf of honest parties in the protocol to form or process onions. The static setting for a fixed set of participants and public key infrastructure requires a separate setup. Additionally, $\mathcal{F}_{OES}$ only runs for a polynomial number of steps (specified in public parameters pp) and can time out while running the algorithms given by $\mathcal{A}$.

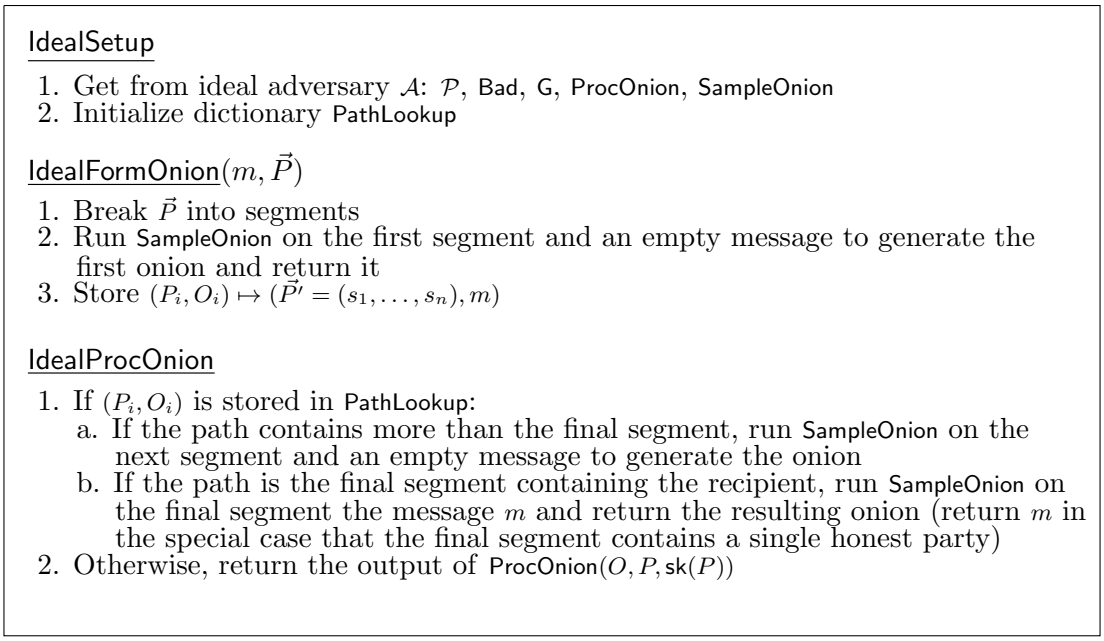

Figure 9: Summary of the Ideal Functionality $\mathcal{F}_{OES}$

During setup, $\mathcal{F}_{OES}$ gets the following algorithms from the ideal adversary $\mathcal{A}$.

1. The set of participants $\mathcal{P}$
2. The set of corrupt parties Bad in $\mathcal{P}$
3. The onion encryption scheme's algorithms
   (a) G *used to generate the honest parties' keys*
   (b) ProcOnion *used to process onions formed by corrupt parties that are not stored by* $\mathcal{F}_{OES}$
4. The p.p.t. algorithm SampleOnion(m, s) that takes a message m and segment s and outputs a tuple of onion layers (o_i, O_j). o_i is for the first party in the segment, and is peelable to O_j, intended for the final (honest) party in the segment. We omit to mention the security parameter 1^λ, public parameters pp, upper bound on the number of layers N and public keys $\mathsf{pk}(s)$ that are also inputs to the algorithm for the sake of readability. $\mathcal{F}_{OES}$ fails if SampleOnion ever samples a repeating key.

The session id sid is specific to all the parameters that the setup creates. $\mathcal{F}_{OES}^{\mathsf{sid}}$ is the session of $\mathcal{F}_{OES}$ that has been set up with this sid.

$\mathcal{F}_{OES}^{\mathsf{sid}}$ generates a public key pair for every honest party $P \in \mathcal{P} \backslash \mathsf{Bad}$ using G and sends the keys to their respective parties. (If working within the global PKI framework, each party then relays his/her key to the global bulletin board functionality [CSV16].)

$\mathcal{F}_{OES}^{\mathsf{sid}}$ also creates the (initially empty) dictionary PathLookup, which supports:

- A method $\mathsf{put}((P, O), (\vec{P'}, m))$ which stores under the label (P, O), a routing path $\vec{P'}$ and message m.
- A method $\mathsf{get}(P, O)$ which looks up the entry $(\vec{P'}, m)$ corresponding to the label (P, O).

This data structure is stored internally in $\mathcal{F}_{OES}^{\mathsf{sid}}$ and is only accessible by $\mathcal{F}_{OES}^{\mathsf{sid}}$.

A.1.2 Forming an onion.

After setting up, the environment $\mathcal{Z}$ can instruct $\mathcal{F}_{OES}$ to form an onion on behalf of an honest party, under a session id sid, message m, and routing path $\vec{P}$.

The goal of the ideal functionality is to create and maintain the information required to handle

both form and process onion requests. The onions should look as though they could have been produced by the OES algorithms FormOnion and ProcOnion. They should not reveal any information to $\mathcal{A}$. By this we mean that $\mathcal{A}$ cannot learn anything of the routing path beyond the honest party with whom the segment ends. Additionally, the onion should be formed information-theoretically independent of any message. Practically, we do so by using an empty message (at least until the onion is reaching an adversarial recipient).

To ensure these properties, the ideal functionality obtains the first segment s_0 of the path $\vec{P}$, where each segment is a series of adversarial parties that usually ends with an honest party (except for the final segment which can culminate with an adversarial recipient).

Then, $\mathcal{F}_{OES}^{\mathsf{sid}}$ samples onions (o_i, O_j) by running the algorithm $\mathsf{SampleOnion}(\perp, s_0)$, where $\perp$ is the empty message and s_0 is the first segment in the path $\vec{P}$. Again, o_i is the onion intended for the first (adversarial) party in the segment, and is peelable to O_j, intended for the last (honest) party in the segment. In the case that there are no honest parties in the path $\vec{P}$, SampleOnion will use the message m and path $s_0 = \vec{P}$.

A.1.3 Processing an onion.

After setup, the environment $\mathcal{Z}$ can instruct an honest party P to process an onion O for the session id sid. To process the onion, party P forwards the instruction to the ideal functionality $\mathcal{F}_{OES}^{\mathsf{sid}}$. There are two possible scenarios.

- There is an entry under the label (P, O) in PathLookup. Then, the ideal functionality takes the next segment of the path $\vec{P}'$. If this is the final segment, $\mathcal{F}_{OES}^{\mathsf{sid}}$ samples an onion using SampleOnion on the real message m and segment s_n. If it is not the final segment, $\mathcal{F}_{OES}^{\mathsf{sid}}$ samples an onion using the empty message $\perp$, and the next segment s. It then forwards the resulting onion o to P.
- There is no entry under the label (P, O) in PathLookup. Then, the ideal functionality peels the onion using the ProcOnion algorithm it obtained from the ideal adversary $\mathcal{A}$. It then forwards the resulting onion o to P.

Upon receiving the response o from $\mathcal{F}_{OES}^{\mathsf{sid}}$, P outputs the session id sid and the onion o.

Remark 6 (On the assumption that keys are consistent with PKI)**.** *In describing our ideal functionality, we assume that the onions are formed using the correct keys. However, an adversarial environment $\mathcal{Z}$ can instruct an honest party to form onions without this dictum. We can subvert this by having PathLookup store every onion layer for an honest party, until it reaches a layer with a mismatched key. To keep the exposition clean, we continue to assume that inputs are well behaved, i.e., router names are valid and keys are correct.*

Remark 7 (On replayed onions)**.** *As noted above 2, the environment can ask honest parties to process an onion that they have already processed. Our functionality does not protect from this attack (neither does the original CL05 construction, or those succeeding it) but can be addressed by directing parties to ignore repeat processing requests.*

A.2 Game-based Security Definition for OES

OnSH is parameterized by the security parameter 1^λ, the adversary $\mathcal{A}$, and the onion encryption scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{FormOnion}, \mathsf{ProcOnion})$.

The challenger controls an honest intermediary in the path. The adversary controls all other intermediaries, as well as the recipient R (the corruptions are modeled as non-adaptive).

- **Setup:** The adversary $\mathcal{A}$ and the challenger $\mathcal{C}$ set up the parties' keys.

1. The adversary $\mathcal{A}$ sends the name of the honest intermediary I and the adversarial parties Bad and their public keys $\mathsf{pk}(\mathsf{Bad})$ to the challenger $\mathcal{C}$.
2. $\mathcal{C}$ generates a key pair $(\mathsf{pk}(I), \mathsf{sk}(I)) \leftarrow \mathsf{KeyGen}(1^\lambda, I)$ and sends $\mathsf{pk}(I)$ to $\mathcal{A}$.

- **First query phase:** $\mathcal{A}$ is allowed to submit queries to have an onion processed by an honest party.

3. $\mathcal{A}$ can direct the honest party to process an arbitrary onion by submitting queries to process onion O on behalf of I. In this case, $\mathcal{C}$ will respond with the output of $\mathsf{ProcOnion}(\mathsf{sk}(I), O)$. $\mathcal{A}$ can submit a polynomial number of such queries.

- **Challenge phase:** $\mathcal{A}$ picks the parameters of the challenge onion and $\mathcal{C}$ responds with the challenge onion.

4. $\mathcal{A}$ sends to $\mathcal{C}$: the message m and the routing path $\vec{P}$, which contains I in the location h i.e. $\vec{P} = (P_0, \ldots I = P_h, \ldots, P_{l-1}, R)$.
5. $\mathcal{C}$ samples bit $b \leftarrow\!\!\$\ \{0,1\}$.
   - If $b = 0$, $\vec{Q} = \vec{P}$, $c = m$.
   - If $b = 1$, $\vec{Q} = (P_0, \ldots, I = P_h)$, $c = \perp$.
6. $\mathcal{C}$ returns the first onion O_0 in the output from running $\mathsf{FormOnion}$ on the message c, the path $\vec{Q}$ and its public keys $\mathsf{pk}(\vec{Q})$ i.e. $\vec{O} \leftarrow \mathsf{FormOnion}(c, \vec{Q}, \mathsf{pk}(\vec{Q}))$.

- **Second query phase:** $\mathcal{A}$ is again allowed to submit queries to have an onion peeled or bruised by the honest party I.

7. If $b = 0$; or the request *is not* to process the onion as the honest party, the challenger processes the request by running the scheme's algorithm.
8. If $(b = 1)$ and *it is the first request of its type*, $\mathcal{C}$ runs $\mathsf{FormOnion}$ on the message m and the path after I i.e., $\vec{O}_{h \to l} \leftarrow \mathsf{FormOnion}(m, \vec{Q}', \mathsf{pk}(\vec{Q}'))$
9. If $b = 1$ and it is *not* the first request of its type, the challenger responds with an error message.

- At the end, $\mathcal{A}$ outputs a guess b' for the bit b and wins if $b' = b$. We define security as follows.

Definition 3 (Onion security)**.** *An onion encryption scheme Σ is onion-secure if there exists a negligible function $v : \mathbb{N} \mapsto \mathbb{N}$ such that every probabilistic polynomial-time adversary $\mathcal{A}$ wins $\mathsf{OnSH}(1^\lambda, \mathcal{A}, \Sigma)$ with advantage at most $v(\lambda)$, i.e., $\left|\Pr\left[\mathcal{A} \text{ wins } \mathsf{OnSH}(1^\lambda, \mathcal{A}, \Sigma\right] - \frac{1}{2}\right| \leq v(\lambda)$.*

A.3 Onion Security Implies SUC-realizability of $\mathcal{F}_{OES}$

We prove the following theorem:

Theorem 4. *If the onion encryption scheme Σ is correct [CL05] and secure (Definition 3 in Appendix A.2), then it SUC-realizes the ideal functionality $\mathcal{F}_{OES}$ (Appendix A.1).*

To do this, we must show that for any static setting (a fixed adversary $\mathcal{A}$, the 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 $v : \mathbb{N} \mapsto \mathbb{R}$ such that

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

A.3.1 Description of the simulator $\mathcal{S}$.

During setup, the ideal adversary (i.e. $\mathcal{S}$) sends to $\mathcal{F}_{OES}$ the set of parties $\mathcal{P}$, the adversarial parties Bad, and the algorithms G, $\mathsf{ProcOnion}$, and $\mathsf{SampleOnion}$.

In order for our construction to be secure, $\mathcal{S}$ must provide these items to $\mathcal{F}_{OES}$, in order to ensure that when the ideal functionality replies to the environment, input by input, the output

history looks like it was produced by running the real protocol. Additionally, in the static setting, we assume w.l.o.g. that these algorithms know which parties are honest, which are corrupt, and all relevant keys.

Send to $\mathcal{F}_{OES}$

$\mathcal{P}$, Bad, KeyGen, ProcOnion, SampleOnion

SampleOnion(m, s)

Simply runs FormOnionon segments of the path, with an empty message if the segment does not contain the recipient, and the message m if it does.

Figure 10: Summary of the Simulator $\mathcal{S}$

Sampling onions. $\mathcal{F}_{OES}^{\mathsf{sid}}$ describes the ideal functionality in the static setting. SampleOnion is very similar to the OES's FormOnion algorithm, except that it allows us (semantically) to distinguish between forming a single onion on a routing path $\vec{P}$ all at once, and forming multiple onions on consecutive segments of the path $\vec{P} = (s_0, \ldots, s_n)$.

When $\mathcal{F}_{OES}^{\mathsf{sid}}$ receives a request to form or process an honestly formed onion, it runs SampleOnion on the first segment of the corresponding path to generate the next onion, and saves the remaining path and message in PathLookup. This continues, with SampleOnion forming new onions for each segment of the path until the final segment is reached, whereupon it uses the message m and final segment s_n to form the final onion. This construction ensures that onions belonging to differnt segments are information-theoretically unrelated to each other.

The algorithm SampleOnion takes as input a message m and segment s, and outputs a tuple of onion layers (o_i, O_j). o_i is onion intended for the first party in the segment and is peelable to O_j, intended for the final (honest) party in the segment. It also takes the security parameter 1^λ, public parameters pp, upper bound on the number of layers N, and public keys $\mathsf{pk}(s)$. If the input to SampleOnion is not given as a segment s consisting of a series of adversarial parties usually ending in an honest party, the algorithm returns an error.

A.3.2 Proof of Theorem 4.

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

Proof idea. An onion encryption scheme SUC-realizes $\mathcal{F}_{OES}$ if the environment cannot distinguish whether an honest onion's evolution (the sequence of onion layers) comes from a single call to FormOnion (the real setting), or if it is produced by $\mathcal{F}_{OES}$ and formed as multiple separate onions. Using this idea, we show that onion layers formed with j calls to SampleOnion are indistinguishable from those that make $j + 1$ calls. This generalizes such that onion layers formed with a single call to FormOnion in the real proctocol are indistinguishable from those formed using multiple calls to SampleOnion in the ideal functionality.

In brief, if the honest party I controlled by the challenger $\mathcal{C}$ does not exist in the correct placement along the path, $\mathcal{B}$ guesses a random bit (see Step 5b). If I appears as the last party in the $j + 1^{th}$ segment, it is in the correct positioning. In this case the challenger gets an opportunity to form the onion for the remaining path $\vec{P}' = (s_j, \ldots, s_n)$. The challenger flips a bit. If $b = 0$, it

produces the onion $\hat{O}_j$ with the entire remainder path and correct message m. If $b = 1$, it produces the onion $\hat{O}_j$ with the truncated path s_j and no message. The challenger returns the onion, knowing it will be asked to process the onion for the honest party I (the next honest party in the path). When $\mathcal{C}$ is asked to process the onion for I, it will either peel the onion to get its successor ($b = 0$) or form a new onion using the remaining path $\vec{P}' = (s_{j+1}, \ldots, s_n)$ and correct message m ($b = 1$). It will return the result $\hat{O}_{j+1}$, which is forwarded to $\mathcal{A}$. If $\mathcal{A}$ can distinguish between case $b = 0$ (where $\mathcal{C}$'s onion was formed correctly and the total number of segments is j) and $b = 1$ (where $\mathcal{C}$'s onion was formed incorrectly, thereby segmenting the onion one extra time and causing total number of segments $j + 1$), then it breaks the OES security game. So, we conclude that $\mathsf{Hybrid}^{i,j}$ and $\mathsf{Hybrid}^{i,j+1}$ are indistinguishable.

Ultimately, we conclude that if an onion encryption scheme is secure, then it SUC-realizes $\mathcal{F}_{OES}$. See below for the formal proof.

Proof. Let $\mathsf{Ideal}(1^\lambda, \mathcal{Z})$ be the view that the environment $\mathcal{Z}$ obtains when the ideal onion routing protocol is run, i.e., when the ideal honest parties query $\mathcal{F}_{OES}$ to form and process onions. $\mathcal{F}_{OES}$ obtains the algorithms KeyGen, ProcOnion and SampleOnion from the ideal adversary (a.k.a the simulator $\mathcal{S}$ described above).

Let $\mathsf{Real}(1^\lambda, \mathcal{Z})$ be the view that the environment $\mathcal{Z}$ sees when the honest participants run Σ's algorithms to form, process, and bruise onions.

Let $\mathsf{Hybrid}^0(1^\lambda, \mathcal{Z})$ be the adversary's view in a game between the challenger and the adversary as follows. The adversary is the environment $\mathcal{Z}$ interacting with the honest participants running the real protocol. The challenger controls the honest parties and also runs, on the side, the ideal functionality $\mathcal{F}_{OES}$. When an honest party is instructed to form an onion, the challenger uses the FormOnion algorithm of the encryption scheme Σ to form onions. For all of an honest party's other queries, the challenger uses $\mathcal{F}_{OES}$ with our simulator $\mathcal{S}$. Thus, the challenger uses the ideal functionality's algorithm IdealProcOnion to process the onions. Note that in this case, IdealProcOnion will just end up calling the real algorithm ProcOnion (that $\mathcal{S}$ supplies to $\mathcal{F}_{OES}$). This can be understood more generally- if an onion was not formed by IdealFormOnion, calling IdealProcOnion to process it will do so using Σ's algorithm ProcOnion.

Claim 1: $\mathsf{Real}(1^\lambda, \mathcal{Z})$ is the same as $\mathsf{Hybrid}^0(1^\lambda, \mathcal{Z})$

Proof of Claim 1: The proof is by construction. All honest parties are running the same algorithms in both experiments. [end of proof of claim]

Let numFOs be the upper bound on the number of honest "form onion" queries from the environment. We now define $\mathsf{Hybrid}^1(1^\lambda, \mathcal{Z})$ through $\mathsf{Hybrid}^{\mathsf{numFOs}}(1^\lambda, \mathcal{Z})$.

$\mathsf{Hybrid}^1(1^\lambda, \mathcal{Z})$: To handle the first "form onion" query, the challenger runs $\mathcal{F}_{OES}$'s algorithm, IdealFormOnion. (See Figure 9.) The algorithm forms an onion on the first segment of the path, with an empty message, saving the remaining path and message in the challenger's local data structure PathLookup. To handle any subsequent "form onion" query, the challenger runs Σ's FormOnion. To process an onion, the challenger always runs $\mathcal{F}_{OES}$'s IdealProcOnion. The output of $\mathsf{Hybrid}^1(1^\lambda, \mathcal{Z})$ is the adversary's (i.e. the environment's) view. We extend this to the following.

$\mathsf{Hybrid}^k(1^\lambda, \mathcal{Z})$: To handle the kth "form onion" query, if $k \leq i$, the challenger runs IdealFormOnion. If $k > i$, the challenger runs Σ's FormOnion. To process an onion, the challenger always runs $\mathcal{F}_{OES}$'s IdealProcOnion. The output of $\mathsf{Hybrid}^1(1^\lambda, \mathcal{Z})$ is the adversary's (i.e. the environment's) view.

Claim 2: $\mathsf{Hybrid}^{\mathsf{numFOs}}(1^\lambda, \mathcal{Z})$ is identical to $\mathsf{Ideal}(1^\lambda, \mathcal{Z})$

Proof of Claim 2: The claim is true by construction. [end of proof of claim]

To complete our proof, we need to show that for each $i, 0 \leq i < \mathsf{numFOs}$, for any p.p.t. adversary $\mathcal{Z}$, the distribution $\mathsf{Hybrid}^{i}(1^{\lambda}, \mathcal{Z})$ is indistinguishable from $\mathsf{Hybrid}^{i+1}(1^{\lambda}, \mathcal{Z})$. To do this, we first define "two-dimensional" hybrids, $\mathsf{Hybrid}^{i,0}, \ldots, \mathsf{Hybrid}^{i,2N_t}$, between Hybrid^{i} and Hybrid^{i+1} such that Hybrid^{i} is equivalent to $\mathsf{Hybrid}^{i,0}$ by construction and $\mathsf{Hybrid}^{i,2N_t}$ is equivalent to Hybrid^{i+1}. (These 2-dimensional hybrids are also parameterized by the security parameter 1^{λ} and the environment $\mathcal{Z}$, but we leave these out since they are now understood by context.) Recall that N is the upper bound on the length of the routing path. Thus, $2N$ is the upper bound on the number of segments per "form onion" query.

At a high level, $\mathsf{Hybrid}^{i,j}$ is the experiment where the first i "form onion" queries, and the first j segments of the $(i+1)^{th}$ "form onion" query are "simulated" and the rest are "real." More formally we modify the ideal functionality's algorithms such that

Let ModifiedIdealFO be the algorithm that on the input of message m, routing path $\vec{P}$, and segment counter segmentCount, modifies Step 1 of IdealFormOnion (see Fig. 9) by placing a delimiter $*$ before the segmentCounter segment of the path. Ex: $\mathsf{ModifiedIdealFO}(m = \perp, \vec{P} = ((p_0, P_1), (P_2), (p_3, r)), \mathsf{segmentCounter} = 2)$ forms an onion on an empty message and the first segment (p_0, P_1) of the path, and saves the real message and delimited path in PathLookup under the next honest party's name and onion: $(P_1, O_1) \rightarrow ((P_2), *, (p_3, r)), m)$.

Let ModifiedIdealPO be the algorithm that on the input of party P and onion O, modifies Step 1a of IdealProcOnion (see Fig. 9). If ModifiedIdealPO encounters the delimiter character $*$ at the head of the path, it runs SampleOnion on the real message and remainder of the path (even if it contains multiple segments).

Intuitively, using the original algorithms IdealFormOnion and IdealProcOnion handle an individual "form onion" request by creating $|\vec{s}|$ onions- one for each segment of the path. However, forming and processing an onion using the modified algorithms means that a request to form an onion with $\mathsf{segmentCounter} = j$ will create only $j + 1$ onions. One onion is created for each of the first j segments, and one onion is created for the remainder of the path (which is peelable to an onion for the recipient).

We now provide a more formal description of the hybrids: $\mathsf{Hybrid}^{i,0}, \ldots, \mathsf{Hybrid}^{i,2N_t}$.

$\mathsf{Hybrid}^{i,1}$: The only time that the challenger in $\mathsf{Hybrid}^{i,1}$ acts differently from the challengers in Hybrid^{i} and Hybrid^{i+1} is in handling the $(i+1)^{th}$ "form onion" query.

The first i "form onion" queries are handled by $\mathcal{F}_{OES}$, as in both Hybrid^{i} and Hybrid^{i+1}.

To respond to the $(i+1)^{th}$ "form onion" query, the challenger in $\mathsf{Hybrid}^{i,1}$ runs the algorithm $\mathsf{ModifiedIdealFO}(m, \vec{P}, \mathsf{segmentCounter} = 1)$. This acts identically to IdealFormOnion, except the path stored in PathLookup has a delimiter placed before the second segment of the path. To handle any subsequent "form onion" query (i.e. after the first $i+1$ "form onion" queries have been made), the challenger runs FormOnion, just as the challengers in Hybrid^{i} and Hybrid^{i+1} would. To process an onion, the challenger runs ModifiedIdealPO. This acts identically to IdealProcOnion, except that upon seeing a delimiter at the head of the path, ModifiedIdealPO forms the onion as though the remainder of the path is the final segment (ie: using the real message, and entire remaining routing path).

We generalize this description for in $\mathsf{Hybrid}^{i,j}$ as follows. In $\mathsf{Hybrid}^{i,j}$: To handle the kth "form onion" query,

1. if $k \leq i$, the challenger runs $\mathsf{IdealFormOnion}$
2. if $k > i + 1$, the challenger runs $\mathsf{FormOnion}$
3. otherwise (if $k = i + 1$), the challenger runs $\mathsf{ModifiedIdealFO}$ with $\mathsf{segmentCounter} = j$

To process the onion, the challenger runs $\mathsf{ModifiedIdealPO}$.

Claim 3: For all $i \in [\mathsf{numFOs}]$, Hybrid^{i} and $\mathsf{Hybrid}^{i,0}$ produce identical results, and $\mathsf{Hybrid}^{i-1,2N}$ and Hybrid^{i} produce identical results.

Proof of Claim 3: The claim is true by construction. [end of proof of claim]

Claim 4: If Σ is OES-secure, then for any $i \in [\mathsf{numFOs}]$ and $j \in [2N - 1]$, $\mathsf{Hybrid}^{i,j}$ and $\mathsf{Hybrid}^{i,j+1}$ are indistinguishable.

Proof of Claim 4: The proof of Claim 4 is by reduction.

For the sake of reaching a contradiction, suppose that there exists an adversary $\mathcal{A}$ and hybrids $\mathsf{Hybrid}^{i,j}$ and $\mathsf{Hybrid}^{i,j+1}$ such that $\mathcal{A}$ can distinguish between $\mathsf{Hybrid}^{i,j}$ and $\mathsf{Hybrid}^{i,j+1}$ with non-negligible advantage. Then, we can construct a reduction $\mathcal{B}$ that can break the OES security of Σ. $\mathcal{B}$ runs $\mathcal{A}$ as a subroutine and interacts with its own challenger $\mathcal{C}$ for the OES security game, as follows:

1. As in the real/ideal/hybrid setup, the reduction $\mathcal{B}$ chooses the identities of all honest parties.
2. Then, $\mathcal{B}$ chooses an honest party, I uniformly at random. For each party P_i other than I, it generates the keys for P_i using Σ's key generation algorithm KeyGen: $(\mathsf{pk}(P_i), \mathsf{sk}(P_i)) \leftarrow \mathsf{KeyGen}(1^\lambda, p, P_i)$.
3. Next, $\mathcal{B}$ queries its challenger $\mathcal{C}$ for the public key $\mathsf{pk}(I)$.
4. Having run the above steps, $\mathcal{B}$ is done setting the stage for an execution of the onion routing protocol with the adversary $\mathcal{A}$. Next, $\mathcal{A}$ provides the public keys of all the participants under the adversarial environment's control.
5. To handle the kth "form onion" query from $\mathcal{A}$, the reduction $\mathcal{B}$ does the following:
   (a) If $k \leq i$: $\mathcal{B}$ runs $\mathsf{IdealFormOnion}$ and sends the output of $\mathsf{IdealFormOnion}$ to $\mathcal{A}$.
   (b) If $k = i + 1$: $\mathcal{B}$ first segments the path $\vec{P}$. If the honest party I does not appear in the jth segment, $\mathcal{B}$ exits the execution and outputs a random bit $b' \leftarrow \{0, 1\}$. Otherwise, $\mathcal{B}$ runs $\mathsf{ModifiedIdealFO}$ with $\mathsf{segmentCounter} = j$.
   (c) If $k > i + 1$: $\mathcal{B}$ runs $\mathsf{FormOnion}$ and sends the output of $\mathsf{FormOnion}$ to $\mathcal{A}$.
6. To handle a processing request:
   (a) If there is a delimiter at the head of the path, $\mathcal{B}$ sends the message m and remainder path $\vec{P}'$ to the challenger $\mathcal{C}$ and forwards its response (the onion $\hat{O}_j$) to $\mathcal{A}$.
   (b) If the processing party requesting a peeled onion is I *and this is the first request to* $\mathcal{B}$ *after returning* $\hat{O}_j$, $\mathcal{B}$ forwards the onion to the challenger $\mathcal{C}$, who returns the onion $\hat{O}_{j+1}$. $\mathcal{B}$ forwards $\hat{O}_{j+1}$ to $\mathcal{A}$.
   (c) If the path does not begin with the delimiter, process using $\mathsf{ModifiedIdealPO}$. (If the processing party is I, then $\mathcal{B}$ does not have the secret key to decrypt it. However, the challenger $\mathcal{C}$ allows oracle access to $\mathsf{ProcOnion}$ in the case that this is not the $j + 1^{th}$ segment.)
7. At the end of the execution, if $\mathcal{A}$ outputs $\mathsf{Hybrid}^{i,j}$, then $\mathcal{B}$ guesses $b' = 0$ (no switch); otherwise if $\mathcal{A}$ outputs $\mathsf{Hybrid}^{i,j+1}$, then $\mathcal{B}$ guesses $b' = 1$ (switch).

Now that we have described the reduction, let's analyze it. Clearly, the reduction runs in polynomial time.

Case $b = 0$. If the challenger $\mathcal{C}$ chooses $b = 0$, then $\mathcal{C}$ forms the challenge onion $\hat{O}_j$ exactly as specified by the reduction $\mathcal{B}$ with message m and path $\vec{P}'$. Thus, the replacement onion $\hat{O}_j$ is

statistically the same as if $\mathcal{B}$ had produced the onion itself, and the adversary's view, in this case, is identical to $\mathsf{Hybrid}^{i,j}$.

Case $b = 1$, $I \neq r$. If the segment j is not the final segment in the path, I is not the recipient r. In this case, $\mathcal{C}$ runs $\mathsf{SampleOnion}$ twice. First, it forms an onion $\hat{O}_j$ the segment s_j, ending in I, and an empty message. It returns $\hat{O}_j$ to $\mathcal{B}$. When it is asked to process the onion on behalf of I immediately following this, it runs $\mathsf{SampleOnion}$ again on the real message m, and remaining path $\vec{P}' = (s_{j+1}, \ldots, s_n)$, and returns the onion as $\hat{O}_{j+1}$.

This is exactly what the challenger in $\mathsf{Hybrid}^{i,j+1}$ does to generate the onion layers corresponding to the segment s_{j+1}.

Case $b = 1$, $I = R$. If the segment j is the final segment of the path, then I is the recipient. In this case, the challenger returns the message m. In this case, a switch still occurs, as the original onion was formed without a message.

Since I is chosen uniformly at random, if $\mathcal{A}$ wins with non-negligible advantage, so too does $\mathcal{B}$. [end of proof of claim]

A.4 Extending to POES

The proof of SUC-realizability we give for the OES described in CL05 is very similar to the proof of SUC-realizability for our own $POES$. We highlight how this can be done below.

The key challenge in adapting this proof to $POES$ is tracking the bruise count. This is done easily enough.

1. $\mathcal{F}_{POES}$ (the ideal functionality for $POES$) is the same as $\mathcal{F}_{OES}$ except that
   (a) The dictionary $\mathsf{PathLookup}$ stores under the label $(P_i, O_{i,j})$, the shortened path $\vec{P}'$, the real message m, and the total bruisecount so far $\mathsf{bruisecount}$.

$$(P_i, O_{i,j}) \mapsto (\vec{P}', m, \mathsf{bruisecount})$$

   (b) $\mathsf{IdealFormOnion}$ forms the onion with 0 bruises and stores each possible onion variant in $\mathsf{PathLookup}$.

$$(o_{i,0}, \vec{O_j} = \{O_{j,0}, \ldots, O_{j,j}\}) \leftarrow \mathsf{IdealFormOnion}(\vec{P}, m)$$
$$(P_i, O_{j,x}) \mapsto (\vec{P}', m, \mathsf{bruisecount} = 0), \quad \forall O_{j,x} \in \vec{O_j}$$

   (c) $\mathsf{IdealProcOnion}$ forms the onion for the next segment with 0 bruises, but records the running bruise count in $\mathsf{PathLookup}$. It also populates the dictionary with every possible valid bruising variant. When forming the onion for the last segment, it forms the sepal matrix such that a master key is accessible if the total bruise count is under threshold.
2. The simulator differs in that it provides $\mathcal{F}_{POES}$ with the algorithms $\mathsf{PeelOnion}$ and $\mathsf{BruiseOnion}$ instead of CL05's $\mathsf{ProcOnion}$. Additionally, the $\mathsf{SampleOnion}$ method takes a bruise count, and will output an onion with that many bruises. $\mathsf{SampleOnion}$ will also output a set of onions $(o_i, \vec{O_j})$, to capture all valid bruising variants of O_j.

Aside from incorporating these changes, the proof does not change.

B Peonies: Proof of Security

Below, we give the supplementary proofs for the Theorem 1.

Lemma 2. *Hybrid_2 and Hybrid_3 are indistinguishable.*

Proof. Assume that there exists a p.p.t adversary $\mathcal{A}$ that can distinguish whether it is in Hybrid_2 or Hybrid_3 with non-negligible (in the security parameter) advantage. We construct the following reduction $\mathcal{B}$. $\mathcal{B}$ plays the role of the challenger for $\mathcal{A}$ and that of the adversary in the PRP security fame against the challenger $\mathcal{C}$.

1. During setup, $\mathcal{B}$ interacts only with $\mathcal{A}$; it does not interact with $\mathcal{C}$. $\mathcal{A}$ sends the names of the honest parties H_m, H_g and the public portions $\mathsf{pk}(\mathsf{Bad})$ of the keys belonging to the adversarial parties to $\mathcal{B}$.
2. $\mathcal{B}$ generates the keys for the honest parties and sends the public portions of the generated keys to $\mathcal{A}$.
3. During the first query phase, $\mathcal{B}$ still interacts only with $\mathcal{A}$. Whenever $\mathcal{A}$ sends $\mathcal{B}$ a query to peel (resp. bruise) an onion O on behalf of an honest party P, $\mathcal{B}$ responds with the output of $\mathsf{PeelOnion}$ (resp. $\mathsf{BruiseOnion}$) on O and P's secret key.
4. During the challenge phase, $\mathcal{A}$ sends to $\mathcal{B}$ the onion parameters: the message m, the routing path $\vec{P} = (M_0, \ldots, M_{l_1-1}, G_0, \ldots, G_{l_2-1}, R)$, such that $H_m = M_{h_0}$ and $H_g = G_{h_1}$, and the metadata sequence $\vec{y} = (y_0, \ldots, y_{l-1})$.
5. $\mathcal{B}$ forms the inner onion layers $\vec{O}_l, \ldots, \vec{O}_{h_0+1}$ by following the $\mathsf{FormOnion}$ algorithm, but deviates from the protocol when forming $\vec{O}_{h_0}$. First, $\mathcal{B}$ forms the dummy ciphertext E_{h_0} by encrypting the dummy key 0 under the public key M_{h_0}. Next, $\mathcal{B}$ sends to $\mathcal{C}$: the header set $\vec{X}^{h_0+1}$, the content C^{h_0+1}, and the sepal matrices $\vec{A}^{h_0+1,b}$ for all valid bruise counts. $\mathcal{C}$ replies with $\vec{X}'^{h_0}$, C'^{h_0}, and the sepal sets $\vec{A}'^{h_0,b}$ for all valid bruise counts. These are obtained by applying either 1) a truly random permutation or 2) a psuedorandom permutation keyed by some key k'_{h_0} that is unknown to $\mathcal{B}$. $\mathcal{B}$ sets the first block of $\vec{X}'^{h_0}$, $X'^{h_0}_{h_0} = (E_{h_0}, I_{h_0})$, then assigns each onion variant $O_{h_0,b} \in \vec{O}_{h_0}$ to be $(\vec{X}'^{h_0}, C'^{h_0}, \vec{A}'^{h_0,b})$. $\mathcal{B}$ finally forms the outer layers $\vec{O}_{h_0-1}, ..., \vec{O}_0$ by following the $\mathsf{FormOnion}$ procedure, building on the modified inner onion $(\vec{X}'^{h_0}, C'^{h_0}, \vec{A}'^{h_0})$. $\mathcal{B}$ sends O_0 to $\mathcal{A}$.
6. Once again $\mathcal{A}$ is allowed to query onions. The first time $\mathcal{A}$ asks to have an onion $O_{h_0,b} \in \vec{O}_{h_0}$ peeled (or bruised), the challenger responds with its peeled version $\vec{O}_{h_0+1,b}$ (resp. $\vec{O}_{h_0+1,b+1}$). Whenever $\mathcal{A}$ queries an onion $O = ((\vec{X}^{h_0}), C, \vec{A}) \notin \vec{O}_{h_0}$, $\mathcal{B}$ responds with $\perp$. For all other queries, $\mathcal{B}$ deals with them in the same way as before.
7. If $\mathcal{A}$ guesses Hybrid_2, $\vec{B}$ guesses that the challenge blocks are psuedorandom. Otherwise if $\mathcal{A}$ guesses Hybrid_3, $\mathcal{B}$ guesses that the blocks are truly random.

The reduction works because $\mathcal{B}$'s advantage is the same as $\mathcal{A}$'s, and the reduction runs in polynomial time. □

Lemma 3. *Hybrid_5 and Hybrid_6 are indistinguishable.*

Proof. The reduction is essentially the same as Lemma 2. The reduction differs only in step 6. Assume that there exists a p.p.t adversary $\mathcal{A}$ that can distinguish whether it is in Hybrid_5 or Hybrid_6 with non-negligible (in the security parameter) advantage. We construct the following reduction $\mathcal{B}$. $\mathcal{B}$ plays the role of the challenger for $\mathcal{A}$ and that of the adversary in the PRP security game against the challenger $\mathcal{C}$.

1. During setup, $\mathcal{B}$ interacts only with $\mathcal{A}$; it does not interact with $\mathcal{C}$. $\mathcal{A}$ sends the names of the honest parties H_m, H_g and the public portions $\mathsf{pk}(\mathsf{Bad})$ of the keys belonging to the adversarial parties to $\mathcal{B}$.
2. $\mathcal{B}$ generates the keys for the honest parties and sends the public portions of the generated keys to $\mathcal{A}$.
3. During the first query phase, $\mathcal{B}$ still interacts only with $\mathcal{A}$. Whenever $\mathcal{A}$ sends $\mathcal{B}$ a query to peel (resp. bruise) an onion O on behalf of an honest party P, $\mathcal{B}$ responds with the output

of PeelOnion (resp. BruiseOnion) on O and P's secret key.

4. During the challenge phase, $\mathcal{A}$ sends to $\mathcal{B}$ the onion parameters: the message m, the routing path $\vec{P} = (M_0, \ldots, M_{l_1-1}, G_0, \ldots, G_{l_2-1}, R)$, such that $H_m = M_{h_0}$ and $H_g = G_{h_1}$, and the metadata sequence $\vec{y} = (y_0, \ldots, y_{l-1})$.
5. $\mathcal{B}$ forms the inner onion layers $\vec{O}_l, \ldots, \vec{O}_{h_0+1}$ by following the FormOnion algorithm, but deviates from the protocol when forming $\vec{O}_{h_0}$. First, $\mathcal{B}$ forms the dummy ciphertext E_{h_0} by encrypting the dummy key 0 under the public key M_{h_0}. Next, $\mathcal{B}$ sends to $\mathcal{C}$: the header set $\vec{X}^{h_0+1}$, the content C^{h_0+1}, and the sepal matrices $\vec{A}^{h_0+1,b}$ for all valid bruise counts. $\mathcal{C}$ replies with $\vec{X}'^{h_0}$, C'^{h_0}, and the sepal sets $\vec{A}'^{h_0,b}$ for all valid bruise counts. These are obtained by applying either 1) a truly random permutation or 2) a psuedorandom permutation keyed by some key k'_{h_0} that is unknown to $\mathcal{B}$. $\mathcal{B}$ sets the first block of $\vec{X}'^{h_0}$, $X'^{h_0}_{h_0} = (E_{h_0}, I_{h_0})$, then assigns each onion variant $O_{h_0,b} \in \vec{O}_{h_0}$ to be $(\vec{X}'^{h_0}, C'^{h_0}, \vec{A}'^{h_0,b})$. $\mathcal{B}$ finally forms the outer layers $\vec{O}_{h_0-1}, ..., \vec{O}_0$ by following the FormOnion procedure, building on the modified inner onion $(\vec{X}'^{h_0}, C'^{h_0}, \vec{A}'^{h_0})$. $\mathcal{B}$ sends O_0 to $\mathcal{A}$.
6. Once again, $\mathcal{A}$ is allowed to query onions. $\mathcal{B}$ handles the first query to peel or bruise an onion $O_{h_0,b} \in \vec{O}_{h_0}$ by running FormOnion on the message m, the routing path $(M_{h_0+1}, ..., R)$, and the associated metadata sequence $(y_{h_0+1}, ..., y_{l-1})$; $\mathcal{B}$ replied with the onion $O_{h_0+1,b}$ for the $h_0 + 1^{th}$ processor on the routing path with b bruises (see Hybrid_5. Whenever $\mathcal{A}$ queries an onion $O = ((\vec{X}^{h_0}), C, \vec{A}) \notin \vec{O}_{h_0}$, $\mathcal{B}$ responds with $\perp$. For all other queries, $\mathcal{B}$ deals with them in the same way as before.
7. If $\mathcal{A}$ guesses Hybrid_5, $\mathcal{B}$ guesses that the challenge blocks are truly random. Otherwise if $\mathcal{A}$ guesses Hybrid_6m $\mathcal{B}$ guesses that the blocks are psuedorandom.

The reduction works because $\mathcal{B}$'s advantage is the same as $\mathcal{A}$'s, and the reduction runs in polynomial time. □

Lemma 4. *Hybrid_6 and Hybrid_7 are indistinguishable.*

Proof. The reduction is essentially the same as in Lemma **??**. The reduction differs only in steps 5 and 6. Assume that there exists a p.p.t adversary $\mathcal{A}$ that can distinguish whether it is in Hybrid_6 or Hybrid_7 with non-negligible (in the security parameter) advantage. We construct the following reduction $\mathcal{B}$. ($\mathcal{B}$) is the challenger for $\mathcal{A}$, but the adversary in the CCA2 security game against a challenger $\mathcal{C}$.)

1. During setup, $\mathcal{A}$ sends the names of the honest parties H_m, H_g, along with the public keys of the adversarial parties $\mathsf{pk}(\mathsf{Bad})$ to $\mathcal{B}$.
2. $\mathcal{B}$ generates the key pair for H_g; that is, $\mathcal{B}$ obtains $(\mathsf{pk}(H_g), \mathsf{sk}(H_g))$ by running $\mathsf{KeyGen}(H_g)$. $\mathcal{C}$ supplies the public key $\mathsf{pk}(H_m)$, and $\mathcal{B}$ forwards $\mathsf{pk}(H_m), \mathsf{pk}(H_g)$ to $\mathcal{A}$.
3. During the first query phase, whenever $\mathcal{A}$ sends an onion $O = (\vec{X}, C, \vec{A})$ to be peeled (resp. bruised) on behalf of H_m, $\mathcal{B}$ sends the ciphertext E to $\mathcal{C}$ to be decrypted, where E is the public portion of the ciphertext in the first header block. Once $\mathcal{C}$ replies with the layer key k, $\mathcal{B}$ uses the PeelOnion (resp. BruiseOnion) algorithm on O and replies to $\mathcal{A}$ with the output. (For queries to peel or bruise on behalf of H_g, $\mathcal{B}$ just runs PeelOnion or BruiseOnion.)
4. During the challenge phase, $\mathcal{A}$ sends to $\mathcal{B}$ the onion parameters: the message m, the routing path $\vec{P} = (M_0, \ldots, M_{l_1-1}, G_0, \ldots, G_{l_2-1}, R)$, such that $H_m = M_{h_0}$ and $H_g = G_{h_1}$, and the metadata sequence $\vec{y} = (y_0, \ldots, y_{l-1})$.
5. $\mathcal{B}$ picks the layer keys $k_0, \ldots, k_l$ and sends the challenge messages $m_0 = k_{h_0}$ and $m_1 = 0$ to $\mathcal{C}$. Once $\mathcal{B}$ obtains the ciphertext $c_b \leftarrow \mathsf{Enc}(\mathsf{pk}(M_{h_0}, m_b))$ from $\mathcal{C}$, it forms the challenge onion O_0 by running FormOnion (on input: the dummy message content $\perp$, the partial path $(M_0, \ldots, M_{h_0})$, and the associated metadata sequence $(y_0, \ldots, y_{h_0})$) except that in forming

the h_0^{th} onion layer, $\mathcal{B}$ uses c_b in place of the ciphertext E_{h_0}. $\mathcal{B}$ sends O_0 to $\mathcal{A}$.

6. Once again, $\mathcal{A}$ is allowed to query onions. $\mathcal{B}$ handles the first query to peel or bruise an onion $O_{h_0,b} \in \vec{O}_{h_0}$ by running $\mathsf{FormOnion}$ on the message m, the routing path $(M_{h_0+1}, \ldots, R)$, and the associated metadata sequence $(y_{h_0+1}, \ldots, y_{l-1})$. $\mathcal{B}$ replies with the onion $O_{h_0+1,b}$ (resp. $O_{h_0+1,b+1}$) for the $h_0 + 1^{th}$ processor on the routing path with b bruises (see Hybrid_6). All other queries are handled in the same manner as before.
7. If $\mathcal{A}$ guesses Hybrid_6, $\mathcal{B}$ guesses 1. Otherwise if $\mathcal{A}$ guesses Hybrid_7, $\mathcal{B}$ guesses 0.

The reduction works because $\mathcal{B}$'s advantage is the same as $\mathcal{A}$'s, and the reduction runs in polynomial time. □

The reductions for $\mathsf{Hybrid}_8, \ldots, \mathsf{Hybrid}_{15}$ convert the "middle layers" of the challenge onion in Hybrid_7 to those in Experiment 1. Thus the reductions for these are the same arguments as those from $\mathsf{Hybrid}_0, \ldots, \mathsf{Hybrid}_7$.

C Peonies Example Diagrams

Here, we describe how the sepal matrix is manipulated over the routing path $\vec{P} = \{M_0, M_1, M_2, G_0, G_1, R\}$, with a bruise tolerance of $d = 1$. M_0 does not bruise the onion, M_1 and M_2 both bruise the onion, exceeding the bruising threshold and rendering the onion undecryptable.

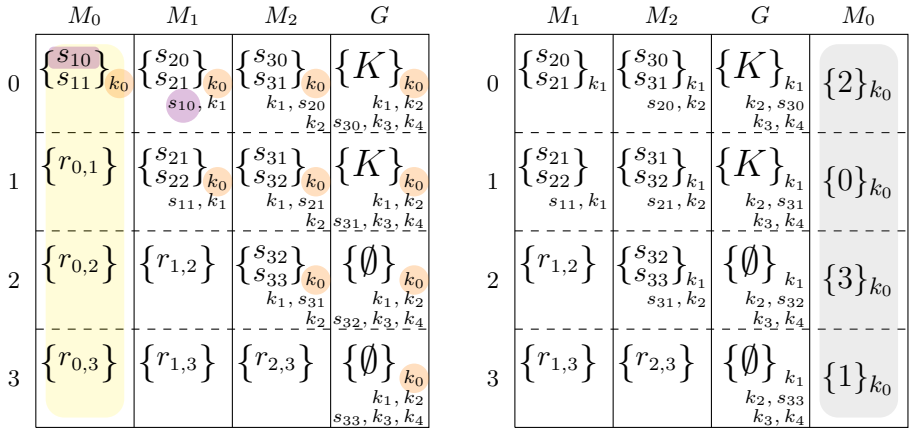

Figure 11: Sepal matrix as generated by the sender (same as Figure 4), and processed by M_0. M_0 chooses *not* to bruise the onion. Accordingly it unlocks the cell $(1, 0)$ using the sepal key $s_{1,0}$ and deletes the first column, inserting a deterministically generated dummy column at the end of the matrix.

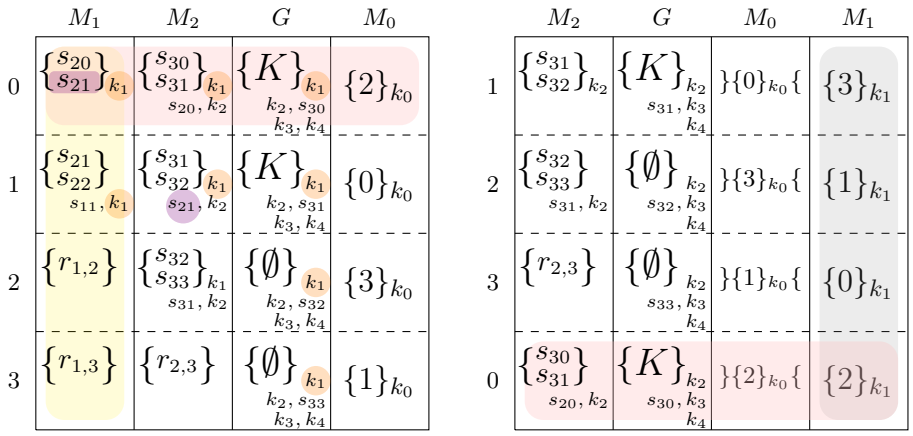

Figure 12: Sepal matrix as processed by M_1. M_1 chooses to bruise the onion. Accordingly it unlocks the cell $(2, 1)$ using the sepal key $s_{2,1}$ and performs a row rotation. It then deletes the first column, inserting a deterministically generated dummy column at the end of the matrix. Note that ordering of the encryptions of the dummy column is based on the numerical ciphertext of the values in previous column. This means any individual row will always have the same dummy values, regardless of the number of bruises/row rotations. We omit layer decryptions on the filler values for readability, instead representing them with the decryption symbol } {.

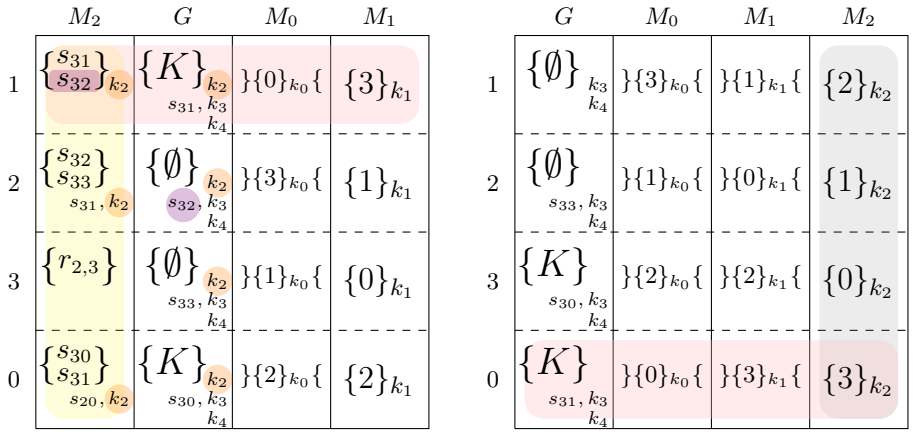

Figure 13: Sepal matrix as processed by M_2. M_2 chooses to bruise the onion. Accordingly it unlocks the cell $(3, 2)$ using the sepal key $s_{3,2}$ and performs a row and column manipulation. Because the onion has now been bruised more than its bruising threshold, we can see that all copies of the master key K have rotated to the bottom of the matrix, and are locked with multiple sepal and layer keys.

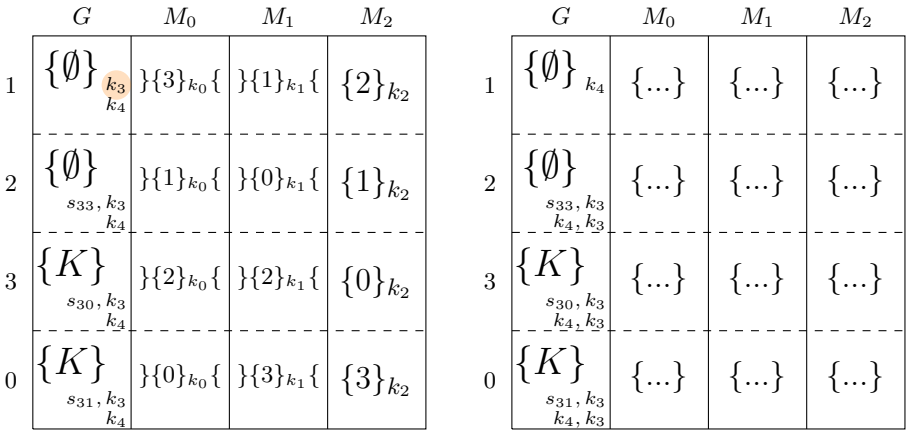

Figure 14: Sepal matrix as processed by G_0. The gatekeepers cannot bruise the onion, and instead remove a layer encryption off of the cell located at $[0, 0]$ (in this case the block $(3, 2)$), and encrypt the rest of the matrix with their layer key. They also do not perform matrix manipulations.

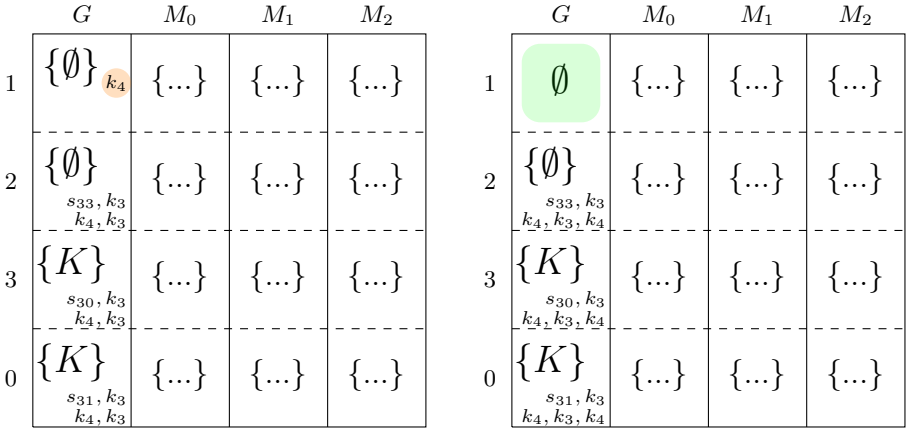

Figure 15: Sepal matrix as processed by G_1. As the final gatekeeper G_1 removes the final layer of encryption off of the matrix to reveal the contents of the final column. Here, the onion was over-bruised, so G_1 is only able to access an encryption of 0, instead of a copy of the master key. G_1 has no way of decrypting the location of the final recipient without K, and so the onion is dropped. Notice how all other cells are locked under multiple layer and sepal keys.

D $\Pi_\bullet$: Proof of Anonymity

Below, we present the proof of Theorem 2.

Roadmap of the proof. We show that $\Pi_\bullet$ is statistically anonymous (Definition 1) under the assumption that the onions are "ideal," as characterized by the ideal functionality for onion security in prior works [CL05, AL21] (reiterated in Appendix A[8]). In particular, we assume that the appearance of each onion on the links does not reveal any information. Consequently, the adversary's view is limited to the volumes of onions on the links and knowledge of how the onions peel at compromised nodes. By Canetti's UC theorem [Can01], this implies the anonymity guarantee we desire.

Let an onion be "good" if it was created by an honest party and meets one of the following criteria: it is a looping onion, a diagnostic onion intended for verification by an honest party, or a message-bearing onion. We demonstrate anonymity even when the adversary has complete knowledge about all other (i.e., "bad") onions: those they create themselves and the diagnostic onions meant for verification by corrupted parties. In the analysis below, all references to quantities of onions pertain exclusively to good onions.

Throughout the analysis, we make frequent use of the Chernoff bound for Poisson trials [MU05, Corollary 4.6] and the Azuma-Hoeffding inequality [MU05, Theorem 13.4]. Each application of these bounds allows us to conclude that, with overwhelming probability (w.o.p.), the size of a sampled set of onions Z is close to its expectation $\mathbb{E}[Z]$. In particular, for any constant $\delta > 0$,

$$\Pr[|Z - \mathbb{E}[Z]| \geq \delta \mathbb{E}[Z]] = \mathsf{negl}(\lambda).$$

(When sampling with replacement, we apply a Chernoff bound; see Section D.1. When sampling without replacement, we instead use the Azuma-Hoeffding inequality; see Section D.2.) Since the total number of such random variables is polynomially bounded, it follows that, with overwhelming probability, *all* of them are simultaneously close to their respective expectations.

Proof. Recall that the execution of the protocol is divided into h epochs. For each party P_j, let α_j be the fraction of message-bearing onions for P_j that the adversary drops before the last round of the first epoch.

Let P_j be an honest party. If the fraction of looping onions missing for P_j is at most $\rho_x = \frac{1}{3}$, then, with overwhelming probability, $\alpha_j \leq \frac{1}{2}$ (Lemma 1).

We define onions as either "unpaired" or "paired" based on their interactions during the *previous* epoch. An onion in a given epoch is considered "unpaired" if it did not merge with any other onion in the previous epoch; conversely, it is "paired" if it did merge. By this definition, every onion in the first epoch is classified as unpaired.

For each epoch i, let β_i represent the fraction of remaining unpaired onions the adversary drops during that epoch (including the last round of the previous epoch). For example, β_2 is calculated as the number of unpaired onions dropped in the second epoch, divided by the total number of unpaired onions present in that same epoch.

Let $\rho_1 = \beta_1$, and for $i \geq 2$, define ρ_i recursively as follows: $\rho_i = \rho_{i-1} + \beta_1(1 - \rho_{i-1})$. Each ρ_i is an estimate of the fraction of diagnostic onions that were dropped by (and including) the ith epoch.

Lemma 5. *If $\rho_i \geq \frac{(1-\kappa)^2}{2}$, then, with overwhelming probability, every honest party P_j observes a fraction of missing checkpoint onions in the ith merging round exceeding $\rho_y = \frac{(1-\kappa)^2}{3}$ (sampling*

[8] This includes a proof that bruisable onion security implies realizability of the analogous functionality for bruisable onions, which is used in the corresponding analysis of $\Pi_{\tilde{o}}$.

without replacement; Azuma-Hoeffding inequality). Conversely, if any honest party P_j observes a fraction of missing checkpoint onions below $\rho_y = \frac{(1-\kappa)^2}{3}$, then, with overwhelming probability, $\rho_i < \frac{1}{2}$ (sampling without replacement; Azuma-Hoeffding inequality).

Case 1: $\Pi_\bullet$ aborted by midpoint. By the "midpoint," let us refer to the diagnostic round occurring in epoch $\frac{h}{2}$. First note that w.o.p., by the midpoint, every honest party aborts the protocol if one or both of these conditions hold:

1. An honest P_j is missing more than one-third of its looping onions at the end of the first epoch. (In this case, the P_j initiates an abort by sending out abort messages. This guarantees that w.o.p., every honest party aborts [ALU21, Lemma 8].)
2. The adversary drops a "significant number" of onions by the midpoint, i.e., $\rho_{h/2} \geq \frac{(1-\kappa)^2}{2}$. (From Lemma 5, in this case, each P_j notices that too many checkpoint onions are missing, at which point they will abort the protocol.)

$\Pi_\bullet$ maintains anonymity in these situations by ensuring no one receives their message. The intermediaries on the routing path for each onion are chosen independently of the inputs. (w.o.p., no onion will be delivered to its recipient because there are necessarily honest parties on the latter half of the routing path.)

Case 2: $\Pi_\bullet$ not aborted by midpoint. From Lemma 1 and Lemma 5, it remains to show that $\Pi_\bullet$ achieves anonymity when each honest P_j receives at least two-thirds of its looping onions, and $\rho_{h/2} < \frac{1}{2}$.

During each epoch, every pair of onions is shuffled with a polylogarithmic number of other onions, a polylogarithmic number of times; this is because $\rho_{h/2} < \frac{1}{2}$ and T is super-polylogarithmic in the security parameter. By the end of the epoch, this process guarantees that the onions are "sufficiently mixed." See prior work for proof of mixing [ALU18,ALU24].[9] As a result, an adversary cannot determine whether a given onion at the end of the epoch originated from its true source (the onion at the start of the epoch) or another onion's source.

Consequently, the only information available to the adversary for adapting its dropping strategy is knowledge about which onions merged early in the epoch and which did not. (Note that other actions, such as injecting onions, cannot help the adversary determine the communication pattern.)

For each epoch i and each honest party P_j: Let $\mathcal{U}_{i,j}$ represent the set of message-bearing onions formed by P_j for the ith epoch. The sizes of these sets are given by $|\mathcal{U}_{1,j}| = z$, $|\mathcal{U}_{2,j}| = \frac{z}{2}$, and so on. Let $\mathcal{V}_{i,j}$ represent the set of message-bearing onions formed by P_j in the ith epoch of the protocol run, so that $p_{i,j} = |\mathcal{V}_{i,j}|/|\mathcal{U}_{i,j}| \leq 1$.

Let the gap g_i be the maximum difference among the $p_{i,\cdot}$ values, i.e., $g_i = \max_{j,k}(p_{i,j} - p_{i,k})$.

We first assume that the adversary does not drop any paired onions. With this assumption, we can show that:

Lemma 6. *For any dropping strategy $(\beta_1, \beta_2, \ldots, \beta_{h/2})$ s.t. $\rho_{h/2} < \frac{(1-\kappa)^2}{2}$, the gap at the midpoint, $g_{h/2}$, is negligibly small.*

We prove Lemma 6 by demonstrating that an optimal strategy to keep $g_{h/2}$ non-negligible is with $\beta_2 = \frac{(1-\kappa)^2}{2}$ and $\beta_3 = \beta_4 = \cdots = \beta_{h/2} = 0$. This is established through a proof by contradiction:

[9] Note that the proof of mixing [ALU24] relies on the corruption rate being $< \frac{1}{2}$. Ando et al. use it to prove that any two onions will simultaneously be at an honest relay (not necessarily the same one) some polylogarithmic number of times. However, the same result in the synchronous setting follows, with corruption rate < 1, from a Chernoff bound [ALU18]. This results in a difference in the corruption rate regimes for which $\Pi_\bullet$ and $\Pi_{\bar{o}}$ are proven to be anonymous.

We begin by assuming that there is a better strategy in which some $\beta_{i\geq 3} > 0$. Let $\gamma = \beta_{i-1} + \frac{\beta_i}{1-\beta_{i-1}}$; we can think of this as the "budget" for dropping onions in epochs $i-1$ and i. We then show that the strategy that is the same, except that $\beta_{i-1} = \gamma$ and $\beta_i = 0$, actually performs better. To do this, we show that the numbers of paired and unpaired onions in a random sample of $\mathcal{U}_k$ are close to their expected values [Azuma-Hoeffding inequality]. This allows us to obtain a high-quality estimate of the size of $\mathcal{V}_{i+1}$ given as a function of β_{i-1}; we show that $\mathcal{V}_{i+1}$ is largest when $\beta_{i-1} = \gamma$ (and $\beta_i = 0$). (See [ALU21] for the full proof.)

Therefore, w.l.o.g., we can assume that the adversary allocates their budget for dropping onions entirely in the second epoch. This implies that $\beta_2 \leq \frac{1}{2}$ and that all other β values are zero. Additionally, since each honest party P_j has $\alpha_j < \frac{1}{2}$, at least some constant fraction (e.g., $\frac{1}{4}$) of P_j's onions will remain in circulation by the end of the second round.

For a message-bearing onion to not exist at the midpoint, the adversary must drop all of its "descendants" (onions that are its children in the binary tree structure used for forming the routing paths for a set of message-bearing onions) during the first two epochs. However, the probability of this happening is negligibly small since there is a super-logarithmic number of descendants [Azuma-Hoeffding inequality]. By applying the union bound, the probability that *any* message-bearing onion has all its descendants dropped upfront is negligibly small. Since the adversary does not drop any more onions after this, w.o.p., for any epoch $t \geq \frac{h}{2}$, the number of message-bearing onions for any sender is $|\mathcal{U}_{t\geq h/2,j}| = \frac{z}{t}$. Thus, anonymity is achieved in this case.

To complete our proof, we point out that if the adversary also drops paired onions, the gap shrinks *faster* because, w.o.p., the adversary drops more onions in $\mathcal{V}_j$ compared to $\mathcal{V}_k$ whenever $|\mathcal{V}_j|$ is larger than $|\mathcal{V}_k|$. □

D.1 Sampling with Replacement

Consider the following sampling problem. We have an opaque bag containing N marbles (think of these as idealized onions, perfectly indistinguishable by feel). A fraction p of the marbles are yellow (onions), while the remaining marbles are red (onions). Suppose we draw n marbles, one at a time, where after each draw the marble is replaced and the next marble is selected uniformly at random (i.e., sampling with replacement).

Let X_i be the indicator random variable that equals 1 if and only if the ith drawn marble is yellow. For example, if the first marble is yellow, the second is red, the third is red, and the fourth is yellow, then $X_1 = X_4 = 1$ and $X_2 = X_3 = 0$. Let

$$X = \sum_{i=1}^{n} X_i$$

denote the total number of yellow marbles in the sample.

Since each draw is independent, the random variables $X_1, \ldots, X_n$ are independent Bernoulli trials with success probability p, and hence

$$\mathbb{E}[X] = pn.$$

It is a standard fact that when $\mathbb{E}[X]$ is polylogarithmic in the security parameter (i.e., $\mathbb{E}[X] = \Omega(\log^a \lambda)$ for some constant $a > 0$) the random variable X is tightly concentrated around its expectation. More precisely, for any constant $\delta \in (0, 1)$,

$$\Pr[|X - \mathbb{E}[X]| > \delta\, \mathbb{E}[X]]$$

is negligible in the security parameter. Equivalently, with overwhelming probability,

$$(1-\delta)\,\mathbb{E}[X] \;\le\; X \;\le\; (1+\delta)\,\mathbb{E}[X].$$

This concentration result follows directly from a standard Chernoff bound for sums of independent Bernoulli (or Poisson) trials [MU05, Corollary 4.6].

D.2 Sampling without Replacement

We also present the analogous result for the similar sampling problem *without replacement*. That is, X_i indicates whether the ith marble is yellow when the previously drawn marbles $1, \ldots, i-1$ were *not* placed back in the bag. In this case, it is also true that, with overwhelming probability,

$$(1-\delta)\,\mathbb{E}[X] \;\le\; X \;\le\; (1+\delta)\,\mathbb{E}[X].$$

Since $X_1, X_2, \ldots, X_n$ are no longer independent, we cannot directly apply Chernoff bounds. Instead, we use the Azuma-Hoeffding inequality, which provides a similar concentration bound for martingales (defined below).

Definition 4 (Martingale [MU05, Definition 13.1])**.** *A sequence of random variables $Z_0, Z_1, \ldots$ is a* martingale *with respect to the sequence $X_0, X_1, \ldots$ if, for all $i \ge 0$, the following conditions hold:*

1. *Z_i is a function of $X_0, X_1, \ldots, X_i$.*
2. *$\mathbb{E}[|Z_i|] < \infty$.*
3. *$\mathbb{E}[Z_{i+1} \mid X_0, X_1, \ldots, X_i] = Z_i$.*

We define the following sequence of random variables:

$$Z_i = \mathbb{E}[X \mid X_1, \ldots, X_i].$$

Then, $Z_0, Z_1, \ldots, Z_n$ forms a martingale (specifically, a *Doob martingale*) by construction.

Intuitively, $Z_0 = \mathbb{E}[X]$ is the expected total number of yellow marbles before any draw is made. Each subsequent Z_i is a "refinement" of this expectation as we observe more draws: it is the conditional expectation of X given the first i draws. Finally, $Z_n = X$, because after observing all n draws, there is no uncertainty left about the total number of yellow marbles.

The Azuma-Hoeffding inequality [MU05, Theorem 13.6] can then be applied to this martingale to obtain the desired concentration result. Roughly speaking, if the martingale differences are bounded by constants, $|Z_{i+1} - Z_i| \le c_i$ for all i, then for any $\delta \in (0,1)$,

$$\begin{aligned}\Pr[|Z_n - Z_0| \ge \delta\,\mathbb{E}[X]] &\le 2\exp\left(-\frac{\delta^2\,\mathbb{E}[X]^2}{2\sum_{i=1}^{n} c_i^2}\right)\\ &= \mathsf{negl}(\lambda) \quad (\text{when } \mathbb{E}[X] = \Omega(\log^a \lambda) \text{ for constant } a > 1).\end{aligned}$$

In our sample-without-replacement problem, the differences $|Z_{i+1} - Z_i|$ are bounded by constants (e.g., $c_i = 1$ since each marble contributes at most 1 to the sum), so the inequality ensures that $X = Z_n$ is tightly concentrated around its expectation $\mathbb{E}[X] = Z_0$, just as in the sample-with-replacement case.