# On the Complexity of Anonymous Communication Through Public Networks

**Megumi Ando**
MITRE, Bedford, MA, USA

**Anna Lysyanskaya**
Brown University, Providence, RI, USA

**Eli Upfal**
Brown University, Providence, RI, USA

## Abstract

Onion routing is the most widely used approach to anonymous communication online. The idea is that Alice wraps her message to Bob in layers of encryption to form an "onion" and routes it through a series of intermediaries. Each intermediary's job is to decrypt ("peel") the onion it receives to obtain instructions for where to send it next. The intuition is that, by the time it gets to Bob, the onion will have mixed with so many other onions that its origin will be hard to trace even for an adversary that observes the entire network and controls a fraction of the participants, possibly including Bob. Despite its widespread use in practice, until now no onion routing protocol was known that simultaneously achieved, in the presence of an active adversary that observes all network traffic and controls a constant fraction of the participants, (a) anonymity; (b) fault-tolerance, where even if a few of the onions are dropped, the protocol still delivers the rest; and (c) reasonable communication and computational complexity as a function of the security parameter and the number of participants.

In this paper, we give the first onion routing protocol that meets these goals: our protocol (a) achieves anonymity; (b) tolerates a polylogarithmic (in the security parameter) number of dropped onions and still delivers the rest; and (c) requires a polylogarithmic number of rounds and a polylogarithmic number of onions sent per participant per round. We also show that to achieve anonymity in a fault-tolerant fashion via onion routing, this number of onions and rounds is necessary. Of independent interest, our analysis introduces two new security properties of onion routing – mixing and equalizing – and we show that together they imply anonymity.

**2012 ACM Subject Classification** Security and privacy → Security protocols

**Keywords and phrases** Anonymity, privacy, onion routing

**Digital Object Identifier** 10.4230/LIPIcs.ITC.2021.9

**Related Version** *Full Version*: `https://arxiv.org/abs/1902.06306`

**Funding** Part of this work was funded by NSF award IIS-1247581 while Megumi was a graduate student at Brown University.

## 1 Introduction

Suppose that Alice wishes to send a message anonymously to Bob. Informally, by *anonymously*, we mean that no one (not even Bob) can distinguish the scenario in which Alice sends a message to Bob from an alternative scenario in which it is Allison who sends a message to Bob. To begin with, Alice can encrypt the message and send the encrypted message to Bob so that only Bob can read the message. However, an eavesdropper observing the sequence of bits coming out of Alice's computer and the sequence of bits going into Bob's computer can still determine that Alice and Bob are communicating with each other if the sequences of bits match. Thus, encryption is not enough.


Onion routing [11] is the most promising approach to anonymous channels to date. In onion routing, messages are sent via intermediaries and wrapped in layers of encryption, resulting in so-called onions; each intermediary's task is to "peel off" a layer of encryption and send the resulting onion to the next intermediary or its final destination. The onion's layers are unlinkable to each other, and so its route through the network cannot be traced from merely observing the sequences of bits that Alice transmits and Bob receives. However, even with Alice sending her message to Bob encoded as an onion, her communication can still be tracked by a resourceful eavesdropper with an extensive view of the network traffic (e.g., an ISP-level or an AS-level adversary) who can observe all Internet traffic.

The adversary who can observe all network traffic is called the *network adversary*. The adversary who, in addition to observing all network traffic, controls a subset of the participants is called the *passive adversary* if it follows the prescribed protocol, or *active* if it does not. The three adversary models – the network adversary, the passive adversary, and the active adversary – are standard for analyzing cryptographic protocols such as multi-party computation (MPC) [21]. The most desirable goal is to achieve security in the presence of the most powerful of these, i.e., the active adversary corrupting as large a fraction of the participants as possible.

It was known how to construct an onion routing protocol that is both efficient and anonymous from the passive adversary who corrupts a constant fraction of the parties. In $\Pi_p$ [3], each user forms an onion bearing his message to its recipient; the users' onions are routed randomly through a network of servers. $\Pi_p$ is anonymous from the passive adversary provided that the onions travel for a superlogarithmic (in the security parameter) number of rounds, and the average number of onions per server per round is also superlogarithmic.

However, $\Pi_p$ isn't anonymous from the active adversary. To see why this is the case, consider the following attack. Suppose that the adversary $\mathcal{A}$ suspects that Alice is communicating with Bob. Because $\mathcal{A}$ is active, he can disrupt Alice's communication by dropping Alice's outgoing onion in the event that Alice's first intermediary is corrupt (the probability of this event is identical to the fraction of parties that are under the adversary's control). If Bob doesn't receive an onion at the end of the protocol, then $\mathcal{A}$ can infer that her suspicion was correct: Alice's interlocutor is Bob!

So what can we do instead? Of course, we could use general-purpose MPC [21]. Every party will receive as input a message and its destination, and every party will receive as output the messages that were meant for him/her. In addition to perfect anonymity, this approach provides fault tolerance: in MPC that is secure against the active adversary, the honest parties are guaranteed to receive their output no matter how much the adversary deviates from the protocol. But the problem with this approach is that relying on *general-purpose* MPC makes this approach too inefficient: the most efficient general MPC protocol still requires that at least some of the participants send and receive $\Omega(N)$ bits, where $N$ is the number of participants. (See Cramer, Damgård, and Nielsen [16].)

Recently proposed protocols, Stadium [30] and Atom [25], are more efficient. However, they are not fault-tolerant: honest parties will abort the protocol run whenever even a single message packet is dropped. Thus, while this approach provides anonymity from the active adversary, it is also extremely fragile: if just one message is dropped (which could be the result of an innocuous fault), the entire network suffers a catastrophic failure. In contrast, we would like to design onion routing protocols that can tolerate faults. Thus, compared to MPC and Stadium-Atom-type protocols, onion routing appears attractive from the efficiency and fault tolerance points of view.

In this paper, we answer these fundamental questions: Can an onion routing protocol be simultaneously anonymous, fault-tolerant, and efficient? What is the communication complexity of anonymous and fault-tolerant onion routing?

## 1.1 Problem setting

Before describing our results in detail, let us first define our problem setting. Let $\mathcal{P} \stackrel{\text{def}}{=} \{P_1, P_2, \ldots, P_N\}$ denote the set of $N$ parties, participating in an onion routing protocol. We assume that the protocol progresses in global rounds and that an onion sent at round $r$ arrives at its destination prior to round $r+1$. Moreover, the adversary is modelled with *rushing*, i.e., the adversary receives onions sent in round $r$ instantaneously in round $r$.[1] We assume that the number $N$ of participants and every other quantity in the protocol is polynomially bounded in the security parameter $\lambda$.

We define an *onion routing protocol* to be a protocol in which the honest parties form and process only message packets that are cryptographic onions. To do this, the honest parties use a secure onion encryption scheme which is a triple of algorithms: $(\mathsf{Gen}, \mathsf{FormOnion}, \mathsf{ProcOnion})$.

See Section 2.1 for more details. During setup of an onion routing protocol, each honest party $P$ generates a public-key pair $(\mathsf{pk}(P), \mathsf{sk}(P)) \leftarrow \mathsf{Gen}(1^\lambda)$ using the onion encryption scheme's key generation algorithm $\mathsf{Gen}$. Each party $P$ publishes his/her public key $\mathsf{pk}(P)$ to a public directory so that everyone knows everyone else's public keys.

Let $\mathcal{M}$ be the space of fixed-length messages. An input $\sigma = (\sigma_1, \ldots, \sigma_N)$ to the protocol is a vector of inputs, where $\sigma_i$ is a set of message-recipient pairs for party $P_i$. For $m \in \mathcal{M}$ and $P_j \in \mathcal{P}$, the inclusion of a message-recipient pair $(m, P_j)$ in input $\sigma_i$ means that party $P_i$ is instructed to send message $m$ to recipient $P_j$. In this paper, we consider the following "benchmark" input space, dubbed the *simple input/output setting (I/O)*. An input $\sigma = (\sigma_1, \ldots, \sigma_N)$ is in the simple I/O setting if there exists a permutation function $\pi : \mathcal{P} \mapsto \mathcal{P}$ such that each party $P \in \mathcal{P}$ is instructed to send a message to party $\pi(P)$ and no other message, i.e., $\forall P \in \mathcal{P}$, $\exists m_P \in \mathcal{M}$ such that $\sigma_P = \{(m_P, \pi(P))\}$. The simple I/O setting is a superset of the spaces considered in prior works [3, 25, 30, 31].

Unless stated otherwise, the adversary is active and can observe the traffic on all communication channels and, additionally, can non-adaptively corrupt and control a constant fraction of the parties. By *non-adaptively*, we mean that the corruptions are made independently of any protocol run.[2] Without loss of generality, this type of corruption is captured by allowing the adversary to select the set $\mathsf{Bad}$ of corrupted parties prior to the beginning of the protocol run. Once the adversary corrupts a party, the adversary can observe the internal state and computations of the corrupted party and arbitrarily alter the behavior of the party.

By $\mathsf{V}^{\Pi,\mathcal{A}}(1^\lambda, \sigma)$, we denote the view of the adversary $\mathcal{A}$ when it interacts with protocol $\Pi$ on input: the security parameter $1^\lambda$ and the instructions $\sigma$. The view consists of all the observations that $\mathcal{A}$ makes during the run: the values and positions of every onion at every round, the states and computations of every corrupted party between every pair of consecutive rounds, the randomness used by $\mathcal{A}$, and the numbers of messages received by the honest parties. The view does not include the honest parties' randomness. $\mathsf{V}^{\Pi,\mathcal{A},\mathsf{Bad}}(1^\lambda, \sigma)$ denotes $\mathcal{A}$'s view given its choice $\mathsf{Bad}$ for the corrupted parties. At the end of the protocol run, each honest party $P_i$ outputs the set $\mathsf{O}_i^{\Pi,\mathcal{A}}(1^\lambda, \sigma)$ of (non-empty) messages from the message space $\mathcal{M}$ that $P_i$ receives from interacting with adversary $\mathcal{A}$ in a run of protocol $\Pi$ on input $\sigma$. We define the *output* $\mathsf{O}^{\Pi,\mathcal{A}}(1^\lambda, \sigma)$ of protocol $\Pi$ in an interaction with adversary $\mathcal{A}$ on input $\sigma$ as the $N$ parties' outputs:[3] $\mathsf{O}^{\Pi,\mathcal{A}}(1^\lambda, \sigma) \stackrel{\text{def}}{=} (\mathsf{O}_1^{\Pi,\mathcal{A}}(1^\lambda, \sigma), \mathsf{O}_2^{\Pi,\mathcal{A}}(1^\lambda, \sigma), \ldots, \mathsf{O}_N^{\Pi,\mathcal{A}}(1^\lambda, \sigma))$.

[1] We do not consider the asynchronous communication model [9] in which Alice's outgoing onions (including her onion to her recipient Bob) can be delayed indefinitely. In such a case, we cannot even guarantee correctness (i.e., message delivery when no party deviates from the protocol).

[2] If we were to allow the adversary to adaptively corrupt parties, then the adversary could easily block all of Alice's onions. For every onion $O$ sent by Alice, the adversary can corrupt the party $P$ who receives $O$ in time to direct $P$ to drop the onion obtained from processing $O$ before the next round.

[3] Technically, the view and the output may depend on other parameters, such as the public parameters

## 1.2 Our results

We now describe our results. Our construction pertains to the problem setting described in Section 1.1. Our lower bound applies more generally to any arbitrary input set (not necessarily constrained to the simple I/O setting).

Following prior work [3, 25, 30, 31], we use a natural game-based definition of anonymity: a protocol is *anonymous* if the adversary cannot distinguish the scenario in which Alice sends a message to Bob while Carol sends one to David, from one in which Alice's message goes to David while Carol's goes to Bob; (see Definition 4). More precisely, for any pair of inputs $(\sigma^0, \sigma^1)$ that agree on the inputs and outputs for the adversarial participants, $\mathsf{O}^{\Pi,\mathcal{A}}(1^\lambda, \sigma^0) \approx \mathsf{O}^{\Pi,\mathcal{A}}(1^\lambda, \sigma^1)$, where "$\approx$" denotes indistinguishability.

We relate anonymity of an onion routing protocol to two new concepts: An onion routing protocol *mixes* if it sufficiently shuffles the honest users' onions making it infeasible for the adversary to trace a received message back to its sender. A protocol *equalizes* if the adversary cannot determine the input from the numbers of messages received by the parties; in other words, the number of messages output by each participant – or the fact that a participant did not receive an output at all – are random variables that are computationally unrelated to the input vector $\sigma$. (See Definitions 5 and 7.)

We show that in many cases, mixing and equalizing implies anonymity, i.e., an onion routing protocol that mixes and equalizes is anonymous. (See Theorem 3 for the formal theorem statement.) We use this to prove that our protocol is anonymous. Anonymity also implies equalizing; this observation is useful for proving a lower bound that (almost) matches our protocol.

### 1.2.1 Anonymous, "robust," and efficient onion routing

As we just explained, our strategy is to construct a protocol that mixes and equalizes.

Intuitively, mixing is the easier one to achieve: the onions need to sufficiently shuffle with other onions traveling over the network to ensure that each of them is hard to trace. This intuition is essentially correct with the caveat that an active adversary can strategically interfere with this process by dropping onions. To ensure that each onion shuffles with a sufficiently large number of onions (formed by an honest party) a sufficiently large number of times, our protocol uses *checkpoint onions* [3] that each intermediary expects to receive, and if a constant fraction (e.g., one-third) of them don't arrive because the adversary dropped them, the protocol aborts.

An active adversary who controls a fraction of the participants can try to "isolate" an honest party Alice from the rest of the network by dropping all of the messages/onions received directly from Alice. In a fault-tolerant network protocol, the remaining participants may still be able to get their messages through to their destinations. In this case, based on who received an output, an adversary can infer who Alice's intended recipient was. This attack explains why equalizing is difficult to achieve.

To overcome this attack, we introduce a new type of onions, called *merging onions*. When two merging onions belonging to the same pair arrive at some intermediary $I$, $I$ recognizes that they are from the same pair (although, other than their next layer and destination, $I$ does not learn anything else about them). The protocol directs $I$ to discard one of them

(denoted, $\mathsf{pp}$) and the parties' states (denoted, $\mathsf{states}$). Thus, we could be more precise by denoting the view and the output as $\mathsf{V}^{\Pi,\mathcal{A}}(1^\lambda, \mathsf{pp}, \mathsf{state}, \sigma)$ and $\mathsf{O}^{\Pi,\mathcal{A}}(1^\lambda, \mathsf{pp}, \mathsf{states}, \sigma)$, but we will use the simpler notation for better readability.

(chosen at random) while sending its mate along. If only one onion of the pair arrived at $I$ while its mate is missing (i.e. the adversary dropped it some time earlier in the protocol run), then $I$ simply sends along the mate that survived, and there is nothing to discard.

Why does this help? Suppose that both Alice and Allison created $2^h$ merging onions; at rounds $r_1$, $r_2$, ..., $r_h$ each of these onions (if it hasn't been deleted yet) will meet a mate. Say, exactly one of Alice's onions is dropped by the adversary at some point prior to round $r_1$, so its mate (the onion it was supposed to pair with at round $r_1$) was not dropped. Also, suppose that none of Allison's onions were dropped. Then at round $r_1$ all but one of Alice's remaining merging onions will meet a mate, and half of them will be dropped, so exactly $2^{h-1}$ of Alice's onions will remain in the system – which is exactly how many of Allison's onions remain. Additional $h-1$ opportunities to merge account for the possibility that the adversary has dropped a larger number of Alice's onions. Merging onions ensure that the number of Alice's onions that remain in the system at the end of the protocol is the same as the number of Allison's onions, i.e., that the protocol equalizes. The fact that Alice was targeted and many of her onions had been dropped upfront doesn't matter because the protocol discards all but one of them anyway! (See Section 4 for a more in-depth description of merging onions and how to construct them.)

**Positive result.** We construct an onion routing protocol $\Pi_{\bowtie}$, pronounced "Pi-butterfly," because it uses a butterfly network. $\Pi_{\bowtie}$ takes advantage of the merging onions technique described above. It is (a) anonymous from the active adversary who can corrupt up to a constant fraction $\kappa < \frac{1}{2}$ of the parties and (b) *robust*, i.e. whenever the adversary drops at most logarithmic (in the security parameter) number of message packets (i.e. onions), $\Pi_{\bowtie}$ delivers the messages from honest senders with overwhelming probability. Moreover, (c) during the execution of the protocol, every honest party transmits up to a polylog (in the security parameter) number of onions: specifically $\gamma_1 \log N \log^{3+\gamma_2} \lambda$ onions, where $N$ is the number of participants, and $\lambda$ is the security parameter. $\gamma_1$ and $\gamma_2$ are parameters that can be set as desired: increasing them increases the rate at which the maximum distance in the adversarial views for any two inputs shrinks. (See Theorem 12 for the precise relationship.)

### 1.2.2 Matching negative result

Our protocol is essentially optimal as far as both the round complexity and the number of onions each participant sends out are concerned. In Section 7, we explain why a protocol that is anonymous and robust in the presence of an active adversary that corrupts a constant fraction of participants requires a polylogarithmic number of onions sent out per participant.

## 1.3 Related work

Our work is inspired by the fact that Tor [18], the most widely adopted anonymous communication system, is also known to have numerous security flaws [23, 27, 29, 32]: Tor is based on a highly efficient design that favors practicality over security and is not secure even from the passive adversary [17]. Moreover, it has been shown to be vulnerable to network traffic correlation attacks [23, 27, 29, 32]. Thus, our goal was to design a protocol that was as close to Tor's efficiency and fault tolerance as possible, while also being provably anonymous. We consider a very specific and narrow problem in the much larger field of anonymous messaging systems. Although our definition of anonymity and adversary models are standard in cryptography, other definitions have been considered [1, 5, 6, 10, 19] and positive results for alternative models are known [4–6].

Other provably anonymous systems exist [12, 14, 15, 28], but they are not nearly as efficient. Achieving anonymous channels using heavier cryptographic machinery has been considered also. One of the earliest examples is Chaum's dining cryptographer's protocol [12]. Rackoff and Simon [28] use secure multiparty computation for providing security from active adversaries. Other cryptographic tools used in constructing anonymity protocols include oblivious RAM (ORAM) and private information retrieval (PIR) [14, 15]. Corrigan-Gibbs et al.'s Riposte solution makes use of a global bulletin board with a latency of days [15].

We are not the first to look into lower bounds on the complexity of anonymous messaging protocols (e.g., [13, 17]). However, all other lower bounds are for the setting where every participant is guaranteed to receive an output, and don't apply to protocols that allow aborts or that allow some participants to receive an output while others' output doesn't make it through.

## 2 Preliminaries

For a set $\mathcal{S}$, we denote the cardinality of $\mathcal{S}$ by $|\mathcal{S}|$, and $s \leftarrow\!\!\$\, \mathcal{S}$ denotes that $s$ is chosen from $\mathcal{S}$ uniformly at random. For an algorithm $A(x)$, $y \leftarrow A(x)$ is the (possibly probabilistic) output $y$ from running $A$ on the input $x$. In this paper, $\log(x)$ is the logarithm of $x$ base 2.

We say that a function $f : \mathbb{N} \mapsto \mathbb{R}$ is negligible in the parameter $\lambda$, written $f(\lambda) = \mathsf{negl}(\lambda)$, if for a sufficiently large $\lambda$, $f(\lambda)$ decays faster than any inverse polynomial in $\lambda$. When $\lambda$ is the security parameter, an event $E_\lambda$ is said to occur with (non-)negligible probability if the probability of $E_\lambda$ can(not) be bounded above by a function negligible in $\lambda$. An event occurs with overwhelming probability (abbreviated, w.o.p.) if its complement occurs with negligible probability. We use the standard notion of a pseudorandom function [20, Chapter 3.6].

### 2.1 Onion encryption schemes

Our work on onion routing builds upon a secure onion encryption scheme [2,7,24]. Recall that an onion encryption scheme is a triple: $(\mathsf{Gen}, \mathsf{FormOnion}, \mathsf{ProcOnion})$. The algorithm $\mathsf{Gen}$ generates a participant key pair, i.e., a public key and a secret key. The algorithm $\mathsf{FormOnion}$ forms onions, and the algorithm $\mathsf{ProcOnion}$ processes onions.

Let $\mathcal{P}$ be a set of participants, and let $\mathsf{Bad} \subseteq \mathcal{P}$ be the set of corrupt parties. For every honest $P \in \mathcal{P} \setminus \mathsf{Bad}$, let $(\mathsf{pk}(P), \mathsf{sk}(P)) \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{pp}, P)$ be the key pair generated for party $P$, where $\lambda$ is the security parameter, and $\mathsf{pp}$, the public parameters. For every corrupt party $P \in \mathsf{Bad}$, let $\mathsf{pk}(P)$ denote $P$'s public key. Let $\mathcal{M}$ be the message space consisting of messages of the same fixed length, and let the nonce space $\mathcal{S}$ consist of nonces of the same fixed length. These lengths may be a function of the security parameter $\lambda$. Here, a *nonce* is really any metadata associated with an onion layer.

$\mathsf{FormOnion}$ takes as input a message $m \in \mathcal{M}$, an ordered list $(Q_1, \ldots, Q_{d-1}, R)$ of parties from $\mathcal{P}$, the public keys $(\mathsf{pk}(Q_1), \ldots, \mathsf{pk}(Q_{d-1}), \mathsf{pk}(R))$ associated with these parties, and a list $(s_1, \ldots, s_{d-1})$ of (possibly empty) nonces from $\mathcal{S}$ associated with the layers of the onion.[4] The party $R$ is interpreted as the *recipient* of the message, and the list $(Q_1, \ldots, Q_{d-1}, R)$ is the *routing path*. The output of $\mathsf{FormOnion}$ is a sequence $(O_1, \ldots, O_d)$ of onions. Such a sequence is referred to as an *evolution*, but every $O_i$ in the sequence is an onion. Because it is

[4] Technically, the input/output syntax and constructions of [2, 7] do not include the sequence $(s_1, \ldots, s_d)$ of nonces but can easily be extended to do so; if we use layered CCA2-secure encryption instead of onion encryption – which is fine for this application – then incorporating the nonces is trivial.

convenient to think of an onion as a layered encryption object where processing an onion $O_i$ produces the next onion $O_{i+1}$, we sometimes refer to the process of revealing the next onion as "decrypting the onion" or "peeling the onion."

For every $i \in [d-1]$, only intermediary party $Q_i$ can peel onion $O_i$ to reveal the next layer, $(O_{i+1}, Q_{i+1}, s_{i+1}) \leftarrow \mathsf{ProcOnion}(\mathsf{sk}(Q_i), O_i, Q_i)$, which contains the *peeled* onion $O_{i+1}$, the *next destination* $Q_{i+1}$ of the onion, and the nonce $s_{i+1}$. Only the recipient $R$ can peel the innermost onion $O_d$ to reveal the message, $m \leftarrow \mathsf{ProcOnion}(\mathsf{sk}(R), O_d, R)$.

In our constructions, a sender of a message $m$ to a recipient $R$ "forms an onion" by generating nonces and running the $\mathsf{FormOnion}$ algorithm on the message $m$, a routing path $(Q_1, \ldots, Q_{d-1}, R)$, the keys $(\mathsf{pk}(Q_1), \ldots, \mathsf{pk}(Q_{d-1}), \mathsf{pk}(R))$ associated with the parties on the routing path, and the generated nonces; the *formed onion* is the first onion $O_1$ from the list of outputted onions.

**Secure onion encryption.** Suppose that (honest) Alice generates an onion carrying a message $m$ for Bob. That is, she generates a string of nonces and runs the algorithm $\mathsf{FormOnion}$ on the inputs: the message $m$, the routing path $(Q_1, \ldots, Q_{i-1}, I, Q_{i+1}, \ldots, Q_{d-1}, \text{Bob})$, the public keys associated with the routing path, and the nonces. Let $O$ denote the onion for intermediary party $I$, i.e., $O$ is the $i^{th}$ onion in the outputted evolution. Suppose that (honest) Carol runs the algorithm $\mathsf{FormOnion}$ on the inputs: the message $m'$, the routing path $(Q'_1, \ldots, Q'_{j-1}, I, Q'_{j+1}, \ldots, Q'_{e-1}, \text{David})$, the public keys associated with the routing path, and some nonces. Let $O'$ denote the onion for intermediary party $I$. Provided that the onion encryption scheme is secure, if party $I$ receives onions $O$ and $O'$ in the same round and consequently processes the two onions in the same batch, then the adversary cannot tell which processed onion resulted from processing $O$ and which resulted from processing $O'$. In other words, onions formed by honest parties "mix" at honest parties. For a precise, cryptographic definition of secure onion encryption, see the recent paper by Ando and Lysyanskaya [2].

## 2.2 Onion routing protocols

In an onion routing protocol, all the packets sent between protocol participants are treated as onions; i.e., upon receipt, they are fed to $\mathsf{ProcOnion}$. Moreover, internally, there are type checks that ensure that these onions are processed properly. There are two cases for processing an honestly formed onion properly: the case where peeling the onion reveals its next layer and destination and the case where it reveals a message for the processing party.

If $Q_i$ runs $\mathsf{ProcOnion}$ and outputs the next layer of the onion $O_{i+1}$ (together with its destination $Q_{i+i}$ and nonce $s_{i+1}$), then the only two options for what an onion routing protocol permits $Q_i$ to do with $O_{i+1}$ is either send it to $Q_{i+1}$ or drop it (if $Q_{i+1} = Q_i$ then this send step is internal to $Q_i$). Which of these actions are taken depends on the specifics of the algorithm and also on the values $(Q_{i+1}, s_{i+1})$. In other words, an onion routing protocol cannot have an onion sent to incorrect destinations or fed as input to another algorithm.

Further, if $Q_i$ runs $\mathsf{ProcOnion}$ and outputs a message $m \neq \perp$, then this message becomes (ultimately, at the end of the protocol) part of $Q_i$'s output, i.e., it will be on the list of messages that have been sent to $Q_i$. In other words, $m$ cannot be internal to the protocol; it must be a message that someone sent to $Q_i$ via the protocol. Conversely, the only way that a message $m$ can be on the list of messages received by $Q_i$ is if $Q_i$ obtained it by peeling one of the onions it received.

These restrictions on protocol design are natural. Indeed, any implementation of onion routing would ensure that it is adhered to by using type checking of the objects created, sent, and processed by the algorithm. Without such a restriction, any protocol can be thought of

as an instance of onion routing protocol, so limiting our attention in this way is meaningful. Note that this places restrictions just on the protocol that the honest parties are executing; the adversary is still free to do anything he wishes: to mismatch types, to route onions incorrectly, to try to rewrap onions, to form and process onions adversarially, etc.

Onion routing serves a purpose: to route messages from senders to recipients. Therefore, it needs to satisfy correctness:

▶ **Definition 1** (Correctness). *A messaging protocol* $\Pi$ *is correct if in an interaction with a passive adversary, it delivers all the messages with overwhelming probability.*

In this paper, we will consider only correct onion routing protocols, but we will analyze their interactions with active adversaries.

Further, the protocols we design in this paper have an additional attractive property of being indifferent:

▶ **Definition 2** (Indifference). *An onion routing protocol* $\Pi$ *is indifferent if two properties hold: (i) The routing path corresponding to each honestly formed onion is of a fixed length. (ii) The sequence of intermediaries, including the recipients of dummy onions, and the sequence of nonces corresponding to each honestly formed onion do not depend on the input.*

The intuition behind this notion is that the contents of the messages sent and received between parties have no bearing on how the messages are routed and transmitted. For protocol design, indifference is an attractive property that allows components of an onion to be in place (and possibly the bulk of the cryptographic computation finished) before the message contents even becomes known. Another attractive feature of indifferent protocols is that their security properties are easier to analyze, as we will explore in the next section. Our negative results apply to all onion routing schemes, indifferent or not.

## 3 Security definitions: anonymity, equalizing, and mixing

**A motivating example.** Consider Ando, Lysyanskaya, and Upfal's very simple protocol $\Pi_p$ ($p$, for passive) in the passive adversary setting [3]. Recall that corrupted parties also follow the protocol in this setting. Let $\mathsf{Servers} \subseteq \mathcal{P}$ be the set of servers which is a subset of $\mathcal{P}$. During the *onion-forming phase*, every party $P$ generates an onion from the message-recipient pair $(m, R)$ in $P$'s input by first choosing $d-1$ servers $(S_1, \ldots, S_{d-1})$, each chosen independently and uniformly at random from $\mathsf{Servers}$. Next, $P$ forms an onion $O$ by running $\mathsf{FormOnion}$ on the message $m$, the routing path $P^{\rightarrow} = (S_1, \ldots, S_{d-1}, R)$, the public keys associated with $P^{\rightarrow}$, and the sequence of empty nonces. At the first round of the *execution phase*, each party $P$ sends its formed onion $O$ to the first server $S_1$ on the routing path. For every round $r \in [d]$, each server $S$ does the following: Between the $r^{th}$ and $(r+1)^{st}$ rounds, $S$ processes all the onions it received at the $r^{th}$ round. At the $(r+1)^{st}$ round, $S$ sends the processed onions to their respective next destinations. At the $d^{th}$ round, each party receives an onion that, once processed, reveals a message $m$ for the party.

$\Pi_p$ is anonymous if the protocol sufficiently shuffles the onions during the execution phase. In prior work [3], Ando, Lysyanskaya, and Upfal showed that sufficient shuffling occurs when the server load (i.e., the average number of onions received by a server at a round: $\frac{N}{|\mathsf{Servers}|}$) and the number of rounds (i.e., $d$) are both superlogarithmic in the security parameter. However, there is no parameter setting for which $\Pi_p$ can be anonymous from the *active* adversary. If $\kappa N$ out of $N$ participants are corrupted, then with probability $\kappa$, the adversary can determine the recipient of any honest party, say Alice: Suppose that during

the onion-forming phase, Alice picks a routing path that begins with an adversarial party $S_1$. During the execution phase, the adversary can direct $S_1$ to drop Alice's onion before the second round. In this case, the adversary can figure out who Alice's recipient is (say, it's Bob) by observing who does not receive an onion at the end of the protocol run.

The motivating example illustrates that while *mixing* (i.e. sufficiently shuffling onions) is helpful for achieving anonymity, it is not enough. To be anonymous, the protocol must also guarantee that the numbers of messages received by the parties don't reveal the input. We call this property, *equalizing*.

Here, we provide formal game-based definitions of anonymity (Section 3.1), equalizing (Section 3.2), and mixing (Section 3.3). Given these definitions, it can be shown that for indifferent onion routing protocols, equalizing and mixing imply anonymity:

▶ **Theorem 3.** *For any adversary class $\mathbb{A}$, an indifferent (Definition 2) onion routing protocol that mixes and equalizes for $\mathbb{A}$ in the simple I/O setting is anonymous for $\mathbb{A}$ in the simple I/O setting, provided that the underlying onion encryption is secure (i.e., UC-realizes the ideal functionality for onion encryption [2]).*

We omit the proof for brevity. We will use Theorem 3 to prove our upper bound in Section 6.3.

## 3.1 Anonymity

Anonymity is a property of a messaging protocol $\Pi$ (i.e., $\Pi$ doesn't have to be an onion routing protocol).

In the anonymity game (for defining anonymity), the adversary necessarily learns the corrupt parties' inputs and received messages. For example, let $N = 4$, and let $P_3$ be a corrupt party. Suppose that the adversary chooses as inputs $\sigma^0 = (\sigma_1^0, \sigma_2^0, \sigma_3^0, \sigma_4^0)$ and $\sigma^1 = (\sigma_1^1, \sigma_2^1, \sigma_3^1, \sigma_4^1)$ such that $\sigma_3^0 \neq \sigma_3^1$. Then, the adversary can determine the input from $P_3$'s input. Suppose that the adversary chooses as inputs $\sigma^0$ and $\sigma^1$ such that $\sigma^0$ contains an instruction to send message $m^0$ to $P_3$, whereas $\sigma^1$ contains an instruction to send message $m^1 \neq m^0$ to $P_3$. Then, the adversary can determine the input from $P_3$'s received message. Thus, the adversary's choice for $(\sigma^0, \sigma^1)$ is constrained to pairs of inputs that differ only in the honest parties' inputs and "outputs."

We define this formally by first defining equivalence classes for inputs as follows. Let $\Sigma$ be a set of input vectors. Let $\mathcal{A}$ be the adversary, and let $\mathsf{Bad}$ be the set of parties controlled by $\mathcal{A}$. Fixing $\mathsf{Bad}$ imposes an equivalence class on $\Sigma$. Each equivalence class is defined by a vector $(e_1, e_2, \ldots, e_N)$. For each corrupted party $P_i \in \mathsf{Bad}$, $e_i = (\sigma_i, \mathcal{M}_i)$ "fixes" the input $\sigma_i$ for $P_i$ and also, the set $\mathcal{M}_i$ of messages instructed to be sent from honest parties to $P_i$. For each honest party $P_i \in \mathcal{P} \setminus \mathsf{Bad}$, $e_i = V_i$ "fixes" the number $V_i$ of messages instructed to be sent from honest parties to $P_i$. An input vector belongs to the equivalence class $(e_1, e_2, \ldots, e_N)$ if for every $P_i \in \mathsf{Bad}$, the input for $P_i$ is $\sigma_i$, the set of messages from honest parties to $P_i$ is $\mathcal{M}_i$, and $e_i = (\sigma_i, \mathcal{M}_i)$; and if for every $P_i \in \mathcal{P} \setminus \mathsf{Bad}$, the number of messages from honest parties to $P_i$ is $V_i$, and $e_i = V_i$. Two input vectors $\sigma^0$ and $\sigma^1$ are equivalent w.r.t. the adversary's choice $\mathsf{Bad}$ for the corrupted parties, denoted $\sigma^0 \equiv_{\mathsf{Bad}} \sigma^1$, if they belong to the same equivalence class imposed by $\mathsf{Bad}$.

We now describe the anonymity game (below); the protocol $\Pi$ is anonymous if this induces indistinguishable adversarial views.

**The anonymity game.** $\mathsf{AnonymityGame}(1^\lambda, \Pi, \mathcal{A}, \Sigma)$ is parametrized by the security parameter $1^\lambda$, a protocol $\Pi$, an adversary $\mathcal{A}$, and a set $\Sigma$ of input vectors.

First, the adversary $\mathcal{A}$ and the challenger $\mathcal{C}$ set up the parties' keys: $\mathcal{A}$ chooses a subset $\mathsf{Bad} \subseteq \mathcal{P}$ of the parties to corrupt and sends $\mathsf{Bad}$ to the challenger $\mathcal{C}$. For each honest party in $\mathcal{P} \setminus \mathsf{Bad}$, $\mathcal{C}$ generates a key pair for the party; the public keys $\mathsf{pk}(\mathcal{P} \setminus \mathsf{Bad})$ of the honest parties are sent to $\mathcal{A}$. $\mathcal{A}$ picks the keys for the corrupted parties and sends the corrupted parties' public keys $\mathsf{pk}(\mathsf{Bad})$) to $\mathcal{C}$.

Next, the input is selected: $\mathcal{A}$ picks two input vectors $\sigma^0, \sigma^1 \in \Sigma$ such that $\sigma^0 \equiv_{\mathsf{Bad}} \sigma^1$ and sends them to $\mathcal{C}$. $\mathcal{C}$ chooses a random bit $b \leftarrow\!\!\$\, \{0,1\}$ and interacts with $\mathcal{A}$ in an execution of protocol $\Pi$ on input $\sigma^b$ with $\mathcal{C}$ acting as the honest parties adhering to the protocol and $\mathcal{A}$ controlling the corrupted parties.

At the end of the execution, $\mathcal{A}$ computes a guess $b'$ for $b$ from its view $\mathsf{V}^{\Pi,\mathcal{A},\mathsf{Bad}}(1^\lambda, \sigma^b)$ and wins the anonymity game if $b' = b$.

The standard notion of anonymity is defined as follows:

▶ **Definition 4** (Anonymity). *A messaging protocol $\Pi(1^\lambda, \mathsf{pp}, \mathsf{states}, \$, \sigma)$ is anonymous from the adversary class $\mathbb{A}$ w.r.t. the input set $\Sigma$ if every adversary $\mathcal{A} \in \mathbb{A}$ wins the anonymity game $\mathsf{AnonymityGame}(1^\lambda, \Pi, \mathcal{A}, \Sigma)$ with only negligible advantage, i.e., $\left|\Pr\left[\mathcal{A} \text{ wins } \mathsf{AnonymityGame}(1^\lambda, \Pi, \mathcal{A}, \Sigma)\right] - \frac{1}{2}\right| = \mathsf{negl}(\lambda)$.*

*The protocol is computationally (resp. statistically) anonymous if the adversaries in $\mathbb{A}$ are computationally bounded (resp. unbounded).*

## 3.2 Equalizing

Here, we introduce a new concept called equalizing, which is closely related to anonymity. Like anonymity, equalizing is a property of a messaging protocol $\Pi$.

Informally, $\Pi$ equalizes if observing how many messages each party received during the protocol run does not reveal whether the protocol ran on $\sigma^0$ or $\sigma^1$. In $\Pi_p$ (in our motivating example), whether Bob receives a message or not exposes who was sending Bob the message: Alice or another party, Allison; so $\Pi_p$ does not equalize. Instead, in an equalizing protocol, the probability that Bob receives a message doesn't depend on the sender's identity. Put another way, Bob is expected to receive the same number of messages in the scenario where Alice is the sender as the one where it is Allison. Formally, equalizing is defined with respect to the equalizing game (below).

**The equalizing game.** $\mathsf{EqualizingGame}(1^\lambda, \Pi, \mathcal{A}, \mathcal{D}, \Sigma)$ is parametrized by the security parameter $1^\lambda$, a protocol $\Pi$, an adversary $\mathcal{A}$, a distinguisher $\mathcal{D}$, and a set $\Sigma$ of input vectors.

The challenger for the equalizing game first interacts with the adversary exactly the same way as the challenger for the anonymity game. (See the previous section, Section 3.1, for the description of the anonymity game.) Recall that at the end of the anonymity game, each honest party $P_r$ outputs the set $\mathsf{O}_r^{\Pi,\mathcal{A}}(1^\lambda, \sigma^b)$ of (non-empty) messages from the message space $\mathcal{M}$ that it obtained during the execution from processing onions. Let $v_r$ be the *number* of messages that $P_r$ received during the run, i.e., $v_r \stackrel{\text{def}}{=} |\mathsf{O}_r^{\Pi,\mathcal{A}}(1^\lambda, \sigma^b)|$. (These statistics are part of the adversary's view in the anonymity game.)

We define the statistics for the corrupt parties differently since $\mathcal{C}$ does not get to observe how many messages the corrupt parties output; indeed it is not even clear what it means for a corrupt party to produce an output. For each recipient $P_r \in \mathsf{Bad}$, let $v_r$ correspond to the number of onions that $\mathcal{C}$ has routed to an adversarial participant $P'$ such that (1) they had been formed by an honest participant with $P_r$ as the recipient; and (2) all the participants after $P'$ on the remainder of this onion's route are controlled by the adversary. In other

words, $v_r$ is the number of onions from honest participants that $P_r$ would receive if, internal to the adversary, all the onions are processed and delivered to their next destinations. We define this formally below.

Let $\mathsf{msPairs}(P_r)$ denote the set of message-sender pairs for $P_r$. That is, for every $(m, P_s) \in \mathsf{msPairs}(P_r)$, the input $\sigma_s$ for $P_s$ includes the message-recipient pair $(m, P_r)$, i.e., $(m, P_r) \in \sigma_s$. Let $\mathsf{receivableOnions}(P_r)$ be the following set of onions: An onion $O$ is in $\mathsf{receivableOnions}(P_r)$ if there exists a message-sender pair $(m, P_s) \in \mathsf{msPairs}(P_r)$ such that

- **i.** $O$ was formed by $\mathcal{C}$ (on behalf of $P_s$) by running $\mathsf{FormOnion}$ on input the message $m$, a routing path $P^{\rightarrow} = (Q_1, \ldots, Q_{d-1}, P_r)$ ending in $P_r$, the public keys $\mathsf{pk}(P^{\rightarrow})$ of the parties on the path, and a sequence $s^{\rightarrow}$ of nonces, i.e., $O \in \{O_1, \ldots, O_d\}$ where $(O_1, \ldots, O_d) \leftarrow \mathsf{FormOnion}(m, (P^{\rightarrow}), \mathsf{pk}(P^{\rightarrow}), s^{\rightarrow})$;
- **ii.** letting $i$ denote the position of $O$ in the output of the $\mathsf{FormOnion}$ call, either $i = 1$, or the $(i-1)^{st}$ intermediary $Q_{i-1}$ on the routing path is honest; and
- **iii.** $O$ is "peelable all the way" by $\mathcal{A}$; i.e., $Q_i, \ldots, Q_{d-1}, P_r$ are all adversarial.

For each adversarial recipient $P_r \in \mathsf{Bad}$, we define the statistic $v_r$ to be the number of onions in $\mathsf{receivableOnions}(P_r)$ that the challenger sent out during the execution.

Let $\mathbf{v} = (v_1, v_2, \ldots, v_N)$. $\mathcal{C}$ provides these statistics $\mathbf{v}$ alone (and not the rest of the view) to the distinguisher $\mathcal{D}$, who outputs a guess $b'$ for the challenge bit and wins the game if $b' = b$, i.e. if it correctly determines whether the challenger ran the protocol on input $\sigma^0$ or $\sigma^1$. The definition for equalizing is as follows:

▶ **Definition 5** (Equalizing). *A messaging protocol $\Pi(1^\lambda, \mathsf{pp}, \mathsf{states}, \$, \sigma)$ equalizes for the adversary class $\mathbb{A}$ w.r.t. the input set $\Sigma$ if for every adversary $\mathcal{A} \in \mathbb{A}$ and distinguisher $\mathcal{D}$, $\mathcal{D}$ wins $\mathsf{EqualizingGame}(1^\lambda, \Pi, \mathcal{A}, \mathcal{D}, \Sigma)$ with negligible advantage, i.e., $\left|\Pr\left[\mathcal{D} \text{ wins } \mathsf{EqualizingGame}(1^\lambda, \Pi, \mathcal{A}, \mathcal{D}, \Sigma)\right] - \frac{1}{2}\right| = \mathsf{negl}(\lambda)$.*

*The protocol computationally (resp. statistically) equalizes if the adversaries and the distinguishers are computationally bounded (resp. unbounded).*

Clearly, a protocol that satisfies anonymity must equalize:

▶ **Theorem 6.** *For any adversary class $\mathbb{A}$, a protocol that is anonymous for $\mathbb{A}$ w.r.t. the input set $\Sigma$ equalizes for $\mathbb{A}$ w.r.t. $\Sigma$.*

**Proof.** If $\mathcal{D}$ can guess $b$ based on the statistics $\mathbf{v}$ alone, then the adversary $\mathcal{A}$ who has access to the entire view of its interaction with $\mathcal{C}$ can guess $b$ also. (It is also easy to see that a protocol need not satisfy anonymity in order to satisfy equalizing. Thus, equalizing is necessary but not sufficient to achieve anonymity.) ◀

## 3.3 Mixing in the simple I/O setting

Mixing is a property of *onion routing* protocols. Informally, an onion routing protocol mixes if the protocol sufficiently shuffles the honest parties' "message-bearing" onions. That is, once an honestly generated onion has traveled far enough, getting peeled at every intermediary, the adversary cannot trace it to the original sender. If the adversary is the recipient of the message contained in the onion, it should not be able to trace it to the sender provided the message itself does not reveal the sender.

Formally, mixing is defined with respect to the mixing game. To keep things simple, we present the definition in the simple I/O setting, but this can be extended to any arbitrary input set.

**The mixing game.** Let $\mathcal{OE} = (\mathsf{Gen}, \mathsf{FormOnion}, \mathsf{ProcOnion})$ be a secure onion encryption scheme. $\mathsf{MixingGame}(1^\lambda, \Pi, \mathcal{A})$ is parametrized by the security parameter $1^\lambda$, an onion routing protocol $\Pi$, and an adversary $\mathcal{A}$.

First, the adversary $\mathcal{A}$ and the challenger $\mathcal{C}$ set up the parties' keys (exactly as we described above for the anonymity game): $\mathcal{A}$ chooses a subset $\mathsf{Bad} \subseteq \mathcal{P}$ of the parties to corrupt and sends $\mathsf{Bad}$ to $\mathcal{C}$. For each honest party in $\mathcal{P} \setminus \mathsf{Bad}$, $\mathcal{C}$ generates a key pair for the party by running the onion encryption scheme's key generation algorithm $\mathsf{Gen}$ and sends the public keys $\mathsf{pk}(\mathcal{P} \setminus \mathsf{Bad})$ of the honest parties to the adversary $\mathcal{A}$. $\mathcal{A}$ picks the keys for the corrupted parties and sends the public-key portions $\mathsf{pk}(\mathsf{Bad})$ to $\mathcal{C}$.

Next, the input is selected: $\mathcal{A}$ identifies a set $\mathsf{S} \subseteq \mathcal{P} \setminus \mathsf{Bad}$ of honest *target senders* and a set $\mathsf{R} \subseteq \mathcal{P}$, $|\mathsf{R}| = |\mathsf{S}|$ of *target receivers*. In addition to $\mathsf{S}$ and $\mathsf{R}$, $\mathcal{A}$ also decides *part* of the input; for every non-target sender $P_s \in \mathcal{P} \setminus \mathsf{S}$, $\mathcal{A}$ chooses a message $m$ and a unique non-target recipient $P_r \in \mathcal{P} \setminus \mathsf{R}$ such that $P_s$'s input becomes $\sigma_s = \{(m, P_r)\}$; and for every target recipient $P_r \in \mathsf{R}$, $\mathcal{A}$ chooses a message $m_r$ to be sent to $P_r$. We call the portion of the input that $\mathcal{A}$ decides "*the partial input vector*," and denote it $\tilde{\sigma}$. $\mathcal{A}$ sends $(\mathsf{S}, \mathsf{R}, \tilde{\sigma})$ to the challenger $\mathcal{C}$. $\mathcal{C}$ supplies the rest of the input vector $\sigma = (\sigma_1, \ldots, \sigma_N)$ by choosing a random bijection $g$ from $\mathsf{S}$ to $\mathsf{R}$; each $P_s \in \mathsf{S}$ is instructed to send the message $m_{g(P_s)}$ to $g(P_s) \in \mathsf{R}$, i.e., $\sigma_s = \{(m_{g(P_s)}, g(P_s)\}$ where the message $m_{g(P_s)}$ was supplied by $\mathcal{A}$ as part of the partial input vector.

Next, $\mathcal{C}$ interacts with $\mathcal{A}$ in an execution of protocol $\Pi$ on input $\sigma$ with $\mathcal{C}$ acting as the honest parties adhering to the protocol and $\mathcal{A}$ controlling the corrupted parties. Whenever the protocol $\Pi$ specifies for an onion to be formed or processed, $\mathcal{C}$ runs the onion encryption scheme's onion-forming algorithm $\mathsf{FormOnion}$ or onion-processing algorithm $\mathsf{ProcOnion}$.

Let $\mathsf{O}_\mathsf{R}$ be the set of onions received by the parties in $\mathsf{R}$.

At the end of the execution, $\mathcal{A}$ chooses two onions $O_s, O_{\bar{s}} \in \mathsf{O}_\mathsf{R}$ and a target sender $P_s \in \mathsf{S}$ and outputs $(O_s, O_{\bar{s}}, P_s)$.

Let an onion $O$ be a "*valid challenge onion*" if (i) there exists a message $m_r \in \mathcal{M}$ and a target recipient $P_r \in \mathsf{R}$ such that $m_r$ is $\mathcal{A}$'s choice for the message to be sent to $P_r$, and (ii) $O$ is the last onion to be received by the recipient over the network in the onion evolution generated by $\mathcal{C}$ on behalf of one of the target senders running $\mathsf{FormOnion}$ on the message $m_r$ and a routing path ending in $P_r$.

Let $\mathsf{sender}(O_s)$ be the sender of $O_s$, and let $\mathsf{sender}(O_{\bar{s}})$ be the sender of $O_{\bar{s}}$. To maximize his chances of winning the game, the adversary wants both $O_s$ and $O_{\bar{s}}$ to be valid challenge onions such that $O_s$ was sent by $P_s$, while $O_{\bar{s}}$ was not. Formally, if $\mathcal{A}$ chose two valid challenge onions, and $\{P_s\} \subset \{\mathsf{sender}(O_s), \mathsf{sender}(O_{\bar{s}})\} \subseteq \mathsf{S}$, then $\mathcal{A}$ wins iff $P_s = \mathsf{sender}(O_s)$. Otherwise, if $\mathcal{A}$ did not choose two valid challenge onions, or if $\{P_s\} \not\subset \{\mathsf{sender}(O_s), \mathsf{sender}(O_{\bar{s}})\}$ or $\{\mathsf{sender}(O_s), \mathsf{sender}(O_{\bar{s}})\} \not\subseteq \mathsf{S}$, then $\mathcal{A}$ wins with probability one-half. See Figure 1 for a quick reference to the mixing game.

We now define mixing as follows.

▶ **Definition 7** (Mixing). *An onion routing protocol* $\Pi(1^\lambda, \mathsf{pp}, \mathsf{states}, \$, \sigma)$ *mixes conditioned on the event* $E$ *for the adversary class* $\mathbb{A}$ *if given* $E$*, every adversary* $\mathcal{A} \in \mathbb{A}$ *wins* $\mathsf{MixingGame}(1^\lambda, \Pi, \mathcal{A})$ *with negligible advantage, i.e.,* $\left|\Pr\left[\mathcal{A} \text{ wins } \mathsf{MixingGame}(1^\lambda, \Pi, \mathcal{A}) \mid E\right] - \frac{1}{2}\right| = \mathsf{negl}(\lambda)$.

*The protocol computationally (resp. statistically) mixes if the adversaries in* $\mathbb{A}$ *are computationally bounded (resp. unbounded).*

Now that we have defined mixing formally, let us walk the reader through our definitional choices. The starting intuition is that this definition needs to capture that it should be hard for the adversary to pinpoint the origin of an onion received by one of the target recipients.

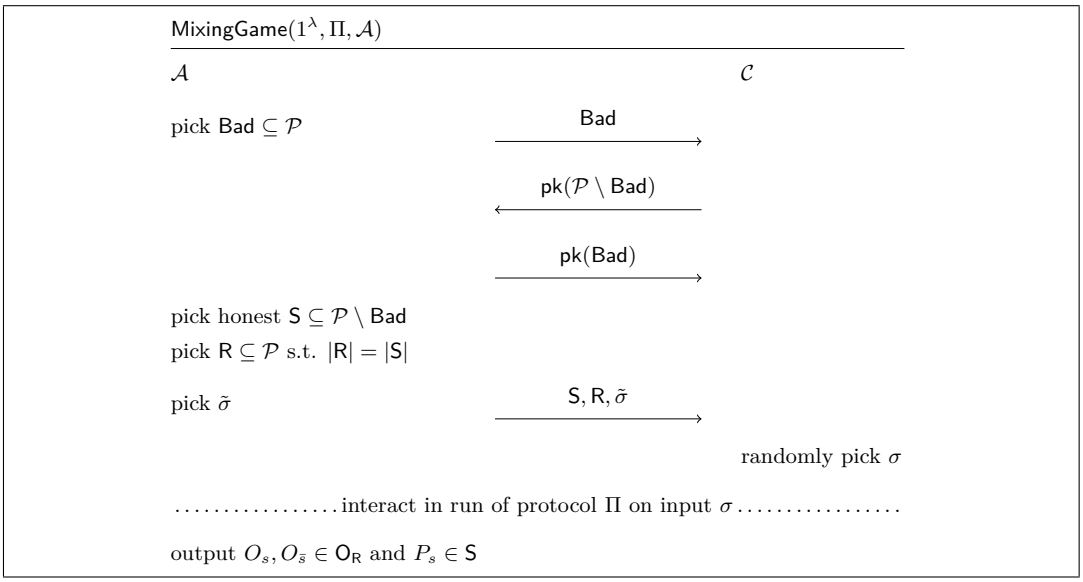


**Figure 1** Schematic of the mixing game.

This goal comes with a caveat that of course an adversary can determine the sender of an onion that one of the target senders has just created, or, more generally, that hasn't traveled very far and hasn't had a chance to mix with any onions from other target senders. Hence, we need to restrict the set of onions on which the adversary can win to a set of onions that have traveled far and have already had a chance to mix with other onions. This is why we have the requirement that the onion be a valid challenge onion. Intuitively, a valid challenge onion is one that was formed by a target sender and has already arrived at its destination, a target recipient, and now the adversary's job is to figure out where it came from.

Next, let us explain why, to win the game, the adversary must produce two valid challenge onions, and correctly attribute one of them to a sender $P_s$, while the other must have originated with another target sender. What does it mean that the adversary cannot trace an onion? One intuitive approach would be to say: the adversary's chances of winning the game where he picks just one onion and guesses its origin are close to a simulator's chances of winning a game where he just guesses a sender, and the challenger picks the onion uniformly at random and independently of the simulator's guess. The problem with this approach is that we don't know the best strategy for such a simulator and with what probability it would succeed. So our approach is to have the adversary pick a sender and two onions. "Mixing" means that, if it so happens that exactly one of them comes from $P_s$ and the other comes from another target sender, then try as he may, the adversary cannot tell which is which any better than by guessing randomly; and if it doesn't happen that way, then the adversary wins with probability one-half.

## 4 Main tools: checkpoint onions and merging onions

We describe the main ingredients for our constructions: checkpoint onions (a tool that was introduced in prior work [3]) and a new tool: merging onions.

## 4.1 Checkpoint onions

Our goal is to achieve anonymity by ensuring that our protocol mixes and equalizes in the presence of an active adversary that drops onions. The challenge is: if the adversary drops too many onions, then the remaining ones don't have enough onions to mix with, and so the resulting protocol will not mix. Checkpoint onions give the honest participants a way of checking that there are still enough onions in the system for mixing to be possible.

A *checkpoint onion* $O$ is a dummy onion (containing the empty message $\perp$) formed by a party $P$ that travels through the network until, at a pre-determined checkpoint round $r$, it arrives at the intermediary $I$, who is expecting it. If it fails to arrive, then $I$ is alerted to the activity of an active adversary. More precisely, let $F_\cdot(\cdot,\cdot)$ be a pseudo-random function over two inputs, keyed by $\mathsf{sk}(P,I)$ which is a secret key shared between $P$ and $I$. Let $b$ be a binary predicate. Let $\mathcal{D}$ be the *diagnostic rounds*; the honest parties test whether enough onions remain in the system after these rounds. For each intermediary $I$ and each round $r \in \mathcal{D}$, $P$ determines whether or not to create a checkpoint onion that will arrive at $I$ at round $r$ by computing $f = F(\mathsf{sk}(P,I),(r,0))$ and then checking if $b(f) = 1$; if so, $P$ creates this checkpoint onion. Similarly, the intermediary $I$ will know to expect a checkpoint onion from $P$ at round $r$ by computing $f = F(\mathsf{sk}(P,I),(r,0))$ and then checking if $b(f) = 1$.

$P$ forms $O$ by running $\mathsf{FormOnion}$ on input the empty message $\perp$, a randomly chosen routing path $P^{\rightarrow} = (I_1,\ldots,I_d)$, the public keys associated with parties on $P^{\rightarrow}$, and a sequence $(s_1,\ldots,s_{d-1})$ of nonces. The nonce $s_r$ which will be received by $I$, is the value that $I$ will know to expect: $s_r = F(\mathsf{sk}(P,I),(r,1))$; the rest are random nonces. The reason that $I$ will know to expect $s_r$ is that $I$ can compute it too, since $\mathsf{sk}(P,I)$ is shared between $P$ and $I$. Of course, the shared key $\mathsf{sk}(P,I)$ need not be set up in advance: it can be generated from an existing PKI, e.g., using Diffie-Hellman.

If the adversary drops an onion belonging to the same evolution as $O$ before it reaches $I$, $I$ will detect it: it will detect that no onion with nonce $s_r$ was received in round $r$. (Since $F$ is pseudorandom, it is highly unlikely that another onion peels to the same nonce value.)

## 4.2 Merging onions

Checkpoint onions help with mixing, but not with equalizing. If our routing protocol just has every sender form one "message-bearing" onion to its recipient and send it along in addition to a set of checkpoint onions (as in the protocol $\Pi_a$ of Ando, Lysyanskaya, and Upfal [3]), then an adversary who targets the sender Alice can cause Alice's recipient Bob to receive the message with a smaller probability than her alternative recipient, Bill; so this protocol will not equalize and, from Theorem 6, has no hope of achieving anonymity.

So how can we design a protocol that equalizes? One approach is to detect when the adversary drops any onions at all (e.g., using verifiable shuffling) [25, 30] and abort when that happens. While this approach equalizes, it is not at all fault-tolerant. To achieve fault tolerance and equalizing, the protocol must be able to react to the adversary dropping onions in a way that is less dramatic than total abort. This can be accomplished by using a new tool: merging onions. The idea here is that a sender $P$ can create two onions, $O_1$ and $O_2$ that bear the same message to the same recipient $R$. Further, they will be routed through the same intermediary $I$, arriving at $I$ at the same round $r$. Let $O_1'$ (resp. $O_2'$) denote the $r^{th}$ layer of $O_1$ (resp. $O_2$) that arrives at $I$ at round $r$. When $I$ peels both $O_1'$ and $O_2'$, $I$ discovers that they are (essentially) the same onion, and only forwards one of them to the next destination. If $I$ receives just one of them (because the other one had been dropped by the adversary), then it forwards it to the next destination too.

Why does this approach help with equalizing? Suppose we have a protocol in which every participant creates two message-bearing onions that merge at round $r$. Suppose that the adversary targets the sender Alice and succeeds in dropping one of her two outgoing merging onions. Since these onions were supposed to merge at round $r$, after round $r$, there are just as many onions for which Alice was the sender (namely, just one onion) as for any other participant. In general, of course, the adversary may drop more than one onion belonging to Alice. In fact, in order to guarantee that any of Alice's onions survive with overwhelming probability when the adversary controls a constant fraction of the network's nodes, Alice needs to send out a superlogarithmic (in the security parameter $\lambda$) number of onions. In order to equalize the number of onions that make it to each destination, our protocol will have to create not a pair, but $2^h = \Omega(\text{polylog}\,\lambda)$ merging onions, organized in a binary tree of height $h$.

We now illustrate how to form $2^h$ merging onions through a toy example for $h = 3$. We first construct a binary tree graph of height $3 = \log 8$. We label the root vertex of the tree $v$, and the left-child and right-child of $v$, $v_0$, and $v_1$. More generally, the left-child of a vertex $v_w$ is $v_{w0}$, and the right-child of $v_w$ is $v_{w1}$, so that the leaf vertices are: $v_{000}$, $v_{001}$, $v_{010}$, $v_{011}$, $v_{100}$, $v_{101}$, $v_{110}$, and $v_{111}$. Each of these leaf vertices corresponds to a separate onion.

Let $y$ denote a fixed number of rounds; this will later correspond to the length of an "epoch." Next, for each vertex $v_i$ of the graph, we choose a random sequence $I_i^{\rightarrow} = (I_i^1, \ldots, I_i^y)$ of $y$ parties and a random sequence $s_i^{\rightarrow} = (s_i^1, \ldots, s_i^y)$ of $y$ nonces, i.e., $\forall j \in [y]$, $I_i^j \leftarrow\!\!\$\, \mathcal{P}$ and $s_i^j \leftarrow\!\!\$\, \mathcal{S}$. Let the "*direct path from a leaf vertex $v_\ell$ to the root*" be the path that begins with $v_\ell$ and recursively moves to its parent vertex until the root vertex $v$ is reached. For example, the direct path from $v_{101}$ to the root is $(v_{101}, v_{10}, v_1, v)$. Let the "*sequence of intermediaries corresponding to leaf vertex $v_\ell$*" be the sequence of parties corresponding to the parties on the direct path from $v_\ell$ to the root, e.g., for $v_{101}$, it is $(I_{101}^{\rightarrow}, I_{10}^{\rightarrow}, I_1^{\rightarrow}, I^{\rightarrow})$, where $I^{\rightarrow}$ is the sequence of parties assigned to the root. Let the "*sequence of nonces corresponding to leaf vertex $v_\ell$*" be the sequence of nonces corresponding to the parties on the direct path from $v_\ell$ to the root, e.g., for $v_{101}$, it is $(s_{101}^{\rightarrow}, s_{10}^{\rightarrow}, s_1^{\rightarrow}, s^{\rightarrow})$, where $s^{\rightarrow}$ is the sequence of nonces assigned to the root.

For each leaf vertex $v_\ell$, we form an onion $O_\ell^1$ using the message $m$ from the input, the routing path $(I_{101}^{\rightarrow}, I_{10}^{\rightarrow}, I_1^{\rightarrow}, I^{\rightarrow}, R)$ where $R$ is the recipient from the input, the public key associated with the routing path, and the sequence $(s_{101}^{\rightarrow}, s_{10}^{\rightarrow}, s_1^{\rightarrow}, s^{\rightarrow})$ of nonces. We can generalize this idea to generate an arbitrarily large set of merging onions by using an appropriately large binary tree.

## 5 A stepping stone construction, $\Pi_\triangle$

Let us extend the toy example construction we just saw to a protocol, $\Pi_\triangle^{x,y,t}$, which is a stepping stone for our main construction. $\Pi_\triangle^{x,y,t}$ is pronounced "Pi-tree" from the fact that the onions' routing paths are structured like a binary tree graph and is parametrized by the number $x$ of merging onions per sender (this is also the expected number of checkpoint onions per sender), the number $y$ of rounds per epoch, and the threshold $t$ for missing checkpoint nonces per diagnostic round. (We will generally omit the superscript for better readability.)

We use a secure onion encryption scheme $\mathcal{OE} = (\mathsf{Gen}, \mathsf{FormOnion}, \mathsf{ProcOnion})$ as a building block. During the setup phase, the participants set up their keys. Every honest party $P$ sets up his/her keys $(\mathsf{pk}(P), \mathsf{sk}(P))$ by running the onion encryption scheme's key generation algorithm $\mathsf{Gen}$.

**The onion-forming phase.** During the onion-forming phase, each honest party $P$ creates two types of onions: merging onions and checkpoint onions.

- On input $\{(m, R)\}$, $P$ forms a set of $x$ merging onions using the number $y$ of rounds in an epoch, the message $m$, and the recipient $R$.
- In addition to merging onions, $P$ generates (on average) $x$ checkpoint onions using the set $\mathcal{D} = \{y, 2y, \ldots, (\log x + 1)y\}$ as the diagnostic rounds. (Appropriate functions are chosen for $F.(\cdot,\cdot)$ and $b(\cdot)$ such that $P$ generates $x$ checkpoint onions in expectation. See *Checkpoint onions* in Section 4 to recall how these functions are used for generating checkpoint onions.)

For both merging onions and checkpoint onions, the length of the routing path is fixed; it is $(\log x + 1)y + 1$.

**The execution phase.** All onions are created during the onion-forming phase and released simultaneously in the first round of the execution phase.

- After each round $r$ of the execution phase, $P$ peels all onions it received at the $r^{th}$ round and merges mergeable onions (i.e., if two onions peel to the same nonce value, drop one of them at random).
- If $r$ is a diagnostic round (i.e., $r \in \mathcal{D}$), $P$ runs the following diagnostic test: Let $\mathsf{Ckpts}(P, r)$ denote the set of checkpoints that $P$ expects to see from peeling the onions between rounds $r$ and $r + 1$. $P$ counts how many checkpoints from $\mathsf{Ckpts}(P, r)$ are missing. If the number exceeds a fixed threshold value $t$, then $P$ aborts. Otherwise, $P$ continues for another round by sending the processed onions to their respective next destinations in random order.
- At the end of the execution phase, $P$ peels the onions it received at the last round and outputs the set of (non-empty) messages it received.

▶ Remark 8. $\Pi_{\triangle}^{x,y,t}$ is anonymous from the adversary who corrupts up to $\kappa$ fraction of the parties when (i) the onion encryption scheme is secure, (ii) the number $x$ of onions formed by each (honest) party is $\Omega\big(2^{\lceil \log(\chi(\log \chi+1))\rceil}\big)$ where $\chi = \max(\sqrt{N \log^{2+\epsilon} \lambda}, \log^{2(1+\epsilon)} \lambda)$, (iii) the number $y$ of rounds per epoch is $\Omega\big(\log^{1+\epsilon} \lambda\big)$, and (iv) the threshold $t$ is $2(1-\delta)(1-\kappa)^3 \kappa \log^{1+\epsilon} \lambda$. (See the full version of the paper for the proof.) The reason that $\Pi_{\triangle}^{x,y,t}$ needs so many onions is that the adversary can target Alice and drop a lot of her onions before the honest participants realize (via checkpoint onions) the presence of an attack and abort. The protocol $\Pi_{\bowtie}$ presented in the next section improves on this by giving the routing paths enough structure that missing onions can be detected sooner.

# 6 Our main construction, $\Pi_{\bowtie}$

In this section, we present our main construction $\Pi_{\bowtie}$ (pronounced "Pi-butterfly"). $\Pi_{\bowtie}$ uses a variant of a butterfly graph described below.

## 6.1 The butterfly network and variants

Recall [26, Chapter 4.5.2] that the *butterfly network* $B = (V(B), E(B))$ is a directed graph on $(n+1)2^n$ vertices. The vertices are organized into $N = 2^n$ rows and $n + 1$ columns, so each vertex has an address $(r, c)$ where $1 \le r \le N$ and $0 \le c \le n$. Vertices in column $i$ represent potential locations of a data packet (here, an onion) at epoch $i$; each participant $P$ has a dedicated row. An edge from $(P, i)$ to $(Q, i + 1)$ means that an onion can travel from participant $P$ to participant $Q$ in epoch $i$. The edges of the specific butterfly network that will be useful for us are $E(B) = \{((P, i), (Q, i + 1)) \mid P = Q$ or binary representations of $P$ and $Q$ differ in position $i + 1$ only$\}$.

Let $J$ and $J'$ be two participants whose binary representation differs in bit $i+1$ only. In $\Pi_{\bowtie}$, epoch $i$ is dedicated to having an onion bounce $y$ times between $J$ and $J'$. This way, by the end of the epoch, the onions that $J$ and $J'$ held at the beginning of the epoch will be mixed together if one of them is honest. More formally, the onions travel along the edges of a *stretched* butterfly network, defined as follows: its $N(ny+1)$ vertices are organized into $N$ rows and $ny+1$ columns; and its edges are: $E(\beta) = \{((P,j),(Q,j+1)) \mid \text{for } i = \lfloor j/y \rfloor, ((P,i),(Q,i+1)) \in E(B)\}$.

However, what if both $J$ and $J'$ are adversarial? Then sending the onions through the stretched butterfly network just once will result in the adversary knowing the $i^{th}$ bit of an onion's destination! So to prevent this, we will send the onions through the *iterated* stretched butterfly network. For an integer $z$, let $\beta^z$ denote the stretched butterfly network iterated $z$ times. More precisely, $\beta^z$ is a directed graph in which the vertices are organized into $N$ rows and $nyz+1$ columns, i.e., a vertex has an address $(r,c)$ where $1 \le r \le N$ and $0 \le c \le nyz$. The edges are as follows: $E(\beta^z) = \{((P,j),(Q,j+1)) \mid \text{for } i = j \bmod ny, ((P,i),(Q,i+1)) \in E(\beta)\}$.

To summarize, we begin with a butterfly network $B$, then we stretch it by $y$ to get $\beta$, then we iterate it $z$ times to get $\beta^z$; see Figure 2. By a "*walk through* $\beta^z$" we mean a sequence $(J_0, \ldots J_{nyz})$ such that, for each $i < nyz$, $((J_i,i),(J_{i+1},i+1)) \in E(\beta^z)$. A random walk from a node $J_0$ is a sequence that begins with $J_0$ such that for $i > 0$, each $J_i$ is a walk selected uniformly at random conditioned on the first $i$ elements being $(J_0, \ldots, J_{i-1})$. A random walk starting at any address can be sampled efficiently.

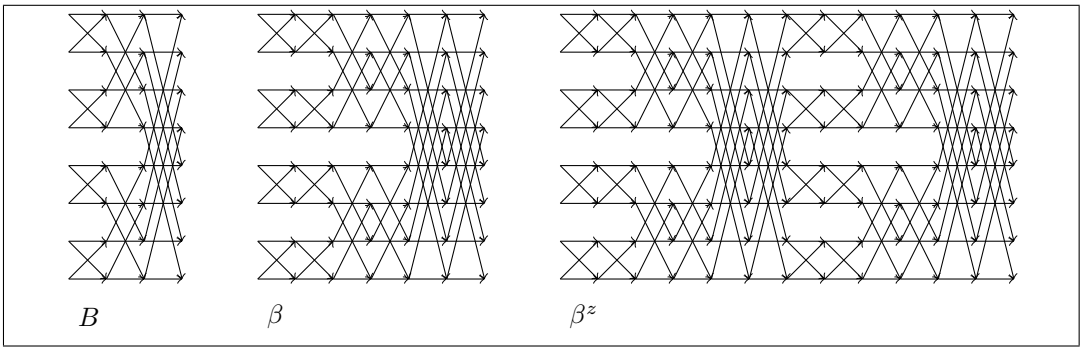


**Figure 2** Diagrams of the butterfly network $B$, the stretched butterfly network $\beta$, and the iterated stretched butterfly network $\beta^z$ for $n = \log(8) = 3$, and $y = z = 2$.

## 6.2 Description of the construction

$\Pi_{\bowtie}^{x,y,z,t}$ consists of setup, the onion-forming phase, and the execution phase. It is parameterized by the number $x$ of merging onions per sender, the number $y$ of rounds per epoch, the number $z$ of iterations of a variant of a butterfly graph, and the threshold $t$ for missing checkpoint nonces. The execution phase is further divided into the *mixing sub-phase* and the *equalizing sub-phase*. The iterated stretched butterfly graph determines routing options for the mixing sub-phase.

Let $\mathcal{OE} = (\mathsf{Gen}, \mathsf{FormOnion}, \mathsf{ProcOnion})$ be a secure onion encryption scheme. During setup, each honest participant $P$ generates its public key pair $(\mathsf{pk}(P), \mathsf{sk}(P))$ using $\mathsf{Gen}$.

### 6.2.1 The onion-forming phase

On input $\{(m, R)\}$, each honest party $P$ generates exactly $x$ merging onions and (on average) $x$ checkpoint onions. To form an onion, $P$ first needs to pick a path for it. Each onion will (potentially) travel to $d \stackrel{\text{def}}{=} (nyz + 1) + y \log x + 1$ parties to reach its destination: the first $nyz + 1$ steps involve a random walk through the iterated stretched butterfly network (the mixing sub-phase), and the next $y \log x + 1$ steps will take the onion through the equalizing sub-phase and to the recipient.

To begin with, $P$ generates the $x$ merging onions as follows: Let $T$ be the binary tree of height $\log x$. Let $k$ be an address of a node in $T$ (i.e., $k$ is a binary string of length at most $\log x$); let $v_k$ denote this node. I.e., $V(T) = \{v_k \mid k \text{ is a binary string, } |k| \leq \log x\}$. To each non-leaf vertex $v_k$ in $T$, $P$ assigns a sequence of $y$ random parties and $y$ random nonces; let $I^{\rightarrow}_{v_k} = (I_{v_k,1}, \ldots, I_{v_k,y})$ denote the sequence of vertices and $s^{\rightarrow}_{v_k} = (s_{v_k,1}, \ldots, s_{v_k,y})$ denote the sequence of nonces corresponding to vertex $v_k$. (Up until this step, this is exactly how merging onions are formed in $\Pi_{\triangle}$.) For each leaf vertex $v_\ell$, $P$ picks a random walk through the iterated stretched butterfly $\beta^z$ and $nyz + 1$ random nonces; let $J^{\rightarrow}_{v_\ell} = (J_{v_\ell,0}, \ldots, J_{v_\ell,nyz})$, denote the random walk, and let $t^{\rightarrow}_{v_\ell} = (t_{v_\ell,0}, \ldots, t_{v_\ell,nyz})$ be the sequence of nonces.

Let $v_\ell$ be a leaf of $T$. Let $v_{\ell,i} = v_{k_i}$ where $k_i$ is the $i$-bit prefix of $\ell$. I.e. $v_{\ell,\ell} = v_\ell$ and $v_{\ell,0} = v_\varepsilon$, and $(v_{\ell,h}, v_{\ell,h-1}, \ldots, v_{\ell,0})$ is the path from $v_\ell$ to the root of the tree, where $h = \log x$. $P$ will create an onion $O_\ell$ for each leaf $v_\ell$. Its routing path is $I^{\rightarrow}_\ell = (J^{\rightarrow}_{v_\ell}, I^{\rightarrow}_{\ell,2}, \ldots, I^{\rightarrow}_{\ell,h}, R)$ where $J^{\rightarrow}_{v_\ell}$ is as defined above, $I^{\rightarrow}_{\ell,i} = I^{\rightarrow}_{k_i}$ where $k_i$ is the $i$-bit prefix of $\ell$, and $R$ is the recipient, and such that $|I^{\rightarrow}_\ell| = d$. Similarly, let $s^{\rightarrow}_\ell = (t^{\rightarrow}_{v_\ell}, s^{\rightarrow}_{\ell,2}, \ldots, s^{\rightarrow}_{\ell,h})$ denote the sequence of nonces corresponding to this path. To form the onion $O_\ell$ corresponding to $v_\ell$, $P$ runs the algorithm FormOnion on the message $m$, the routing path $I^{\rightarrow}_\ell$, the public keys associated with the routing path, and the nonce sequence $s^{\rightarrow}_\ell$.

After forming the merging onions, $P$ generates the checkpoint onions. Just as in $\Pi_{\triangle}$, the execution phase consists of epochs, and the last round of every epoch is a diagnostic round. Here, each epoch lasts $y$ rounds, thus round $r > 0$ is a diagnostic round if $r$ is a multiple of $y$. For each diagnostic round $r$ and for each intermediary $I$, $P$ uses the pseudorandom function $F_{\mathsf{sk}(P,I)}(r, 0)$ to determine whether to form a checkpoint onion to send to $I$ at round $r$, and if so, calculates the nonce $s = F_{\mathsf{sk}(P,I)}(r, 1)$.

When $F_{\mathsf{sk}(P,I)}(r, 0) = 1$, $P$ generates a checkpoint onion to be verified by party $I$ in round $r$. Recall that $d \stackrel{\text{def}}{=} (nyz + 1) + y \log x + 1$; so round $d$ is the last round of the execution phase. Since the checkpoint onion should not be distinguishable from a merging one during the mixing sub-phase, it needs to travel over the edges of the iterated stretched butterfly network for the first $nyz + 1$ rounds, and follow a random path through the network during the equalizing sub-phase, all the way until the last round $d$.

As a result, for $r \geq nyz + 1$, $P$ generates the routing path by first picking a random walk $J^{0 \to nyz} = (J_0, \ldots, J_{nyz})$ through the iterated stretched butterfly network starting at a random node $J_0$, and then choosing each participant on the next part of the path $J^{nyz+1 \to r-1} = (J_{nyz+1}, \ldots, J_{r-1})$ uniformly at random from $\mathcal{P}$. Next, $J_r = I$, and each router on the remaining stretch of the path $J^{r+1 \to d}$ is, again, chosen uniformly at random from $\mathcal{P}$. So the resulting routing path is $J^{\rightarrow}_{I,r} = (J^{0 \to nyz}, J^{nyz+1 \to r-1}, J_r, J^{r+1 \to d})$. $P$ chooses the corresponding nonces $\{s_{I,r,j}\}_{j \in \{0,\ldots,d-1\} \setminus \{r\}}$ uniformly at random, sets $s_{I,r,r} = s$, and gives the resulting routing path, sequence $(s_{I,r,0}, \ldots, s_{I,r,d-1})$ of nonces and the empty message to FormOnion to obtain checkpoint onion $O_{I,r}$.

If $r \leq nyz$, then round $r$ occurs during the mixing sub-phase, as the onion is making its way through the butterfly network. So its path has to be formed in such a way that it arrives at $I$ at round $r$; but it needs to be a randomly chosen path conditioned on this event (so that

a checkpoint onion's path is distributed the same way as one of a merging onion). Let $J^{0\to nyz}$ be a random walk through $\beta^z$ that is at address $I$ at round $r$. Let each intermediary in the sequence $J^{nyz+1\to d}$ be chosen uniformly at random from $\mathcal{P}$. Again, for $j \neq r$, $0 \leq j \leq d-1$, the nonce $s_{I,r,j}$ is chosen at random, while $s_{I,r,r} = s$. Let $J^{\to}_{I,r} = (J^{0\to nyz}, J^{nyz+1\to d})$. Run FormOnion on input the routing path $J^{\to}_{I,r}$, sequence of nonces $s^{\to}_{I,r}$ and the empty message to obtain checkpoint onion $O_{I,r}$.

▶ Remark 9. In both $\Pi_{\triangle}$ and $\Pi_{\bowtie}$, the onion layers are tagged with their respective round number to prevent replay attacks. If by peeling an onion received at round $r$, an honest relaying party observes a round number $r' \neq r$, the party drops the onion. (We can, therefore, assume that replay attacks do not happen. We can safely do so since the security of the onion encryption scheme prevents the adversary from modifying the onions formed by honest participants in any meaningful way. See, for example Ando and Lysyanskaya's work on onion encryption [2], for a sufficiently strong construction.)

### 6.2.2 The execution phase

At the beginning of the execution phase, each party $P$ is *live*. $P$'s status will change from live to *aborted* if it ever receives a special abort message from another party. An aborted party sends the special abort message to a random sample of $x$ parties. (A slight technicality is that, since all messages must be onions, the abort message is a specially formed onion.)

- For each $r \in \{0, \ldots, d-1\}$, each live honest party $P$ first peels all the onions it received at the $r^{th}$ round. It merges onions that are mergeable: if it received two onions that have the same nonce, then it drops one of them, selected at random, and sends the other one to its next destination.
- If $r$ is a diagnostic round (i.e., a multiple of $y$), then $P$ runs the diagnostic test: $P$ compares the number of checkpoint onions it expects to receive with the number it received. For every participant $Q \in \mathcal{P}$, if $F_{\mathsf{sk}(Q,P)}(r,0) = 1$, then $P$ expects to receive a checkpoint onion with nonce $s = F_{\mathsf{sk}(Q,P)}(r,1)$ in this round. In the mixing sub-phase, if fewer than $t$ checkpoint onions are missing so far in the protocol run (not just in this round, but cumulatively), then $P$ continues the run by processing all the other onions. Otherwise, $P$'s status changes: it is no longer live but becomes an *aborted* party. In the equalizing sub-phase, change status to aborted if there are $t$ or more missing checkpoint onions in this round, else continue.
- At the last round (round $d$) of the execution phase, $P$ peels the onions it received and outputs the set of (non-empty) messages it received.

## 6.3 Proof that $\Pi_{\bowtie}$ is anonymous, robust, and efficient

In this section, we will prove that there exists a parameter setting (for $x$, $y$, $z$, and $t$) such that $\Pi_{\bowtie}$ is simultaneously anonymous, fault-tolerant, and efficient.

Our measure of efficiency is *onion cost per user*, which measures how many onions are transmitted by each user in the protocol. This is an appropriate measure when the parties pass primarily onions to each other. It is also an attractive measure of complexity because it is algorithm-independent: If we measured complexity in bits, it would change depending on which underlying encryption scheme was used. Since an onion contains as many layers as there are intermediaries, its bit complexity scales linearly with the number of intermediaries. (We assume that every message $m$ can be contained in a single onion.) To translate our lower bound from onion complexity to bits, we will consider onions to be at least as long (in bits) as the message $m$ being transmitted and the routing information. More formally,

▶ **Definition 10** (Onion cost). *Let* $\mathsf{out}_i^{\Pi,\mathcal{A}}(1^\lambda, \sigma)$ *denote the number of onions formed by an honest party that party* $P_i$ *transmits directly to another party in a protocol run of* $\Pi$ *with adversary* $\mathcal{A}$*, security parameter* $\lambda$ *and* $\sigma$*. The onion cost of* $\Pi$ *is* $\mathsf{OC}^{\Pi,\mathcal{A}}(1^\lambda, \Sigma) \stackrel{\text{def}}{=} \mathbb{E}_{\sigma,i,\$}\left[\mathsf{out}_i^{\Pi,\mathcal{A}}(1^\lambda, \sigma)\right]$*. The expectation is taken over the input* $\sigma \leftarrow\!\!\$\, \Sigma$*, the party* $P_i \leftarrow\!\!\$\, \mathcal{P}$*, and the randomness* $\$$ *of the protocol.*

*For an adversary class* $\mathbb{A}$*, the onion cost of* $\Pi$ *interacting with* $\mathbb{A}$ *w.r.t.* $\Sigma$ *is the maximum onion cost over the adversaries in* $\mathbb{A}$*, i.e.,* $\mathsf{OC}^{\Pi,\mathbb{A}}(1^\lambda, \Sigma) \stackrel{\text{def}}{=} \max_{\mathcal{A}\in\mathbb{A}} \mathsf{OC}^{\Pi,\mathcal{A}}(1^\lambda, \Sigma)$.

Our formal notion of fault tolerance is *robustness*, defined below:

▶ **Definition 11** (Robustness). *A messaging protocol* $\Pi$ *is robust if in every interaction in which the adversary drops at most a logarithmic (in the security parameter) number of message packets,* $\Pi$ *delivers all messages sent out by honest participants w.o.p.*

Let $\mathbb{A}_\kappa$ denote the class of active adversaries who can corrupt up to a constant $\kappa$ fraction of the participants. In this section, we will prove the following upper bound on onion cost:

▶ **Theorem 12.** *For any constants* $\kappa < \frac{1}{2}$ *and* $\gamma_1, \gamma_2 > 0$*, there is a setting of* $x$*,* $y$*,* $z$*, and* $t$ *such that* $\Pi_{\bowtie}^{x,y,z,t}$ *is robust and anonymous from the adversary class* $\mathbb{A}_\kappa$ *with onion cost at most* $\gamma_1 \log N \log^{3+\gamma_2} \lambda$ *(in the presence of* $\mathbb{A}_\kappa$*), where* $\lambda$ *is the security parameter and* $N = \omega(\log \lambda)$ *is the number of participants.*

**Proof.** Recall that the number of corruptions is $\kappa < \frac{1}{2}$. Set $\epsilon_1$ such that $\gamma_1 = 6\epsilon_1^3$ and $\epsilon_2$ such that $\gamma_2 = 3\epsilon_2$. Let $x = y = z = \epsilon_1 \log^{1+\epsilon_2} \lambda$. Let an onion (layer) be *commutable* if (i) an honest party formed it, and (ii) it is not a checkpoint onion for verification by an adversarial party (more precisely, it does not belong to the same evolution as a checkpoint onion for verification by an adversarial party); and let $t = \frac{W}{3}$, where $W = \frac{(1-\kappa)x}{z \log N + \log x}$ is the expected number of commutable checkpoint nonces at a party at a diagnostic round.

Having set the parameters, we wish to show that the protocol $\Pi_{\bowtie}^{x,y,z,t}$ (a) is robust; (b) has onion cost $\mathsf{OC} \leq \gamma_1 \log N \log^{3+\gamma_2} \lambda$; and (c) is anonymous, provided that the underlying onion encryption scheme is secure.

Part (a) is true by inspection.

To see why (b) follows, recall that each participant forms $x$ merging onions and, on average, $x$ checkpoint onions; let $X$ be the maximum number of onions formed by an honest party. Each of these onions will need to be processed in each round, so $\mathsf{OC} \leq Xd$, where $d$ is the number of rounds. Using Chernoff bounds, $X < 3x$ with overwhelming probability. The number of rounds is $d = (nyz + 1) + y \log x + 1$; for our setting of parameters, therefore, $\mathsf{OC} \leq 6\epsilon_1^3 \log N \log^{3(1+\epsilon_2)} \lambda$.

We show part (c) via a series of lemmas that follow. First, we invoke the UC composition theorem of Canetti [8] in order to replace cryptographic algorithms for onion encryption with ideal encryption; this allows our further analysis to assume that onions reveal nothing to an intermediary $I$ other than the information that is intended for $I$ (Lemma 13). Let $\Pi_{\bowtie}^{ideal}$ be the resulting protocol. Next, we argue that $\Pi_{\bowtie}^{ideal}$ is an indifferent onion routing protocol (Lemma 15). This is helpful because then we will be able to invoke Theorem 3. Third, we discard, for the purposes of analysis, all the checkpoint onions that are checked by the adversary; we show that if a protocol mixes (resp. equalizes) in this setting, then it mixes (resp. equalizes). Finally, we show that in this setting, $\Pi_{\bowtie}$ mixes (Lemma 16) and equalizes (Lemma 17). Then, putting it all together, we get our desired result. ◀

▶ **Lemma 13.** *Let* $\Pi$ *be onion routing protocol that makes use of an onion encryption scheme that is UC-secure [8] under a computational assumption* $A$*. Let* $\Pi^{ideal}$ *be the same protocol, but the onion encryption scheme is replaced by the ideal onion encryption functionality of Camenisch and Lysyanskaya [7]. If* $\Pi^{ideal}$ *is anonymous, then* $\Pi$ *is anonymous under assumption* $A$*.*

**Proof.** The Lemma follows by the UC composition theorem of Canetti [8]. ◀

▶ Remark 14. Since CCA2-secure public-key encryption UC-realizes the ideal public-key encryption functionality of Canetti, and in $\Pi_{\bowtie}$, the adversary already knows how many layers of a given onion have already been peeled, forming onions by using CCA2-secure encryption to encrypt each layer will also result in an anonymous $\Pi_{\bowtie}$.

▶ **Lemma 15.** $\Pi^{ideal}_{\bowtie}$ *is indifferent.*

**Proof.** In $\Pi^{ideal}_{\bowtie}$, the length of each routing path is fixed, and the intermediaries and nonces of honestly formed onion layers do not depend on the input $\sigma$ to the protocol. The procedure for generating intermediaries and nonces takes as input only the values $x$, $y$, and $z$. Thus, by definition, $\Pi^{ideal}_{\bowtie}$ is indifferent. ◀

For the subsequent lemmas (Lemmas 16-18), we analyze only *commutable* onions.

▶ **Lemma 16.** *With parameters* $x$*,* $y$*,* $z$*, and* $t$ *defined as above,* $\Pi^{ideal}_{\bowtie}$ *mixes for the adversary who corrupts up to half of the parties.*

**Proof sketch.** If $\Pi^{ideal}_{\bowtie}$ delivers messages in the final round $d$, then w.o.p., the adversary dropped (at most) a constant fraction of the commutable checkpoint onions before the last epoch: The adversary cannot drop more than a constant fraction of all commutable onions without also dropping a proportional number of checkpoint onions. This is because if the adversary were to drop more than a constant fraction of all commutable onions, then, from known probability concentration bounds [22], w.o.p., the adversary would drop close to a proportional number of checkpoint onions, which, in turn, would cause all honest parties to abort the run. Combining this with Chernoff bounds we get: during each round of the penultimate epoch $e$, each honest party processed a polylogarithmic (in the security parameter) number of commutable onions. From Chernoff bounds, we also get: during epoch $e$, each commutable onion went to an honest party a polylogarithmic number of times. Thus, either the $\Pi^{ideal}_{\bowtie}$ aborts, or it sufficiently shuffles the commutable onions during the penultimate epoch since shuffling for a polylogarithmic number of rounds with a polylogarithmic number of other onions is sufficient for mixing. Either way, $\Pi^{ideal}_{\bowtie}$ mixes. ◀

▶ **Lemma 17.** *With parameters* $x$*,* $y$*,* $z$*, and* $t$ *defined as above,* $\Pi^{ideal}_{\bowtie}$ *equalizes for the adversary who corrupts up to half of the parties, who also receives everything about non-commutable onions as an auxiliary input.*

Before proving Lemma 17, let us prove the following:

▶ **Lemma 18.** *Let* $\Pi^{ideal}_{\bowtie}$ *run with parameters* $x$*,* $y$*,* $z$*, and* $t$ *are as defined above on input* $\sigma$*, with* $\mathcal{A}$ *corrupting up to half of the participants, and receiving an auxiliary input about non-commutable onions as an auxiliary input. If there is an unaborted honest party at the beginning of the equalizing phase, then with overwhelming probability for each honest party* $P$*, at least* $\frac{1-\kappa}{9}$ *of* $P$*'s merging onions remained undropped by the adversary at the end of the mixing phase. (Recall that* $\kappa$ *is the corruption rate.)*

**Proof sketch.** In the first round, the adversary $\mathcal{A}$ knows the sender of each commutable onion. As the protocol progresses, $\mathcal{A}$ loses track of this information. Thus, $\mathcal{A}$'s optimal tactic is to target Alice upfront by dropping every onion that might have come from Alice that is routed to an adversarial party during the first three rounds of the first epoch (as well as the last round of the epoch).

In the first round, some of Alice's onions route to a corrupt party; $\mathcal{A}$ drops all of these. However, from Chernoff bounds, w.o.p., at least a constant fraction of Alice's onion go to an honest party first. Let $O$ be such an onion, and let $P$ be the honest party that receives $O$ in the first round. Recall that during each epoch of the mixing phase, $P$ shuffles onions back and forth with another party $P'$. $\mathcal{A}$ can attempt to drop $O$ if $P'$ is corrupt. However, even if $P'$ is corrupt, $\mathcal{A}$ cannot drop $O$ if it arrives at $P$ first and remains at $P$ during rounds 2 and 3 (and return to $P$ at round $y$) – so, using probability concentration bounds, $\frac{1-\kappa}{9}$ of the time. Thus, even if $\mathcal{A}$ employs the optimal tactic for dropping Alice's onions, (at least) $\frac{1-\kappa}{9}$ of Alice's onions will make it to the equalizing phase. Since $\mathcal{A}$ cannot do better than this, this proves Lemma 18. ◀

**Proof sketch of Lemma 17.** From Lemma 18, if $\Pi_{\bowtie}^{ideal}$ continues into the equalizing phase, then a constant fraction of each honest party's merging onions are still in play at the start of the equalizing phase. However, Lemma 18 does not guarantee that there will be an epoch $i > nyz$ such that the number of Alice's merging onions at epoch $i$, $\mathsf{numMO}_{\mathsf{Alice},i}$, will be close to that of Allison's, $\mathsf{numMO}_{\mathsf{Allison},i}$. To prove that $\Pi_{\bowtie}^{ideal}$ equalizes, we need to show that there exists an epoch $i \leq d$ such that (for any two parties Alice and Allison), $\mathsf{numMO}_{\mathsf{Alice},i} \approx \mathsf{numMO}_{\mathsf{Allison},i}$. If $\mathcal{A}$ doesn't drop any commutable onions during the equalizing phase, then this condition is satisfied by the merging of onions.

So what can $\mathcal{A}$ do? The only information that $\mathcal{A}$ has for guessing where any commutable onion came from is which onions are part of a mergeable pair and which are not; this is because the onions are shuffled during the mixing phase and each epoch of the equalizing phase. Let a *singleton* be a commutable onion that is not part of a mergeable pair; note that it can be either a checkpoint onion or a merging onion. W.l.o.g., suppose that $\mathcal{A}$ dropped more of Alice's onions upfront (during the mixing phase) than Allison's. Then, at the start of the equalizing phase, it is likely that more singletons are Alice's merging onions than Allison's merging onions. So, $\mathcal{A}$ can attempt to prevent the numbers of merging onions from evening out by dropping singletons. We can show that the best that $\mathcal{A}$ can do is to drop as many singletons as possible (without causing the protocol to be aborted) at the beginning of the equalizing phase. (Of course, $\mathcal{A}$ could also drop onions that belong in a mergeable pair, but this would only help to even out the numbers of merging pairs.) Even if $\mathcal{A}$ does this, there exists an epoch $i \leq d$ such that $\mathsf{numMO}_{\mathsf{Alice},i} \approx \mathsf{numMO}_{\mathsf{Allison},i}$.

Armed with Lemma 18 and the above analysis, we can prove that $\Pi_{\bowtie}^{ideal}$ equalizes. If the adversary drops too many onions during the mixing phase, then $\Pi_{\bowtie}^{ideal}$ equalizes since every honest party stops participating (Lemma 18), and so no one receives their message. Otherwise, $\Pi_{\bowtie}^{ideal}$ equalizes since enough of each sender's merging onions make it to the equalizing phase (Lemma 18), and the numbers of merging onions are eventually evened out by the merging of onions (above). ◀

## 7 Our lower bound: polylog onion cost is required

In this section, we present our lower bound: an onion routing protocol can be anonymous from the active adversary only if the onion cost is superlogarithmic in the security parameter. Our lower bound holds for protocols that are minimally functional for the active adversary.

We call this notion weakly robustness, defined below. The reason this definition is weaker than robustness (Definition 11) is that here we only insist that the protocol guarantee delivery for senders whose onions are never dropped.

▶ **Definition 19** (Weakly robust). *Let* $\Pi(1^\lambda, \mathsf{pp}, \mathsf{states}, \$, \sigma)$ *be an onion routing protocol and let* $\mathcal{A}$ *be an adversary attacking* $\Pi$ *that drops at most* $\mathcal{O}(\log(\lambda))$ *onions.* $\Pi$ *is weakly robust if whenever* $\mathcal{A}$ *doesn't drop any onions sent by honest party* $P$*,* $P$*'s message will be delivered to its recipient with overehlming probability.*

▶ **Theorem 20.** *If the onion routing protocol* $\Pi(1^\lambda, \mathsf{pp}, \mathsf{states}, \$, \sigma)$ *is weakly robust and (computationally) anonymous from the adversary* $\mathcal{A}$ *who corrupts up to a constant fraction of the parties and drops at most* $f(\lambda) = \mathcal{O}(\log(\lambda))$ *onions, then the onion cost of* $\Pi$ *interacting with* $\mathcal{A}$ *is* $\omega(f(\lambda))$*.*

**Proof sketch.** Let us give the intuition for the proof of this theorem. If an honest $P_i$ sends out only $\mathcal{O}(\log(\lambda))$ onions, then an adversary that chooses which participants to corrupt uniformly at random has a $1/\lambda^{\mathcal{O}(1)}$ chance of controlling each and every participant that ever receives an onion directly from $P_i$. (This is because $\mathcal{O}(\log(\lambda)) = \mathcal{O}(\log(N))$, since $\lambda$ and $N$ are polynomially related.) Thus with non-negligible probability it can cut off $P_i$ entirely by dropping all of the onions it sends out, guaranteeing that the intended recipient of $P_i$'s message never receives the message; yet, by weak robustness (Definition 19), we can show that there will be some recipient whose probability of receiving his message is high. Therefore, $\Pi$ will not equalize (Definition 5): based on who failed to receive the message, it is possible to determine whether $P_i$'s intended recipient was Bob or Bill. Since it does not equalize, by Theorem 6, it is not anonymous. See the full version of this paper for the proof. ◀

## 8 Conclusion and future work

Here, we mention a few extensions of our results: We proved that the required onion cost for an onion routing protocol to provide robustness and (computational) anonymity from the active adversary is polylogarithmic in the security parameter. Our proof for the lower bound can be used to prove the stronger result that polylogarithmic onion cost is required even when (i) the adversary observes the traffic on only $\Theta(1)$ fraction of the links and or when (ii) the security definition is weakened to (computational) differential privacy. (iii) Also, while we explicitly showed this to be the case for the simple I/O setting, the result holds more generally whenever any party can send a message to any other party.

We also proved the existence of a robust and anonymous onion routing protocol with polylogarithmic (in the security parameter) onion cost. (iv) This result also extends beyond the simple I/O setting; our onion routing protocol is anonymous w.r.t. any input set where the size of each party's input is fixed.

There is a small gap between our lower and upper bounds. A natural direction for future work is to close this gap.

## References

1 Mário S. Alvim, Miguel E. Andrés, Konstantinos Chatzikokolakis, Pierpaolo Degano, and Catuscia Palamidessi. Differential privacy: on the trade-off between utility and information leakage. In *FAST 2011*, pages 39–54. Springer, 2011.

2 Megumi Ando and Anna Lysyanskaya. Cryptographic shallots: A formal treatment of repliable onion encryption. *IACR Cryptol. ePrint Arch.*, 2020:215, 2020. URL: https://eprint.iacr.org/2020/215.

3 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. doi:10.4230/LIPIcs.ICALP.2018.144.

4 Michael Backes, Ian Goldberg, Aniket Kate, and Esfandiar Mohammadi. Provably secure and practical onion routing. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 369–385. IEEE, 2012.

5 Michael Backes, Aniket Kate, Praveen Manoharan, Sebastian Meiser, and Esfandiar Mohammadi. Anoa: A framework for analyzing anonymous communication protocols. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 163–178. IEEE, 2013.

6 Ron Berman, Amos Fiat, and Amnon Ta-Shma. Provable unlinkability against traffic analysis. In Ari Juels, editor, *FC 2004*, volume 3110 of *LNCS*, pages 266–280. Springer, Heidelberg, February 2004.

7 Jan Camenisch and Anna Lysyanskaya. A formal treatment of onion routing. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 169–187. Springer, Heidelberg, August 2005. doi:10.1007/11535218_11.

8 Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. doi:10.1109/SFCS.2001.959888.

9 Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 42–51. ACM, 1993. doi:10.1145/167088.167105.

10 Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. *Information and Computation*, 206(2–4):378–401, 2008.

11 David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981. doi:10.1145/358549.358563.

12 David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, January 1988. doi:10.1007/BF00206326.

13 Miranda Christ. New lower bounds on the complexity of provably anonymous onion routing. Undergraduate honors thesis, Brown University, Providence, RI 02912 USA, 2020.

14 David A. Cooper and Kenneth P. Birman. Preserving privacy in a network of mobile computers. In *1995 IEEE Symposium on Security and Privacy*, pages 26–38. IEEE Computer Society Press, 1995. doi:10.1109/SECPRI.1995.398920.

15 Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE Computer Society Press, May 2015. doi:10.1109/SP.2015.27.

16 Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure multiparty computation*. Cambridge University Press, 2015.

17 Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In *2018 IEEE Symposium on Security and Privacy*, pages 108–126. IEEE Computer Society Press, May 2018. doi:10.1109/SP.2018.00011.

18 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.

19 Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 523–540. Springer, Heidelberg, May 2004. doi:10.1007/978-3-540-24676-3_31.

**20** Oded Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, Cambridge, UK, 2001.

**21** Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987. doi:10.1145/28395.28420.

**22** Don Hush and Clint Scovel. Concentration of the hypergeometric distribution. *Statistics & Probability Letters*, 75(2):127–132, 2005.

**23** Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul F. Syverson. Users get routed: traffic correlation on tor by realistic adversaries. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 337–348. ACM Press, November 2013. doi:10.1145/2508859.2516651.

**24** Christiane Kuhn, Martin Beck, and Thorsten Strufe. Breaking and (partially) fixing provably secure onion routing. *CoRR*, abs/1910.13772, 2019. arXiv:1910.13772.

**25** 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. doi:10.1145/3132747.3132755.

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

**27** Lasse Øverlier and Paul Syverson. Locating hidden servers. In *2006 IEEE Symposium on Security and Privacy*, pages 100–114. IEEE Computer Society Press, May 2006. doi:10.1109/SP.2006.24.

**28** Charles Rackoff and Daniel R. Simon. Cryptographic defense against traffic analysis. In *25th ACM STOC*, pages 672–681. ACM Press, May 1993. doi:10.1145/167088.167260.

**29** 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.

**30** 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, Shanghai, China, October 28-31, 2017*, pages 423–440. ACM, 2017. doi:10.1145/3132747.3132783.

**31** 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. doi:10.1145/2815400.2815417.

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