XRD: Scalable Messaging System with Cryptographic Privacy

Albert Kwon, *MIT;* David Lu, *MIT PRIMES;* Srinivas Devadas, *MIT*

https://www.usenix.org/conference/nsdi20/presentation/kwon

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Open access to the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20) is sponsored by

XRD: Scalable Messaging System with Cryptographic Privacy

Albert Kwon *MIT*
David Lu *MIT PRIMES*
Srinivas Devadas *MIT*

Abstract

Even as end-to-end encrypted communication becomes more popular, private messaging remains a challenging problem due to metadata leakages, such as who is communicating with whom. Most existing systems that hide communication metadata either (1) do not scale easily, (2) incur significant overheads, or (3) provide weaker guarantees than cryptographic privacy, such as differential privacy or heuristic privacy. This paper presents XRD (short for Crossroads), a metadata private messaging system that provides cryptographic privacy, while scaling easily to support more users by adding more servers. At a high level, XRD uses multiple mix networks in parallel with several techniques, including a novel technique we call aggregate hybrid shuffle. As a result, XRD can support 2 million users with 228 seconds of latency with 100 servers. This is 13.3× and 4× faster than Atom and Pung, respectively, which are prior scalable messaging systems with cryptographic privacy.

1 Introduction

Many Internet users today have turned to end-to-end encrypted communication like TLS [18] and Signal [40], to protect the *content* of their communication in the face of widespread surveillance. While these techniques are starting to see wide adoption, they unfortunately do not protect the *metadata* of communication, such as the timing, the size, and the identities of the end-points. In scenarios where the metadata are sensitive (e.g., a government officer talking with a journalist for whistleblowing), encryption alone is not sufficient to protect users' privacy.

Given its importance, there is a rich history of works that aim to hide the communication metadata, starting with mix networks (mix-nets) [10] and dining-cryptographers networks (DC-Nets) [11] in the 80s. Both works provide formal privacy guarantees against global adversaries, which has inspired many systems with strong security guarantees [14, 55, 35, 53]. However, mix-nets and DC-nets require the users' messages to be processed by either centralized servers or every user in the system, making them difficult to scale to millions of users. Systems that build on them typically inherit the scalability limitation as well, with overheads increasing (often superlinearly) with the number of users or servers [14, 55, 35, 53]. For private communication systems, however, supporting a large user base is imperative to providing strong security; as aptly stated by prior works, "anonymity loves company" [20, 49]. Intuitively, the adversary's goal of learning information about a user naturally becomes harder as the number of users increases.

As such, many recent messaging systems have been targeting scalability as well as formal security guarantees. Systems like Stadium [52] and Karaoke [37], for instance, use differential privacy [22] to bound the information leakage on the metadata. Though this has allowed the systems to scale to more users with better performance, both systems leak a small bounded amount of metadata for each message, and thus have a notion of "privacy budget". A user in these systems then spends a small amount of privacy budget every time she sends a sensitive message, and eventually is not guaranteed strong privacy. Users with high volumes of communication could quickly exhaust this budget, and there is no clear mechanism to increase the privacy budget once it runs out. Scalable systems that provide stronger cryptographic privacy like Atom [34] or Pung [5], on the other hand, do not have such a privacy budget. However, they rely heavily on expensive cryptographic primitives such as public key encryption and private information retrieval [3]. As a result, they suffer from high latency, in the order of ten minutes or longer for a few million users, which impedes their adoption.

This paper presents a point-to-point metadata private messaging system called XRD that aims to marry the best aspects of prior systems. Similar to several recent works [5, 34, 52], XRD scales with the number of servers. At the same time, the system cryptographically hides all communication metadata from an adversary who controls the entire network, a constant fraction of the servers, and any number of users. Consequently, it can support virtually unlimited amount of communication without leaking privacy against such an adversary. Moreover, XRD only uses cryptographic primitives that are significantly faster than the ones used by prior works, and can thus provide lower latency and higher throughput than prior systems with cryptographic security.

A XRD deployment consists of many servers. These servers are organized into many small chains, each of which acts as a local mix-net. Before any communication, each user creates a *mailbox* that is uniquely associated with her, akin to an e-mail address. In order for two users Alice and Bob to have a conversation in the system, they first pick a number of chains using a specific algorithm that XRD provides. The algorithm guarantees that *every* pair of users intersects at one of the chains. Then, Alice and Bob send messages addressed to their own mailboxes to all chosen chains, except to the chain where their choices of chains align, where they send their messages for each other. Once all users submit their messages, each chain shuffles and decrypts the messages, and forwards the shuffled messages to the appropriate mailboxes. Intuitively, XRD protects the communication metadata because (1) every pair of users is guaranteed to meet at

a chain which makes it equally likely for any pair of users to be communicating, and (2) the mix-net chains hide whether a user sent a message to another user or herself.

One of the main challenges of XRD is addressing active attacks by malicious servers, where they tamper with some of the users' messages. This challenge is not new to our system, and several prior works have employed expensive cryptographic primitives like verifiable shuffle [14, 55, 35, 52, 34] or incurred significant bandwidth overheads [34] to prevent such attacks. In XRD, we instead propose a new technique called *aggregate hybrid shuffle* that can verify the correctness of shuffling more efficiently than traditional techniques.

XRD has two significant drawbacks compared to prior systems. First, with N servers, each user must send $O(\sqrt{N})$ messages in order to ensure that every pair of users intersects. Second, because each user sends $O(\sqrt{N})$ messages, the workload of each XRD server is $O(M/\sqrt{N})$ for M users, rather than $O(M/N)$ like many prior scalable messaging systems [34, 52, 37]. Thus, prior systems could outperform XRD in deployment scenarios with large numbers of servers and users, since the cost of adding a single user is higher and adding servers is not as beneficial in XRD.

Nevertheless, our evaluation suggests that XRD outperforms prior systems with cryptographic guarantees if there are less than a few thousand servers in the network. XRD can handle 2 million users (comparable to the number of daily Tor users [1]) in 228 seconds with 100 servers. For Atom [34] and Pung [5, 4], two prior scalable messaging systems with cryptographic privacy, it would take over 50 minutes and 15 minutes, respectively. (These systems, however, can defend against stronger adversaries, as we detail in §2 and §7.) Moreover, the performance gap grows with more users, and we estimate that Atom and Pung require at least 1,000 servers in the network to achieve comparable latency with 2 million or more users. While promising, we find that XRD is not as fast as systems with weaker security guarantees: Stadium [52] and Karaoke [37], for example, would be 3× and 23× faster than XRD, respectively, in the same deployment scenario. In terms of user costs, we estimate that 40 Kbps of bandwidth is sufficient for users in a XRD network with 2,000 servers, and the bandwidth requirement scales down to 1 Kbps with 100 servers.

In summary, we make the following contributions:

- Design and analyze XRD, a metadata private messaging system that can scale by distributing the workload across many servers while providing cryptographic privacy.
- Design a technique called aggregate hybrid shuffle that can efficiently protect users' privacy under active attacks.
- Implement and evaluate a prototype of XRD on a network of commodity servers, and show that XRD outperforms existing cryptographically secure designs.

2 Related work

In this section, we discuss related work by categorizing the prior systems primarily by their privacy properties, and also discuss the scalability and performance of each system.

Systems with cryptographic privacy. Mix-nets [10] and DC-Nets [11] are the earliest examples of works that provide cryptographic (or even information theoretic) privacy guarantees against global adversaries. Unfortunately, they have two major issues. First, they are weak against active attackers: adversaries can deanonymize users in mix-nets by tampering with messages, and can anonymously deny service in DC-Nets. Second, they do not scale to large numbers of users because all messages must be processed by either a small number of servers or every user in the system. Many systems that improved on the security of these systems against active attacks [14, 55, 35, 53] suffer from similar scalability bottlenecks. Riposte [13], a system that uses "private information storage" to provide anonymous broadcast, also requires all servers to handle a number of messages proportional to the number of users, and thus faces similar scalability issues.

A recent system Atom [34] targets both scalability and strong anonymity. Specifically, Atom can scale *horizontally*, allowing it to scale to larger numbers of users simply by adding more servers to the network. At the same time, it provides sender anonymity [46] (i.e., no one, including the recipients, learns who sent which message) against an adversary that can compromise any fraction of the servers and users. However, Atom employs expensive cryptography, and requires the message to be routed through hundreds of servers in series. Thus, Atom incurs high latency, in the order of tens of minutes for a few million users.

Pung [5, 4] is a system that aims to provide metadata private messaging between honest users with cryptographic privacy. This is a weaker notion of privacy than that of Atom, as the recipients (who are assumed to be honest) learn the senders of the messages. However, unlike most prior works, Pung can provide private communication even if *all servers* are malicious by using a cryptographic primitive called computational private information retrieval (CPIR) [12, 3]. Its powerful threat model comes unfortunately at the cost of performance: Though Pung scales horizontally, the amount of work required per user is proportional to the total number of users, resulting in the total work growing superlinearly with the number of users. Moreover, PIR is computationally expensive, resulting in throughput of only a few hundred or thousand messages per minute per server.

Systems with differential privacy. Vuvuzela [53] and its horizontally scalable siblings Stadium [52] and Karaoke [37] aim to provide differentially private (rather than cryptographically private) messaging. At a high level, they hide the communication patterns of honest users by inserting dummy messages that are indistinguishable from real messages, and reason carefully about how much information is leaked at

each round. They then set the system parameters such that they could support a number of sensitive messages; for instance, Stadium and Karaoke target 10^4 and 10^6 messages, respectively. Up to that number of messages, the systems allow users to provide a plausible cover story to "deny" their actual actions. Specifically, the system ensures that the probability of Alice conversing with Bob from the adversary's perspective is within e^{ε} (typically, $e^{\varepsilon} \in [3, 10]$) of the probability of Alice conversing with any other user with only a small failure probability δ (typically, $\delta = 0.0001$). This paradigm shift has allowed the systems to support larger numbers of users with lower latency than prior works.

Unfortunately, systems with differential privacy suffer from two drawbacks. First, the probability gap between two events may be sufficient for strong adversaries to act on. For instance, if Alice is ten times as likely to talk to Bob than Charlie, the adversary may act assuming that Alice is talking to Bob, despite the plausible deniability. Second, there is a "privacy budget" (e.g., 10^4 to 10^6 messages), meaning that a user can deny a limited number of messages with strong guarantees. Moreover, for best possible security, users must constantly send messages, and deny every message. For instance, Alice may admit that she is not in any conversation (thinking this information is not sensitive), but this could have unintended consequences on the privacy of another user who uses the cover story that she is talking with Alice. The budget could then run out quickly if users want the strongest privacy possible: If a user sends a message every minute, she would run out of her budget in a few days or years with 10^4 to 10^6 messages. Although the privacy guarantee weakens gradually after the privacy budget is exhausted, it is unclear how to raise the privacy levels once they have been lowered.

These shortcomings can particularly affect journalists and their sources. Many journalists mention the importance of long-term relationships with their sources for their journalistic process [41, 43], and the difficulty of maintaining private relationships with them. In a differentially private system, if the journalist and the source are ten times as likely to be talking as two other users, then the adversary might simply assume the journalist and the source's relationship. Furthermore, these relationships often last many years [43], which could cause the privacy budget to run out. This may put the journalist and the source in jeopardy.

Scalable systems with other privacy guarantees. The only private communication system in wide-deployment today is Tor [21]. Tor currently supports over 2 million daily users using over 6,000 servers [1], and can scale to more users easily by adding more servers. However, Tor does not provide privacy against an adversary that monitors significant portions of the network, and is susceptible to traffic analysis attacks [19, 30]. Its privacy guarantee weakens further if the adversary can control some servers, and if the adversary launches active attacks [29]. Similar to Tor, most free-route mix-nets [44, 24, 49, 15, 39] (distributed mix-nets where each messages is routed through a small subset of servers) cannot provide strong privacy against powerful adversaries due to traffic analysis and active attacks.

Loopix [47] is a recent iteration on free-route mix-nets, and can provide fast asynchronous messaging. To do so, each user interacts with a semi-trusted server (called "provider" in the paper), and routes her messages through a small number of servers (e.g., 3 servers). Each server inserts small amounts of random delays before routing the messages. Loopix then reasons about privacy using entropy. Unfortunately, the privacy guarantee of Loopix weakens quickly as the adversary compromises more servers. Moreover, Loopix requires the recipients to trust the provider to protect themselves.

3 System model and goals

XRD aims to achieve the best of all worlds by providing cryptographic metadata privacy while scaling horizontally without relying on expensive cryptographic primitives. In this section, we present our threat model and system goals.

3.1 Threat model and assumptions

A deployment of XRD would consist of hundreds to thousands of servers and a large number of users, in the order of millions. Similar to several prior works on distributed private communication systems [34, 52], XRD assumes an adversary that can monitor the entire network, control a fraction f of the servers, and control up to all but two honest users. We assume, however, that there exists a public key infrastructure that can be used to securely share public keys of online servers and users with all participants at any given time. These keys, for example, could be maintained by key transparency schemes [36, 42, 51].

XRD does not hide the fact that users are using XRD. Thus, for best possible security, users should stay online to avoid intersection attacks [31, 16]. XRD also does not protect against large scale denial-of-service (DoS) attacks. It can, however, recover from a small number of benign server failures or disruptions from malicious users. In addition, XRD provides privacy even under DoS, server churn, and user churn (e.g., Alice goes offline unexpectedly without her conversation partner knowing). We discuss the availability properties further in §5 and §7.3.

Finally, XRD assumes that the users can agree to start talking at a certain time out-of-band. This could be done, for example, via two users exchanging this information offline, or by using systems like Alpenhorn [38] that can initiate conversations privately.

Cryptographic primitives. XRD assumes existence of a group of prime order p with a generator g in which discrete log is hard and the decisional Diffie-Hellman assumption holds. We will write $\mathsf{DH}(g^a, b) = g^{ab}$ to denote Diffie-Hellman key exchange. In addition, XRD makes use of authenticated encryption.

Authenticated encryption [6]: XRD relies on an authenticated encryption scheme for confidentiality and integrity, which consists of the following algorithms:

- $c \leftarrow \mathsf{AEnc}(\mathsf{s}, \mathsf{nonce}, m)$. Encrypt message m and authenticate the ciphertext c using a symmetric key s and a nonce nonce. Typically, s is used to derive two other keys for encryption and authentication.
- $(b, m) \leftarrow \mathsf{ADec}(\mathsf{s}, \mathsf{nonce}, c)$. Check the integrity of and decrypt ciphertext c using the key s and a nonce nonce. If the check fails, then $b = 0$ and $m = \perp$. Otherwise, $b = 1$ and m is the underlying plaintext.

In general, the adversary cannot generate a correctly authenticated ciphertext without knowing the secret key used for ADec. We use Encrypt-then-HMAC for authenticated encryption, which has the additional property that the ciphertext serves as a commitment to the underlying plaintext with the secret key being the opening to the commitment [28].

3.2 Goals

XRD has three main goals.

Correctness. Informally, the system is correct if every honest user successfully communicates with her conversation partner after a successful execution of the system protocol.

Privacy. Similar to prior messaging systems [53, 5, 52, 37], XRD aims to provide relationship unobservability [46], meaning that the adversary cannot learn anything about the communication between two honest users. Informally, consider any honest users Alice, Bob, and Charlie. The system provides privacy if the adversary cannot distinguish whether Alice is communicating with Bob, Charlie, or neither. XRD only guarantees this property among the honest users, as malicious conversation partners can trivially learn the metadata of their communication. We provide a more formal definition in Appendix B. (This is a weaker privacy goal than that of Atom [34], which aims for sender anonymity.)

Scalability. Similar to prior work [34], we require that the system can handle more users with more servers. If the number of messages processed by a server is $C(M,N)$ for M users and N servers, we require that $C(M,N) \to 0$ as $N \to \infty$. $C(M,N)$ should approach zero polynomially in N so that adding a server introduces significant performance benefits.

4 XRD overview

Figure 1 presents the overview of a XRD network. At a high level, XRD consists of three different entities: users, mix servers, and mailbox servers. Every user in XRD has a unique *mailbox* associated with her, similar to an e-mail address. The mailbox servers maintain the mailboxes, and are only trusted for availability and not privacy.

To set up the network, XRD organizes the mix servers into many chains of servers such that there exists at least one honest server in each chain with overwhelming probability (i.e., an anytrust group [55]). Communication in XRD is carried

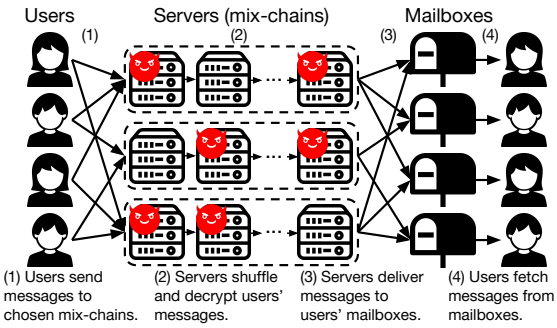

Figure 1: Overview of XRD operation.

out in discrete rounds. In each round, each user selects a fixed set of ℓ chains, where the set is determined by the user's public key. She then sends a fixed size message to each of the selected chains. (If the message is too small or large, then the user pads the message or breaks it into multiple pieces.) Each message contains a destination mailbox, and is onion-encrypted for all servers in the chain.

Once all users submit their messages, each chain acts as a local mix-net [10], decrypting and shuffling messages. During shuffling, each server also generates a short proof that allows other servers to check that it behaved correctly. If the proof does not verify, then the servers can identify who misbehaved. If all verification succeeds, then the last server in each chain forwards the messages to the appropriate mailbox servers. (The protocol for proving and verifying the shuffle is described in §6.) Finally, the mailbox servers put the messages into the appropriate mailboxes, and each user downloads all messages in her mailbox at the end of a round.

The correctness and security of XRD is in large part due to how each user selects the chains. As we will see in §5, the users are required to follow a specific algorithm to select the chains. The algorithm guarantees that *every* pair of users have at least one chain in common and the choices of the chains are publicly computable. For example, every user selecting the same chain will achieve this property, and thus correctness and security. In XRD, we achieve this property while distributing the load evenly.

Let us now consider two scenarios: (1) a user Alice is not in a conversation with anyone, or (2) Alice is in a conversation with another user Bob. In the first case, she sends a dummy message encrypted for herself to each chain that will come back to her own mailbox. We call these messages *loopback messages*. In the second case, Alice and Bob compute each other's choices of chains, and discover at which chain they will intersect. If there are multiple such chains, they break ties in a deterministic fashion. Then, Alice and Bob send the messages encrypted for the other person, which we call *conversation messages*, to their intersecting chain. They also send loopback messages on all other chains.

Security properties. We now argue the security informally. We present a more formal definition and arguments of privacy in Appendix B. Since both types of messages are en-

crypted for owners of mailboxes and the mix-net hides the origin of a message, the adversary cannot tell if a message going to Alice's mailbox is a loopback message or a conversation message sent by a different user. This means that the network pattern of all users is the same from the adversary's perspective: each user sends and receives exactly ℓ messages, each of which could be a loopback or a conversation message. As a result, the adversary cannot tell if a user is in a conversation or not. Moreover, we choose the chains such that every pair of users intersects at some chain (§5), meaning the probability that Alice is talking to a particular honest user is the same for all honest users. This hides the conversation metadata.

The analysis above, however, only holds if the adversary does not tamper with the messages. For instance, if the adversary drops Alice's message in a chain, then there are two possible observable outcomes in this chain: Alice receives (1) no message, meaning Alice is not in a conversation in this chain, or (2) one message, meaning someone intersecting with Alice at this chain is chatting with Alice. This information leakage breaks the security of XRD. We propose a new protocol called aggregate hybrid shuffle (§6) that efficiently defends against such an attack.

Scalability properties. Let n and N be the number of chains and servers in the network, respectively. Each user must send at least $\sqrt{n}$ messages to guarantee every pair of users intersect. To see why, fix ℓ, the number of chains a user selects. Those chains must connect a user Alice to all M users. Since the total number of messages sent by users is $M \cdot \ell$, each chain should handle $\frac{M \cdot \ell}{n}$ messages if we distribute the load evenly. We then need $\frac{M \cdot \ell}{n} \cdot \ell \geq M$ because the left hand side is the maximum number of users connected to the chains that Alice chose. Thus, $\ell \geq \sqrt{n}$. In §5, we present an approximation algorithm that uses $\ell \approx \sqrt{2n}$ to ensure all users intersect with each other while evenly distributing the work. This means that each chain handles $\approx \frac{\sqrt{2}M}{\sqrt{n}}$ messages, and thus XRD scales with the number of chains. If we set $n = N$ and each server appears in k chains for $k << \sqrt{N}$, which means $C(M,N) = \frac{k\sqrt{2}M}{\sqrt{N}} \to 0$ polynomially as $N \to \infty$ (§3.2). We show that k is logarithmic in N in §5.2.1.

5 XRD design

We now present the details of a XRD design that protects against an adversary that does *not* launch active attacks. We then describe modifications to this design that allows XRD to protect against active attacks in §6.

5.1 Mailboxes and mailbox servers

Every user in XRD has a mailbox that is publicly associated with her. In our design, we use the public key of a user as the identifier for the mailbox, though different public identifiers like e-mail addresses can work as well. The mailboxes are maintained by the mailbox servers, with simple put and get functionalities to add and fetch messages to a mailbox.

Algorithm 1 Mix server routing protocol

Server i is in chain x which contains k servers with its mixing key pair $(\mathsf{mpk}_i = g^{\mathsf{msk}_i}, \mathsf{msk}_i)$. In each round r, it receives a set of ciphertexts $\{c_i^j = (g^{x_j}, \mathsf{AEnc}(\mathsf{DH}(\mathsf{mpk}_i, x_j), r||\mathsf{x}, c_{i+1}^j)\}_{j \in [M]}$, either from an upstream server if $i \neq 1$, or from the users if $i = 1$.

1. **Decrypt and shuffle:** Compute $c_{i+1}^j = \mathsf{ADec}(\mathsf{DH}(g^{x_j}, \mathsf{msk}_i), r||\mathsf{x}, c_i^j)$ for each j, and randomly shuffle $\{c_{i+1}^j\}$.

2a. **Relay messages:** If $i < k$, then send the shuffled $\{c_{i+1}^j\}$ to server $i+1$.

2b. **Forward messages to mailbox:** If $i = k$, then each decrypted message is of the form $(\mathsf{pk}_u, \mathsf{AEnc}(\mathsf{s}, r||\mathsf{x}, m_u))$, where pk_u is the public key of a user u, s is a secret key, and m_u is a message for the user. Send the message to the mailbox server that manages mailbox pk_u.

5.2 Mix chains

XRD uses many parallel mix-nets to process the messages. We now describe their formation and operations.

5.2.1 Forming mix chains

We require the existence of an honest server in every chain to guarantee privacy. To ensure this property, we use public randomness sources [7, 50] that are unbiased and publicly available to randomly sample k servers to form a chain, similar to prior works [34, 52]. We set k large enough such that the probability that all servers are malicious is negligible. Concretely, the probability that a chain of length k consists only of malicious servers is f^k. Then, if we have n chains in total, the probability there exists a group of only malicious servers is less than $n \cdot f^k$ via a union bound. Finally, we can upper bound this to be negligible. For example, if we want this probability to be less than 2^{-64} for $f = 20\%$, then we need $k = 32$ for $n < 2000$. This makes k depend logarithmically on N. In XRD, we set $n = N$ for N servers, meaning each server appears in k chains on average.

We "stagger" the position of a server in the chains to ensure maximal server utilization. For instance, if a server is part of two chains, then it could be the first server in one chain and the second server in the other chain. This optimization has no impact on the security, as we only require the existence of an honest server in each group. This helps minimize the idle time of each server.

5.2.2 Processing user messages

After the chains form, each mix server i generates a *mixing key pair* $(\mathsf{mpk}_i = g^{\mathsf{msk}_i}, \mathsf{msk}_i)$, where msk_i is a random value in $\mathbb{Z}_p$. The public mixing keys $\{\mathsf{mpk}_i\}$ are made available to all participants in the network, along with the ordering of the keys in each chain. Now, each chain behaves as a mix-net [10]: users submit some messages onion-encrypted using the mixing keys (§5.3), and the servers decrypt and shuffle the messages in order. Algorithm 1 describes this protocol.

5.2.3 Server churn

Some servers may go offline in the middle of a round. Though XRD does not provide additional fault tolerance mechanisms, only the chains that contain failing servers are affected. Furthermore, the failing chains do not affect the security since they do not disturb the operations of other chains and the destination of the messages at the failing chain remains hidden to the adversary. Thus, conversations that use chains with no failing servers are unaffected. We analyze the empirical effects of server failures in §7.3.

5.3 Users

We now describe how users operate in XRD.

5.3.1 Selecting chains

XRD needs to ensure that all users' choices of chains intersect at least once, and that the choices are publicly computable. We present a scheme that achieves this property. Upon joining the network, every user is placed into one of $\ell+1$ groups such that each group contains roughly the same number of users, and such that the group of any user is publicly computable. This could be done, for example, by assigning each user to a pseudo-random group based on the hash of the user's public key. Every user in a group is connected to the same ℓ servers specified as follows. Let C_i be the ordered set of chains that users in group i are connected to. We start with $C_1 = \{1,\ldots,\ell\}$, and build the other sets inductively: For $i = 1,\ldots,\ell$, group $i+1$ is connected to $C_{i+1} = \{C_1[i], C_2[i], \ldots, C_i[i], C_i[\ell]+1, \ldots, C_i[\ell]+(\ell-i)\}$, where $C_x[y]$ is the y^{th} entry in C_x.

By construction, every group is connected to every other group: Group i is connected to group j via $C_i[j]$ for all $i < j$. As a result, every user in group i is connected to all others in the same group (they meet at all chains in C_i), and is connected to users in group j via chain $C_i[j]$.

To find the concrete value of ℓ, let us consider C_ℓ. The last chain of C_ℓ, which is the chain with the largest index, is $C_\ell[\ell] = \ell^2 - \sum_{i=1}^{\ell-1} i = \frac{\ell^2+\ell}{2}$. This value should be as close as possible to n, the number of chains, to maximize utilization. Thus, $\ell = \lceil \sqrt{2n+0.25} - 0.5 \rceil \approx \lceil \sqrt{2n} \rceil$. Given that $\ell \geq \sqrt{n}$ (§4), this is a $\sqrt{2}$-approximation.

5.3.2 Sending messages

After choosing the ℓ mix chains, the users send one message to each of the chosen chains as described in Algorithm 2. At a high level, if Alice is not talking with anyone, Alice generates ℓ loopback messages by encrypting dummy messages (e.g., messages with all zeroes) using a secret key known only to her, and submits them to the chosen chains. If she is talking with another user Bob, then she first finds where they intersect by computing the intersection of Bob's group and her group (§5.3.1). If there is more than one such chain, then she breaks the tie by selecting the chain with the smallest index. Alice then generates $\ell-1$ loopback messages and one encrypted message using a secret key that Alice and Bob shares. Finally, Alice sends the message for Bob to the intersecting chain, and sends the loopback messages to the other chains. Bob mirrors Alice's actions.

Algorithm 2 User conversation protocol

Consider two users Alice and Bob with key pairs $(\mathsf{pk}_A = g^{\mathsf{sk}_A}, \mathsf{sk}_A)$ and $(\mathsf{pk}_B = g^{\mathsf{sk}_B}, \mathsf{sk}_B)$ who are connected to sets of ℓ chains C_A and C_B (§5.3.1). The network consists of chains $1,\ldots,n$, each with k servers. Alice and Bob possess the set of mixing keys for each chain. Alice performs the following in round r.

1a. **Generate loopback messages:** If Alice is not in a conversation, then Alice generates ℓ loopback messages: $m_{\mathsf{x}} = (\mathsf{pk}_A, \mathsf{AEnc}(\mathsf{s}_A^{\mathsf{x}}, r||\mathsf{x}, 0))$ for $\mathsf{x} \in C_A$, where $\mathsf{s}_A^{\mathsf{x}}$ is a chain-specific symmetric key known only to Alice.

1b. **Generate conversation message:** If Alice is in a conversation with Bob, then she first computes the shared key $\mathsf{s}_{AB} = \mathsf{DH}(\mathsf{pk}_B, \mathsf{sk}_A)$, and the symmetric encryption key for Bob $\mathsf{s}_B = \mathsf{KDF}(\mathsf{s}_{AB}, \mathsf{pk}_B, r)$ where KDF is a secure key derivation function (e.g., HKDF [33]). Alice then generates the conversation message: $m_{\mathsf{x}_{AB}} = (\mathsf{pk}_B, \mathsf{AEnc}(\mathsf{s}_B, r||\mathsf{x}, \mathsf{msg}))$, where msg is the plaintext message for Bob and $\mathsf{x}_{AB} \in C_A \cap C_B$ is the first chain in the intersection. She also generates $\ell-1$ loopback messages m_{x} for $\mathsf{x} \in C_A, \mathsf{x} \neq \mathsf{x}_{AB}$.

2. **Onion-encrypt messages:** For each message m_{x}, let $c_{k+1} = m_{\mathsf{x}}$, and let $\{\mathsf{mpk}_i\}$ be the mixing keys for chain $\mathsf{x} \in C_A$. For $i = k$ to 1, generate a random value $x_i \in \mathbb{Z}_p$, and compute $c_i = (g^{x_i}, \mathsf{AEnc}(\mathsf{DH}(\mathsf{mpk}_i, x_i), r||\mathsf{x}, c_{i+1}))$. Send c_1 to chain x.

3. **Fetch messages:** At the end of the round, fetch and decrypt the messages in her mailbox, using ADec with matching $\mathsf{s}_A^{\mathsf{x}}$ or $\mathsf{s}_A = \mathsf{KDF}(\mathsf{s}_{AB}, \mathsf{pk}_A, r)$.

5.3.3 User churn

Like servers, users might go offline in the middle of a round, and XRD aims to provide privacy in such situations. However, the protocol presented thus far does not achieve this goal. If Alice and Bob are conversing and Alice goes offline without Bob knowing, then Alice's mailbox will receive Bob's message while Bob's mailbox will get one fewer message. Thus, by observing mailbox access counts and Alice's availability, the adversary can infer their communication.

To solve this issue, we require Alice to submit two sets of messages in round r: the messages for the current round r, and *cover messages* for round $r+1$. If Alice is not communicating with anyone, then the cover messages will be loopback messages. If Alice is communicating with another user, then one of the cover messages will be a conversation message indicating that Alice has gone offline.

If Alice goes offline in round τ, then the servers use the cover messages submitted in $\tau-1$ to carry out round τ. Now, there are two possibilities. If Alice is not in a conversation, then Alice's cover loopback messages are routed in round τ,

and nothing needs to happen afterwards. If Alice is conversing with Bob, then at the end of round τ, Bob will get the message that Alice is offline via one of the cover messages. Starting from round $\tau+1$, Bob now sends loopback messages instead of conversation messages to hide the fact that Bob was talking with Alice in previous rounds. This could be used to end conversations as well. Malicious servers cannot fool Bob into thinking Alice has gone offline by replacing Alice's messages with her cover messages because the honest servers will ensure Alice's real messages are accounted for using our defenses described in §6.

6 Aggregate hybrid shuffle

Adversarial servers can tamper with the messages to leak privacy in XRD. For example, consider a mix-net chain where the first server is malicious. This malicious server can replace Alice's message with a message directed at Alice. Then, at the end of the mixing, the adversary will make one of two observations. If Alice was talking to another user Bob, Bob will receive one fewer message while Alice would receive two messages. The adversary would then learn that Alice was talking to Bob. If Alice is not talking to anyone on the tampered chain, then Alice would receive one message, revealing the lack of conversation on that chain.

Prior works [14, 55, 35, 52, 34] have used traditional verifiable shuffles [45, 25, 8, 27] to prevent these attacks. At a high level, verifiable shuffles allow the servers in the chain (one of which is honest) to verify the correctness of a shuffle of another server; namely, that the plaintexts underlying the outputs of a server is a valid permutation of the plaintexts underlying the inputs. Unfortunately, these techniques are computationally expensive, requiring many exponentiations.

In XRD, we make an observation that help us avoid traditional verifiable shuffles. For a meaningful tampering, the adversary necessarily has to tamper with the messages *before* they are shuffled by the honest server. Otherwise, the adversary does not learn the origins of messages. For example, after dropping a message in a server downstream from the honest server, the adversary might observe that Alice did not receive a message. The adversary cannot tell, however, whether the dropped message was sent by Alice or another user, and does not learn anything about Alice's communication pattern. (Intuitively, the adversarial downstream servers do not add any privacy in any case.) In this section, we describe a new form of verifiable shuffle we call *aggregate hybrid shuffle* (AHS) that allows us to take advantage of this fact. In particular, the protocol guarantees that the honest server will receive and shuffle all honest users' messages, or the honest server will detect that someone upstream (malicious servers or users) misbehaved. We will then describe how the honest server can efficiently identify all malicious participants who deviated from the protocol, without affecting the privacy of honest users. Table A in Appendix A summarizes the notations used in AHS.

6.1 Key generation with AHS

When the chain is created, the servers generate three key pairs: *blinding*, *mixing*, and *inner* key pairs. The inner keys are per-round keys, and each server i generates its own inner key pair $(\mathsf{ipk}_i = g^{\mathsf{isk}_i}, \mathsf{isk}_i)$. The other two keys are long-term keys, and are generated in order starting with the first server in the chain. Let $\mathsf{bpk}_0 = g$. Starting with server 1, server $i = 1,\ldots,k$ generates $(\mathsf{bpk}_i = \mathsf{bpk}_{i-1}^{\mathsf{bsk}_i}, \mathsf{bsk}_i)$ and $(\mathsf{mpk}_i = \mathsf{bpk}_{i-1}^{\mathsf{msk}_i}, \mathsf{msk}_i)$ in order. In other words, the base of the public keys of the server i is $\mathsf{bpk}_{i-1} = g^{\prod_{a<i}\mathsf{bsk}_a}$. The public mixing key of the last server, for example, would be $\mathsf{mpk}_k = \mathsf{bpk}_{k-1}^{\mathsf{msk}_k} = g^{\mathsf{msk}_k \cdot \prod_{a<k}\mathsf{bsk}_a}$. Each server also has to prove to all other servers that it knows the private keys that match the public keys in zero-knowledge [9]. All public keys are made available to all servers and users.

6.2 Sending messages with AHS

Once the servers generate the keys, user Alice can submit a message to a chain. To do so, Alice now employs a double-enveloping technique to encrypt her message [26]: she first onion-encrypts her message for all servers using the inner keys, and then onion-encrypts the result with the mixing keys. Let *inner ciphertext* be the result of the first onion-encryption, and *outer ciphertext* be the final ciphertext. The inner ciphertexts are encrypted using $\prod_i \mathsf{ipk}_i$ as the public key, which allows users to onion-encrypt in "one-shot": i.e., $e = (g^y, \mathsf{AEnc}(\mathsf{DH}(\prod_i \mathsf{ipk}_i, y), r, m))$ in round r with message m and a random y. Without y, one must know all $\{\mathsf{isk}_i\}$ to compute $\mathsf{DH}(\prod_i \mathsf{ipk}_i, y)$, which makes this a "one-shot" onion-encryption. To generate the outer ciphertext for chain x, Alice performs the following.

1. Generate her outer Diffie-Hellman key: a random $x \in \mathbb{Z}_p$ and (g^x, x).
2. Generate a NIZK that proves she knows x that matches g^x (using knowledge of discrete log proof [9]).
3. Let $c_{k+1} = e$, and let $\{\mathsf{mpk}_i\}$ for $i \in [k]$ be the mixing keys of the servers in the chain. For $i = k$ to 1, compute $c_i = \mathsf{AEnc}(\mathsf{DH}(\mathsf{mpk}_i, x), r||\mathsf{x}, c_{i+1})$.

$c = (g^x, c_1)$ is the final outer ciphertext. This is nearly identical to Algorithm 2, except that the user does not generate a fresh pair of Diffie-Hellman keys for each layer of encryption. To submit the message, Alice sends c and the NIZK to all servers in the chain.

6.3 Mixing with AHS

Before mixing begins in round r, the servers in chain x have $c^j = (X_1^j = g^{x_j}, c_1 = \mathsf{AEnc}(\mathsf{DH}(\mathsf{mpk}_1, x_j), r||\mathsf{x}, c_2^j))$ for user j. The servers first verify all NIZKs the users submit, and agree on the inputs for this round. This can be done, for example, by sorting the users' ciphertexts, hashing them using a cryptographic hash function, and then comparing the hashes. Then, starting with server 1, server $i = 1,\ldots,k$ perform the following:

1. **Decrypt and shuffle:** Similar to Algorithm 1, decrypt each message. Each message is of the form $(X_i^j, c_i^j = \mathsf{AEnc}(\mathsf{DH}(\mathsf{mpk}_i, x_j), r||\mathsf{x}, c_{i+1}^j)$. Thus, $(b^j, c_{i+1}^j) = \mathsf{ADec}(\mathsf{DH}(X_i^j, \mathsf{msk}_i), r||\mathsf{x}, c_i^j)$. If any decryption fails (i.e., $b^j = 0$ for some j), then mixing halts and the server can start the blame protocol described in §6.4. Randomly shuffle $\{c_{i+1}^j\}$.
2. **Blind and shuffle:** Blind the users' Diffie-Hellman keys $\{X_i^j\}$ using the blinding key: $X_{i+1}^j = (X_i^j)^{\mathsf{bsk}_i}$ for each j. Then, shuffle the keys using the same permutation as the one used to shuffle the ciphertexts.
3. **Generate zero-knowledge proof:** Generate a proof that $(\prod_j X_i^j)^{\mathsf{bsk}_i} = \prod_j X_{i+1}^j$ by generating a NIZK that shows $\log_{\prod_j (X_i^j)}(\prod_j X_{i+1}^j) = \log_{\mathsf{bpk}_{i-1}} \mathsf{bpk}_i (= \mathsf{bsk}_i)$. Send the NIZK with the shuffled $\{X_{i+1}^j\}$ to all other servers in the chain. All other servers verify this proof using $\{X_i^j\}$ they received previously, $\{X_{i+1}^j\}$, bpk_{i-1}, and bpk_i.
4. **Forward messages:** If $i < k$, then send the shuffled $\{(X_{i+1}^j, c_{i+1}^j)\}$ to server $i+1$.

When the last server finishes shuffling and no server reports any errors during mixing, our protocol guarantees that the honest server mixed all the honest users' messages successfully. At this point, the servers reveal their private per-round inner keys $\{\mathsf{isk}_i\}$. With this, the last server can decrypt the inner ciphertexts to recover the users' messages.

Analysis. We first argue correctness of AHS (i.e., every message is successfully delivered if every participant followed the protocol) by showing that the encryption and decryption keys match at all layers. Consider the key user j used to encrypt the message for server i and the Diffie-Hellman key server i receives. User j encrypts the message using the key $\mathsf{DH}(\mathsf{mpk}_i, x_j) = g^{x_j \cdot \mathsf{msk}_i \prod_{a<i} \mathsf{bsk}_a}$. The Diffie-Hellman key server i receives is $X_i^j = g^{x_j \cdot \prod_{a<i} \mathsf{bsk}_a}$. The key exchange then results in $\mathsf{DH}(X_i^j, \mathsf{msk}_i) = g^{\mathsf{msk}_i \cdot x_j \prod_{a<i} \mathsf{bsk}_a}$, which is the same as the one the user used.

The scheme protects against honest-but-curious adversaries (i.e., does not reveal anything about the permutation used to shuffle the messages), as the inputs and outputs of a server look random: If decisional Diffie-Hellman is hard, then $g^{x \cdot \mathsf{bsk}_i}$ is indistinguishable from a random value given g^x and g^{bsk_i} for random x and bsk_i. Thus, by observing $\{g^{x_j}\}$ (input) and $\{g^{x_{\pi(j)} \cdot \mathsf{bsk}_i}\}$ (output) of an honest server where π is a random permutation, the adversary cannot learn anything about the relationships between the inputs and outputs.

We now provide a high level analysis that the honest server will always detect upstream servers tampering with honest users' messages. The detailed proof is in Appendix A. Let server h be the honest server. First, since we only need to consider upstream adversaries, we will simplify the problem, and view all upstream malicious servers as one collective server with private blinding key $\mathsf{bsk}_A = \sum_{i<h} \mathsf{bsk}_i$. For the adversary to successfully tamper, it must generate $\{X_h^j\}$ such that $(\prod X_1^j)^{\mathsf{bsk}_A} = \prod X_h^j$; otherwise it would fail the NIZK verification in step 3 of §6.3. Let $X_T \neq \emptyset$ be the set of honest users whose messages were tampered. The adversary needs to know the keys used for authenticated decryption to generate valid ciphertexts that differ from the users' ciphertexts. However, the adversary cannot compute $((g^{x_j})^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$ for $j \in X_T$ (the keys used for authenticated encryption), since x_j and msk_h are unknown random values and Diffie-Hellman is hard. Thus, to tamper with messages undetected, the adversary needs to change the users' Diffie-Hellman keys (i.e., $X_h^j \neq (X_1^j)^{\mathsf{bsk}_A}$ for $j \in X_T$), such that it can compute the keys used for authenticated decryption (i.e., $(X_h^j)^{\mathsf{msk}_h}$ for $j \in X_T$).

In the beginning of a round, the adversary controlled users have to prove their knowledge of discrete logs of their Diffie-Hellman keys after seeing the honest users' keys. The adversarial user are thus forced to generate keys independently of the honest users' input. Then, the adversary's goal is essentially to find $\{X_h^j\}_{j \in X_T}$ such that $(\prod_{j \in X_T} X_1^j)^{\mathsf{bsk}_A} = \prod_{j \in X_T} X_h^j$, with $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$. Assume the adversary is successful. Then, it could compute $((\prod_{j \in X_T} X_1^j)^{\mathsf{bsk}_A})^{\mathsf{msk}_h} = \prod_{j \in X_T} (X_h^j)^{\mathsf{msk}_h}$, since it knows $(X_h^j)^{\mathsf{msk}_h}$ for $j \in X_T$ (recall that these are the keys used for authenticated decryption, which the adversary must know). This means that the adversary computed $((\prod_{j \in X_T} X_1^j)^{\mathsf{bsk}_A})^{\mathsf{msk}_h} = g^{\mathsf{msk}_h \cdot \mathsf{bsk}_A \cdot \sum_{j \in X_T} x_j}$ only given $\{g^{x_j}\}_{j \in X_T}$, bsk_A, and $(g^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$, where $\{x_j\}_{j \in X_T}$ and msk_h are random values independent of bsk_A. This breaks the Diffie-Hellman assumption, and thus the adversary must not be able to tamper with messages undetected.

6.4 Blame protocol

There are two ways an honest server can detect misbehavior: a NIZK fails to verify or an authenticated decryption fails. If a malicious user cannot generate a correct NIZK in step 2 in §6.2 or if a malicious server misbehaves and cannot generate a correct NIZK in step 3 in §6.3, then the misbehavior is detected and the adversary is immediately identified. In the case where a server finds some misauthenticated ciphertexts, the server can start a blame protocol that allows the server to identify who misbehaved. The protocol guarantees that users are identified if and only if they purposefully sent misauthenticated ciphertexts. In addition, the protocol ensures that honest users remain private in all cases, even if malicious servers try to falsely accuse honest users.

Once server h identifies an misauthenticated ciphertext, it starts the blame protocol by revealing the problem ciphertext (X_h^j, c_h^j). Then, the servers execute the following:

1. For $i = h-1, \ldots, 1$, the servers reveal X_i^j that matches X_{i+1}^j (i.e., $(X_i^j)^{\mathsf{bsk}_i} = X_{i+1}^j$). Each server proves to all other servers it calculated X_{i+1}^j correctly by showing that $\log_{X_i^j}(X_{i+1}^j) = \log_{\mathsf{bpk}_{i-1}}(\mathsf{bpk}_i) (= \mathsf{bsk}_i)$ with a NIZK [9].

2. For $i = h-1, \ldots, 1$, the servers reveal c_i^j that matches c_{i+1}^j (i.e., $c_i^j = \mathsf{AEnc}(\mathsf{DH}(X_i^j, \mathsf{msk}_i), r||\mathsf{x}, c_{i+1}^j)$. Each server proves it correctly decrypted the ciphertext by revealing the key used for decryption $k_i^j = (X_i^j)^{\mathsf{msk}_i}$, and showing that $\log_{X_i^j}(k_i^j) = \log_{\mathsf{bpk}_{i-1}}(\mathsf{mpk}_i)$ with a NIZK. The other servers can verify the correctness of the decryption operation by checking the NIZKs and decrypting the ciphertext themselves.

3. All servers check that c_1^j revealed by the first server matches the user submitted ciphertext (§6.2).

4. Similar to step 2, server h (the accusing server) reveals its Diffie-Hellman exchanged key $k_h = (X_h^j)^{\mathsf{msk}_h}$, and shows that $\log_{X_h^j}(k_h) = \log_{\mathsf{bpk}_{h-1}}(\mathsf{mpk}_h)$. All servers verify that $\mathsf{ADec}(k_h, r||\mathsf{x}, c_h^j)$ fails.

If there are multiple problem ciphertexts, the blame protocol can be carried out in parallel for each ciphertext. Steps 1 and 2 can be done simultaneously as well. If the servers successfully carry out the blame protocol, then they have identified actively malicious users. At this point, those ciphertexts are removed from the set, and the upstream servers are required to repeat the AHS protocol; since the accusing servers have already removed all bad ciphertexts, the servers just have to repeat step 3 of §6.3 to show the keys were correctly computed. If any of the above steps fail, then the servers delete their private inner keys.

Analysis. The accusing server and the upstream servers are required to reveal the exchanged key used to decrypt the ciphertexts, and the correctness of the key exchange is proven through the two NIZKs in step 1 and step 2. All servers can use the revealed keys to ensure that the submitted original ciphertext decrypts to the problem ciphertext. Since the outer ciphertext behaves as a commitment to all layers of encryption (§3), the servers get a verifiable chain of decryption starting with the outer ciphertext to the problem ciphertext if a user submits misauthenticated ciphertext. Moreover, if an honest user submits a correctly authenticated ciphertext, she will never be accused successfully, since an honest user's ciphertext will authenticate at all layers. Thus, a user is identified if and only if she is malicious.

Importantly, the users' privacy is protected even after a false accusation. After a malicious server accuses an honest user, the malicious server learns either the outer ciphertext (if the server is upstream of server h) or the inner ciphertext (if the server is downstream of server h) the user sent. In either case, the message remains encrypted for the honest server, by the mixing key in the former case or by the inner key in the latter case. The blame protocol will fail when the malicious server fails to prove that the honest user's ciphertext is misauthenticated, and the ciphertext will never be decrypted. As such, the adversary never learns the final destination of the user's message, and XRD protects the honest users' privacy.

7 Implementation and evaluation

To evaluate XRD, we wrote a prototype in approximately 4,000 lines of Go. We used the NIST P-256 elliptic curve [2] for our cryptographic group, and used AES with SHA-256-based key derivation function [33] and HMAC [32] for our authenticated encryption scheme. The servers communicated using streaming gRPC over TLS. Our prototype assumes that the servers' and users' public keys, and the public randomness for creating the chains are provided by a higher level functionality. Finally, we set the number of chains n equal to the number of servers N (§5.2.1).

In this section, we investigate the cost of users and the performance of XRD for different network configurations. For majority of our experiments, we assumed $f = 0.2$ (i.e., 80% of the servers are honest) unless indicated otherwise. We used 256 byte messages, similar to evaluations of prior systems [53, 52, 5]; this is about the size of a standard SMS message or a Tweet. We used c4.8xlarge instances on Amazon EC2 for our experiments, which has 36 Intel Xeon E5-2666 CPUs with 60 GB of memory and 10 Gbps links.

We compare the results against four prior systems: Stadium [52], Karaoke [37], Atom [34], and Pung [5, 4]. For Stadium and Karaoke, we report the performance for $e^\varepsilon = 10$ and $e^\varepsilon = 4$, respectively, (meaning the probability of Alice talking with Bob is within $10\times$ and $4\times$ the probability of Alice talking with any other user) and allow up to 10^8 rounds of communication with strong security guarantees ($\delta < 10^{-4}$), which are the parameters the used for evaluation in their papers. We show results for Pung with XPIR (used in the original paper [5]) and with SealPIR (used in the follow-up work [4]) when measuring user overheads, and only with XPIR when measuring end-to-end latency. We do this because SealPIR significantly improves the client performance and enables more efficient batch messaging, but introduces extra server overheads in the case of one-to-one messaging.

As mentioned in §2, the four systems scale horizontally, but offer different security properties. To summarize, Stadium and Karaoke provide differential privacy guarantees against the same adversary assumed by XRD. Thus, their users can send a limited number of sensitive messages with strong privacy, while XRD users can do so for unlimited messages. Atom provides cryptographic sender anonymity [46] under the same threat model. Finally, Pung provides messaging with cryptographic privacy against an adversary who can compromise all servers rather than a fraction of servers. To the best of our knowledge, we are not aware of any scalable private messaging systems that offer the same security guarantee under a similar threat model to XRD.

7.1 User costs

We first characterize computation and bandwidth overheads of XRD users using a single core of a c4.8xlarge instance. In order to ensure that every pair of users intersects, each user sends $\sqrt{2N}$ messages (§5.3.1). This means that

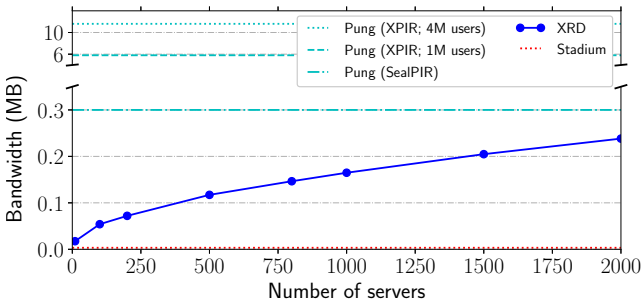

Figure 2: Required user bandwidth per round as a function of number of servers in the network.

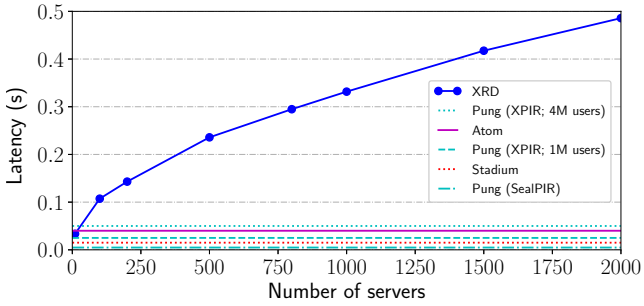

Figure 3: Required user computation as a function of number of servers with a single core. The computation could easily be parallelized with more cores for XRD.

the overheads for users increase as we add more servers to the network, as shown in Figure 2 and 3. This is a primary weakness of XRD, since our horizontal scalability comes at a higher cost for users. Still, the cost remains reasonable even for large numbers of servers. With 2,000 servers, each user must submit about 238 KB of data. For 1 minute rounds, this translates to about 40 Kbps of bandwidth requirement. A similar trend exists for computation overhead as well, though it remains relatively small: it takes less than 0.5s with fewer than 2,000 servers in the network. Computation could also be easily parallelized with more cores, since users can generate the messages for different chains independently. The cover messages make up half of the client overhead (§5.3.3).

User costs in prior works do not increase with the number of servers. Still, Pung with XPIR incurs heavy user bandwidth overheads due to the cost of PIR. With 1 million users, Pung users transmit about 5.8 MB, which is about 25× worse than XRD when there are fewer than 2,000 servers. Moreover, per user cost of XPIR is proportional to the total number of users: the bandwidth cost increases to 11 MB of bandwidth for 4 million users. The SealPIR variant, however, is comparable to that of XRD, as the users can compress the communication using cryptographic techniques. Stadium, Karaoke, and Atom incur minimal user bandwidth cost, with less than a kilobyte of bandwidth overhead (only Stadium is shown Figure 2). Thus, for users with heavily limited resources, prior works can be more desirable than XRD.

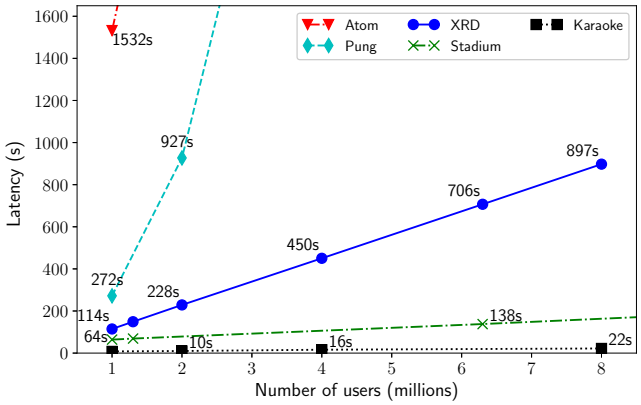

Figure 4: End-to-end latency of XRD and prior systems with varying numbers of users with 100 servers.

7.2 End-to-end latency

Experiment setup. To evaluate the end-to-end performance, we created a testbed consisting of up to 200 c4.8xlarge instances. We ran the instances within the same data center to avoid bandwidth costs, but added 40-100ms of round trip latency between servers using the Linux `tc` command to simulate a more realistic distributed network. Our evaluations consider all parts of the AHS protocol (§6), but assume all users are following the protocol. We then evaluate the blame protocol (§6.4) separately.

We used many c4.8xlarge instances to simulate millions of users, and also used ten more c4.8xlarge instances to simulate the mailboxes. We generate all users' messages before the round starts, and measure the critical path of our system by measuring the time between the last user submitting her message and the last user downloading her message.

We estimate the latency of Pung with M users and N servers by evaluating it on a single c4.8xlarge instance with M/N users. This is the best possible latency Pung can achieve because (1) Pung is embarrassingly parallel, so evenly dividing users across all the servers should be ideal [5, §7.3], and (2) we are ignoring the extra work needed for coordination between the servers (e.g., for message replication). For Stadium, we report the latency when the length of each mix chain is nine servers. For Karaoke, we report the numbers reported in their paper.

We focus on the following questions in this section, and compare against prior work:

- What is the end-to-end latency of XRD, and how does it change with the number of users?
- How does XRD scale with more servers?
- What is the effect of f, the fraction of malicious servers, on latency?
- How fast is the blame protocol?

Number of users. Figure 4 shows the end-to-end latency of XRD and prior works with 100 servers. XRD was able to handle 2 million users in 228s, and the latency scales linearly

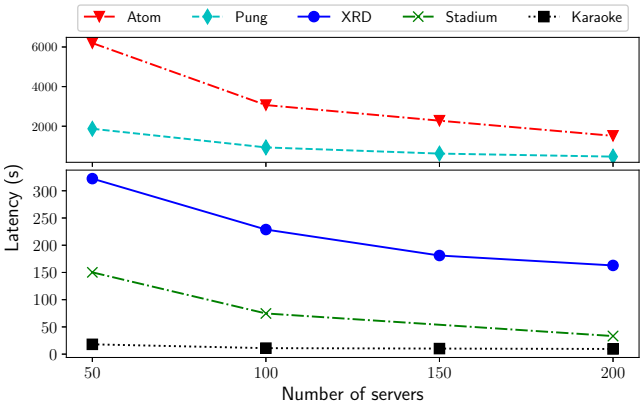

Figure 5: End-to-end latency of XRD for varying numbers of servers with 2 million users. We show Pung and Atom on a different time scale.

with the number of users. This is 13.3× and 4× faster than Atom and Pung, and 3× and 22.8× worse than Stadium and Karaoke for the same deployment scenario. Though processing a single message in XRD is faster than doing so in Stadium (since Stadium relies on verifiable shuffle, while XRD uses AHS), the overall system is still slower. This is because each XRD user submits many messages. For example, each user submits 15 messages with 100 servers, which is almost equivalent to adding 15 users who each submit one message. Unfortunately, the performance gap would grow with more servers due to each user submitting more messages (the rate at which the gap grows would be proportional to $\sqrt{2N}$). While XRD cannot provide the same performance as Stadium or Karaoke with large numbers of users and servers, XRD can provide stronger cryptographic privacy.

When compared to Pung, the speed-up increases further with the number of users since the latency of Pung grows superlinearly. This is because the server computation per user increases with the number of users. With 4 million users, for example, XRD is 8× faster. For Atom, the latency increases linearly, but with higher slope. This is due to its heavy reliance on expensive public key cryptography and long routes for the messages (over 300 servers).

Scalability. Figure 5 shows how the latency decreases with the number of servers with 2 million users.We experimented with up to 200 servers, and observed the expected scaling pattern: the latency of XRD reduces as $\sqrt{2/N}$ with N servers (§4). In contrast, prior works scale as $1/N$, and thus will outperform XRD with enough servers. Still, because XRD employs more efficient cryptography, XRD outperforms Atom and Pung with less than 200 servers.

To estimate the performance of larger deployments, we extrapolated our results to more servers. We estimate that XRD can support 2 million users with 1,000 servers in about 84s, while Stadium and Karaoke can do so in about 8s and 2s, respectively. (At this point, the latency between servers would be the dominating factor for Stadium and Karaoke.) This

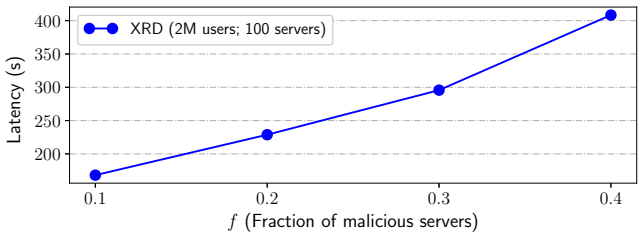

Figure 6: Latency of XRD for different values of f.

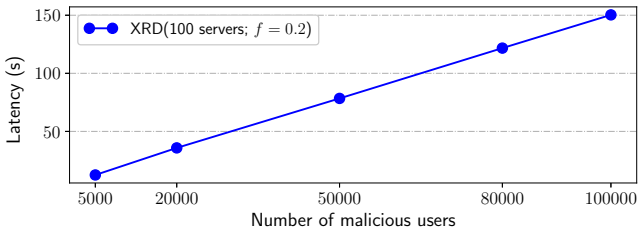

Figure 7: Latency of blame protocol.

gap increases with more users, as described previously. For Atom and Pung, we estimate that the latency would be comparable to XRD with about 3,000 servers and 1,000 servers in the network, respectively, for 2 million users. Pung would need more servers with more users to catch up to XRD due to the superlinear increase in latency with the number of users.

Impact of f. During setup, the system administrator should make a conservative estimate of f to form the chains. Larger f affects latency because it increases in the length of the chains k (§5.2.1). Concretely, with $n = 100$, k must satisfy $100 \cdot f^k < 2^{-64}$. Thus, $k > \frac{\log(2^{-64}/100)}{\log(f)}$, which means that the length of a chain (and the latency) grows as a function of $\frac{-1}{\log(f)}$. Figure 6 demonstrates this effect. The latency grows slowly for $f < 0.5$. This function, however, grows rapidly when $f >> 0.5$, and thus the latency would be significantly worse when considering larger values of f.

Atom would experience the same effect since its mix chains are created using the same strategy as XRD. Karaoke also experiences similar increase in latency with f, as every message must be routed through a number of servers proportional to $|\frac{1}{\log f}|$ as well. Stadium would face more significant increase in latency with f as its mix chains similarly get longer with f, and the length of the chains has a superlinear effect on the latency due to zero-knowledge proof verification [52, §10.3]. The latency of Pung does not increase with f since it already assumes $f = 1$.

Blame protocol. Malicious users could send misauthenticated ciphertexts to trigger the blame protocol, and slow down the system. Since malicious users' messages are removed as soon as servers find them, the users cause the most slowdown when the misauthenticated ciphertexts are discovered at the last server. The performance of the blame protocol also depends on the number of malicious users. In Figure 7, we therefore show the latency of the blame protocol

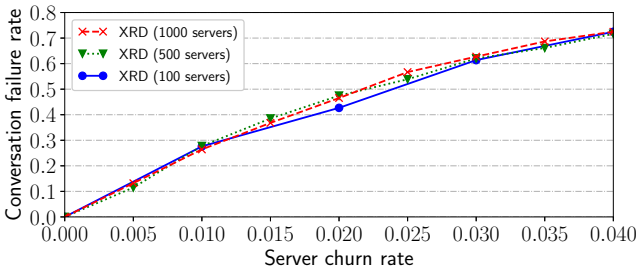

Figure 8: Fraction of conversations that fail in a given round due to server failures for different server churn rates.

as a function of the number of malicious users in a chain of 32 servers when the last server detects misbehavior. The blame protocol requires two discrete log equivalence proofs and decryption per user for each layer of encryption (§6.4).

Concretely, if 5,000 users misbehave in a chain, the blame protocol takes about 13s. This cost increases linearly with the number of users: if 100,000 users misbehave in a chain (which corresponds to approximately a third of all users being malicious with 100 servers and 2 million users in the network), the protocol takes about 150s. As a result, the overall round would take 378s, instead of 228s, for 2 million users. Still, this is 8× and 2.4× faster than Atom and Pung. In practice, the blame protocol could be faster since the honest server is likely to be not the last server. For example, if the honest server is the 16$^{\text{th}}$ server, then the blame protocol would take half the time. The overall round would then take around 303s.

While this is a significant increase in latency, malicious users are removed from the network once servers identify them. To cause serious slowdowns across many rounds, the adversary needs to constantly create new malicious users. Then, by employing defenses against Sybil attacks (e.g., imposing a small cost for user registration such as proof-of-work [23] or CAPTCHA [54]), we could limit the effectiveness of an adversary.

Malicious users can have varying degree of impact in prior systems. Pung and Karaoke are not affected by malicious users, and Stadium has a blame protocol similar to XRD to handle malicious users. Atom, however, is the significantly affected: a single malicious user can launch a DoS attack in the faster variant used for comparison in this paper [34, §4.4]. The slower variant of Atom that can handle adversarial users is at least 4× slower [34, §6], meaning it would be at least 30× slower than XRD overall.

7.3 Availability

To estimate the effect of server churn on a XRD network, we simulated deployment scenarios with 2 million users and different numbers of servers. We assumed that all users were in a conversation, and show the fraction of the users whose conversation messages did not reach their partner in Figure 8. For example, if 1% of the servers fail in a given round (comparable to server churn rate in Tor [1]), then we expect about 27% of the conversations to experience failure, and the end-points would have to resend their conversation messages. Unfortunately, the failure rate quickly increases with the server churn rates, reaching 70% with 4% server failures, as more chains contain at least one failing server. Thus, it would be easy for the adversary who controls a non-trivial fraction of the servers to launch a denial-of-service attack. Addressing this concern remains important future work.

When compared to Pung, the availability guarantees can be significantly worse, assuming Pung replicates all users' messages across all servers. In this case, the conversation failure rate would be equal to the server churn rate with users evenly distributed across all Pung servers, and the users connected to the failing servers could be rerouted to other servers to continue communication. Atom can tolerate any fraction γ of the servers failing using threshold cryptography [17], but the latency grows with γ. For example, to tolerate $\gamma = 1\%$ servers failing, we estimate that Atom would be about 10% slower [34, Appendix B]. Stadium and Karaoke, however, are more affected by server churn. Stadium uses two layers of parallel mix-nets, and fully connects the chains across the two layers. As a result, even one server failure would cause the whole system to come to a halt. (Stadium does not provide a fault recovery mechanism, and the security implications of continuing the protocol without the failing chains are not analyzed [52].) Similarly, Karaoke uses layers of interconnected mixing servers, and a single server failure results in the failure of the whole network.

8 Conclusion

XRD provides a unique design point in the space of metadata private communication systems by achieving cryptographic privacy and horizontal scalability using efficient cryptographic primitives. XRD organizes the servers into multiple small chains that process messages in parallel, and can scale easily with the number of servers by adding more chains. We hide users' communication patterns by ensuring every user is equally likely to be talking to any other user, and hiding the origins of users' messages through mix-nets. We then efficiently protect against active attacks using a novel technique called aggregate hybrid shuffle. Our evaluation on a network of 100 servers demonstrates that XRD can support 2 million users in 228 seconds, which is 13× and 4× faster than Atom and Pung, two prior systems with cryptographic privacy guarantees.

Acknowledgements

We thank Jun Wan, Daniel Grier, Ling Ren, Alin Tomescu, Zack Newman and Kyle Hogan for valuable feedback and discussion during this project. We also thank Paul Grubbs for pointing out a flaw in our initial proof. Finally, we thank the anonymous reviewers of NSDI, and our shepherd Raluca Ada Popa. This work was partially supported by a National Science Foundation grant (1813087).

References

[1] Tor metrics portal. https://metrics.torproject.org.

[2] Mehmet Adalier. Efficient and secure elliptic curve cryptography implementation of curve p-256. 2015.

[3] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *PETS*, 2016(2):155–174, 2016.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 1011–1028.

[5] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, pages 551–569, GA, 2016. USENIX Association.

[6] Mihir Bellare, Phillip Rogaway, and David Wagner. Eax: A conventional authenticated-encryption mode. Cryptology ePrint Archive, Report 2003/069, 2003. `https://eprint.iacr.org/2003/069`.

[7] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On Bitcoin as a public randomness source. `https://eprint.iacr.org/2015/1015.pdf`, 2015.

[8] Justin Brickell and Vitaly Shmatikov. Efficient anonymity-preserving data collection. In *KDD*, pages 76–85, New York, NY, USA, 2006. ACM.

[9] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. Technical report, 1997.

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

[11] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, March 1988.

[12] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, November 1998.

[13] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE Symposium on Security and Privacy*, pages 321–338, May 2015.

[14] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable anonymous group messaging. In *CCS*, pages 340–350, New York, NY, USA, 2010. ACM.

[15] George Danezis, Roger Dingledine, David Hopwood, and Nick Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *IEEE Symposium on Security and Privacy*, pages 2–15, 2003.

[16] George Danezis and Andrei Serjantov. Statistical disclosure or intersection attacks on anonymity systems. In Jessica Fridrich, editor, *Information Hiding*, pages 293–308, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[17] Alfredo De Santis, Yvo Desmedt, Yair Frankel, and Moti Yung. How to share a function securely. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 522–533, New York, NY, USA, 1994. ACM.

[18] Tim Dierks and Eric Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, RFC Editor, August 2008. `http://www.rfc-editor.org/rfc/rfc5246.txt`.

[19] Roger Dingledine. One cell is enough to break tor's anonymity. `https://blog.torproject.org/one-cell-enough-break-tors-anonymity`, February 2009.

[20] Roger Dingledine and Nick Mathewson. Anonymity loves company: Usability and the network effect.

[21] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, pages 303–320. USENIX Association, August 2004.

[22] Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'06, pages 1–12, Berlin, Heidelberg, 2006. Springer-Verlag.

[23] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *CRYPTO*, pages 139–147, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[24] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *CCS*, CCS '02, pages 193–206, New York, NY, USA, 2002. ACM.

[25] Jun Furukawa and Kazue Sako. An efficient scheme for proving a shuffle. In *CRYPTO*, pages 368–387. Springer-Verlag, 2001.

[26] Philippe Golle, Sheng Zhong, Dan Boneh, Markus Jakobsson, and Ari Juels. Optimistic mixing for exit-polls. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 451–465. Springer, 2002.

[27] Jens Groth and Steve Lu. Verifiable shuffle of large size ciphertexts. In *PKC*, pages 377–392, 2007.

[28] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO*, pages 66–97. Springer International Publishing, 2017.

[29] Amir Houmansadr and Nikita Borisov. The need for flow fingerprints to link correlated network flows. In Emiliano De Cristofaro and Matthew Wright, editors, *Privacy Enhancing Technologies*, pages 205–224, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[30] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. Users get routed: Traffic correlation on Tor by realistic adversaries. In *CCS*, November 2013.

[31] Dogan Kedogan, Dakshi Agrawal, and Stefan Penz. Limits of anonymity in open environments. In Fabien A. P. Petitcolas, editor, *Information Hiding*, pages 53–69, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[32] H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication. RFC 2104, RFC Editor, February 1997.

[33] H. Krawczyk and P. Eronen. Hmac-based extract-and-expand key derivation function (hkdf). RFC 5869, RFC Editor, May 2010.

[34] 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*, SOSP '17, pages 406–422, New York, NY, USA, 2017. ACM.

[35] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. In *PETS*, volume 2016, pages 115–134, 2015.

[36] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962, RFC Editor, June 2013.

[37] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Fast and strong metadata privacy with low noise. In *OSDI*, Carlsbad, CA, 2018. USENIX Association.

[38] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *OSDI*, 2016.

[39] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. In *SIGCOMM*, pages 303–314, New York, NY, USA, 2013. ACM.

[40] Moxie Marlinspike and Trevor Perrin. Signal specifications. `https://https://signal.org/docs/`.

[41] Susan E. McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. Investigating the computer security practices and needs of journalists. In *USENIX Security*, pages 399–414, Washington, D.C., August 2015. USENIX Association.

[42] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, Washington, D.C., 2015. USENIX Association.

[43] Steve Mills. Defining the delicate and often difficult relationship between reporters and sources. `https://www.propublica.org/article/ask-propublica-illinois-reporters\-and-sources-relationship`, April 2018.

[44] Prateek Mittal and Nikita Borisov. Shadowwalker: Peer-to-peer anonymous communication using redundant structured topologies. In *CCS*, CCS '09, pages 161–172, New York, NY, USA, 2009. ACM.

[45] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *CCS*, pages 116–125, New York, NY, USA, 2001. ACM.

[46] Andreas Pfitzmann and Marit Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. August 2010.

[47] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *USENIX Security Symposium*, pages 1199–1216, Vancouver, BC, 2017. USENIX Association.

[48] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli Maurer, editor, *EUROCRYPT*, pages 387–398, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[49] Michael K. Reiter and Aviel D. Rubin. Anonymous web transactions with crowds. *Communications of the ACM*, 42(2):32–48, February 1999.

[50] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *IEEE Symposium on Security and Privacy*, pages 444–460, May 2017.

[51] Alin Tomescu and Srinivas Devadas. Catena: Efficient non-equivocation via bitcoin. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 393–409, May 2017.

[52] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *SOSP*, SOSP '17, pages 423–440, New York, NY, USA, 2017. ACM.

[53] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *SOSP*, pages 137–152. ACM, 2015.

[54] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. Captcha: Using hard ai problems for security. In Eli Biham, editor, *EUROCRYPT*, pages 294–311, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[55] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *OSDI*, pages 179–182, Hollywood, CA, 2012. USENIX Association.

A Security of aggregate hybrid shuffle

The adversary's goal is to have an upstream server successfully tamper with some messages without getting detected by the honest server. To model this, we consider the following security game between three parties: the client, the adversary, and the verifier. All parties are given the total number of users M. The client controls users in set $X_H \subset [M]$ (this models the honest users), and the adversary controls users in set $X_A = [M] \setminus X_H$. In addition, the adversary controls servers $1,\ldots,h-1$, and the verifier controls server h (i.e., the honest server). To simplify the presentation, we assume the adversary uses the identity permutation for all servers, but it is easy to adapt this proof to any arbitrary permutation. Table A lists the notation used in aggregate hybrid shuffle, and summarizes their purposes.

1. The adversary sends the client and the verifier the public keys bpk_i and mpk_i and ipk_i for $i = 1,\ldots,h-1$. It also generates NIZKs to prove that it knows the values $\mathsf{bsk}_i = \log_{\mathsf{bpk}_{i-1}}(\mathsf{bpk}_i)$ and $\mathsf{msk}_i = \log_{\mathsf{bpk}_{i-1}}(\mathsf{mpk}_i)$ for $i = 1,\ldots,h-1$, where $\mathsf{bpk}_0 = g$. It sends the public keys and NIZKs to the verifier.
2. The verifier verifies the NIZKs. The verifier then generates the key pairs $(\mathsf{bpk}_h = \mathsf{bpk}_{i-1}^{\mathsf{bsk}_h}, \mathsf{bsk}_h)$, $(\mathsf{mpk}_h = \mathsf{bpk}_{i-1}^{\mathsf{msk}_h}, \mathsf{msk}_h)$, and $(\mathsf{ipk}_h, \mathsf{isk}_h)$, and sends the public keys to the client and the adversary.
3. The client generates random $\{x_j\}_{j\in X_H}$, and $\{c^j = (X_1^j = g^{x_j}, c_1^j)\}_{j\in X_H}$ using the protocol described in §6.2. It also generates a NIZK that it knows x_j that corresponds to X_1^j for each j, and sends both $\{c^j\}$ and the NIZKs to the adversary and the verifier.
4. The adversary generates its input messages $\{c^j = (X_1^j, c_1^j)\}_{j\in X_A}$ (not necessarily by following the protocol in §6.2). It generates a NIZK that shows it knows the discrete log of X_1^j and sends $\{c^j\}_{j\in X_A}$ and the NIZKs to the client and the verifier.
5. The verifier verifies all NIZKs.
6. For $i = 1,\ldots,h-1$, the adversary sends the verifier $\{X_{i+1}^j\}_{j\in[M]}$, and a NIZK that shows
$$\left(\prod_{j=1}^{M} X_i^j\right)^{\mathsf{bsk}_i} = \prod_{j=1}^{M} X_{i+1}^j$$
by proving that
$$\log_{\prod_{j=1}^{M} X_i^j}\left(\prod_{j=1}^{M} X_{i+1}^j\right) = \log_{\mathsf{bpk}_{i-1}}(\mathsf{bpk}_i)\ .$$
It also sends the ciphertexts $\{c_h^{j\prime}\}$ to the verifier.
7. The verifier verifies all NIZKs, and checks that $\mathsf{ADec}((X_h^j)^{\mathsf{msk}_h}, c_h^{j\prime}) = (1,\cdot)$ for all $j \in [M]$.

The game halts if the verifier fails to verify any NIZKs or authenticated decryption ever fails (i.e., returns $(0,\cdot)$). The adversary wins the game if the game does not halt and it has successfully tampered with some messages. In other words, the adversary wins if

1. $\left(\prod_{j=1}^{M} X_i^j\right)^{\mathsf{bsk}_i} = \prod_{j=1}^{M} X_{i+1}^j$ for all $i = 1,\ldots,h-1$,
2. there exists $X_T \subset X_H$ such that $|X_T| > 0$ and for all $j \in X_T$ one of the two properties is true: (1) $(X_1^j)^{\prod_{i<h}\mathsf{bsk}_i} \neq X_h^j$, or (2) $(X_1^j)^{\prod_{i<h}\mathsf{bsk}_i} = X_h^j$ and $c_h^j \neq c_h^{j\prime}$ (i.e., the adversary tampered with some messages),
3. and $\mathsf{ADec}((X_h^j)^{\mathsf{msk}_h}, c_h^{j\prime}) = (1,\cdot)$ for all $j \in [M]$.

We will now show that if the adversary can win this game, then it can also break Diffie-Hellman. Assume the adversary won the game. Let $\mathsf{bsk}_A = \prod_{i<h}\mathsf{bsk}_i$ be the product of the private blinding key of the adversary. If the adversary won, then the first condition implies that $(\prod_{j=1}^{M} X_1^j)^{\mathsf{bsk}_A} = \prod_{j=1}^{M} X_h^j$. Now, consider three boolean predicates for j: $c_h^j \stackrel{?}{=} c_h^{j\prime}$, $(X_1^j)^{\mathsf{bsk}_A} \stackrel{?}{=} X_h^j$, and $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h})$, where $\mathsf{KNOW}(x) = 1$ if the adversary knows (or can compute) x, and 0 otherwise. There are eight possible combinations of the predicates, and we consider

Table 1: Summary of notation used in aggregate hybrid shuffle.

Notation	Description
M	The total number of users.
N	The total number of servers.
n	The total number of chains.
$(\mathsf{isk}_i, \mathsf{ipk}_i = g^{\mathsf{isk}_i})$	Per-server private-public key pair used to encrypt and decrypt the inner most layer of ciphertexts. Used to protect the messages under active attacks by malicious servers or users.
$(\mathsf{bsk}_i, \mathsf{bpk}_i)$	Per-server private-public key pair used to blind the users' Diffie-Hellman keys. After shuffling and decrypting the messages, each server blinds the Diffie-Hellman keys by raising them to the private key bsk_i. This hides which output message is associated with which input while preserving some structures of the Diffie-Hellman keys, which allows us to generate efficient zero-knowledge proofs to prove the correctness of the shuffle. The public component is used to derive the public keys used to encrypt messages for the mix chain (i.e., $\{\mathsf{mpk}_i\}$). bpk_i is $g^{\prod_{k\le i}\mathsf{bsk}_k}$ for server i in a mix chain.
$(\mathsf{msk}_i, \mathsf{mpk}_i)$	Per-server private-public key pair used to encrypt and decrypt messages for shuffling. mpk_i is $(g^{\prod_{k<i}\mathsf{bsk}_k})^{\mathsf{msk}_i}$ for server i in a mix chain.
x^j	The private Diffie-Hellman key of user j used to encrypt the messages for a mix chain. The user uses the same x^j for all layers of encryption, but the effective private Diffie-Hellman key of a layer changes after a server processes the messages due to blinding.
X_i^j	The public Diffie-Hellman key of user j used to encrypt her message for server i. If everyone is behaving correctly, then this should be $(g^{\prod_{k<i}\mathsf{bsk}_k})^{x^j}$.
c_i^j	The actual ciphertext component that encrypts the message. User j will derive the encryption key using its private key x^j and the public mixing key mpk_i. Server i will derive the decryption key using its private mixing key msk_i and X_i^j.

each combination for $j \in X_H$. We indicate which combinations are possible for the adversary to satisfy, given that all authenticated decryptions were successful.

1. $c_h^j \neq c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 0$**: IMPOSSIBLE.** Since the adversary does not know the key used to decrypt (i.e., $(X_h^j)^{\mathsf{msk}_h}$), it cannot generate a valid ciphertext.

2. $c_h^j \neq c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1$**: POSSIBLE.** Since the adversary knows the key used to decrypt it could generate a valid ciphertext.

3. $c_h^j \neq c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 0$**: IMPOSSIBLE.** Same argument as case 1.

4. $c_h^j \neq c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1$**: IMPOSSIBLE.** If possible, then the adversary can break the Diffie-Hellman assumption. Namely, given only $X_1^j = g^{x_j}$, bsk_A, and $(g^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$ for random x_j and msk_h and an independently generated bsk_A, it can compute $(X_h^j)^{\mathsf{msk}_h} = g^{x_j \cdot \mathsf{bsk}_A \cdot \mathsf{msk}_h}$. If this were possible, then given g^a and g^b for random a and b, the adversary could generate an bsk_A, compute $(g^b)^{\mathsf{bsk}_A}$, and compute $g^{a \cdot b \cdot \mathsf{bsk}_A}$. It could then break the Diffie-Hellman assumption and compute g^{ab} by raising $g^{a \cdot b \cdot \mathsf{bsk}_A}$ to bsk_A^{-1}.

5. $c_h^j = c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 0$**: IMPOSSIBLE.** If possible, then c_h^j authenticates under two different keys $(X_1^j)^{\mathsf{bsk}_A \cdot \mathsf{msk}_h}$ and $(X_h^j)^{\mathsf{msk}_h}$. However, if we model the underlying hash function of HMAC as a random oracle, then the HMAC instantiated with a random key is a random function. This implies that the probability that the same ciphertext authenticates under two different keys (i.e., collision of two random functions) is negligible when using Encrypt-then-HMAC.

6. $c_h^j = c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1$**: POSSIBLE.** Since the adversary knows and controls the key used for decryption (in particular, the HMAC), it may be able to find a key that is different from the key used to encrypt that results in valid decryption.

7. $c_h^j = c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 0$**: POSSIBLE.** This corresponds to an untampered message.

8. $c_h^j = c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1$**: IMPOSSIBLE.** Same argument as case 4.

Thus, there are three possible combinations of predicates: $(c_h^j \neq c_h^{j'},\ (X_1^j)^{\mathsf{bsk}_A} \neq X_h^j,\ \mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1)$, $(c_h^j = c_h^{j'},\ (X_1^j)^{\mathsf{bsk}_A} \neq X_h^j,\ \mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1)$, and $(c_h^j = c_h^{j'},\ (X_1^j)^{\mathsf{bsk}_A} = X_h^j,\ \mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 0)$. The first two cases correspond to X_T (tampered messages), and the second corresponds exactly to $X_H \setminus X_T$ (untampered messages).

Similarly, consider $j \in X_A$. The adversary generates the ciphertexts $\{c_h^{j'}\}$ for the verifier. Thus, the adversary must

know $(X_h^j)^{\mathsf{msk}_h}$, the key used to authenticate the ciphertext, for $j \in X_A$.

Now, we consider the product of the users' Diffie-Hellman keys. Because all NIZKs have to be verified, we have that

$$\left(\prod_{j=1}^{M} X_1^j\right)^{\mathsf{bsk}_A} = \left(\prod_{j=1}^{M} X_h^j\right) .$$

Consider $X_U = X_H \setminus X_T$, i.e., the set of messages that did not change. Then, we can divide both sides by the values associated with X_U since $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$ for $j \in X_U$:

$$\left(\prod_{j \in X_T \cup X_A} X_1^j\right)^{\mathsf{bsk}_A} = \left(\prod_{j \in X_T \cup X_A} X_h^j\right) ,$$

since $X_T \cup X_A = [M] \setminus X_U$. We can rewrite this as

$$\left(\prod_{j \in X_T} X_1^j\right)^{\mathsf{bsk}_A} = \left(\prod_{j \in X_T \cup X_A} X_h^j\right) / \left(\prod_{j \in X_A} X_1^j\right)^{\mathsf{bsk}_A} . \quad (1)$$

Based on our analysis of the possible predicates for X_T and X_A, the adversary must know $(X_h^j)^{\mathsf{msk}_h}$ for $j \in X_T \cup X_A$. Moreover, the adversary knows $\log_g(X_1^j)$ for $j \in X_A$ (it was required to prove the knowledge in step 4 of the game). Thus, the adversary can compute $((X_1^j)^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$ by computing $((g^{\mathsf{bsk}_A})^{\mathsf{msk}_h})^{\log_g(X_1^j)}$ for $j \in X_A$ (it knows $\mathsf{mpk}_h = (g^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$). As a result, it can compute

$$\left(\prod_{j \in X_T \cup X_A} \left(X_h^j\right)^{\mathsf{msk}_h}\right) / \prod_{j \in X_A} \left(\left(X_1^j\right)^{\mathsf{bsk}_A}\right)^{\mathsf{msk}_h} .$$

This, however, is

$$\begin{aligned}
&\left(\prod_{j \in X_T \cup X_A} \left(X_h^j\right)^{\mathsf{msk}_h}\right) / \prod_{j \in X_A} \left(\left(X_1^j\right)^{\mathsf{bsk}_A}\right)^{\mathsf{msk}_h} \\
&= \left(\left(\prod_{j \in X_T \cup X_A} X_h^j\right) / \left(\prod_{j \in X_A} X_1^j\right)^{\mathsf{bsk}_A}\right)^{\mathsf{msk}_h} \\
&= \left(\left(\prod_{j \in X_T} X_1^j\right)^{\mathsf{bsk}_A}\right)^{\mathsf{msk}_h} ,
\end{aligned}$$

where the last step uses the equality from equation 1. This means that given $\{X_1^j = g^{x_j}\}$, bsk_A, and $(g^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$ for unknown random $\{x_j\}$ and msk_h, and bsk_A that was generated independently of $\{x_j\}$ and msk_h, the adversary was able to compute $g^{\mathsf{bsk}_A \cdot \mathsf{msk}_h \cdot \sum_{j \in X_T} x_j}$. This essentially breaks Diffie-Hellman.

In more detail, consider the following adversary A_{DH} that tries to break Diffie-Hellman. A_{DH} is given g^a and g^b for random a and b, and is asked to compute g^{ab}. To compute this, A_{DH} simulates the above game by simulating the clients and the honest server. To do so, A_{DH} chooses the blinding keys $\{\mathsf{bsk}_i\}$, and sets $\mathsf{mpk}_h = (\prod_i g^{\mathsf{bsk}_i})^b = (g^b)^{\mathsf{bsk}_A}$; i.e., the private mixing key of the honest server is b. Then, it generates the inputs from the client by generating a random r_j for $j \in X_H$ and setting $g^{x_j} = g^{a+r_j}$. It also generates the inputs associated with each message. The adversary (a simulator) can generate valid NIZKs of knowledge of discrete logs in the random oracle model [48]. For the first layer of encryption (which is for the "honest server" whose secret key is the unknown b), generate random values for c_h^j, since ciphertexts are indistinguishable from random values. For the other layers of encryption, it can generate correctly authenticated ciphertexts by using g^{x_j}, $\{\mathsf{bsk}_i\}_{i<h}$, and $\{\mathsf{msk}_i\}_{i<h}$ without knowing the value of a.

At the end of the game, the adversary can compute $g^{\mathsf{bsk}_A \cdot b \cdot (\sum_{j \in X_T} x_j)}$, as described previously. From this, the adversary can compute the following:

$$\begin{aligned}
g^{\mathsf{bsk}_A \cdot b \cdot (\sum_{j \in X_T} x_j)} &= g^{\mathsf{bsk}_A \cdot b \cdot (\sum_{j \in X_T} (a + r_j))} \\
&= g^{\mathsf{bsk}_A \cdot b \cdot (|X_T| a + \sum_{j \in X_T} r_j)} .
\end{aligned}$$

Since the adversary knows bsk_A and r_j for $j \in X_T$, it can compute

$$g^{|X_T| ab} = (g^{\mathsf{bsk}_A \cdot b \cdot (|X_T| a + \sum_{j \in X_T} r_j)} / (g^b)^{\mathsf{bsk}_A \sum_{j \in X_T} r_j})^{\mathsf{bsk}_A^{-1}} .$$

From this, it can recover g^{ab}, and break Diffie-Hellman. Therefore, the adversary cannot win the above game if Diffie-Hellman is hard, meaning that it could satisfy at most two out of the three conditions to win the game. In turn, this implies that the honest server will always catch an upstream malicious server misbehaving.

B Security game and proof sketches

We define the security of our system using the following security game played between a challenger and an adversary. Both the challenger and the adversary are given the set of users $[M] = \{1, 2, \ldots, M\}$, the set of servers $[N]$, the fraction f of servers the adversary can compromise, and the number of chains n.

1. The adversary selects the set of malicious servers $A_s \subset [N]$ such that $|A_s| \leq f \cdot N$, and the set of malicious users $A_c \subset [M]$ such that $|A_c| \leq M - 2$. The adversary sends A_s and A_c to the challenger. Let $H_s = [M] \setminus A_s$ and $H_c = [N] \setminus A_c$ denote the set of honest servers and users.
2. The challenger computes the size of each chain k as a function of f and n, as described in §5.2. Then, it creates n mix chains by repeatedly sampling k servers per group at random. The challenger sends the chain configurations to the adversary.
3. The adversary and the challenger generate blinding keys, mixing keys, and inner keys as described in §6.1 and Appendix A.

4. The adversary picks some honest users $H_t \subset H_c$ such that $|H_t| \geq 2$. It generates sets of chains $\{C_x\}$ for $x \in H_t$ such that $C_x \cap C_y \neq \emptyset$ for all $x, y \in H_t$. For $\mathsf{c} \in [n]$, let $U_\mathsf{c} = \{x \in H_t : \mathsf{c} \in C_x\}$. For each chain c, it also generates the potential messages $\{m^\mathsf{c}_{xy}\}$ for $x, y \in U_\mathsf{c}$ where m^c_{xy} is the message that may be sent from user x to user y in chain c. The adversary sends H_t, $\{\{m^\mathsf{c}_{xy}\}\}$, and $\{C_x\}$ to the challenger.
5. The challenger first verifies that every pair of C_x and C_y intersects at least once. If this is not the case, the game halts. The challenger then performs the following for each chain $\mathsf{c} \in [n]$. First, it creates conversation pairs for each chain c $\{(X_i, Y_i)_\mathsf{c}\} \subset U_\mathsf{c} \times U_\mathsf{c}$ at random such that every $x \in U_\mathsf{c}$ appears in exactly one of the pairs. In other words, every user has a unique conversation partner per chain. (If $X_i = Y_i$, then that user is talking with herself.) For each $(x, y) \in \{(X_i, Y_i)_\mathsf{c}\}$, the challenger onion-encrypts the messages m^c_{xy} from x to y and m^c_{yx} from y to x with the keys of servers in chain c. Then, it uses the protocol described in §6.2 to submit the ciphertexts and the necessary NIZKs.
6. The adversary generates inputs to the chains for the users in A_c, and sends them to the chains.
7. The challenger and the adversary take turns processing the messages in each chain. Within a chain, they perform the following for $i = 1, \ldots, k$:
   (a) If server $i \in H_s$, the challenger performs protocol described in §6.3 to shuffle and decrypt the messages, and also generate an AHS proof. The challenger then sends the proof to the adversary, and the resulting messages to the owner of server $i+1$.
   (b) If server $i \in A_s$, the adversary generates some messages along with an AHS proof. Then, sends the AHS proof to the challenger, and sends the messages to the owner of server $i+1$.

   The challenger verifies all AHS proofs.
8. The challenger and the adversary decrypt the final result of the shuffle (i.e., the inner ciphertexts).
9. The challenger samples a random bit $b \leftarrow 0, 1$. If $b = 0$, then send the adversary $\{(X_i, Y_i)_\mathsf{c}\}$ for $\mathsf{c} \in [n]$. If $b = 1$, then sample random conversation pairs $\{(X'_i, Y'_i)_\mathsf{c}\} \subset U_\mathsf{c} \times U_\mathsf{c}$ for each chain with the same constraint as in step 5, and send the adversary the newly sampled pairs.
10. The adversary makes a guess b' for b.

The adversary wins the game if the game does not come to a halt before the last step and $b' = b$. The adversary need not follow the protocol described in this paper. The advantage of the adversary in this game is $|\Pr[b' = b] - \frac{1}{2}|$. We say that the system provides metadata private communication if the advantage is negligible in the implicit security parameter.

Note that this game models a stronger version of XRD, which allows users to communicate with multiple users on different chains rather than only one user. We could change the game slightly to force the challenger to send loopback messages in step 5 to model having just one conversation.

Proof sketches. First, we argue that the adversary needs to tamper with messages prior to the last honest server shuffling, as stated in §6. To see why, consider an adversary that only tampers with the messages after the last honest server. The adversary can learn the recipients of all messages, but not the senders. As a result, the adversary does not learn anything about whether two users $x, y \in U_\mathsf{c}$ received messages because there exists a conversation pair $(x, y)_\mathsf{c}$, or because there were two conversation pairs $(x, x)_\mathsf{c}$ and $(y, y)_\mathsf{c}$. This means that any set of conversation pairs is equally likely to be sampled by the challenger from the adversary's view, meaning that the adversary does not gain any advantage. Thus, we consider an adversary who tampers with messages prior to the honest server processing the messages.

In this scenario, the adversary in step 7 must follow the protocol (e.g., no tampering with the messages), as analyzed in Appendix A. Given this restriction, we now argue that the adversary does not learn anything after playing the security game by describing how a simulator of an adversary could simulate the whole game with only the public inputs and the private values of the adversary.

The simulator can simulate step 3 by generating random public keys. It can simulate step 5 by generating random values in place of the ciphertexts that encrypt the users' messages $\{\{m^\mathsf{c}_{xy}\}\}$, since the ciphertexts are indistinguishable from random. It then randomly matches a user in U_c to one of the generated random values for each chain c, and sets the destination of each message as the matched user. It onion-encrypts the final message using the randomly generated public keys and the adversary's public keys. In step 7, the adversary simulates the challenger by randomly permuting the messages, and removing a layer of the encryption from the messages. (It can remove a layer of encryption since it knows all layers of onion-encryption.) Finally, it could simulate the challenger's last challenge by picking sets of randomly generated conversation pairs, subject to the constraints in step 5. The distribution of the messages generated and exchanged in the security game and in the simulator are indistinguishable for a computationally limited adversary.