Albert Kwon*, David Lazar, Srinivas Devadas, and Bryan Ford

Riffle

An Efficient Communication System With Strong Anonymity

Abstract: Existing anonymity systems sacrifice anonymity for efficient communication or vice-versa. Onion-routing achieves low latency, high bandwidth, and scalable anonymous communication, but is susceptible to traffic analysis attacks. Designs based on DC-Nets, on the other hand, protect the users against traffic analysis attacks, but sacrifice bandwidth. Verifiable mixnets maintain strong anonymity with low bandwidth overhead, but suffer from high computation overhead instead.

In this paper, we present Riffle, a bandwidth and computation efficient communication system with strong anonymity. Riffle consists of a small set of anonymity servers and a large number of users, and guarantees anonymity among all honest clients as long as there exists at least one honest server. Riffle uses a new hybrid verifiable shuffle technique and private information retrieval for bandwidth- and computation-efficient anonymous communication. Our evaluation of Riffle in file sharing and microblogging applications shows that Riffle can achieve a bandwidth of over 100KB/s per user in an anonymity set of 200 users in the case of file sharing, and handle over 100,000 users with less than 10 second latency in the case of microblogging.

Keywords: anonymous communication; verifiable shuffle; private information retrieval

DOI 10.1515/popets-2016-0008
Received 2015-08-31; revised 2015-12-02; accepted 2015-12-02.

1 Introduction

The right to remain anonymous is a fundamental right in a democratic society and is crucial for freedom of speech [49]. Anonymizing networks based on relays such as Tor [26] have been gaining popularity as a practical privacy enhancing technology among users seeking higher levels of privacy. However, such systems are susceptible to traffic analysis attacks [36, 43] by powerful adversaries such as an authoritarian government or a state controlled ISP, and have recently been attacked by even weaker adversaries monitoring only the users' traffic [14, 32, 39, 51].

There are two major branches of work that offer traffic analysis resistance even in the presence of a powerful adversary. The first is Dining-Cryptographer Networks (DC-Nets) [16], which offer information theoretically secure anonymous communication for users as long as one other participant is honest. Dissent [53] improved upon DC-Nets by moving to the *anytrust* model, where the network is organized as servers and clients, and guarantees anonymity as long as there exists one honest server. The second is verifiable mixnets, based on mix networks [18]. In this design, the mixes use a verifiable shuffle [7, 13, 28, 37] to permute the ciphertexts, and produce a third-party verifiable proof of the correctness of the shuffle without revealing the actual permutation. Similar to DC-Net based systems, verifiable mixnets guarantee anonymity as long as one mix in the network is honest.

Both designs, however, suffer from serious drawbacks. DC-Nets and DC-Net based systems, by design, implement a broadcast channel. That is, they were primarily designed for the case where one client messages everyone in the network. Thus, when multiple users wish to communicate simultaneously, every user must transfer a message of size proportional to the number of clients who wish to communicate. As a result, DC-Net based designs suffer from a large bandwidth overhead, and only scale to a few thousand clients [23, 53]. Verifiable mixnets, on the other hand, allow the clients to send messages of size proportional only to their own messages, and thus can be bandwidth efficient. However, the high computation overhead of verifiable shuffles has prevented verifiable mixnets from supporting high bandwidth communication.

In this paper, we present Riffle, a system for bandwidth- and computation-efficient anonymous communication. Riffle addresses the problems of DC-Nets and verifiable mixnets, while offering the same level of anonymity. At a high level, Riffle is organized as servers and clients, similar to previous works [21, 23, 53]. Rif-

***Corresponding Author: Albert Kwon:** MIT, E-mail: kwonal@mit.edu
David Lazar: MIT, E-mail: lazard@mit.edu
Srinivas Devadas: MIT, E-mail: devadas@mit.edu
Bryan Ford: EPFL, E-mail: bryan.ford@epfl.ch

fle focuses on minimizing the bandwidth consumption of the clients, who may be connecting from bandwidth-constrained environments such as their mobile phones, and reducing the computation overhead on the servers so they can support more clients. Specifically, the clients in Riffle consume upstream bandwidth proportional only to the size of their messages rather than the number of clients, and the server computation only involves fast symmetric key cryptography in the common case. This allows the users to exchange messages efficiently, making it suitable for applications like file sharing that no existing strong anonymity system can support well. Moreover, Riffle provides strong anonymity for all clients as long as one of the servers is honest.

Riffle achieves bandwidth and computation efficiency by employing two privacy primitives: a new hybrid verifiable shuffle for upstream communication, and private information retrieval (PIR) [20] for downstream communication. Our novel hybrid verifiable shuffle scheme avoids using an expensive verifiable shuffle in the critical path, and employs authenticated encryption to improve both bandwidth and computation overhead of the shuffle without losing verifiability. We also propose a novel application of private information retrieval in the anytrust setting. Previous strong anonymity systems made a trade-off between computation and bandwidth by either broadcasting all messages to all users (low computation, high bandwidth) [21, 23, 53], or using computationally expensive PIR (high computation, low bandwidth) [47]. In the anytrust model, we show that PIR can minimize the download bandwidth with minimal computation overhead.

We also develop a Riffle prototype and two applications. The first is an anonymous file sharing application, where each message is large and is potentially of interest to only a small number of users. Sharing large files is a scenario that has not been considered carefully by previous strong anonymity systems [7, 21, 53]; we propose a new file sharing protocol that leverages and illustrates Riffle's bandwidth-efficiency. The second application is for anonymous microblogging, similar to applications studied in previous works [21, 53]. In this setting, each user posts small messages to the server, and each message is of interest to many users.

Our prototype demonstrates effectiveness for both applications. In file sharing, the prototype achieves high bandwidth (over 100KB/s) for more than 200 clients. In microblogging, the prototype can support more than 10,000 clients with latency less than a second, or handle more than 100,000 clients with latency of less than 10 seconds. We show that our results are orders of magnitude better in terms of both scalability and bandwidth compared to previous systems offering traffic analysis resistance [7, 48, 53] in comparable scenarios.

This paper makes the following contributions:

1. A hybrid verifiable shuffle that uses symmetric encryption, and avoids expensive public key verifiable shuffling [7, 13, 28, 37] in the common case.
2. A novel application of private information retrieval [20] in anytrust settings.
3. A bandwidth- and computation-efficient anonymous communication system that is resilient against traffic analysis attacks and malicious clients.
4. Evaluation of Riffle that demonstrates efficiency in two contrasting applications.

In Section 2, we describe related work. In Section 3 and Section 4, we explain the threat model and deployment model of Riffle and describe the protocol in detail. We then describe our file sharing protocol in Section 5, and evaluate our prototype in Section 6. Finally, we discuss future work in Section 7, and conclude in Section 8.

2 Background and Related Work

In this section, we describe related work, focusing on the trade-offs made by existing anonymity systems.

2.1 Proxy-Based Anonymity Systems

Tor [26] is a popular anonymity system that focuses on scalability and low latency by routing users' messages through decentralized volunteer relays without delays or cover traffic. While this design allows Tor to scale to millions of users [6], a powerful adversary who can observe traffic going in and out of the relay network (e.g., a state controlled ISP) can deanonymize users [36, 43]. One recently-discovered attack enables even a *local* adversary, observing only entry-relay traffic, to deanonymize users with high probability [14, 32, 33, 39, 51].

Mix networks (mixnets) [18] and systems that build on them [24, 27, 44–46] aim to prevent traffic analysis attacks by routing each user's message through a set of anonymity servers called *mixes*. Mixes collect many users' inputs and shuffle them before sending any out, making it difficult to correlate inputs and the outputs even for a global adversary. Because the mixes can be distributed and decentralized, mixnets can scale to a large number of clients [45], and provide reasonable latency and bandwidth when the mixes are wellprovisioned. With malicious mixes in the network, however, mixnets fail to provide the level of anonymity required for sensitive activities like whistleblowing. Several proposed attacks [38, 40, 41, 52] allow a malicious mix to deanonymize users by dropping, modifying, or duplicating input messages before sending them out.

Aqua [34] aims to provide low-latency and high-bandwidth anonymity with traffic analysis resistance by using Tor-like infrastructure with mixes. Aqua attaches each client to exactly one mix called the edge mix, and provides k-anonymity among the k honest clients connected to the same edge mix. In its threat model, Aqua assumes that the honest clients are connected to uncompromised edge mixes, and that the adversary controls the network and edge mix on only one end of the path between two clients. Though Aqua provides traffic analysis resistance in this model, these assumptions result in security problems similar to Tor: it is not readily possible for clients to determine which edge mix is uncompromised, and powerful adversaries controlling both ends of the circuit can still deanonymize clients.

2.2 Anonymity versus Bandwidth

Unlike systems using anonymizing proxies, Dining Cryptographer Networks (DC-Nets) [16] provide information theoretic anonymity even in the face of global adversaries as long as there exists at least one other honest participant. However, they require communication between *every* user to broadcast one message from a single user, and thus incur high bandwidth overhead. As a result, systems that build on DC-Nets could not scale to more than a few tens of clients [22, 30]. Recent DC-Net based systems [23, 53] use the *anytrust* model, where the network is organized as servers and clients, and guarantee anonymity as long as one of the servers is honest. The new model allowed these designs to scale to a few thousand clients with strong anonymity, but they still suffer from a bandwidth penalty proportional to the number of clients and the message size.

Riposte [21] is a recent system optimized for anonymous microblogging. Each client in Riposte uses a novel "private information storage" technique to write into a shared database maintained by multiple servers. It follows a threat model similar to Dissent, where anonymity is guaranteed as long as one server is honest, and offers good throughput for small messages. However, each Riposte client must submit a message proportional to the square root of the size of the whole database (i.e., collection of *all* clients' data), making it unsuitable for sharing large messages among many clients.

2.3 Anonymity versus Computation

The classic mixnets approach described earlier may be strengthened against malicious mixes by using verifiable shuffles [7, 13, 28, 37]. In this design, when a mix shuffles the inputs, it also generates a zero-knowledge proof that the outputs form a valid permutation of the input ciphertexts, while hiding the permutation itself.[1] Using the proof and the ciphertexts, other parties can verify that the mix did not tamper with any message, while learning nothing about the permutation. Assuming at least one of the mixes is honest, a verifiable mixnet is secure even with compromised mixes in the network: The honest mix alone shuffles inputs sufficiently to thwart traffic analysis attacks, and malicious mixes cannot tamper with messages without generating a bad proof. However, generation and verification of such proofs is computationally expensive, resulting in high latency and low bandwidth. The state-of-the-art verifiable shuffle by Bayer and Groth [7], for instance, takes 2 minutes to prove and verify a shuffle of 100,000 short messages.

Another privacy primitive that trades computation for anonymity is private information retrieval (PIR) [20]. While most anonymity systems mentioned previously focus on protecting the senders' anonymity, PIR protects the privacy of the receiver. In PIR, a client accesses some data in a database managed by a server or a collection of servers, and the goal is to hide *which* data was accessed. There are variants of PIR for different settings [19, 20, 29], but many schemes have complex formulation, and incur significant overheads in computation. On the other hand, the original PIR scheme proposed by Chor *et al.* [20] is efficient in terms of both computation and bandwidth, but has a weaker threat and usage model than other schemes: it requires multiple servers each with a copy of the database, and at least one of the servers needs to be honest. However, this is precisely the setting of anytrust, and we show that efficient PIR can be used in a practical system to minimize the downstream bandwidth overhead.

1 Inputs to a verifiable shuffle are probabilistically encrypted and re-randomized by the mix, preventing attackers from associating inputs with outputs to learn the permutation.

Table 1. Notations used.

Terminology	Description
C	Set of Riffle clients
n	Number of clients
S	Set of Riffle servers
m	Number of servers
b	Size of a message
λ	Security parameter

3 Models and Assumptions

This section presents the system and threat models of Riffle, and its assumptions. We also briefly define its security properties. Table 1 summarizes notations used.

3.1 System Model and Goals

A Riffle system consists of *clients* and *servers*, as illustrated in Figure 1. The clients form the anonymity set of individuals who wish to communicate anonymously, and the servers collectively form an anonymity provider. For deployment, we assume that each server is run by separately administered entities, such as different commercial or non-profit anonymity providers. Each client in a Riffle group is connected to his or her preferred server called *primary server*, chosen based on factors such as location, hosting organization, etc. This paper focuses on supporting anonymous intra-group communication among clients in the same group, as in a chatroom or file-sharing group, and not on Tor-like anonymous communication with other groups or the Internet at large. Section 7 discusses other possible usage models.

We assume that the most precious resource in Riffle's setting is the bandwidth between clients and servers. Provisioning a high-bandwidth network between a small number of servers is feasible and already common (e.g., between data centers, large companies, etc). However, we cannot expect all clients to have high-bandwidth connections to the servers due to clients connecting from locations with poor infrastructure, expensive mobile connections, etc. Therefore, Riffle focuses on minimizing client-server bandwidth requirements. We discuss the impact of the bandwidth between servers in Sections 6 and 7.

This paper assumes that a Riffle group is already established, focusing on the operation of a single group. That is, we do not consider how the clients select the servers to ensure presence of an honest server, or how each client determines the appropriate group

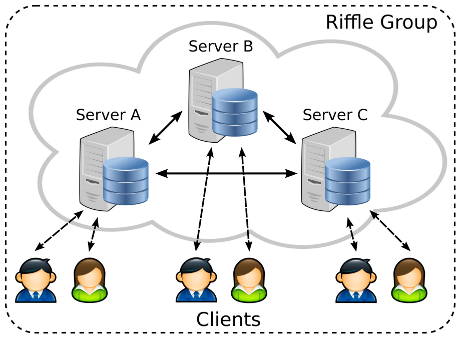


Fig. 1. Deployment model of Riffle

(anonymity set) size. Previous works [30, 48] have explored these problems in detail, and their solutions are applicable to Riffle as well.

Finally, Riffle aims to prevent traffic analysis attacks. To do so, communication in Riffle is carried out in *rounds*, similar to previous designs with traffic analysis resistance [21, 53]. In each round, every client sends and receives a message, even if he or she does not wish to communicate that round.

3.2 Threat Model

Riffle assumes an *anytrust* model for its threat model [21, 53]. Riffle does not depend on a fraction of the servers, or even a particular server, being honest to guarantee anonymity. We rely only on the assumption that there *exists* an honest server. In particular, despite the clients being connected to only one server, we guarantee anonymity of *all* honest clients in a group as long as there exists an honest server in the group. Apart from having one honest server, we do not limit the adversary's power: we allow the adversary to control any number of servers and all but k clients for any $k \geq 2$. Riffle aims to provide anonymity among the k honest clients, even in the presence of many malicious clients and servers.

Riffle requires that all network communication be done through authenticated and encrypted channels, such as TLS. Moreover, variable-length messages must be subdivided into fixed-length blocks and/or padded to prevent privacy leakage through message size. We assume this is done for all communication, and henceforth omit this detail to simplify presentation. We also note that our focus is on anonymity and not confidentiality of the messages. If confidentiality is desired, then public keys can be shared through a round of anonymous communication, and end-to-end encryption can be used in subsequent rounds.

3.3 Security Properties

Riffle provides three main security properties. The first is correctness, which guarantees that Riffle is a valid communication system.

Definition 1. *The protocol is correct if, after a successful run of the protocol, every honest client's messages are available to all honest clients.*

In addition to correctness, Riffle aims to provide two anonymity properties: sender anonymity and receiver anonymity. The first is crucial for users who share security critical information, such as whistleblowers and protest organizers. For instance, a journalist uploading a sensitive document exposing government corruption would not want the post to be traceable back to him or her. Informally, sender anonymity is the property that no adversary described in Section 3.2 can determine which honest client sent which messages better than guessing randomly.

Definition 2. *The protocol provides sender anonymity if, for every round of communication, the probability of any efficient adversary successfully guessing the honest client that sent a particular honestly generated message is negligibly close (in the implicit security parameter) to* $\frac{1}{k}$ *where* k *is the number of honest clients.*

Intuitively, this means that the adversary cannot deanonymize an honest sender (i.e., link a message to a client) better than guessing at random from the set of honest clients. We formalize this definition in Appendix A.

Receiver anonymity is the complementary property to sender anonymity; for instance, a user downloading documents exposing government corruption could face prosecution for simply accessing the sensitive material. The property states that no adversary can learn which messages were downloaded by an honest client.

Definition 3. *The protocol provides receiver anonymity if, for every round of communication, the probability of any efficient adversary successfully guessing which of the* n *messages was received by any particular honest client is negligibly close (in the implicit security parameter) to* $\frac{1}{n}$*, where* n *is the number of available messages.*

The definition for receiver anonymity is with respect to $\frac{1}{n}$, where n is the total number of messages (which is equal to the total number of malicious and honest clients), instead of $\frac{1}{k}$. For sending messages, honest clients can only hide among the other honest clients since the malicious clients' messages are already known to the adversary. For receiving, however, the clients do not produce any new messages, and the only information available to the adversary is the metadata, which we aim to hide as much as possible. We thus would like receiving messages to be as secure as all clients broadcasting all messages, which hides as much metadata as possible and limits the probability of the adversary successfully learning which message an honest client was actually receiving to $\frac{1}{n}$.

4 Riffle Architecture

This section first starts with straw-man approaches that fail to achieve our goals but illustrate the challenges, and a baseline protocol that addresses the security problems of the straw-man solutions with some inefficiency. We then present the cryptographic primitives used in Riffle and the final Riffle protocol. We also describe how to hold a malicious client accountable, without any privacy leakage and performance overhead during regular operation. Finally, we analyze the asymptotic bandwidth requirement, and provide a security argument for Riffle.

4.1 Straw-man Protocols

We outline two straw-man protocols: the first fails to provide sender anonymity, while the second fails to provide receiver anonymity.

4.1.1 PIR-Only

We first consider a scheme that only uses PIR:

1. Clients upload their encrypted messages to the servers. If a client has no message to send, then he/she sends a random string.
2. The servers share all ciphertexts with each other.
3. All clients perform PIR to download a message at a particular index. If a client has no message to download, then he/she downloads a message at a random index.

Though other details need to be specified (e.g., which key to use for message encryption, how to learn the index of the message to perform PIR, etc.), any simple variant of this scheme where clients naively upload messages cannot provide sender anonymity. When user i

uploads a message for user j, user j can collude with the malicious servers to learn who uploaded the message.

4.1.2 Shuffle-Only

We can also consider a scheme that only uses shuffling.

1. Clients download all public keys of the servers.
2. Clients upload onion-encrypted ciphertexts to the servers.
3. The servers shuffle and onion-decrypt the ciphertexts. The final server publishes the plaintext messages.
4. Each client downloads the block of interest.

There are several problems with this scheme. First, if the shuffle is not verifiable, then this scheme does not provide sender anonymity. For example, if the first server is malicious, then it can duplicate an honest client's message and put it in place of a malicious client's message. When the plaintexts are revealed at the end, the first server can see the duplicate messages. The probability of the adversary correctly guessing the honest client that sent the duplicated message is now 1, and thus fails to provide sender anonymity.

Even if the shuffle is verifiable, this scheme fails to provide receiver anonymity: the servers immediately learn which message each client downloaded.

4.2 First Attempt: Baseline Protocol

We now propose a naive solution that will serve as the basis for the full Riffle protocol. To avoid the shortcomings of the straw-man solutions, we use both verifiable shuffle and broadcast for communication. This protocol is carried out in *epochs*, each of which consist of two phases: setup and communication. The setup phase is used to share keys and happens only once at the beginning of an epoch. The communication phase consists of multiple rounds, and clients upload and download messages to and from the servers in each round, as discussed in Section 3.

During the setup phase, every server S_i generates a public key p_i, and all keys are published to the clients and servers in the group. Each round in the communication phase consists of three stages: (1) upload, (2) shuffle, and (3) download. In the upload stage, client C_j onion-encrypts his or her message with all public keys $\{p_i : i \in [1, m]\}$, and uploads the ciphertext to his or her primary server. Once all ciphertexts are uploaded, the first server S_1 collects the ciphertexts.

In the shuffle stage, starting with S_1, the servers perform a verifiable shuffle and verifiable decryption. Each server sends the proof of decryption and shuffle along with the decrypted ciphertexts to all other servers, who will then verify the proofs. Decryption, shuffling, and verification of proofs are repeated until the last server finally reveals all plaintexts to the servers.

Finally, in the download stage, all plaintext messages are broadcast to all clients through the clients' primary servers. Though the download bandwidth overhead grows linearly with the number of clients due to broadcast, this design significantly reduces the upload bandwidth overhead between the clients and the servers compared to DC-Net based designs, as the ciphertext each client uploads is proportional only to the actual message.

4.3 Hybrid Verifiable Shuffle

Despite significant bandwidth savings, the computational overhead of verifiable shuffles makes the baseline protocol unsuitable for high bandwidth communication. As a concrete example, the state-of-the-art verifiable shuffle proposed by Bayer and Groth [7] takes more than 2 minutes to shuffle 100,000 ElGamal ciphertexts, each of which corresponds to a small message (e.g., a symmetric key). Furthermore, randomized public key encryption schemes commonly used for verifiable shuffle, such as ElGamal, often result in ciphertext expansion of at least 2, which halves the effective bandwidth.

To address the issues of traditional verifiable shuffles, we propose a new hybrid verifiable shuffle. In a hybrid shuffle, a traditional verifiable shuffle, such as [7, 13, 28, 37], is performed only once to share encryption keys that are used throughout an epoch. Instead of public key cryptography, we then use authenticated encryption with the shared keys, and verify the shuffle through authenticating ciphertexts. Intuitively, we are bootstrapping verifiablility from the initial verifiable shuffle of keys.

The shuffle can be described as a protocol carried out among three non-colluding parties: the client, the prover, and the verifier. The goal is for the client to send R sets of n messages to the verifier using the prover, while satisfying two properties. First, the verifier does not learn the order of the messages in each set (zero-knowledge property). Second, the verifier should be able to check that the prover did not tamper or drop any messages (verifiability of shuffles). The details of the shuffle are presented in Algorithm 1, and the se-

Algorithm 1 Hybrid Shuffle

```
1. Share Keys: Prover $P$ and verifier $V$ generate public-private key pairs $(s_P, p_P)$ and $(s_V, p_V)$, and publish public keys $p_P$ and $p_V$. Client $C$ shares keys $\{k'_j\}_{i\in[n]}$ with $P$.
2. Shuffle Keys: $C$ generates keys $\{k_j\}_{j\in[n]}$ for verifier $V$, and sends $\{Enc_{p_P}(Enc_{p_V}(k_j))\}_{j\in[n]}$ to $P$ and $V$. $P$ performs a verifiable decryption and verifiable shuffle using a random permutation $\pi$, and sends $\{Enc_{p_V}(k_{\pi(j)})\}_{j\in[n]}$ to $V$. $V$ verifies the shuffle and decryption, and decrypts to learn $\{k_{\pi(j)}\}_{j\in[n]}$.
3. Send Messages: For $r = 1, \ldots, R$,
    (a) Shuffle: To send messages $\{M^r_j\}_{j\in[n]}$, $C$ onion-encrypts the messages and sends $\{AEnc_{k'_j,r}(AEnc_{k_j,r}(M^r_j))\}_{j\in[n]}$ to $P$, where $AEnc$ is an authenticated encryption scheme that uses $r$ as the nonce. $P$ decrypts a layer of encryption using $\{k'_j\}_{j\in[n]}$, permutes them using the same $\pi$, and sends $\{AEnc_{k_{\pi(j)},r}(M^r_{\pi(j)})\}_{j\in[n]}$ to $V$.
    (b) Verify: $V$ verifies the ciphertexts by checking authenticity using the keys $\{k_{\pi(j)}\}_{j\in[n]}$ and $r$, decrypts a layer, and learns $\{M^r_{\pi(j)}\}_{j\in[n]}$.
```

curity properties of the hybrid shuffle are analyzed in Section 4.8. We describe how we use the hybrid shuffle in our protocol in Section 4.5; in Riffle, every server at some point in the protocol behaves as the prover, and the servers' downstream server behaves as the verifier.

4.4 Private Information Retrieval in Riffle

There are situations where a client is not interested in the majority of the messages. For example, consider two clients chatting through Riffle. If the messages are shuffled using the same permutation every round, the clients can learn the expected location of the messages (i.e., the index) after one round of communication, but the clients are forced to download *all* available messages in case of broadcast. Moreover, as we will see in Section 5, the indices can sometimes be learned by downloading small meta data which are asymptotically smaller than the actual messages. In these scenarios, we can use multi-server private information retrieval (PIR) proposed by Chor *et al.* [20] to improve download efficiency. In this PIR scheme, let I_j be the index (location) of the message client C_j wants to download. To download the message, C_j first generates $m-1$ random bit masks each of length n, and computes a mask such that the XOR of all m masks results in a bit mask with a 1 only at position I_j. Each mask is sent to a server, and each server S_i XORs the messages at positions with 1 in the bit mask to generate its response r_{ij} for C_j. Finally, C_j downloads all $\{r_{ij}\}_{i\in[m]}$ and XORs them together to learn the plaintext message.

Although this scheme is fairly efficient, we can further reduce the bandwidth overhead using pseudorandom number generators (PRNGs). To avoid sending masks to all servers every round, C_j shares an initial mask with all servers during the setup phase. Each server then updates the mask internally using a PRNG every round, and C_j only sends a mask to its primary server S_{p_j} to ensure the XOR of all masks has a 1 only in position I_j.

To avoid downloading a message from every server, C_j can ask S_{p_j} to collect all responses and XOR them together. However, doing so naively results in S_{p_j} learning the message C_j downloaded. To prevent this problem, C_j shares another set of secrets with every server during the setup, and each server XORs its secret into the response. S_{p_j} can now collect the responses and XOR them, while learning nothing about which message C_j is interested in. Finally, C_j can download the response from S_{p_j} (i.e., the message XORed with the shared secrets), and remove the secrets to recover the message. Similar to the masks, the servers and the clients can internally update the secrets using a PRNG. Since PIR hides which data was accessed, the sharing of masks and secrets need not be through a verifiable shuffle; we do not need to hide which secret is associated with which client. However, each client must perform PIR every round to remain resistant to traffic analysis attacks even if the client is not interested in any message. Algorithm 2 demonstrates the exact details of the optimized PIR.

4.5 Riffle Protocol

We now present the full Riffle protocol. During the setup phase, the clients share three sets of pairwise secrets with the servers: (1) $\{k_{ij}\}$ using a verifiable shuffle (used in the hybrid shuffle), and (2) $\{m_{ij}\}$ and (3) $\{s_{ij}\}$ using simpler methods like Diffie-Hellman [25] (used in the PIR). Each server generates permutations π_i for verifiable shuffle, and retain them for future use during the communication phase. Key k_{ij} will be at position $\pi_{i-1}(\ldots(\pi_1(j))\ldots)$ in S_i at the end of the setup.

In round r of the communication phase, the protocol uses hybrid shuffle for upload and PIR or broadcast for download. In the upload stage, each client

Algorithm 2 Private Information Retrieval

```
1. Setup (Share Secrets): Each client $C_j$ shares two secrets $m_{ij}$ and $s_{ij}$ with each $S_i$ except for the primary server $S_{p_j}$ it is connected to. This step happens only once per epoch.
2. Download:
   (a) Mask Generation: Let index $I_j$ be the index of the message $C_j$ wants to download. $C_j$ generates $m_{p_j j}$ such that $\bigoplus_i m_{ij} = e_{I_j}$ where $e_{I_j}$ is a bit mask with 1 only in slot $I_j$. $C_j$ then sends $m_{p_j j}$ to $S_{p_j}$.
   (b) Response Generation: Each server $S_i$ computes the response $r_{ij}$ for $C_j$ by computing the XOR sum of the messages at the positions of 1s in the $m_{ij}$, and XORing the secret $s_{ij}$. Specifically, response $r_{ij} = \left(\bigoplus_\ell m_{ij}[\ell] \cdot M_\ell\right) \oplus s_{ij}$, where $M_\ell$ is the $\ell$th plaintext message. Then, the servers send $r_{ij}$ to $S_{p_j}$, and $S_{p_j}$ computes $r_j$:
       $r_j = \bigoplus_i r_{ij} = \left(\bigoplus_i \bigoplus_\ell m_{ij}[\ell] M_\ell\right) \oplus \left(\bigoplus_i s_{ij}\right)$
           $= M_{I_j} \oplus \left(\bigoplus_i s_{ij}\right)$
   (c) Message Download: $C_j$ downloads $r_j$ from $S_{p_j}$, and XORs all $\{s_{ij}\}_{i\in[m]}$ to compute the message of interest, $M_{I_j} = r_j \oplus \left(\bigoplus_i s_{ij}\right)$.
   (d) Update Secrets: Both $C$ and $S$ apply PRNG to their masks and secrets to get fresh masks and secrets.
```

C_j onion-encrypts a message using $\{k_{ij}\}_{i\in[m]}$, and uploads the ciphertexts to S_1 via C_j's primary server. In the shuffle stage, starting with S_1, each server S_i authenticates and decrypts the ciphertexts using the shared keys $\{k_{ij}\}_{j\in[n]}$, shuffles them using the π_i saved from the setup phase, and sends the result to the next server. This means in S_i, ciphertext $AEnc_{k_{ij},r}(\ldots(AEnc_{k_{mj},r}(M_j^r))\ldots)$ of C_j is at position $\pi_{i-1}(\ldots(\pi_1(j))\ldots)$, which is where the matching key k_{ij} is. The last server finally reveals the plaintext messages to all servers. We note that the final permutation of the messages is $\pi = \pi_m(\pi_{m-1}(\ldots(\pi_2(\pi_1))\ldots))$.

In the download stage, the clients either perform PIR (Section 4.4) with the masks and the secrets, or download all messages through broadcast. We will see a concrete scenario where PIR is used instead of broadcast in Section 5. The final Riffle protocol is described in Algorithm 3.

Algorithm 3 Riffle Protocol

```
1. Setup:
   (a) Shuffle Keys:
       i. Each server $S_i$ generates public keys pairs, and publishes the public key $p_i$ to the clients. $S_i$ also generates permutation $\pi_i$.
       ii. Each client $C_j$ generates key $k_{ij}$ for $S_i$, and encrypts them with keys $p_1, \ldots, p_i$ for $i = 1, \ldots, m$. $C_j$ submits $m$ onion-encrypted $\{k_{ij}\}_{i\in[m]}$ to $S_1$ via the primary server.
       iii. From $S_1$ to $S_m$, $S_i$ verifiably decrypts the encrypted keys. $S_i$ then retains $\{k_{ij}\}_{j\in[n]}$, verifiably shuffles the other keys using $\pi_i$, and sends the shuffled keys to $S_{i+1}$. The servers verify the decryption and shuffle.
   (b) Share Secrets: Every pair of $S_i$ and $C_j$ generates pairwise secrets $m_{ij}$ and $s_{ij}$, used for PIR (Algorithm 2) in the download stage.
2. Communication: In round $r$,
   (a) Upload: $C_j$ onion-encrypts the message $M_j^r$ using authenticated encryption with $\{k_{ij}\}_{i\in[m]}$ and $r$ as a nonce: $AEnc_{1,\ldots,m}(M_j^r) = AEnc_{k_{1j},r}(\ldots(AEnc_{k_{mj},r}(M_j^r))\ldots)$. $C_j$ then sends $AEnc_{1,\ldots,m}(M_j^r)$ to $S_1$ via $C_j$'s primary server.
   (b) Shuffle: From $S_1$ to $S_m$, $S_i$ authenticates, decrypts, and shuffles ciphertexts using the $\pi_i$, and sends the shuffled $\{AEnc_{i+1,\ldots,m}(M_j^r)\}_{j\in[n]}$ to $S_{i+1}$. $S_m$ shares the final plaintext messages with the all servers.
   (c) Download: The clients download the plaintext message(s) through PIR or broadcast.
```

4.6 Accusation

In DC-nets [16] or DC-net based designs [53], it is easy for a malicious client to denial-of-service attack the whole network. Namely, any client can XOR arbitrary bits at any time to corrupt a message from any user. This problem has led to complex, and often costly, solutions to hold a client accountable, such as trap protocols [50], trap bits [53], and verifiable DC-nets [23], or limited the size of the group to minimize the attack surface [30].

Without any precautions, a similar attack is possible in Riffle as well: during upload, a malicious client could send a mis-authenticated ciphertext, and the system will come to a halt. However, unlike DC-nets, shuffling does not allow one client's input to corrupt others' inputs. Leveraging this fact, Riffle provides an efficient way to hold a client accountable while introducing no overhead during regular operation and leaking no privacy of honest clients: when a server S_i detects that a ciphertext at its position j is mis-authenticated, it begins the accusation process by revealing j to S_{i-1}. S_{i-1} then reveals $j' = \pi_{i-1}^{-1}(j)$, and this continues until S_1 reveals the client who sent the problem ciphertext.[2]

A malicious client, however, cannot perform a similar attack during download. When the messages are broadcast, there is nothing that a client can do to disrupt the communication. When the clients use PIR to download the message, there is no notion of mis-authenticated ciphertexts or "illegal" messages to cause the system to halt, as masks and secrets are random values. Moreover, similar to the upload stage, a malicious client cannot corrupt other clients' messages since every client generates its own masks and secrets.

4.6.1 Accusation with Malicious Servers

Revealing the inverse permutation unconditionally, however, could be abused by a malicious server: any server could flag an error for any client, and that client will be deanonymized since other servers have no way to verify the claim. To prevent this problem, Riffle uses the ciphertexts of the keys used during the setup phase as the commitments of the keys for all server secrets. We describe the commitment scheme in detail in Section 4.6.2, and the accusation algorithm in Algorithm 4. After each step of Algorithm 4, the party responsible for the verification sends out the proof generated at each step, signed with its private key, to all other servers. The accusation continues if and only if every server agrees that the proof is correct. Intuitively, the goal of this algorithm, apart from successful accusation, is to ensure no server misbehaves by checking that

1. The revealed keys are the ones that were shared between the accused client and the servers.
2. The revealed ciphertexts are encryptions / decryptions of each other, which shows that all of them have originated from a single client.

When any step fails or times-out, or the accusation finishes, every server sends all the transcript of the accusation thus far (i.e., all the messages and proofs exchanged among the servers) to all clients. The clients can now verify the accusation. If the transcript verifies, then the group collectively removes the accused client. If the transcript does not verify, then the clients determine which server caused the failure using the transcript. The clients and the other servers then kick out the server that caused the failure from the group. In both cases, a new Riffle group is formed after an accusation.

Algorithm 4 Accusation in Riffle

```
Server $S_i$ starts the accusation by revealing position $j$ of the potentially malicious client, the problem ciphertext $E_{ij}$, and the associated key $k_{ij}$. For $\ell = i-1, \dots, 1$,
  1. $S_\ell$ verifies $E_{\ell+1,j}$ is the ciphertext at location $j$.
  2. All servers verify the authenticity of $E_{\ell+1,j}$ using $k_{\ell+1,j}$.
  3. All servers verify $k_{\ell+1,j}$ with the commitment (Section 4.6.2).
  4. $S_\ell$ reveals $j' = \pi_\ell^{-1}(j)$, the corresponding secret $k_{\ell j'}$, and ciphertext $E_{\ell j'}$ to all other servers.
  5. All servers verify that decryption of $E_{\ell j'}$ using $k_{\ell j'}$ is $E_{\ell+1,j}$. Set $j = j'$.
Each server sends the transcript of the above steps to all clients once the accusation finishes.
```

4.6.2 Key Commitments

Step 2 of Algorithm 4 requires the commitments of the keys to be tied to the actual keys that were shared during the setup phase. Otherwise, a malicious server may commit a key k' that is different from the key k it shared with the client, reveal k' for accusation (which matches the commitment), and falsely accuse a client. Riffle thus uses the ciphertexts from the setup as the commitment for the keys, which have a proof, as part of the initial verifiable shuffle, that they originated from the clients.

In general, checking that a ciphertext is an encryption of a plaintext either requires the randomness used to encrypt the original message or the secret key. However, we can prove this without revealing either values with a zero-knowledge proof if we use ElGamal encryption. The protocol is carried out between a prover and a verifier, where the prover is the server revealing the key for accusation, and the verifier is any other server in the group. The prover has its secret key s, and the prover and the verifier both have access to the plaintext message k' (the key revealed for accusation), a ciphertext $c = (g^r, k \cdot g^{rs})$ for some random r (the ciphertext from the setup phase), and the public key g^s.

2 Any inadvertent transmission of a mis-authenticated ciphertexts by an honest client should be handled at a level below the Riffle protocol. For example, a network error should be handled by the underlying TCP connection. We discuss a malicious server tampering with messages in Section 4.8.3.

The goal now is to prove that c is an encryption of k' (i.e., $k = k'$), which shows that the revealed k' is the key shared between the server and the accused client. To do so, the verifier first independently computes $\frac{k}{k'} \cdot g^{rs}$. The prover then generates the following two proofs:

1. Prover knows an s' such that $g^{rs'} = \frac{k}{k'} \cdot g^{rs}$.
2. $\log_{g^r}(\frac{k}{k'} \cdot g^{rs'}) = \log_g(g^s)$.

Proof 1 shows that $k = k'$ if $s' = s$, and Proof 2 shows that $s' = s$. Therefore, the two proofs together form a proof that $k = k'$, and that c is an encryption of k'. We can use zero-knowledge proof-of-knowledge [15] for 1, and Chaum-Pedersen proof [17] for 2.

4.7 Bandwidth Overhead

Riffle achieves near optimal bandwidth between a client and a server when sending a message. A client only uploads a ciphertext of layered authenticated encryption of size $b + m\lambda$, where b is the size of the message, m is the number of servers, and λ is the size of the message authentication codes (MACs) [8] used with the authenticated encryption [9]. If the client is interested in only one message and the index is known, then the only overhead is sending the mask of size n, the number of clients, to the primary server. The total upstream bandwidth is then $b + m\lambda + n$, and the total downstream bandwidth is b per client per round. We note that even though upstream bandwidth grows linearly with n, it only requires 1 bit per client. In the general case where the index of the message is not known, the download bandwidth is nb per client due to the broadcast, but the upload bandwidth decreases by n, the size of the mask.

The bandwidth requirement between the servers, on the other hand, grows linearly with the number of users. Every server must download n ciphertexts, and upload n ciphertexts (with one removed layer) to the next downstream server. The last server also needs to send the plaintexts to all other servers as well. Furthermore, though PIR reduces the download bandwidth overhead of the clients, the server to server bandwidth increases: with our optimizations, each server needs to send r_i to the clients' primary servers. Therefore, the servers also need $(m-1)nb$ additional bandwidth for the PIR responses. In total, the server to server bandwidth requirement per round is approximately $3(m-1)nb$. Even though the total grows linearly with the number of clients, we note that this is asymptotically better than previous anytrust systems. For example, Dissent [53] requires $m(m-1)nb$ to perform a DC-Net among the servers with all clients' data.

4.8 Security Analysis

We first sketch an analysis of the security of the hybrid shuffle. We then describe how Riffle provides the three security properties from Section 3.3. Finally, we briefly argue the security of the accusation mechanism.

4.8.1 Security of Hybrid Shuffle

We need to show two different properties of the shuffle: (1) verifiability and (2) zero-knowledge. To show verifiability, let us assume that the authenticated encryption used by Riffle is secure against forgery, following the proposal of [9]. Then, for prover P to tamper with the inputs and not be detected by verifier V, P needs to generate outputs such that some of the outputs are not decryptions of the inputs, but still authenticate properly under the keys of P. However, ciphertexts are unforgeable and the keys of V are unknown since P only sees the encrypted k_i's. Therefore, P cannot generate such outputs. We also use the round number, which is provided internally by P, as the nonce to authenticated encryption to guarantee the freshness of the data, and stop replay attacks by P. Therefore, shuffle and decryption of P is valid if and only if V can authenticate all output ciphertexts, and the shuffle can be verified.

To show zero-knowledge, let us assume that the encryption scheme is semantically secure [31], as assumed by [9]. This means that V learns only negligible information by observing the input ciphertexts and the output ciphertexts of P. Then, V can simulate P's responses by encrypting random values with the keys stored on V: the keys are stored in the same permutation as the expected output of P, and the ciphertexts output by P are indistinguishable from encryption of random values if the encryption is semantically secure. Finally, the keys in V do not leak the permutation of P since the verifiable shuffle used to share the keys is zero-knowledge. Therefore, the hybrid shuffle is also zero-knowledge[3].

3 It does reveal one bit of information that indicates if the permutation used for hybrid shuffle is the same as the permutation used to shuffle keys during setup: if a different permutation is used, the messages will not authenticate correctly. However, the shuffle *only* reveals this bit, and is zero-knowledge with respect to the actual permutation.

4.8.2 Security of Riffle

Correctness. If the protocol is carried out faithfully, then the servers will simply shuffle the messages and the final server publishes all plaintext messages to every server. The messages are also available to the clients via PIR or broadcast. Thus, Riffle satisfies the correctness property.

Sender Anonymity. Sender anonymity relies on the verifiability and zero-knowledge property of the verifiable and hybrid shuffle. During the upload and shuffle stages of the protocol, S_i, an upstream server, is the prover P, and S_{i+1}, a downstream server, is the verifier V of the hybrid shuffle. Intuitively, verifiability ensures that the protocol is carried out faithfully; otherwise the honest server will flag an error. Moreover, since the shuffle is zero-knowledge, the honest server's permutation π_H is unknown to the adversary. Thus, the final permutation of the messages is also unknown to the adversary, and no malicious client or server can link a message to a honest client. We defer the detailed security argument to Appendices A and B.

We note that though the shuffle is now only verifiable by the next downstream server, this does not result in privacy leakage. Any messages that pass through the honest server will be valid, and the downstream malicious servers can only denial-of-service the clients by tampering with the messages rather than deanonymizing them since the honest server's permutation is unknown to them. Moreover, even though Riffle uses the same permutation every round, the adversary can only learn that two messages were sent by the same client, but can never learn which client sent the messages.

Receiver Anonymity. The anonymity of the downloads depends on the security of PIR and PRNGs. The security of PIR used in Riffle was proven by Chor *et al.* [20]. Intuitively, if the masks are generated at random, then the mth mask cannot be inferred from the other $m-1$ masks. The optimization to reduce mask sharing is secure if the PRNG is cryptographically secure, which says that the masks cannot be distinguished from truly random masks. Finally, collecting the responses at one server is also secure as long as not all secrets are known to the malicious servers. Therefore, the adversary cannot learn the index of the downloaded message for any honest client.

4.8.3 Security of Accusation

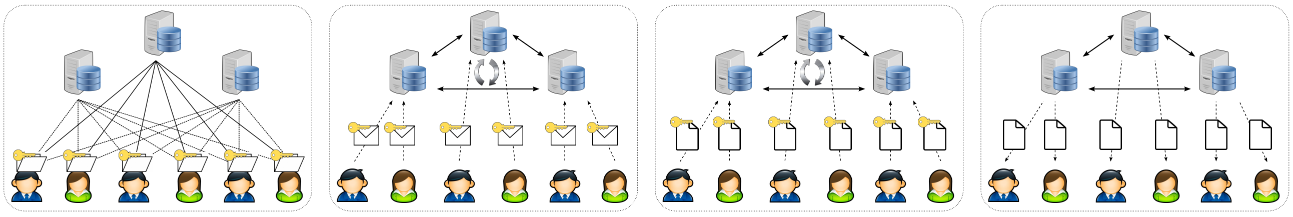
If no server misbehaves, then the accusation will reveal only the malicious client: every step of Algorithm 4 will verify successfully, and the servers reveals inverse permutations related only to the accused client.

In the case with malicious servers, let S_i be the first misbehaving server, and let S_H be the honest server. We first note that the transcripts from all servers remove the possibility of clients being incorrectly convinced that S_H is malicious, thereby removing S_H from the group. There are two cases: (1) $i > H$ (malicious downstream server), or (2) $i < H$ (malicious upstream server).

In the first case, eventually the honest server S_H will see all ciphertexts $E_{H+1}, \ldots, E_i$ flagged by the downstream servers along with the associated secrets. S_H first checks the commitments; since the commitments are the ciphertexts of the keys used during setup, the servers cannot use a different key than the key shared with the client. S_H also checks the validity of the ciphertexts by (1) confirming E_{H+1} is the ciphertext given to S_{H+1} and (2) checking that the decryptions of E_{H+1} matches $E_{H+2}, \ldots, E_i$. If S_H finds any mismatching ciphertexts, it will not reveal the permutation, thus preserving anonymity. If the commitment is binding and a ciphertext cannot decrypt to two different values using one key, then the malicious servers cannot generate correct decryptions of E_{H+1} that are different from the expected $E_{H+2}, \ldots, E_i$. Therefore, no downstream server can misbehave without getting caught.

In the second case ($i < H$), there are three scenarios. First, S_i starts the accusation. Here, S_H would not reveal its permutation, and preserves anonymity of all clients. Second, S_i misbehaves during an accusation but does not start it. In this scenario, there is a real misbehaving client (since S_i is the first misbehaving server), and S_H only reveals the inverse permutation of the malicious client. Thus, either the malicious client is revealed, or the transcript reveals that S_i is malicious.

In the last scenario, S_{H-1} sends a mis-authenticated ciphertext and tricks S_H into starting a (false) accusation, thereby deanonymizing one client. Since the key used by the malicious server is not revealed yet, S_H has no way of stopping this attack the first time. However, after the accusation, S_H and the clients will check the accusation transcript and remove S_i from the group. Moreover, since π_H is unknown, S_i cannot *target* any particular client, and can only do this for a random honest client. We believe that these two properties will sufficiently disincentivize a malicious server from carrying out this attack.



(a) Setup: Clients share the torrent files anonymously.

(b) Request: Each client requests a file by uploading the hash of the file using Riffle.

(c) Upload: Each client uploads an encrypted file based on the requests using Riffle.

(d) Download: Each client downloads the file he or she requested using PIR.

Fig. 2. Anonymous File Sharing Protocol

5 Anonymous File Sharing

The efficiency of Riffle makes it suitable for bandwidth-intensive applications like file sharing. In this section, we describe in detail a new anonymous file sharing protocol using Riffle as the underlying mechanism.

5.1 File Sharing Protocol

File sharing within a Riffle group is similar to that of BitTorrent [2], despite the differences in the system model (client-server in Riffle versus peer-to-peer in BitTorrent). When a client wants to share a file, he or she generates a torrent file, which contains the hashes of all *blocks* (the smallest unit used for file sharing) of the file. Then, using Riffle, the client uploads the torrent file to the servers. The servers play the role of torrent trackers in BitTorrent, and manage all available files in that group. In the simplest design, the file descriptors are broadcast to all connected clients, and clients can locally choose a file to download. Since the torrent files are fairly small even for large files (only a few 100KB in practice) and sharing them is a one-time cost, we assume broadcasting them is inexpensive and focus on sharing blocks.

With the torrent files distributed, the clients can now share files anonymously using Riffle. There are three major steps:

1. **Requesting Blocks:** Each C_j identifies a file F of interest, and the hashes of the blocks of the file $\vec{H_F}$ via its torrent file. C_j then requests a block of F by uploading the hash of the block $H_j \in \vec{H_F}$ to S_{p_j} using Riffle. When a client has no blocks to request, he or she sends a random value as a (non-)request to remain traffic analysis resistant. All requests $\vec{H}_\pi$ are broadcast to the clients at the end of this step.
2. **Uploading Blocks:** Each C_j checks if it possesses any requested block by checking the hashes of the blocks it owns with $\vec{H}_\pi$. If a matching block M_j is found, then C_j uploads M_j using Riffle. Once the plaintext blocks are available to the servers, each server broadcasts the hashes of the available blocks $\vec{H'}_\pi$.
3. **Downloading Blocks:** From H_j and $\vec{H'}_\pi$, C_j learns the index I_j of the block C_j requested. Using PIR, C_j downloads the block.

The rounds can (and should) be pipelined for performance since each request is independent. That is, we allow many outstanding requests. Figure 2 summarizes our file sharing protocol. We note that though an adversary can learn which files a client is interested in due to the same permutation used in each round, the clients still remain anonymous.

5.2 Bandwidth Overhead in File Sharing

To share a block among users in peer-to-peer file sharing such as BitTorrent [2], each client only needs to request and download a block. The client also may need to upload a file if it receives a request. If each request is of size h, each client consumes $h + b$ of upload bandwidth (assuming it has a block to upload to another client), and b of download bandwidth. In Riffle with PIR, there are three sources of bandwidth overhead for a client: (1) downloading the requests, (2) uploading MACs of authenticated encryption, and (3) the mask uploaded to the primary server. Thus, the total bandwidth between a client and a server is $h + b + n + 2m\lambda$ of upload, and $b + 2hn$ of download. We note that even though both bandwidths grow with the number of clients, n and $2hn$ are much smaller than b in file sharing scenarios for a reasonable number of clients and block size.

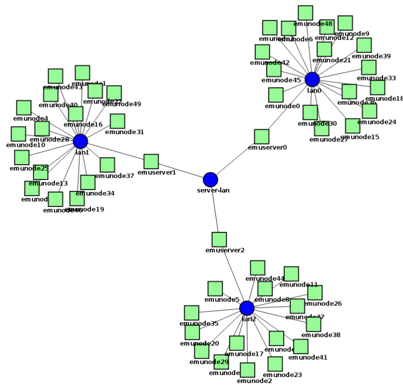


Fig. 3. Testbed topology.

6 Prototype Evaluation

In this section, we describe our prototype implementation, and evaluation.

6.1 Implementation

We have implemented a Riffle prototype in Go using Go's native crypto library along with the DeDiS Advanced Crypto library [1]. We used ElGamal encryption using Curve25519 [11] and Neff's shuffle [37] with Chaum-Pederson proofs [17] for the verifiable shuffle and decryption. For authenticated encryption, we use Go's Secretbox implementation [4, 5], which internally uses Salsa20 [12] for encryption and Poly1305 [10] for authentication. For the pseudo-random number generator used to update the masks and the secrets needed for PIR, we used keyed AES.

Our prototype supports two different modes of operation: (1) file sharing, and (2) microblogging. File sharing mode implements the protocol described in Section 5. Microblogging mode implements the Riffle protocol without PIR: each client submits a small message per round, and the plaintext messages are broadcast to all clients. We use broadcast because we assume that the users in microblogging scenarios are interested in many messages posted by different users.

6.2 Evaluation

To evaluate our system, we used the Emulab [3] testbed. Emulab provided a stable environment to test our prototype, while allowing us to easily vary the group topology and server configurations. The servers were connected to each other via a shared LAN, and the clients were distributed evenly among all servers. The clients connected to their primary server using one shared 100 Mbps link with 20ms delay, and the servers were connected to each other through a 1 Gbps link with 10ms delay. Due to resource and time constraints, we used 50 physical nodes to simulate all clients. Each server was equipped with an 8-core Intel Xeon E5530 Nehalem processor[4], and the majority of the client nodes were using a dual core Intel Pentium 4 processor. As shown in the next sections, we have found that the "outdated" processors did not impact our results much. Figure 3 shows the testbed topology used for the majority of our experiments.

6.2.1 File sharing

We implemented the file sharing application described in Section 5, including the optimizations to PIR. We simulated users sharing large files by first creating a pool of files, each of which was 300MB. From the pool, each client chose one file to request and one file to share, and each file was divided into 256KB blocks, similar to BitTorrent's typical block size [42]. We have experimented with different block sizes, and found that the block size changed the effective bandwidth by less than 5% in all experiments as we varied it from 128KB to 1MB.

Figure 4 shows the total time spent and the effective bandwidth (the size of the shared file over the total time spent) when sharing a 300MB file. We also plotted the "ideal" Riffle, where we computed the expected time to share a 300MB file based on our analytic bandwidth model (Section 5.2), assuming computation is free and network bandwidth is perfect. We have created a similar model for Dissent [53] as well for comparison.

In the experiments, Riffle provides good performance for up to 200 clients, supporting 100KB/s of effective bandwidth per client. Our prototype matches the analytical model fairly closely, showing that the amount of computation is minimal, and the primary limitation is the server to server bandwidth. If the servers were connected through 10 Gbps connections, we expect the effective bandwidth to improve by an order of magnitude. The discrepancy between the idealized model and the prototype for larger numbers of clients is due to two factors: first, the ideal model ignores cost of computation, which increases linearly with the number of clients. Though symmetric decryption is inexpensive, the cost

4 The most powerful machines with Sandy Bridge Xeon with 10 Gigabit Ethernet were not readily available.

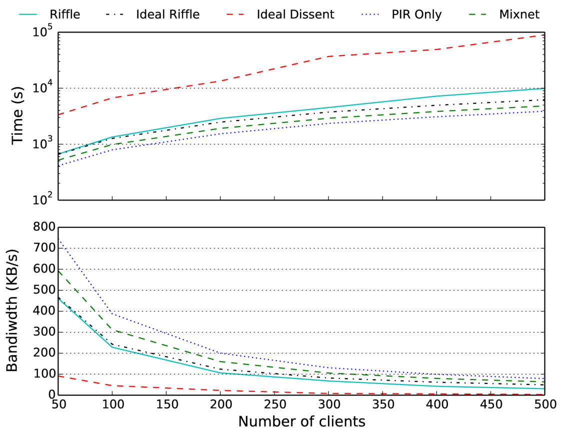


Fig. 4. Average time taken to share a 300MB file and effective bandwidth for varying numbers of clients with 3 servers. Note that the y-axis of the time graph is in log scale to properly display the time taken for Dissent [53].

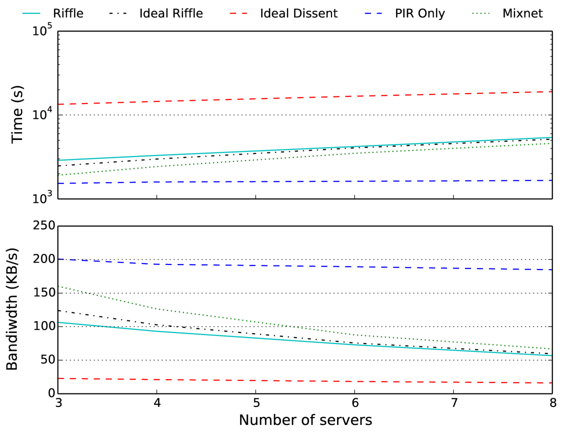


Fig. 5. Average time taken to share a 300MB file and effective bandwidth for 200 clients with varying numbers of servers. The y-axis of the time graph is in log scale.

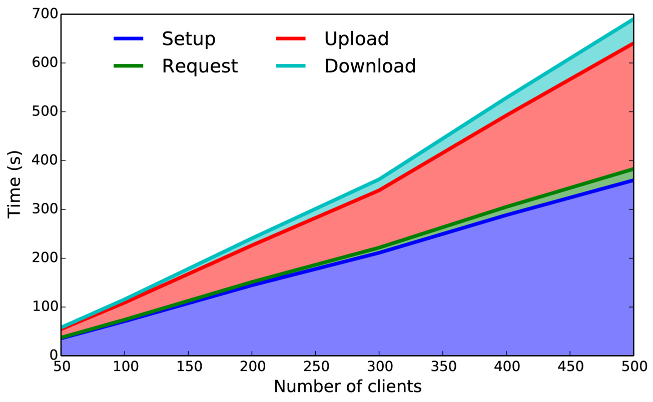


Fig. 6. Breakdown of the total time taken to share a 10MB file for varying numbers of clients with 3 servers. Sharing a 10MB file consists of 40 rounds with 256KB blocks.

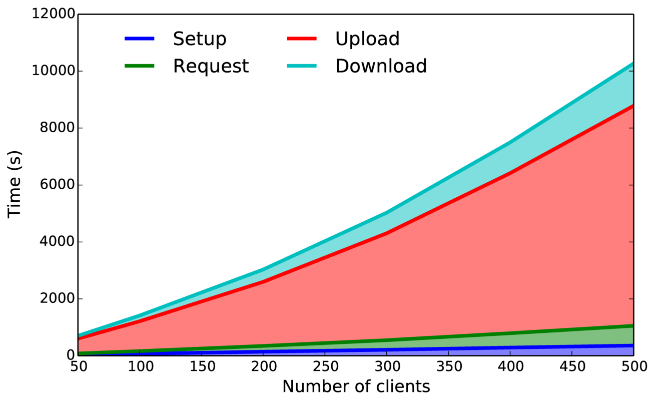


Fig. 7. Breakdown of the total time taken to share a 300MB file for varying numbers of clients with 3 servers. Sharing a 300MB file consists of 1200 rounds with 256KB blocks.

becomes non-negligible when the number of clients is large. Second, the effective bandwidth per client decreases since we are sharing a 100 Mbps link among a few hundred clients.

Figure 4 also depicts performance of other anonymous communication solutions: the two straw-man solutions (Section 4.1), PIR-only and shuffle only (Mixnet), and the ideal model of Dissent [53]. PIR-only models the situation where there are 3 servers with replicated data, and clients upload to their primary servers, the servers share all ciphertexts, and download using PIR. The Mixnet here consists of 3 mixes, and implements a point-to-point connection; that is, the final plaintexts are not broadcast, but sent to their intended destinations by the final mix. Though it is hard to compare directly to Aqua [34], the performance of Mixnet is similar to that of Aqua, assuming all clients have similar transfer rate. We note that Mixnet here has a similar threat model as Aqua: as long as the first mix is honest, this should provide sender anonymity.

When comparing to Dissent, we see an order of magnitude speed up. This is expected since the client-server bandwidth overhead in Dissent grows linearly with the number of clients, and each client only has access to a small amount of bandwidth. Riffle also performs comparably to the two straw-man designs: PIR-only and Mixnet can support up to 400 and 300 clients while Riffle can support up to 200 clients with bandwidth of 100KB/s. Though Riffle is not as efficient as the straw-man solutions, Riffle provides anonymity in a stronger threat model: we provide sender and receiver anonymity as long as *any* of the servers is honest, while PIR-only fails to provide sender anonymity, and Mixnet and Aqua require the first mix to be honest to provide anonymity.

We also tested the impact of different server configurations on performance. Figure 5 shows the effective bandwidth of 200 clients sharing 300MB files as we varied the number of servers. As observed in our analytical model (Section 5.2), the server to server bandwidth requirement grows with the number of servers, so the average bandwidth of the clients drops as we increase the number of servers.

Finally, we evaluated the full system performance of Riffle, including the setup phase of an epoch. Figure 6 and Figure 7 present the breakdown of the time spent to share a 10MB file and a 300MB file for different numbers of clients. Specifically, the graph shows the time spent in the setup phase (verifiable shuffle of keys, and sharing secrets for PIR), and the three steps of file sharing rounds (request, upload, and download). When sharing a 10MB file, the verifiable shuffle took more than half of the total time, proving to be quite costly. However, as demonstrated by Figure 7, the verifiable shuffle is a one-time operation at the beginning of an epoch, and the cost becomes less significant for longer epochs. Moreover, the verifiable shuffle used by the Riffle prototype [37] is not the most efficient shuffle, and we can reduce the setup time if we implement faster verifiable shuffles [7]. We also note that with more powerful machines, the verifiable shuffle should be much faster, as computation dominates the overhead for shuffles.

6.2.2 Microblogging

We simulated a microblogging scenario by creating tens to hundreds of thousands of clients each of whom submitted a small message (160 byte or 320 byte) per round, and broadcasting the messages at the end of each round. Due to resource limitation, we created hundreds to thousands of "super" clients, each of which submitted hundreds of messages to simulate a large number of users. For this experiment, we fixed the number of servers at 3 as we varied the number of clients.

Figure 8 shows the average latency of posting one message. For latency sensitive microblogging, we can support up to 10,000 users with less than one second latency with 160 byte messages. If the messages can tolerate some delay, we can support more than 100,000 users with less than 10 seconds of latency. This figure also demonstrates the ability of Riffle to exchange latency for message size: if we reduce the message size, the latency also decreases proportionally. Because Riffle is bandwidth and computation efficient, the latency is determined solely by the total number of bits of the messages in a round. This makes it easy to make a conscientious trade-off between the latency, the message size, and the number of clients.

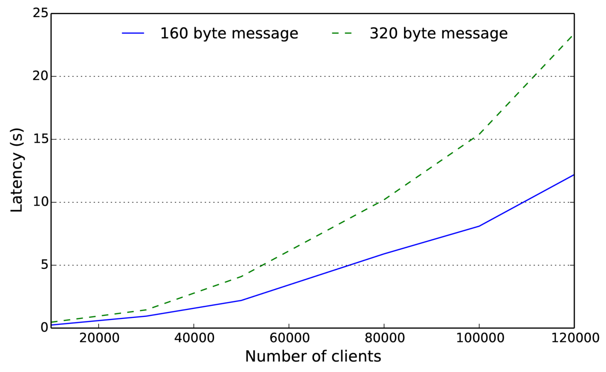


Fig. 8. Average latency to share a microblogging post for different message sizes. The number of servers here was fixed at 3.

Riffle can support an order of magnitude more clients compared to Dissent [53] for latency sensitive microblogging (10,000 in Riffle vs. 1,000 in Dissent). For a large number of clients, Riffle outperforms previous works [7, 53] by orders of magnitude. For instance, it takes 2 minutes just to verifiably shuffle messages of 100,000 users [7], ignoring network communication. In Riffle, it takes a fraction of a second to shuffle and verify, and takes less than 10 seconds in total for 100,000 users, including the time spent in the network. Finally, though it is hard to compare Riffle to Riposte [21] due to lack of a database in Riffle, we expect Riffle to perform better as the database of microblog posts grows larger: the bandwidth requirement of Riposte clients grows with the size of the database, while the bandwidth requirement of Riffle clients remains the same.

7 Discussion and Future Work

In this section, we discuss a few limitations and some aspects of Riffle we did not consider in the paper, and how they could be addressed in future work.

Alternate usage model. In this paper, we have assumed that the clients want to communicate with others in the same group. However, the clients may also want to interact with (1) clients in other Riffle groups, or (2) the general Internet. In the first setting, we can use ideas from Herbivore [30], and connect the servers of different groups to each other. The clients can then communicate with any client in any group through the network of servers. In the second setting, we could use the Riffle servers as "exit" nodes: each client can submit his or her message for someone outside the group through Riffle, and the servers interact with the Internet on the client's behalf. For example, each client uploads a BitTorrent request for a file to the servers. The servers then participate in a regular BitTorrent protocol in the open Internet, and the clients can use PIR to securely download their files. Moreover, the servers could use a highly scalable anonymity system with a large anonymity set (e.g., Tor [26]) to expand the anonymity set of the clients beyond just one Riffle group.

Malicious Servers. Verifiability and zero-knowledge properties of the hybrid shuffle (Section 4.8) prevent any malicious server from deanonymizing a client, and make it possible for an honest server to reveal a malicious server. However, since the identity of the honest server is unknown, the clients cannot rely on the claims of the servers. Concretely, there is no mechanism to prevent $m - 1$ malicious servers from claiming that the one honest server is malicious. Though this scenario should be rare with independently administered servers, we hope to provide a mechanism to prevent such an attack in the future.

Server to server bandwidth. As noted in our bandwidth analysis and evaluation, the server to server bandwidth requirement grows linearly with the number of clients. This quickly became the bottleneck of Riffle, and limited the effective throughput (Section 6). One potential solution is for each provider to manage a "super" server that consists of smaller servers, and each smaller server handles a fraction of the traffic. Essentially, we can increase the server to server bandwidth by replicating the connections.

Though we hope to lower the actual overhead in future work, we believe that some of the cost is fundamental to the anytrust model assumed by Riffle. Namely, if there is only one honest server and the identity of the honest server is unknown, it seems necessary for all data to pass through all servers.

Network churn and group changes. In a realistic network, different clients can have drastically different connection speeds, and clients can leave or join the group dynamically. The default Riffle protocol requires that everyone in a group submit a message every round to maintain anonymity, which makes the overall latency as bad as the worst individual latency. Worse, any changes in the group require Riffle to perform an expensive verifiable shuffle to create a new permutation for the new set of active clients.

To handle the case of clients dropping out, we currently ask each client to submit some cover traffic to the first server. When a client disconnects, we use the cover traffic to carry on the rounds with the active clients, and perform the verifiable shuffle in the background to create a new permutation. This option has two significant problems. First, it wastes bandwidth, and does not allow for dynamic growth of the group. The setup phase needs to be run again when a new client joins the group. Second, the first server can now choose to send cover traffic instead of the real traffic. This means that the first server can denial-of-service a subset of users. However, the first server cannot manufacture cover traffic, since it does not have the honest server's symmetric key for that user, and can only use the legitimate cover message in place of a real message. It therefore cannot deanonymize a client by creating fake cover traffic, and checking where the fake cover traffic is in the final permutation. Tolerating network churn and changes in the group more effectively is deferred to future work.

Intersection attacks. A powerful adversary monitoring clients and network over longer periods of time can correlate the presence of some messages with the online status of the clients [35]. For instance, if messages related to a protest are only posted when a particular client is online, then the adversary can link the messages to the client. Though Riffle does not protect against this class of attacks, it could benefit from prior work on mitigating these attacks [54].

8 Conclusion

Riffle is an anonymous communication system that provides traffic analysis resistance and strong anonymity while minimizing the bandwidth and computation overhead. We achieved this by developing a new hybrid shuffle, which avoids expensive verifiable shuffles in the critical path for uploading messages, and using private information retrieval for downloading messages. We have also demonstrated through a prototype the effectiveness of Riffle in anonymous file sharing and microblogging scenarios, and that strong anonymity can indeed scale to a large number of users with good bandwidth.

Acknowledgements

We would like to thank Ling Ren, Alin Tomescu, Chris Fletcher, Charles Herder, and Marten Van Dijk for the important discussions regarding some key aspects of this system, and security of the hybrid shuffle. We also thank the anonymous reviewers and our shepherd Jonathan Katz for their helpful feedback and suggestions. This material was supported by National Science Foundation Frontier CNS-1413920.

References

[1] Advanced crypto library for the go language. https://github.com/DeDiS/crypto.

[2] Bittorrent. https://bittorrent.com.

[3] Emulab network emulation testbed. http://www.emulab.net/.

[4] Secret-key authenticated encryption. http://nacl.cr.yp.to/secretbox.html.

[5] Secretbox - godoc. https://godoc.org/golang.org/x/crypto/nacl/secretbox.

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

[7] S. Bayer and J. Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *Proceedings of the 31st Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'12, pages 263–280, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. pages 1–15. Springer-Verlag, 1996.

[9] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *J. Cryptol.*, 21(4):469–491, Sept. 2008.

[10] D. Bernstein. The poly1305-aes message-authentication code. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer Berlin Heidelberg, 2005.

[11] D. J. Bernstein. Curve25519: new diffie-hellman speed records. In *In Public Key Cryptography (PKC), Springer-Verlag LNCS 3958*, page 2006, 2006.

[12] D. J. Bernstein. New stream cipher designs. chapter The Salsa20 Family of Stream Ciphers, pages 84–97. Springer-Verlag, Berlin, Heidelberg, 2008.

[13] J. Brickell and V. Shmatikov. Efficient anonymity-preserving data collection. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 76–85, New York, NY, USA, 2006. ACM.

[14] X. Cai, X. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*, October 2012.

[15] J. Camenisch and M. Stadler. Proof systems for general statements about discrete logarithms. Technical report, 1997.

[16] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptol.*, 1(1):65–75, Mar. 1988.

[17] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 89–105, London, UK, UK, 1993. Springer-Verlag.

[18] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, Feb. 1981.

[19] B. Chor and N. Gilboa. Computationally private information retrieval (extended abstract). In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 304–313, New York, NY, USA, 1997. ACM.

[20] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, Nov. 1998.

[21] H. Corrigan-Gibbs, D. Boneh, and D. Mazieres. Riposte: An Anonymous Messaging System Handling Millions of Users. *ArXiv e-prints*, Mar. 2015.

[22] H. Corrigan-Gibbs and B. Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 340–350, New York, NY, USA, 2010. ACM.

[23] H. Corrigan-Gibbs, D. I. Wolinsky, and B. Ford. Proactively accountable anonymous messaging in verdict. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 147–162, Washington, D.C., 2013. USENIX.

[24] G. Danezis, R. Dingledine, D. Hopwood, and N. Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *In Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 2–15, 2003.

[25] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, Nov 1976.

[26] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, August 2004.

[27] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 193–206, New York, NY, USA, 2002. ACM.

[28] J. Furukawa and K. Sako. An efficient scheme for proving a shuffle. In *In Proc. of CRYPTO '01*, pages 368–387. Springer-Verlag, 2001.

[29] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In P. Nguyen and E. Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 640–658. Springer Berlin Heidelberg, 2014.

[30] S. Goel, M. Robson, M. Polte, and E. G. Sirer. Herbivore: A Scalable and Efficient Protocol for Anonymous Communication. Technical Report 2003-1890, Cornell University, Ithaca, NY, February 2003.

[31] S. Goldwasser and S. Micali. Probabilistic encryption; how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 365–377, New York, NY, USA, 1982. ACM.

[32] D. Herrmann, R. Wendolsky, and H. Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naive-bayes classifier. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 31–42, New York, NY, USA, 2009. ACM.

[33] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas. Circuit fingerprinting attacks: Passive deanonymization of tor hidden services. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 287–302, Washington, D.C., Aug. 2015. USENIX Association.

[34] S. Le Blond, D. Choffnes, W. Zhou, P. Druschel, H. Ballani, and P. Francis. Towards efficient traffic-analysis resistant anonymity networks. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 303–314, New York, NY, USA, 2013. ACM.

[35] N. Mathewson and R. Dingledine. Practical traffic analysis: extending and resisting statistical disclosure. In *4th International Workshop on Privacy Enhancing Technologies*, May 2004.

[36] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of tor. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, pages 183–195, Washington, DC, USA, 2005. IEEE Computer Society.

[37] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, CCS '01, pages 116–125, New York, NY, USA, 2001. ACM.

[38] L. Nguyen and R. Safavi-naini. Breaking and mending resilient mix-nets. In *Proc. PET'03, Springer-Verlag, LNCS 2760*, pages 66–80. Springer-Verlag, LNCS, 2003.

[39] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 103–114, October 2011.

[40] B. Pfitzmann. Breaking an efficient anonymous channel. In *In EUROCRYPT*, pages 332–340. Springer-Verlag, 1995.

[41] B. Pfitzmann and A. Pfitzmann. How to break the direct rsa-implementation of mixes. In *Advances in Cryptology—EUROCRYPT '89 Proceedings*, pages 373–381. Springer-Verlag, 1990.

[42] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In M. Castro and R. van Renesse, editors, *Peer-to-Peer Systems IV*, volume 3640 of *Lecture Notes in Computer Science*, pages 205–216. Springer Berlin Heidelberg, 2005.

[43] J.-F. Raymond. Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems. In H. Federrath, editor, *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, pages 10–29. Springer-Verlag, LNCS 2009, July 2000.

[44] M. K. Reiter and A. D. Rubin. Anonymous web transactions with crowds. *Commun. ACM*, 42(2):32–48, Feb. 1999.

[45] M. Rennhard and B. Plattner. Introducing morphmix: Peer-to-peer based anonymous internet usage with collusion detection. In *Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society*, WPES '02, pages 91–102, New York, NY, USA, 2002. ACM.

[46] L. Sassaman, B. Cohen, and N. Mathewson. The pynchon gate: A secure method of pseudonymous mail retrieval. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, WPES '05, pages 1–9, New York, NY, USA, 2005. ACM.

[47] R. Sion and B. Carbunar. On the computational practicality of private information retrieval.

[48] E. G. Sirer, S. Goel, and M. Robson. Eluding carnivores: File sharing with strong anonymity. In *In Proc. of ACM SIGOPS European Workshop*, 2004.

[49] A. Teich, M. S. Frankel, R. Kling, and Y. Lee. Anonymous communication policies for the internet: Results and recommendations of the aaas conference. *Information Society*, 15(2), 1999.

[50] M. Waidner and B. Pfitzmann. The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability. In *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*, EUROCRYPT '89, pages 690–, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[51] T. Wang and I. Goldberg. Improved website fingerprinting on tor. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2013)*. ACM, November 2013.

[52] D. Wikström. Four practical attacks for "optimistic mixing for exit-polls", 2003.

[53] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182, Hollywood, CA, 2012. USENIX.

[54] D. I. Wolinsky, E. Syta, and B. Ford. Hang with your buddies to resist intersection attacks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer Communications Security*, CCS '13, pages 1153–1166, New York, NY, USA, 2013. ACM.

A Sender Anonymity

Uploading in Riffle consists of the following algorithms:

- $\mathit{KeyUpload}(\{k_{ij}\}_{i\in[m]}, \{p_i\}_{i\in[m]}) \to \{c_{ij}\}_{i\in[m]}$.
  Client C_j uses *KeyUpload* to generate the onion-encryption of the symmetric keys. *KeyUpload* takes the symmetric keys $\{k_{ij}\}_{i\in[m]}$ and the servers' public keys $\{p_i\}_{i\in[m]}$, and produces the ciphertexts $\{c_{ij}\}_{i\in[m]}$.
- $\mathit{KeyShuffle}(\{\vec{c_\ell}\}_{\ell\in[i,m]}, s_i) \to (\vec{k_i}, \{\vec{c_\ell}'\}_{\ell\in[i+1,m]}, \pi_i)$.
  Server S_i uses *KeyShuffle* to generate the (encrypted) keys that it sends to the next server. *KeyShuffle* takes in the onion-encrypted keys and a secret key s_i, and generates the keys for S_i, encrypted keys to send to next server, and the permutation used.
- $\mathit{Upload}(M_j, \{k_{ij}\}_{i\in[m]}) \to u_j$.
  Client C_j uses *Upload* to generate the value that it sends to the servers. *Upload* takes the message M_j and the symmetric keys $\{k_{ij}\}$, and generates the upload value u_j.
- $\mathit{Shuffle}(\vec{u_i}, \vec{k_i}, \pi_i) \to \vec{u_{i+1}}$.
  Server S_i uses *Shuffle* to generate the (encrypted) messages that it sends to the next server. *Shuffle* takes the vector of messages $\vec{u_i}$, the shared keys of S_i

$\vec{k_i}$, and the permutation π_i. It generates an updated vector of messages $\vec{u_{i+1}}$.

We present the security game which models the Riffle message sending protocol. The game is played between an adversary who controls a subset of servers and clients, and a challenger.

1. The adversary selects a subset of servers $S_M \subset S$ to be the set of malicious servers such that $|S_M| < m$. We denote the set of honest servers as S_H (i.e., $S_H = S \setminus S_M$). The adversary also selects a subset of clients $C_M \subset C$ to be the set of malicious clients such that $|C_M| \leq m-2$. We similarly denote C_H to be the set of honest clients.
2. The challenger and the adversary generate the servers' public keys in order:
   * The challenger generates public and private keys $\{p_i\}_{i:S_i \in S_H}$ and $\{s_i\}_{i:S_i \in S_H}$, and reveals the $\{p_i\}$ to the adversary.
   * The challenger generates public and private keys $\{p_i\}_{i:S_i \in S_M}$ and $\{s_i\}_{i:S_i \in S_M}$, and reveals the $\{p_i\}$ to the challenger.
3. The challenger and the adversary generate the states for the clients in the following order.
   * The challenger generates keys $\{k_{ij}\}_{i\in[m]}$, uses *KeyUpload* to generate $\{c_{ij}\}_{i\in[m]}$ for $C_j \in C_H$, and reveals the $\{c_{ij}\}$ to the adversary.
   * The adversary generates keys $\{k_{ij}\}_{i\in[m]}$ and $\{c_{ij}\}_{i\in[m]}$ for $C_j \in C_M$, and reveals the $\{c_{ij}\}$ to the adversary. The adversary may generate them using a malicious strategy.
   - The challenger and the adversary generate $\{\vec{c_\ell}\}_{\ell\in[m]}$.
   * For $i = 1, \ldots, m$,
     - If $S_i \in S_H$, then the challenger uses *KeyShuffle* with $\{\vec{c_\ell}\}_{\ell\in[i,m]}$ and s_i to generate $\{\vec{c_\ell}\}_{\ell\in[i+1,m]}$ and reveals it to the adversary. S_i keeps $\vec{k_i}$ and π_i.

       If $S_i \in S_M$, then the adversary generates $\{\vec{c_\ell}\}_{\ell\in[i+1,m]}$, and reveals it to the challenger. The adversary may generate it using a malicious strategy given any the previous states.
4. For rounds $r = 1, \ldots, R$, where R is polynomial in the security parameter, repeat the following:
   * The challenger and the adversary generate the upload values.
     - The adversary sends C_H and any states to the client oracle.
     - The client oracle sends the challenger the messages for C_H; $\{M_i\}_{i:C_i \in C_H}$.
     - The challenger generates $\{u_j\}_{j:C_j \in C_H}$ using *Upload* and $\{M_i\}_{i:C_i \in C_H}$, and reveals the $\{u_j\}$ to the adversary.
     - The adversary picks messages $\{M_j\}_{j:C_j \in C_M}$ and generates $\{u_j\}_{j:C_j \in C_M}$. The adversary then reveals the $\{u_j\}$ to the challenger.
     - The challenger and the adversary generate $\vec{u_1}$.
   * For $i = 1, \ldots, m$,
     - If $S_i \in S_H$, then the challenger uses *Shuffle* and u_i to generate $\vec{u_{i+1}}$, and reveals it to the adversary.

       If $S_i \in S_M$, then the adversary generates $\vec{u_{i+1}}$. The adversary again may generate it using a malicious strategy given the previous messages and other states. The adversary then reveals $\vec{u_{i+1}}$ to the challenger.
5. The adversary sends $\{\pi_i\}_{i:S_i \in S_M}$ to the challenger. The adversary then selects an I and I' such that $C_I, C_{I'} \in C_H$ and sends them to the challenger.
6. The challenger computes $\pi = \pi_m(\ldots(\pi_2(\pi_1))\ldots)$. The challenger then flips a coin b. If $b = 1$, then the challenger sends the adversary $\pi(I)$, the permuted index of I. Otherwise, the challenger sends $\pi(I')$.
7. The adversary makes a guess b' for b. The adversary wins the game if it guesses b correctly;

Note that any message the adversary sends need not be generated using the algorithms described in this section. We also model the clients via a client oracle because we cannot stop a client from sending information revealing his or her identity. For instance, the adversary may influence or fool an honest client into sending his or her IP. Our goal is to show that the probability of the adversary winning this game is no better than random guessing for any messages sent by the client.

Let W_I be the event that an adversary guesses $\pi(I)$ of an index I from π, where π is a unknown random permutation on $[k]$ and k is the number of honest clients. Let W_R be the event that an adversary successfully deanonymizes a client (i.e., wins the security game). We say that the protocol provides sender anonymity if the probability of any real world adversary successfully attacking the protocol is negligibly close $Pr[W_I]$. In other words,

$$|Pr[W_I] - Pr[W_R]| \leq neg(\lambda) \tag{1}$$

where λ is the implicit security parameter. In other words, the adversary cannot guess any client's index in the final permutation better than guessing at random[5].

B Proof of Sender Anonymity

Anonymity of the uploads relies on the fact that both the traditional and hybrid shuffles are verifiable and zero-knowledge. The verifiability ensures that any efficient malicious server cannot deviate from the protocol without getting caught. Specifically, this forces the adversaries to use *KeyUpload*, *KeyShuffle*, *Upload* and *Shuffle* to generate all messages.

We now use a sequence of games argument to analyze the security game in Appendix A. First, with the protocol being executed faithfully, the zero-knowledge verifiable shuffle in the setup phase (i.e., key sharing) ensures that the adversary learns only negligible information about the secret keys and the permutations of the honest server. Furthermore, π_H, the honest server's permutation, is generated randomly independently of all other permutations, and thus the final permutation $\pi = \pi_m(\pi_{m-1}(\ldots(\pi_1)\ldots))$ is also random. The adversary then only has negligible probability of learning π. We can now replace collective shuffling and decryption Step 3 with a shuffle and decryption by a trusted party that uses an unknown random permutation π, and introduce only negligible change in the advantage of the adversary.

Second, the zero-knowledge property of the hybrid shuffle guarantees that any information gained from observing the outputs during the communication phase is also negligible. After setup and polynomially many (in the security parameter) rounds of communication, the adversary has only negligible probability of learning the permutation π_H of the honest server S_H. Using the same arguments, we can again replace the collective shuffle and decryption in Step 4 with a shuffle and decryption by a trusted party.

With the two replacements, the adversary now wins the game if it can guess an index from an unknown random permutation of honest clients. Since we have only introduced negligible advantage to the adversary when making these replacements, this probability of an adversary winning the game satisfies Equation 1.

5 Here, we are assuming that the client oracle is not outputting information that will give unfair advantage to the adversary. If we do not make such an assumption, W_I needs to be changed to the adversary successfully guessing $\pi(I)$ given the encrypted messages from the client oracle and the permutation of the plaintext messages (using π) from the client oracle.