Clarion: Anonymous Communication from Multiparty Shuffling Protocols

Saba Eskandarian
UNC Chapel Hill

Dan Boneh
Stanford University

***Abstract*—This paper studies the role of *multiparty shuffling protocols* in enabling more efficient metadata-hiding communication. We show that the process of shuffling messages can be expedited by having servers collaboratively shuffle and verify secret-shares of messages instead of using a conventional mixnet approach where servers take turns performing independent verifiable shuffles of user messages. We apply this technique to achieve both practical and asymptotic improvements in anonymous broadcast and messaging systems. We first show how to build a three server anonymous broadcast scheme, secure against one malicious server, that relies only on symmetric cryptography. Next, we adapt our three server broadcast scheme to a k-server scheme secure against $k-1$ malicious servers, at the cost of a more expensive per-shuffle preprocessing phase. Finally, we show how our scheme can be used to significantly improve the performance of the MCMix anonymous messaging system.**

We implement our shuffling protocol in a system called Clarion and find that it outperforms a mixnet made up of a sequence of verifiable (single-server) shuffles by $9.2\times$ for broadcasting small messages and outperforms the MCMix conversation protocol by $11.8\times$.

I. Introduction

While end to end encrypted communication has recently become widespread, systems that additionally hide sensitive communication *metadata* remain an active area of research. Many metadata-hiding communication systems adopt a mixnet-based approach [13] and require the platform's servers to somehow shuffle user messages before revealing them, breaking the connection between a message and its author. A primary performance obstacle in these systems is the cost of each server computing a *verifiable* shuffle of user messages [6] to protect against misbehavior by malicious servers. Metadata-hiding communication systems have adopted a diverse array of approaches to minimize or avoid this cost. Several recent systems use modified mixnets with additional techniques to reduce what part of the shuffle needs to be verified, trade off some security for better performance, or perform several smaller shuffles instead of a single big one [58, 57, 44, 41, 43, 45].

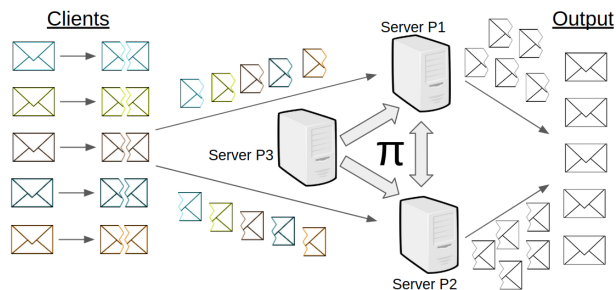

Fig. 1: Message flow in our three server anonymous broadcast system

This work presents a new approach to anonymous communication based on *multiparty shuffling protocols*. In this setting, instead of having servers take turns shuffling user data, each server holds a secret share of the data from the beginning, and the servers collaboratively shuffle the data, conducting integrity checks before and after the shuffle. The general message flow in the system is shown in Figure 1. While each server performs a shuffle on the data, the cost of verifying the full shuffle using our techniques is significantly reduced compared to a typical approach where each server individually proves the integrity of the shuffle it performed. This approach can either be used as a drop-in replacement for parts of existing systems that use verifiable shuffles or to build new standalone protocols.

We begin by building a three-server anonymous broadcast scheme that provides security against one malicious server. Instead of relying on each server producing a proof that it participated honestly in the shuffling protocol, our approach relies on a series of integrity checks performed by the servers during the shuffle computation that ensure the shuffle has been conducted honestly. Server communication and computation costs are $O(N\ell)$ for processing N messages of length ℓ each, and client communication and computation costs are $O(\ell)$. Asymptotically, this matches a standard messaging system that provides no privacy guarantees. Our scheme only requires symmetric-key cryptographic primitives and thus has excellent practical performance properties as well.

Next, we show how to adapt our three-server anonymous broadcast scheme to a k server scheme secure against $k-1$ malicious servers. This scheme maintains the strong performance of the three server scheme and reduces the asymptotic cost of a k-server shuffle to $O(k)$ as opposed to $O(k^2)$ in a naïve adaptation, but it comes at the cost of a more expensive per-shuffle preprocessing phase. We show how to realize this preprocessing phase by combining the secret-shared shuffle techniques of Chase et al. [12] with new optimizations. Given the addition of a preprocessing phase, this approach is particularly well-suited to settings like an online event with live anonymous comments: the preprocessing information can be prepared prior to the event, and comments can be delivered quickly during the live event.

Finally, we show how to use our system to achieve order of magnitude performance improvements over the MPC-based MCMix [3] anonymous messaging system while simultaneously upgrading parts of the system's security. One reason we are able to obtain such a significant improvement is our new lightweight multiparty shuffling protocol. Another reason is that end users format their submissions so that the shuffling servers cannot corrupt the execution of the MPC protocol. This further simplifies the protocol.

We implement and evaluate our scheme in a system called Clarion, comparing it to several prior works. Our three server broadcast scheme outperforms servers running the verifiable shuffle of Bayer and Groth [6] by $9.2\times$ for 32 Byte messages and by $3.0\times$ for 160 Byte messages. When used to build the conversation protocol for an anonymous messaging scheme, our approach outperforms verifiable shuffles by $8.2\times$ for 160 Byte messages. Compared to the MCMix system [3], we improve the performance of the conversation protocol by $11.8\times$ for 160 Byte messages and $10.8\times$ for 1KB messages. We also improve the performance of the dialing protocol, used to initiate new conversations, by over $2\times$. Finally, increasing the number of servers from three to five in our k-server broadcast scheme increases the speedup over verifiable shuffles from $2.9\times$ to $4.0\times$ for 160 Byte messages, demonstrating increasing benefits as the number of servers increases. Our implementation is open source and both our code and raw evaluation data can be found at `https://github.com/SabaEskandarian/Clarion`.

In summary, we make the following contributions:

- Demonstrate a new application of multiparty shuffling to anonymous communication.
- Build a new three-server anonymous broadcast scheme with optimal asymptotic performance using only symmetric cryptography.
- Extend our three-server scheme to k servers with security against $k-1$ malicious servers, at the cost of introducing a per-shuffle preprocessing phase.
- Show how to use our broadcast scheme to improve both the dialing and conversation protocols of the MCMix [3] anonymous messaging scheme.

II. Design Goals

We will broadly target two classes of metadata-hiding communication: *anonymous broadcast* and *anonymous messaging*. Anonymous broadcast allows users to anonymously send messages which are then broadcast to the world in batches, hiding the identity of each message's author among the set of all users who sent a message in that batch. Anonymous messaging allows users to initiate and conduct private conversations without the platform learning which users are talking to each other. Anonymous messaging protocols are usually split into two parts. A *dialing* protocol allows users to initiate conversations, and a *conversation* protocol allows two users who have started a conversation to send messages back and forth. Both anonymous broadcast and anonymous messaging make use of *cover traffic*, dummy messages sent by users who are not actively communicating, sent solely for the purpose of increasing the anonymity set enjoyed by real users. For our purposes, both will also proceed in a series of synchronized rounds, with message delivery happening once per round.

Throughout this paper, we will assume that the servers providing the broadcast/messaging services have key pairs used to establish private, authenticated communication channels between each other and with clients via TLS. All messages sent using our techniques need to be padded to a fixed length to hide message size metadata.

Security requirements. At a high level, we require three kinds of security properties for a multiparty shuffle. Taken together, these properties imply that, in our broadcast and messaging protocols, any honest author of a message can maintain anonymity against a malicious server and coalition of malicious clients.

- Confidentiality: no client or server, possibly colluding, will learn anything about the permutation that was used to shuffle the messages (except for inputs to the shuffle that the client chose itself).
- Integrity: a message sent to the shuffle by an honest author will come out of the shuffle unmodified, even in the presence of a malicious server, or any attempted modification will be detected.

- Disruption-resistance: a malicious client cannot launch an in-protocol denial of service attack.

To achieve security, we will make a *non-collusion* assumption among the servers. In particular, in Section IV, we will assume that no two servers collude in an attempt to break confidentiality or integrity guarantees. In other words, we assume that at most one of the three servers required by our scheme will deviate from the prescribed protocol. In Section V, where we extend our protocol to k servers, we will make the weaker assumption that there is at least one honest server who does not deviate from the protocol. That is, we expect security to hold even if up to $k-1$ servers maliciously collude in an attempt to break confidentiality or integrity guarantees. Our schemes do not provide security if all the servers collude to deanonymize clients. This form of non-collusion assumption is used by many metadata-hiding communication systems as well as the private telemetry functionality employed by Mozilla Firefox [20, 31, 1].

Note that we will trust the servers for *availability*, meaning that we will expect them to remain online and deliver the results of the protocol. We do not aim to protect against DoS attacks by malicious servers, nor against network-level DoS attacks. However, even a malicious server who decides to abort the protocol and prevent messages from being successfully broadcast will be unable to violate the confidentiality or integrity of messages. On the other hand, we do aim to protect against disruptive clients.

We will formalize our security goals using a simulation-based definition with an ideal functionality described in Appendix A which captures both security requirements. This means that the adversary, given access to the ideal functionality, can produce a simulated transcript distributed indistinguishably from that of the adversary interacting with real servers and clients in the protocol.

Limitations. As in all anonymity systems, a user's anonymity set depends on the presence of cover traffic, either from other real, honest users of the system or from non-users who wish to increase the size of the anonymity set available to those who need metadata-hiding communication. A user's anonymity set for a given message is the number of honest users whose messages are sent in the same round.

In the anonymous broadcast setting, our schemes will hide which client sent each message within a given round, but an adversary who can observe traffic across many rounds, as the set of clients sending messages changes, will be able to make inferences about who is sending which messages. These *intersection attacks* affect many anonymous broadcast systems and can be mitigated using orthogonal techniques [48, 61] or additional cover traffic.

Additional security properties. Trusting the servers for availability implies some potential limitations of our approach. Foremost among these is that a malicious server can simply cease to operate and prevent the system from successfully delivering messages. Moreover, a malicious server could prevent a particular user from being able to use the system to deliver messages, or even decide to abort the system once it sees the output (in the broadcast setting) but before publishing messages to the world. Prior work has studied how to prevent these scenarios with security properties named *robustness*, *censorship resistance*, and *fairness*, respectively, and shown how to achieve them, albeit at additional cost [47, 2]. We compare our techniques to these works, both quantitatively and qualitatively, in Appendix D.

The choice of which security properties to include in a system has ramifications for where it can be deployed and for what purposes it is best-suited. Systems where all servers must remain online are clearly not useful in settings where it is likely that a powerful outside actor, e.g., a nation-state adversary, would try to censor or otherwise take down the servers. On the other hand, allowing servers to block abusive users and view messages before broadcasting them to the world may be more compatible with platforms that wish to conduct a degree of moderation to ensure compliance with legal or ethical requirements. Our work fits in this latter category.

III. Background

This section provides background information on notation and cryptographic tools that will be used throughout this paper.

Notation. From here on we will use the following notation:

- Let G be a finite abelian group where the group operation is written additively.
- Let $\boldsymbol{x} \in G^N$ be a length-N vector. We use $[\boldsymbol{x}]$ to denote an additive secret sharing of $\boldsymbol{x}$. If there are two shares, we use $[\boldsymbol{x}]_1$ and $[\boldsymbol{x}]_2$ to denote the two shares. In particular, $[\boldsymbol{x}]_1$ and $[\boldsymbol{x}]_2$ are elements in G^N and $\boldsymbol{x} = [\boldsymbol{x}]_1 + [\boldsymbol{x}]_2$.
- For a vector $\boldsymbol{x} = (x_1, \ldots, x_N) \in G^N$ and a permutation $\pi : \mathbb{Z}_N \to \mathbb{Z}_N$, we let $\pi(\boldsymbol{x})$ denote the permuted vector $\left(x_{\pi(1)}, \ldots, x_{\pi(N)}\right) \in G^N$.

Secret-shared shuffle. The starting point for our shuffling scheme is an abstract problem called a *secret-shared shuffle*. The problem is to design a secure protocol between two parties P_1 and P_2 for the following functionality:

- *input:* let $\boldsymbol{x} \in G^N$ be a secret vector; Party P_1 has secret share $[\boldsymbol{x}]_1 \in G^N$ and Party P_2 has secret share $[\boldsymbol{x}]_2 \in G^N$.
- *output:* P_1 has $[\boldsymbol{s}]_1 \in G^N$ and P_2 has $[\boldsymbol{s}]_2 \in G^N$ such that $\boldsymbol{s} = \pi(\boldsymbol{x})$ and $\pi : \mathbb{Z}_N \to \mathbb{Z}_N$ is a random permutation.

Neither party should learn anything else. In particular, they should learn nothing else about $\boldsymbol{x}$ or the random permutation π.

This problem was recently studied by Chase, Ghosh, and Poburinnaya [12]. Their first step is a 2-party *share translation protocol* which is a simpler problem defined as follows. The parties take no input, and when the protocol terminates:

- P_2 holds random $\boldsymbol{a}, \boldsymbol{b} \in G^N$.
- P_1 holds a random permutation $\pi_1 : \mathbb{Z}_N \to \mathbb{Z}_N$ and $\boldsymbol{\Delta} \in G^N$ such that $\boldsymbol{\Delta} = \pi_1(\boldsymbol{a}) - \boldsymbol{b}$.

Neither party learns anything else about the other party's output.

Once they have a share translation protocol, Chase et al. are able to use it to allow the parties P_1, P_2 to blindly shuffle a secret-shared message using a permutation π held by one of the parties. This process is repeated twice with permutations π_1 and π_2 held by each party, respectively, to achieve a full secret-shared shuffle. Chase et al. construct a 2-party share translation protocol using oblivious transfer (OT) and PRGs. As we will not be using their share translation protocol construction, we omit the details and only state how the resulting share translation is used to conduct a secret-shared shuffle.

The parties use share translation to build a two-party secret-shared shuffle protocol that works as follows: P_2 masks its input share $[\boldsymbol{x}]_2$ using $\boldsymbol{a}$ and sends $\boldsymbol{z}_1 \leftarrow [\boldsymbol{x}]_2 - \boldsymbol{a}$ to P_1. P_1 sets its output to $[\boldsymbol{s}]_1 \leftarrow \pi_1([\boldsymbol{x}]_1 + \boldsymbol{z}_1) + \Delta$, and P_2 sets its output to $[\boldsymbol{s}]_2 \leftarrow \boldsymbol{b}$. Clearly $[\boldsymbol{s}]_1 + [\boldsymbol{s}]_2 = \pi_1(\boldsymbol{x})$. To achieve a full secret-shared shuffle where neither party can determine what permutation was applied to $\boldsymbol{x}$, the parties repeat this process a second time with the roles reversed.

Beaver triples. Our final scheme will require a small multiparty computation (MPC) [36] where the servers verify a MAC on a message while that message remains secret-shared. This will require them to add and multiply secret-shared values. Addition and multiplication by non-secret-shared values can be done non-interactively by each server on its own. However, we will require *beaver triples* to efficiently compute multiplications of two secret-shared values [7].

A Beaver triple is a secret sharing $[a], [b], [c]$ between parties such that $a \cdot b = c$ in $\mathbb{Z}_p$. A Beaver triple enables the parties to efficiently multiply secret-shared quantities that they hold. To multiply secret-shared values $[x]$ and $[y]$, each party begins by computing and publishing $[x-a]$ and $[y-b]$, which reveals the combined values $\epsilon = x - a$ and $\delta = y - b$. Then, each party locally computes the share $[z] \leftarrow [c] + [x] \cdot \delta + [y] \cdot \epsilon - \frac{\epsilon\delta}{N}$, where N is the number of parties involved in the computation. It is easy to verify that this results in each party holding a share $[z]$ of $z = x \cdot y$. Since each beaver triple multiplication requires interaction between parties, MPC protocols using them benefit from mimizing the overall number of multiplications and the multiplicative depth of any arithmetic computations on secret-shared data. More details on beaver triple MPC can be found in a number of excellent online resources (e.g., [56, 62]).

In our three-server protocols, we will have the third server generate the shares of $[a], [b]$, and $[c]$ and give them to the two shuffling servers. Note that the third server could maliciously create malformed beaver triples that do not satisfy the required relationship, but because our threat model prohibits collusion between the servers, the third server, even if it is malicious, cannot cooperate with another server to reveal user secrets. Thus, sending incorrect beaver triples can damage availability but not confidentiality or integrity. In the k-server setting, beaver triples will be created in a maliciously-secure preprocessing phase.

IV. Three Server Anonymous Broadcast

In this section, we present our three-server protocol for anonymous broadcast. We will first describe how three semihonest servers can shuffle secret shared user data such that no server learns the permutation that was applied to the messages sent by the clients. Then we show how to add integrity to build a full anonymous broadcast scheme.

A. *Shuffling Secret-Shared Data*

We will start by using the share translation approach of Chase et al. to shuffle messages, but instead of having two servers generate the share translations using a protocol between them, we have a third server generate the share translation protocol outputs on their behalf. Roughly speaking, each of the first two servers picks random values for $\pi, \boldsymbol{a}, \boldsymbol{b}$, and sends them to the third server, who sends back $\boldsymbol{\Delta} = \pi(\boldsymbol{a}) - \boldsymbol{b}$. Since this process involves the third server learning the permutations used by the first two servers, we additionally have the first two servers share a permutation π_{12}, which they apply to the messages before beginning the shuffle protocol.

Unfortunately, this shuffle involves communication $8 \cdot |\boldsymbol{x}| + 2|\pi|$ bytes of communication between the servers: $2|\pi|$ to send the permutations, $|\boldsymbol{x}|$ for each of the $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{\Delta}$ values sent from/to the two shuffling servers, and another $2|\boldsymbol{x}|$ to perform the Chase et al. secret-shared shuffle given the share translation values.

Our final shuffle is a heavily optimized version of the protocol described above that reduces the overall communication cost to $2 \cdot |\boldsymbol{x}| + 32$ bytes, a reduction of over $4\times$. First, the two shuffling servers generate the values $\pi, \boldsymbol{a}, \boldsymbol{b}$ from random seeds and only send the seeds to the third server, replacing $4 \cdot |\boldsymbol{x}| + 2|\pi|$ bytes of communication with short random seeds.

Second, and more importantly, observe that at the end of two consecutive shuffles, P_1 will hold the share $[\boldsymbol{s}]_1 = \boldsymbol{b}_2$ and P_2 will hold the share

$$[\boldsymbol{s}]_2 = \pi_2\Big(\boldsymbol{b}_1 + \pi_1(\boldsymbol{x} - \boldsymbol{a}_1) + \boldsymbol{\Delta}_1 - \boldsymbol{a}_2\Big) + \boldsymbol{\Delta}_2.$$

It turns out that several of these values are redundant, and performance can be improved by merging or removing them. That is, the secret-shared shuffle can be more efficient than simply doing two share translations back-to-back. In particular, by adjusting how the share translations are used, we can entirely eliminate the need for one party to get the vector $\boldsymbol{\Delta}$ and for the other party to generate a vector $\boldsymbol{b}$.

Our secret-shared shuffle protocol. Before describing the protocol, let us first define the concept of a *shuffle correlation*, which resembles the outputs of two share translation protocols, but removes the components we will no longer need.

Definition IV.1. A **2-party shuffle correlation** is a setting where two parties P_1 and P_2 hold the following data:

- P_1 holds random vectors $\boldsymbol{a}'_2, \boldsymbol{b}_2 \in G^N$ and a random permutation $\pi_1 : \mathbb{Z}_N \to \mathbb{Z}_N$,
- P_2 holds a random vector $\boldsymbol{a}_1 \in G^N$, a random permutation $\pi_2 : \mathbb{Z}_N \to \mathbb{Z}_N$, and a vector $\boldsymbol{\Delta}_2 \in G^N$ such that $\boldsymbol{\Delta}_2 = \pi_2\Big(\pi_1(\boldsymbol{a}_1) + \boldsymbol{a}'_2\Big) - \boldsymbol{b}_2$.

The parties know nothing else about each other's values. □

In Section IV-C, where we describe our full anonymous broadcast, we will explain how the third server can generate a shuffle correlation. For now we will assume that the two parties already have a shuffle correlation, and show how to use it to obtain a protocol for the secret-shared shuffle functionality defined in Section III.

Protocol IV.2 (secret-shared shuffle)**.**
input: P_1 has $[\boldsymbol{x}]_1 \in G^N$ and P_1's shuffle correlation quantities $\boldsymbol{a}'_2, \boldsymbol{b}_2$ and π_1. P_2 has $[\boldsymbol{x}]_2 \in G^N$ and P_2's shuffle correlation quantities $\boldsymbol{a}_1, \boldsymbol{\Delta}_2$, and π_2.
The secret-shared shuffle protocol proceeds as follows:

1) P_2 sends $\boldsymbol{z}_2 \leftarrow [\boldsymbol{x}]_2 - \boldsymbol{a}_1$ to P_1.
2) P_1 sends $\boldsymbol{z}_1 \leftarrow \pi_1(\boldsymbol{z}_2 + [\boldsymbol{x}]_1) - \boldsymbol{a}'_2$ to P_2 and sets $[\boldsymbol{s}]_1 \leftarrow \boldsymbol{b}_2$.
3) P_2 sets $[\boldsymbol{s}]_2 \leftarrow \pi_2(\boldsymbol{z}_1) + \boldsymbol{\Delta}_2$.

At the end of the protocol $[\boldsymbol{s}]_1 + [\boldsymbol{s}]_2 = \pi_2\big(\pi_1(\boldsymbol{x})\big)$. Indeed,

$$\begin{aligned}
[\boldsymbol{s}]_1 + [\boldsymbol{s}]_2 &= \pi_2(\boldsymbol{z}_1) + \boldsymbol{\Delta}_2 + \boldsymbol{b}_2 \\
&= \pi_2(\boldsymbol{z}_1) + \pi_2\big(\pi_1(\boldsymbol{a}_1) + \boldsymbol{a}'_2\big) \\
&= \pi_2\big(\pi_1(\boldsymbol{z}_2 + [\boldsymbol{x}]_1) - \boldsymbol{a}'_2\big) + \pi_2\big(\pi_1(\boldsymbol{a}_1) + \boldsymbol{a}'_2\big) \\
&= \pi_2\big(\pi_1(\boldsymbol{z}_2 + [\boldsymbol{x}]_1) + \pi_1(\boldsymbol{a}_1)\big) \\
&= \pi_2\big(\pi_1([\boldsymbol{x}]_1 + [\boldsymbol{x}]_2)\big) = \pi_2\big(\pi_1(\boldsymbol{x})\big).
\end{aligned}$$

Hence the protocol correctly implements the secret-shared shuffle functionality. Theorem A.2 shows that this protocol is a secure honest-but-curious protocol for this functionality.

B. Integrity

We observe that since all servers hold shares of the data from the beginning of the protocol, a malicious server cannot surreptitiously drop messages without detection, as other servers would notice the change in the number of messages. This significantly reduces the set of attacks the malicious server can potentially mount, limiting it to modifying the contents of its shares. As a result, instead of having servers provide proofs that they have correctly shuffled messages, we will have clients send messages with a special structure such that the servers can later check that they have not been tampered with during the shuffling process.

Adding MACs. As a first attempt at providing integrity, consider a scheme where client messages are MACed with a key k chosen by the client. The MAC and the key are secret-shared along with the message and sent to the servers, so actual message sent by a client is a share of $(\mathsf{k}, \boldsymbol{m}, t)$, where $t \leftarrow \mathsf{MAC}(k, \boldsymbol{m})$. The servers run the secret-shared shuffle protocol on this data.

At the end of the protocol the servers each hash their shares and send each other the hashes. Then they reveal their shares to one another. The exchanged hashes ensure that a malicious server cannot change its shares in an attempt to forge MACs after seeing the keys. The high entropy of the shares, which include shares of MAC keys

unknown to the servers, mean that the hashes hide the shares. Finally, each server verifies the MAC for each message, and only publishes the messages for which the MAC is valid. This ensures that messages that were tampered with are discarded.

Unfortunately, this design does not actually achieve any confidentiality against a malicious server. When receiving a message share from a user whom it wants to surveil, a malicious server can flip some bits in the message share, ensuring that the MAC for that message will fail to verify. Later, the malicious server knows that the message with a MAC that does not verify is the one from the user whose message it corrupted.

Blind MAC verification. To avoid the attack sketched above, we require the servers to *blindly* verify the MACs on all messages before revealing their shares to each other, aborting the entire protocol if any MAC fails to verify. We make this process efficient by using a Carter-Wegman one-time MAC [59] that is optimized to reduce the communication and round complexity of the computation needed to blindly verify the MACs. We carry out the blind validation of the MAC using Beaver triples [7, 56, 62].

In our scheme we use a Carter-Wegman one-time MAC defined as follows: the MAC on a message $\boldsymbol{m} \in \mathbb{Z}_p^\ell$ using a random key $\mathsf{k} \in \mathbb{Z}_p^\ell$ is computed as $t \leftarrow \Sigma_{j=1}^{\ell} \mathsf{k}_j \cdot m_j$. Note that the circuit for computing a MAC has multiplicative depth 1. The keys themselves are generated by expanding two key seeds ks_1 and ks_2 with a pseudorandom generator (PRG) $G : \mathbb{Z}_p \rightarrow \mathbb{Z}_p^\ell$ and adding the results together, i.e., $\mathsf{k} \leftarrow G(\mathsf{ks}_1) + G(\mathsf{ks}_2)$. This allows the client to only send the servers their corresponding key seeds instead of a full vector of keys, reducing communication costs. The security of the MAC follows directly from the DeMillo-Lipton-Schwartz-Zippel lemma [28, 63, 54].

We implement the MAC check by having the servers take the difference $[d_i] \leftarrow [t_i] - [\Sigma_{j=1}^{\ell} \mathsf{k}_j \cdot m_j]$ between the shares of the MAC tag provided by the client and the MAC tag they have blindly computed. The servers check all the MACs at once by taking the sum $[d] \leftarrow \Sigma_i [d_i]$ over all the messages before revealing their shares, thus hiding which particular MAC failed to verify. This does not harm the security of the MAC because under our MAC scheme, this sum is no different than taking a MAC of a single message that is the concatenation of all the separate messages. If $d = 0$, then the servers accept the MAC. Otherwise, the MAC is rejected. The servers send each other commitments to their shares of d before revealing them, preventing a malicious server from waiting to see the other server's share and then producing a fake share that forces $d = 0$.

As described thus far, our scheme is vulnerable to attack by a malicious server who intentionally corrupts the MAC in an attempt to learn part of a user's message, e.g., by incrementing a single key share by one, so that the difference of the provided tag and the computed tag is the corresponding message block. This attack results in the broadcast aborting, but not before the malicious server learns part of a user's message.

We handle this by having the client encrypt its message under a randomly chosen key ek_i, and set the actual message that gets MACed to be the concatenation of the encrypted message and the encryption key, i.e., $(\boldsymbol{c}_i, \mathsf{ek_i})$ where $\boldsymbol{c}_i \leftarrow \mathsf{Enc}(\mathsf{ek}_i, \boldsymbol{m}_i)$. This ensures that no meaningful information is leaked by an incorrect MAC verification check. Since each encryption key is only used for a single message, it suffices for the encryption to be one-time semantically secure. Thus, we implement encryption with AES in counter mode where the count always starts at zero. Thus the final structure of the message sent from each user to the servers is $(\mathsf{ks}, [t], [\boldsymbol{c}], [\mathsf{ek}])$, which includes a share of an encryption of the user's message, a share of the key under which the message is encrypted, a share of the key under which a MAC is computed (after the key shares are expanded by a PRG), and a share of the resulting MAC tag.

Optimizations. Encrypting the client messages enables two final optimizations. First, we can hash messages to commit to them instead of using a commitment scheme because the encrypted messages and ek_i have high entropy and a malicious server who incorrectly computes a MAC will learn nothing from a hash of the other server's share.

More importantly, we can reduce to the number of Beaver triples required to one triple per message. This is accomplished by having the servers reveal their shares of $\boldsymbol{c}_i$ before the MAC verification. Thus the MAC tag for each message will be computed as $[\mathsf{ek}_i] \cdot [k_{\ell+1}] + \Sigma_{j=1}^{\ell} c_{i,j} \cdot [k_j]$, which only includes a single multiplication between secret-shared values. Revealing the ciphertexts reveals nothing about the underlying messages, including whether they were tampered with before the shuffle, so they can be safely revealed without harm to security.

Handling a malicious client. Adding the blind MAC verification creates an opening where a malicious client can launch an in-protocol denial of service attack. It only needs to send a message with an incorrect MAC tag in each batch of messages, and the servers will always

abort the broadcast. The issue is that the current protocol has no way of distinguishing whether a malicious server or client has corrupted the MAC. Recall that we allow servers to abort the protocol because they are trusted for availability, but clients should not be allowed to launch in-protocol DoS attacks. To mitigate this issue, we have the servers conduct another blind MAC verification just as they receive each message, allowing them to filter out any faulty messages sent by clients before beginning the shuffle. Since this check is for protecting against malicious clients, the servers do not need to hash their shares before revealing them to each other when checking the MAC. The optimization to reduce the number of beaver triples also does not apply to this initial blind MAC verification, as revealing ciphertexts both before and after the permutation would make them linkable.

C. The Complete Protocol

We now present the full three server scheme. We denote the three servers as parties P_1, P_2, P_3 and refer to servers P_1 and P_2 together as the "shuffling servers." There are additionally N' clients who send messages $\boldsymbol{m}_1, ..., \boldsymbol{m}_{N'}$ where $\boldsymbol{m}_i \in \mathcal{M}$ for all $i \in [N']$, to the servers in shares. Messages will all be of a fixed length, and the system processes N of these messages in each round. P_1 and P_2 share a secret seed from which they generate a fresh permutation π_{12} over the elements of $\mathbb{Z}_N$ for each run of the protocol. The client and servers will make use of (1) an encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ where $\mathsf{Enc} : \mathbb{Z}_p \times \mathcal{M} \rightarrow \mathbb{Z}_p^\ell$ is defined as $\mathsf{Enc}(\mathsf{ek}, \boldsymbol{m}) = \boldsymbol{m} + H(\mathsf{ek})$ for a random oracle $H : \mathbb{Z}_p \rightarrow \mathbb{Z}_p^\ell$, instantiated with AES in counter mode and modeled as a distinct random oracle from the one used for other hashes in the protocol, and (2) a PRG $G : \mathbb{Z}_p \rightarrow \mathbb{Z}_p^{\ell+1}$. For our purposes, p is a 128-bit prime. The protocol proceeds as follows. We prove its security in Appendix A.

Clients send messages. Each client samples two MAC key seeds $\mathsf{ks}_{i,1}, \mathsf{ks}_{i,2} \in \mathbb{Z}_p$, from which it computes the one-time MAC key

$$\mathbf{k}_i = (\mathsf{k}_{i,1}, \ldots, \mathsf{k}_{i,\ell+1}) \leftarrow G(\mathsf{ks}_{i,1}) + G(\mathsf{ks}_{i,2}) \in \mathbb{Z}_p^{\ell+1}.$$

It also samples an encryption key $\mathsf{ek}_i \in \mathbb{Z}_p$ and produces the ciphertext $\boldsymbol{c}_i \leftarrow \mathsf{Enc}(\mathsf{ek}_i, \boldsymbol{m}_i)$.

It then computes the fixed-length MAC tag $t_i \leftarrow \mathsf{k}_{i,\ell+1} \cdot \mathsf{ek_i} + \Sigma_{j=1}^{\ell} \mathsf{k}_{i,j} \cdot c_{i,j}$. It sends the share $(\mathsf{ks}_{i,1}, [t_i]_1, [\boldsymbol{c}_i]_1, [\mathsf{ek}_i]_1)$ to P_1 and the share $(\mathsf{ks}_{i,2}, [t_i]_2, [\boldsymbol{c}_i]_2, [\mathsf{ek}_i]_2)$ to P_2.

Seed expansion. For each message, the servers compute vectors of shares $[\mathbf{k}_i]_1 \leftarrow G(\mathsf{ks}_{i,1})$ and $[\mathbf{k}_i]_2 \leftarrow G(\mathsf{ks}_{i,2})$ respectively. Let $\boldsymbol{x}_i = (\mathbf{k}_i, t_i, \boldsymbol{c}_i, \mathsf{ek}_i) \in \mathbb{Z}_p^{2\ell+2}$ and $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$. Server P_1 has a share $[\boldsymbol{x}]_1$ and P_2 has $[\boldsymbol{x}]_2$.

Blind MAC verification. The shuffling servers use an MPC to check that $t_i = \mathsf{k}_{i,\ell+1} \cdot \mathsf{ek_i} + \sum_{j=1}^{\ell} \mathsf{k}_{i,j} \cdot c_{i,j}$ for every entry $i \in [N]$ of $[\boldsymbol{x}]$. Each verification requires $\ell + 1$ multiplications of secret shared data. All of this can be done in parallel in one round using $(\ell + 1)N$ beaver triples. These beaver triples are supplied by the third server P_3.

The equality check is computed by the servers locally taking the difference of shares $[d_i] \leftarrow [t_i] - [\mathsf{k}_{i,\ell+1} \cdot \mathsf{ek_i} + \Sigma_{j=1}^{\ell} \mathsf{k}_{i,j} \cdot c_{i,j}]$ and revealing their shares of d_i, with verification passing when $d_i = 0$. Messages for which the MAC fails to verify are removed.

To ensure an appropriate anonymity set for each message, the servers wait until N messages have passed this blind MAC verification before proceeding to the next step of the protocol.

Secret-shared shuffle. The shuffling servers now have vectors $[\boldsymbol{x}]_1$ and $[\boldsymbol{x}]_2$. They begin by each permuting their shares according to π_{12} to compute $[\boldsymbol{x}'] \leftarrow \pi_{12}([\boldsymbol{x}])$.

Next, server P_1 chooses a random seed, expands it to get $\pi_1 : \mathbb{Z}_N \rightarrow \mathbb{Z}_N$ and $\boldsymbol{a}'_2, \boldsymbol{b}_2 \in \mathbb{Z}_p^{N \times (2\ell+2)}$ (these are length-N vectors with entries of size $2\ell + 2$), and sends the seed to P_3, who expands it to recover the same values. P_2 does the same to get $\pi_2, \boldsymbol{a}_1$ and give them to P_3. Server P_3 computes $\boldsymbol{\Delta}_2 \leftarrow \pi_2\Big(\pi_1(\boldsymbol{a}_1) + \boldsymbol{a}'_2\Big) - \boldsymbol{b}_2$ and sends it back to P_2.

The shuffling servers now have a shuffle correlation, which they use to execute a secret-shared shuffle using Protocol IV.2 on their shared vector $\boldsymbol{x}'$. At the end of the protocol the shuffling servers hold shares of a vector $\boldsymbol{s}$, where $\boldsymbol{s} = \pi_2(\pi_1(\pi_{12}(\boldsymbol{x})))$. No single server knows all three permutations used to shuffle $\boldsymbol{x}$.

Second blind MAC verification. The servers run a second blind MAC verification for every entry $([\mathbf{k}_i], [t_i], [\boldsymbol{c}_i], [\mathsf{ek}_i]) \in [\boldsymbol{s}]$. They begin by revealing their shares of $[\boldsymbol{c}_i]$ to each other to recover all the $\boldsymbol{c}_i$. They then use an MPC with N beaver triples to check

$$\Sigma_{i=1}^{N} [t_i] = \Sigma_{i=1}^{N} \Big([\mathsf{k}_{i,\ell+1}] \cdot [\mathsf{ek}_i] + \Sigma_{j=1}^{\ell} [\mathsf{k}_{i,j}] \cdot c_{i,j} \Big).$$

The beaver triples are supplied by the third server. As in the first blind MAC verification, all the multiplications can be done in one round. Note, however, that this time we verify all the MACs together as one batch, so if any MAC fails to verify, the servers output abort and stop, without revealing which particular MAC failed.

The equality check is performed by taking the difference

$$[d] \leftarrow [\Sigma_{i=1}^{N} t_i] - [\Sigma_{i=1}^{N}\Big(\mathsf{k}_{i,\ell+1} \cdot \mathsf{ek}_i + \Sigma_{j=1}^{\ell} \mathsf{k}_{i,j} \cdot c_{i,j}\Big)],$$

revealing the shares of d, and checking that $d = 0$. To prevent a malicious shuffling server from lying about its share to force the result to be zero, we have the servers first send each other hashes of their shares of d before revealing the actual shares.

Output phase. The shuffling servers hash their shuffled shares and send the hashes to each other. Once it has received the other shuffling server's hash, each server sends over its share of $\boldsymbol{s}$ and checks that the share it receives matches the hash. For the purposes of our security proof in Appendix A, we model the hash function as a random oracle.

The shuffling servers then merge the shares to recover $(\mathbf{k}_i, t_i, \boldsymbol{c}_i, \mathsf{ek}_i) \in \mathbb{Z}_p^{2\ell+2}$ for $i \in [N]$. They check each MAC, aborting if a verification fails, recover $\boldsymbol{m}_i \leftarrow \mathsf{Dec}(\mathsf{ek}_i, \boldsymbol{c}_i)$, and make the messages available to clients for download.

Complexity. Our scheme requires $O(N\ell)$ field multiplications and PRG output evaluations to shuffle N messages of length ℓ. Communication costs between the servers and between the servers and the client are also $O(N\ell)$. The cost for a client to send a message of length ℓ is $O(\ell)$, and its communication cost is exactly $\ell + 3$ elements of $\mathbb{Z}_p$ sent to each server. This is asymptotically optimal, as a system that simply receives and broadcasts N messages with no security guarantees must also do $O(N\ell)$ computation and communication.

We discuss potential alternative MAC schemes that could be used to instantiate our design in Appendix B

V. k-Server Scheme

In this section, we show how to adapt the shuffling technique used for three server anonymous broadcast in Section IV to any $k > 2$ servers with malicious security against $k - 1$ servers. This increase in security comes at the cost of a more expensive per-shuffle preprocessing phase that replaces the role of the third server in the previous scheme. Since this scheme offers fast online time but requires more expensive precomputation, it is ideal for settings like an online event with live anonymous comments: the preprocessing information can be prepared while the event is planned, and comments can be delivered quickly during the live event.

The k-server scheme involves server parties $P_1, ..., P_k$ and is very similar to the three server scheme. Messages sent by clients are identical to those sent in the three-server scheme except that the key vector k_i is generated by more shares:

$$\mathbf{k}_i \leftarrow G(\mathsf{ks}_{i,1}) + ... + G(\mathsf{ks}_{i,k}),$$

where each $\mathsf{ks}_{i,j}$ is sent to server P_j. The protocol for conducting integrity checks is identical to that of the three server case, except the multiplications require beaver triples shared among k servers instead of only two. Moreover, instead of using a third server to generate the triples, they are generated in a per-round preprocessing phase using standard techniques [24, 38].

Since the client input and server-side verification procedures require only minimal changes compared to the three-server setting, we will focus our attention on changing the shuffling stage of the protocol to efficiently accommodate k servers.

A. k-Server Shuffling

As described in Section III, the share translation protocol of Chase et al. [12] roughly allows one party to apply a permutation to the shares held by another party without revealing the permutation. This naturally implies a k-server shuffle protocol made up of pairwise share translations between all the parties. In a setting where each P_i holds permutation π_i, the protocol begins with P_1 doing a share translation protocol separately with each party $P_j, j \neq 1$, where the other parties' shares are permuted according to π_1. Then, one at a time, each subsequent P_i repeats this process with all other parties $P_j, j \neq i$, until every party's share has been permuted under all permutations $\pi_1, ..., \pi_k$. This protocol would result in communication and computation cost $O(k^2 N\ell)$. In this section, we show how improve this naïve protocol, reducing the cost to $O(kN\ell)$.

The important observation is that when party P_i uses the share translation protocol to shuffle the shares of all the other parties, every party $P_j \neq P_i$ has as its output a random value $\boldsymbol{b}$ that does not depend on the user data being shuffled. When it is time for the the next party to shuffle, all the parties mask their random outputs with another random value $\boldsymbol{a}$ and send that to P_{i+1}. But of the values being masked, only P_i holds data that actually depends on user inputs at this point. The observation that most of the computation and communication in the naïve protocol involves shares that do not depend on user inputs allows us to push that computation into a preprocessing phase or even remove it altogether.

Our final protocol reduces the computation of the naïve protocol to only include those steps that depend on user

data. As before, the protocol begins with all the servers masking the initial shares they have received from the users and sending them to P_1, who shuffles them. Then P_1 masks its resulting share and sends it to P_2, which shuffles only the share from P_1. This process continues, with each server P_i shuffling its share and sending it to server P_{i+1}, until the final server P_k has performed its shuffle. Then the server P_k applies a single correction value $\boldsymbol{\Delta}_k$, generated during the preprocessing phase, that removes the now-permuted masks that each server added to its data before sending it to P_1. We formalize the protocol below.

(i) Offline phase. A pre-processing protocol described in Section V-B below generates a k-party shuffle correlation, defined as follows.

Definition V.1. A **k-party shuffle correlation** is a situation where for $i = 1, \ldots, k$ server P_i holds:

- random vectors $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{a}'_i \in G^N$ and
- a random permutation $\pi_i : \mathbb{Z}_N \to \mathbb{Z}_N$.

Server P_k holds an additional vector $\boldsymbol{\Delta}_k \in G^N$ where:

$$\boldsymbol{\Delta}_k = \pi_k(\ldots \pi_2(\pi_1(\Sigma_{i=2}^k \boldsymbol{a}_i) + \boldsymbol{a}'_1) + \boldsymbol{a}'_2) \ldots + \boldsymbol{a}'_{k-1}) - \Sigma_{i=1}^{k-1} \boldsymbol{b}_i.$$

The servers know nothing else about each other's values.

(ii) Collect data. For $i = 1, \ldots, k$ server P_i receives a share $[\boldsymbol{x}]_i \in G^N$ from the N end users (each user contributes one entry in $[\boldsymbol{x}]_i$).

(iii) Shuffle. The k servers do:

1) All servers P_i, where $i \neq 1$, produce $\boldsymbol{z}_i \leftarrow [\boldsymbol{x}]_i - \boldsymbol{a}_i$ and send $\boldsymbol{z}_i$ to server P_1.
2) Server P_1 computes $\boldsymbol{z}'_1 \leftarrow \pi_1(\Sigma_{i=2}^k \boldsymbol{z}_i) - \boldsymbol{a}'_1$ and sends $\boldsymbol{z}'_1$ to server P_2. Then it sets its output to $[\boldsymbol{s}_1] \leftarrow \boldsymbol{b}_1$.
3) For $i \in \{2...k-1\}$, server P_i computes $\boldsymbol{z}'_i \leftarrow \pi_i(\boldsymbol{z}'_{i-1}) - \boldsymbol{a}'_i$ and sends it to P_{i+1}. Then it sets its output to $[\boldsymbol{s}]_i \leftarrow \boldsymbol{b}_i$.
4) Server P_k outputs $[\boldsymbol{s}]_k \leftarrow \pi_k(\boldsymbol{z}'_{k-1}) + \boldsymbol{\Delta}_k$.

The servers now hold shares of $[\boldsymbol{s}]$, which is a secret sharing of $\pi_k(...\pi_1([\boldsymbol{x}])...)$, and no set of $k-1$ servers knows all the permutations used to shuffle $[\boldsymbol{x}]$. This completes the shuffling protocol.

We prove security of this shuffle in Appendix A.

B. Preprocessing Phase

The offline phase of our k-server scheme has two goals: (i) generate the beaver triples needed for blind MAC verification, and (ii) prepare the k-party shuffle correlation needed for the k-server shuffle. We use standard techniques for generating beaver triples [24, 38] and the protocol of Chase et al. [12] to produce share translations $\boldsymbol{a}_{i,j}, \boldsymbol{b}_{i,j}, \boldsymbol{\Delta}_{i,j}$ for all $i, j \in [k], i \neq j$ such that $\boldsymbol{\Delta}_{i,j} = \pi_i(\boldsymbol{a}_{i,j}) - \boldsymbol{b}_{i,j}$, with P_i holding $\pi_i, \boldsymbol{\Delta}_{i,j}$ and P_j holding $\boldsymbol{a}_{i,j}, \boldsymbol{b}_{i,j}$. Note that producing these share translations will be slightly more expensive than reported by Chase et al. [12] because they estimated performance for semihonest security, whereas our instantiation of their protocol would require malicious security. In practice, this only requires replacing semihonest-secure oblivious transfers with malicious-secure ones.

From here, we need to process these share translation values to generate the k-party shuffle correlation required for our shuffling protocol. In particular, we need to produce the vectors $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{a}'_i$ for each party P_i and additionally $\boldsymbol{\Delta}_k$ for party P_k. We define the values of the k-party shuffle correlation as follows.

- $\boldsymbol{a}_i \leftarrow \boldsymbol{a}_{1,i}$
- $\boldsymbol{b}_i \leftarrow \boldsymbol{b}_{k,i}$
- $\boldsymbol{a}'_i \leftarrow \Sigma_{j \in [k] \setminus i}^k \boldsymbol{b}_{i,j} + \boldsymbol{\Delta}_{i,j} + \boldsymbol{a}_{i+1,j}$
- $\boldsymbol{\Delta}_k \leftarrow \Sigma_{j \in [k-1]} \boldsymbol{\Delta}_{k,j}$

Of these, $\boldsymbol{a}_i, \boldsymbol{b}_i$, and $\boldsymbol{\Delta}_k$ can be computed by each server using only its own outputs from the Chase et al. protocol. However, $\boldsymbol{a}'_i$ requires summing vectors held by different servers. This can be achieved with a simple MPC, involving only additions, where each server shares all its vectors $[\boldsymbol{a}_{i,j}], [\boldsymbol{b}_{i,j}]$, and $[\boldsymbol{\Delta}_{i,j}]$. The servers then compute the appropriate sums $[\boldsymbol{a}'_i]$ over the shares and send the shares of each sum $\boldsymbol{a}'_i$ to P_i.

Omitting message sizes and log factors that vary based on tunable parameters, the Chase et al. protocol has cost $\tilde{O}(N)$ to produce share translations for shuffling N messages, meaning that the preprocessing phase has complexity $\tilde{O}(k^2 N)$. If k becomes large enough that the k^2 factor outweighs the benefits of using the Chase et al. scheme over generic MPC, the parties could use an alternative approach where all values except $\boldsymbol{\Delta}_k$ are chosen uniformly at random. Then an MPC is used to compute $\boldsymbol{\Delta}_k$ such that $\boldsymbol{\Delta}_k = \pi_k(...\pi_2(\pi_1(\Sigma_{i=2}^k \boldsymbol{a}_i) + \boldsymbol{a}'_1) + \boldsymbol{a}'_2) ... + \boldsymbol{a}'_{k-1}) - \Sigma_{i=1}^{k-1} \boldsymbol{b}_i$. This MPC would consist only of additions and k permutations, which can be implemented with oblivious sorts, resulting in overall cost $\tilde{O}(kN)$.

VI. Anonymous Messaging

We now show how to use our three-server anonymous broadcast technique to improve the MPC-based anonymous messaging scheme MCMix [3]. Although the actual design of MCMix will not be relevant to our discussion here, the core technique used in MCMix is to sort the data

passing through the system in various ways to exchange messages between communicating parties, resulting in $O(N \log N)$ complexity to handle N messages. MCMix, like many other anonymous messaging systems, consists of a conversation protocol that delivers actual messages and a dialing protocol that initiates conversations. MCMix operates in a three server setting, but its implementation only offers security against a single *semihonest* server. In addition to improving performance, the conversation protocol we introduce will achieve security against a single *malicious* server. Our improvements to the dialing protocol use the MCMix dialing protocol in a black-box way, so they inherit the semihonest security of the MCMix protocol.

Conversation protocol. We first observe that anonymous broadcast can be used as a drop-in replacement for the conversation protocol in many anonymous messaging protocols, including MCMix. What the communicating parties A and B need from the dialing protocol is a shared encryption key k_{AB} for an underlying end-to-end encrypted messaging scheme, another shared secret s_{AB} used to facilitate message transfer, and knowledge of whether they will or will not be receiving a message in a given round. Clients use the shared secret s_{AB} for message transfer as a PRF key which is expanded to different pseudorandom "addresses" $\mathsf{addr}_{rAB} \leftarrow F(s_{AB}, (r, \text{“A, B”}))$, $\mathsf{addr}_{rBA} \leftarrow F(s_{AB}, (r, \text{“B, A”}))$ (sometimes called a "dead drops") for each round r of communication.

Using the information from the dialing protocol, a client involved in a conversation sends a message to the anonymous broadcast system that includes the ciphertext it wants delivered as well as the address it shares with its conversation partner. Specifically, the message a client A wanting to send $\boldsymbol{m}$ to B sends to the broadcast system in round r is $\mathsf{addr}_{rAB} || \mathsf{Enc}_{\mathsf{E2E}}(\mathsf{k}_{AB}, \boldsymbol{m})$, where $\mathsf{Enc}_{\mathsf{E2E}}$ is some end-to-end encrypted messaging scheme. Client B sends a message $\mathsf{addr}_{rBA} || \mathsf{Enc}_{\mathsf{E2E}}(\mathsf{k}_{AB}, \boldsymbol{m'})$ where $\boldsymbol{m'}$ can either be a real message or simply a string of 0s if B doesn't have a message to simultaneously send back to A. Note k_{AB} may differ for each message sent under end-to-end encrypted messaging schemes that employ ratcheting [51]. A client C not sending a message in round r uses a randomly chosen address $\mathsf{addr}_{rC} \xleftarrow{R} \mathbb{Z}_p$ and encrypts a message consisting of all zeros under a random key k_C.

Clients attempting to read messages simply need to query the servers for the address from which they want to read, and clients not receiving messages query the same random address they included in the message they sent. That is, A, would query the servers for the message that begins with addr_{rBA}, B would query for the message that begins with addr_{rAB}, and C would query for the message that begins with addr_{rC}. Since the servers can see all the final messages in the anonymous broadcast, they can send each client its requested messages.

Every client always sends and then retrieves a single message identified by a random address through the broadcast system, so the behavior of clients having conversations and not having conversations is indistinguishable. Moreover, every message includes a random address string followed by a ciphertext. This means the anonymous renders conversation partners unlinkable because nothing about the messages themselves or which address a client accesses reveals which parties are speaking to each other.

Using this approach for messaging gives the conversation protocol the $O(N\ell)$ complexity of our shuffling scheme, removing a $\log N$ factor compared to MCMix, and also upgrades security to allow one malicious server instead of a semihonest one.

There is an additional optimization that improves performance when using an anonymous broadcast scheme for messaging as described here. In this special case, the addresses addr_r are the only parts of the client messages that actually need to be MACed and blindly verified in the broadcast scheme because the rest of the message is protected by the authenticated encryption of the underlying end-to-end encryption scheme used to protect the messages.

Dialing protocol. Our shuffling technique can also be used to improve the MCMix dialing protocol. With our improvement, server costs will be $O(N)$ in the number of total messages and $O(N_R \log N_R)$ in the number $N_R \leq N$ of real, non-cover messages, i.e., the messages that are actually initiating conversations. This results in an overall cost $O(N + N_R \log N_R)$. In the original MCMix scheme, the cost is $O(N \log N)$ for all messages. This clearly leaks the number of real conversations to the servers, but gives no information about which users are involved in real conversations. This technique was first suggested by Chase et al. [12] in the context of applying secret-shared shuffles to MPC on private set intersections.

The idea is for clients to add a "cover" bit to all messages, indicating whether each message is cover traffic or not. The servers start by doing a shuffle, revealing the cover bits, and separating the cover messages from the non-cover messages. Then they perform the usual MCMix dialing protocol on the non-cover messages. At the end of the protocol, the cover messages are added back in at

the same locations from which they were removed, the cover bits are dropped from each message, and another shuffle occurs using the inverse of the permutations used in the original shuffle. This puts the cover messages back in their original locations, without the servers learning which entries were cover messages and which were real.

Although the shuffles used in this transformation have security against malicious servers themselves, the overall protocol inherits the semihonest security of the MCMix dialing implementation.

VII. Implementation

We implemented our schemes in a system called Clarion, which includes our three server anonymous broadcast system, the variant of the broadcast system used for the anonymous messaging conversation protocol, and the multi-server version of our broadcast protocol, all in Go. We did not implement the multi-server preprocessing or the modified MCMix dialing protocol. Performance estimates for the main components of both can be found in prior work [12, 38, 3]. Our code is open source and available at `https://github.com/SabaEskandarian/Clarion`.

We use the Goff library for fast finite field arithmetic in Go [9]. We instantiate our hash function with SHA256 and our encryption scheme with AES in counter mode. Although the formalization of our scheme requires a cipher with keyspace $\mathbb{Z}_p$, the statistical distance between $\mathbb{Z}_p$ and a uniformly random 128-bit key is negligible for our choice of $p = 2^{128} - 159$.

Clarion includes a number of performance optimizations, several of which we discuss in Appendix C.

VIII. Evaluation

We evaluated Clarion on Google cloud using compute instances spread across the us-east, us-central, and us-west regions and running Ubuntu Linux with 2, 8, or 16 vCPUs and 48 or 64GB of RAM. We set the power of the instances used for each comparison to match the machines used to evaluate the works to which we compare. All performance numbers for Clarion are averages of 5 runs, and our evaluation uses the malicious-secure versions of our protocols that include blind MAC verification.

Throughout our evaluation, on all message sizes, message batch sizes, and evaluation instance configurations, the client computation to prepare a message never exceeded 1ms. Thus we focus our evaluation on server-side performance, where most costs are incurred.

Evaluation overview. We begin by comparing our three server shuffling technique to a system where each server computes a separate verifiable shuffle. We compare the two approaches when they are used for anonymous broadcast (Section IV) and anonymous messaging conversation protocols (Section VI). We find that we outperform verifiable shuffles for anonymous broadcast of 32 Byte messages by $9.2\times$, but our performance becomes worse when message sizes increase to 1KB. We perform significantly better in the messaging setting, where we deliver 160 Byte messages $8.2\times$ faster than the verifiable shuffle-based system and 1KB messages $1.9\times$ faster.

Next, we evaluate the effectiveness of our approach in improving the performance of an existing MPC-based anonymous messaging system, MCMix [3]. We find that our system has a conversation protocol that delivers 160 Byte messages $11.8\times$ faster than MCMix. We also estimate the performance gains our optimizations can have on the MCMix dialing protocol.

Finally, we evaluate the performance of the online phase of our k-server protocol (Section V), showing that the performance cost of adding a server is less than that of adding a server to the verifiable shuffle system, and that Clarion significantly outperforms prior work that does not take advantage of a preprocessing phase.

Comparison to verifiable shuffles. We conduct comparisons of our three server shuffling system to a two server system where each server runs a separate verifiable shuffle, one at a time. The difference in the number of servers is so that both systems offer security against 1 malicious server, ensuring a fair comparison. The verifiable shuffle we compare to is the optimized Bayer-Groth shuffle proof initially built and used in the Stadium system [6, 57]. We calculate the system's total shuffling time as the time to compute two verifiable shuffles minus the time to verify the first shuffle proof (because this can be done in parallel with preparing the second proof). Our reported times underestimate the true running time because they only shuffle one group element per message instead of the whole message, as this is the most expensive part of the verifiable shuffle. We also do not include the cost of network communication for the verifiable shuffle, but we do for our own system. This comparison was run using Google cloud instances with 16 vCPUs and 64GB of RAM.

Figure 2 shows the performance of Clarion on various message sizes as compared to the verifiable shuffle system. For a batch size of 1 million messages, Clarion outperforms verifiable shuffles by $9.2\times$ on 32 Byte messages and $3.0\times$ for 160 Byte messages, but is $1.8\times$ slower for 1KB messages. This is because the cost of blind MACs in Clarion is linear in the length of messages,

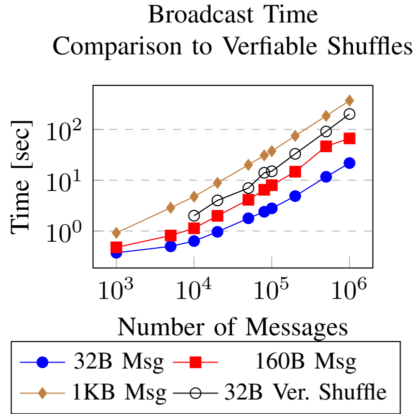

Fig. 2: Performance of our three server anonymous broadcast scheme for various message sizes compared to a verifiable shuffle system with 2 servers. For 1M 32 Byte messages, our scheme shuffles the messages 9.2× faster.

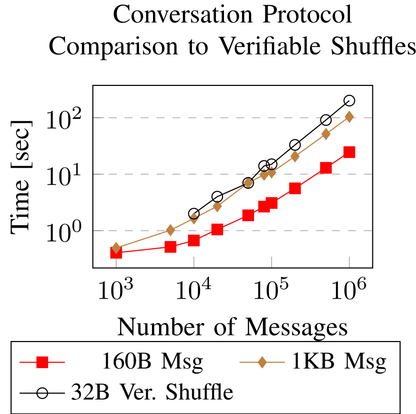

Fig. 3: Performance of our anonymous messaging conversation protocol for various message sizes compared to a verifiable shuffle system with 2 servers. For 1M 160 Byte messages, our scheme shuffles the messages 8.2× faster.

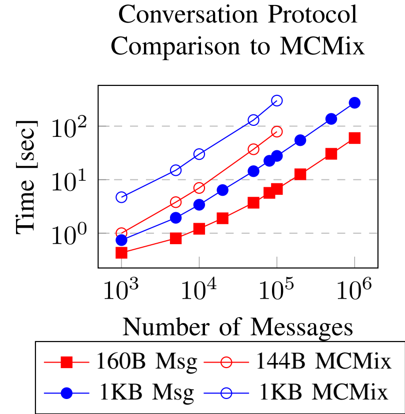

Fig. 4: Performance of our messaging conversation protocol for various message sizes compared to MCMix [3]. For 100,000 160 Byte messages, our scheme shuffles the messages 11.8× faster than MCMix.

but the verifiable shuffle only needs to run on a fixed-size component of each message. Although a full verifiable shuffle system implementation would also need to do additional computation, roughly corresponding to the cost of evaluating AES for the length of each message, we do not expect that this would significantly increase costs. We conclude that our approach can lead to significant performance improvements for anonymous broadcast in shorter messages, but a verifiable shuffle system will perform better on large messages.

The situation changes, however, when we consider the conversation protocol in anonymous messaging schemes, as shown in Figure 3. Here we outperform verifiable shuffles by 8.2× for 160 Byte messages and 1.9× for 1KB messages. The improvement is due to the fact that in a conversation protocol, the servers in Clarion only need to blindly verify MACs on a short identifier for each message, not the messages themselves, which are ultimately verified by communicating clients using the underlying end-to-end encryption scheme.

Comparison to MCMix. Figure 4 compares our conversation protocol with the reported performance of the MCMix conversation protocol. For this comparison, we run our scheme on Google cloud instances with 2 vCPUs and 48GB of RAM to match the MCMix evaluation. The figure shows that we outperform the MCMix protocol by 11.8× when delivering 100,000 160 Byte messages and by 10.8× when delivering 1KB messages.

We also estimate the performance improvement we can achieve over MCMix by using our technique to make the cost of processing cover traffic cheaper than real traffic in the dialing protocol (as described in Section VI). The performance benefit will vary depending on what fraction of the messaging service's users are initiating conversations at any time, so we will assume conservatively that perhaps 20% of the service's user base starts a new conversation in each dialing round. This means that the cost of dialing with 500,000 users, the largest number of dialing messages on which MCMix was evaluated, will be the cost of two shuffles using our scheme on 500,000 messages, plus the cost of the original MCMix scheme on 100,000 messages. This comes out to under 100 seconds in our improved version versus over 200 seconds in the original scheme, a more than 2× improvement. The improvement will increase as we reduce the fraction of users that we assume are starting a conversation in each dialing round.

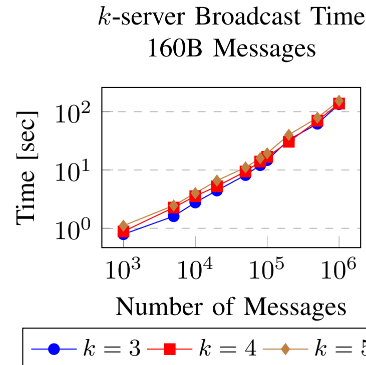

Fig. 5: Performance of online phase of the k-server broadcast.

k**-server scheme.** Figure 5 shows the performance of the online portion of our k-server protocol broadcasting 160 Byte messages, excluding preprocessing, for three, four, and five servers. This evaluation was conducted on Google cloud instances with 8 vCPUs and 64GB of RAM. Compared to verifiable shuffles run by 3, 4, or 5 servers, which would provide the same security properties, Clarion mixes 1 Million messages $2.9\times$, $3.6\times$, and $4.0\times$ faster, respectively. Thus the more servers are desired, the more shuffling secret shared messages improves over using verifiable shuffles.

In Appendix D, we compare our k-server protocol with other systems that use MPC techniques for anonymous communication.

IX. Related Work

In general, Clarion can be thought of as sitting between mixnet approaches [13, 25, 33, 42, 41, 43, 45] and more recent MPC techniques for anonymous communication [47, 2, 3]. By having all servers involved in each shuffle and imposing a special structure on messages sent by clients, we make the task of proving that a shuffle was computed honestly much easier.

A large class of systems also build anonymous communication from DC-nets and related approaches [14, 35, 22, 60, 23, 21]. Compared to these works, we remove the linear server-side work for each request. Since communication-efficient DC-net approaches often rely on DPFs [34, 11, 10, 21, 32, 49], we also get around the communication and efficiency issues that arise when scaling DPFs past two servers. For example, while systems like Riposte [21] or Express [32] process individual messages faster than Clarion, their overall $O(N^2)$ complexity to handle N messages makes them much slower, e.g., Riposte estimates 11 hours to get an anonymity set of 1,000,000 users whereas Clarion performs a shuffle of similar size in minutes.

We similarly avoid the linear server-side cost per message involved when using PIR approaches [18, 50, 19, 53, 39, 40, 8, 5, 4, 17]. Our work offers a security/performance tradeoff compared to Pung [5], which is a single-server system secure against a malicious server and therefore does not need to make any kind of non-collusion assumption. Clarion offers stronger performance and lower costs than Pung, but it also makes a stronger non-collusion assumption among the participating servers.

The most widely used anonymity system today is Tor [30]. Tor and similar low-latency anonymity systems achieve strong performance but are vulnerable to traffic analysis by a passive adversary with a view of enough of the network [29, 37]. Recent impossibility results suggest that this limitation may be necessary [26, 27].

Recent iterations on the low-latency mix network design, e.g., HORnet [15], Loopix [52], and TARAnet [16], provide stronger resistance to passive attacks, and even a degree of protection against active attacks, while still incurring minimal latency overheads. As an example of the tradeoffs involved in these systems, the Loopix anonymity system provides low latency on the order of seconds to process messages because it does not require the protocol to run in synchronized rounds, which lead to high latencies in round-based systems. On the other hand, Loopix measures security by observing the expected difference in likelihood, from the perspective of an attacker, that a given message originated from one sender versus another. Whereas this measure of security varies based on system parameters in Loopix and other low-latency anonymity systems, the difference in likelihood in Clarion will always be zero, as Clarion's confidentiality property guarantees that the source of a message is completely hidden among the set of potential senders. Thus Clarion offers a stronger security guarantee at the cost of significantly higher latency.

Works based on differential privacy [58, 46, 57, 44] achieve high throughput at the expense of only providing differentially private security guarantees that gradually exhaust a privacy budget over time as more messages are sent. Since these systems also often use verifiable shuffling as part of their solutions, our techniques may have applications in speeding them up as well.

X. Conclusion and Future Work

We have shown how to use multiparty shuffling protocols to build faster metadata-hiding communication systems in the three-server and k-server settings. Our k-server scheme comes at the cost of a more expensive preprocessing phase. This raises the interesting question of whether we can reduce the cost of preprocessing or if it would be possible to replace the per-shuffle preprocessing with a one-time preprocessing that can be reused. We leave this as a compelling problem for future work.

Acknowledgments

We would like to thank our shepherd Pedro Moreno-Sanchez for helpful comments in preparing the final version of this paper. This work was funded by NSF, DARPA, a grant from ONR, and the Simons Foundation. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA. Google provided GCP credits used to perform our evaluation.

REFERENCES

[1] J. Aas and T. Geoghegan. Introducing isrg prio services for privacy respecting metrics. https://www.abetterinternet.org/post/introducing-prio-services/, 2020.

[2] I. Abraham, B. Pinkas, and A. Yanai. Blinder: MPC based scalable and robust anonymous committed broadcast. *IACR Cryptol. ePrint Arch.*, 2020:248, 2020.

[3] N. Alexopoulos, A. Kiayias, R. Talviste, and T. Zacharias. Mcmix: Anonymous messaging via secure multiparty computation. In *USENIX Security*, 2017.

[4] S. Angel, H. Chen, K. Laine, and S. T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy*, 2018.

[5] S. Angel and S. T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, 2016.

[6] S. Bayer and J. Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT*, 2012.

[7] D. Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[8] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. *PoPETs*, 2015.

[9] G. Botrel, G. Gutoski, and T. Piellard. Goff. https://github.com/ConsenSys/goff, 2018.

[10] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *EUROCRYPT*, 2015.

[11] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS*, 2016.

[12] M. Chase, E. Ghosh, and O. Poburinnaya. Secret shared shuffle. *IACR Cryptol. ePrint Arch.*, 2019:1340, 2019.

[13] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–88, 1981.

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

[15] C. Chen, D. E. Asoni, D. Barrera, G. Danezis, and A. Perrig. HORNET: high-speed onion routing at the network layer. In *ACM CCS*, 2015.

[16] C. Chen, D. E. Asoni, A. Perrig, D. Barrera, G. Danezis, and C. Troncoso. TARANET: traffic-analysis resistant anonymity at the network layer. In *IEEE European Symposium on Security and Privacy, EuroS&P*, 2018.

[17] R. Cheng, W. Scott, B. Parno, I. Zhang, A. Krishnamurthy, and T. Anderson. Talek: a Private Publish-Subscribe Protocol. Technical Report UW-CSE-16-11-01, University of Washington Computer Science and Engineering, Seattle, Washington, Nov 2016.

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

[19] D. A. Cooper and K. P. Birman. Preserving privacy in a network of mobile computers. In *IEEE Symposium on Security and Privacy*, 1995.

[20] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, 2017.

[21] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE Symposium on Security and Privacy*, 2015.

[22] H. Corrigan-Gibbs and B. Ford. Dissent: accountable anonymous group messaging. In *ACM CCS*, 2010.

[23] H. Corrigan-Gibbs, D. I. Wolinsky, and B. Ford. Proactively accountable anonymous messaging in verdict. In *USENIX Security*, 2013.

[24] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[25] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *IEEE Symposium on Security and Privacy*, 2003.

[26] D. Das, S. Meiser, E. Mohammadi, and A. Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In *IEEE Symposium on Security and Privacy*, 2018.

[27] D. Das, S. Meiser, E. Mohammadi, and A. Kate. Comprehensive anonymity trilemma: User coordination is not enough. *PoPETs*, 2020(3):356–383, 2020.

[28] R. A. DeMillo and R. J. Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7(4):193–195, 1978.

[29] R. Dingledine. One cell is enough to break tor's anonymity, 2009.

[30] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[31] S. Englehardt. Next steps in privacy-preserving telemetry with prio. https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/, 2019.

[32] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. *CoRR*, abs/1911.09215, 2019.

[33] M. J. Freedman and R. T. Morris. Tarzan: a peer-to-peer anonymizing network layer. In *ACM CCS*, 2002.

[34] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *EUROCRYPT*, 2014.

[35] S. Goel, M. Robson, M. Polte, and E. G. Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical report, Cornell University, 2003.

[36] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, 1987.

[37] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. F. Syverson. Users get routed: traffic correlation on tor by realistic adversaries. In *ACM CCS*, 2013.

[38] M. Keller, E. Orsini, and P. Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS*, 2016.

[39] L. Kissner, A. Oprea, M. K. Reiter, D. X. Song, and K. Yang. Private keyword-based push and pull with applications to anonymous communication. In *ACNS*, 2004.

[40] L. Kissner, A. Oprea, M. K. Reiter, D. X. Song, and K. Yang. Private keyword-based push and pull with applications to anonymous communication. In *ACNS*, 2004.

[41] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford. Atom: Horizontally scaling strong anonymity. In *SOSP*, 2017.

[42] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. *PoPETs*, 2016.

[43] A. Kwon, D. Lu, and S. Devadas. XRD: scalable messaging system with cryptographic privacy. *CoRR*, abs/1901.04368, 2019.

[44] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *OSDI*, 2018.

[45] D. Lazar, Y. Gilad, and N. Zeldovich. Yodel: strong metadata security for voice calls. In *SOSP*, 2019.

[46] D. Lazar and N. Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *OSDI*, 2016.

[47] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. K. Miller. Honeybadgermpc and asynchromix: Practical asynchronous MPC and its application to anonymous communication. In *ACM CCS*, 2019.

[48] N. Mathewson and R. Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *PETS*, pages 17–34, 2004.

[49] Z. Newman, S. Servan-Schreiber, and S. Devadas. Spectrum: High-bandwidth anonymous broadcast with malicious security. *IACR Cryptol. ePrint Arch.*, 2021:325, 2021.

[50] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, 1997.

[51] T. Perrin and M. Marlinspike. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016.
[52] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis. The loopix anonymity system. In *USENIX Security*, 2017.
[53] L. Sassaman, B. Cohen, and N. Mathewson. The pynchon gate: a secure method of pseudonymous mail retrieval. In *ACM WPES*, 2005.
[54] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.
[55] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
[56] N. Smart and P. Scholl. FHE-MPC notes. https://homes.esat.kuleuven.be/~nsmart/FHE-MPC/Lecture8.pdf, November 2011.
[57] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. Stadium: A distributed metadata-private messaging system. In *SOSP*, 2017.
[58] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP*, 2015.
[59] M. N. Wegman and L. Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.
[60] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *OSDI*, 2012.
[61] D. I. Wolinsky, E. Syta, and B. Ford. Hang with your buddies to resist intersection attacks. In *ACM CCS*, pages 1153–1166, 2013.
[62] D. Wu. Lecture notes on secret sharing and Beaver triples. https://crypto.stanford.edu/cs355/18sp/lec7.pdf, 2018.
[63] R. Zippel. Probabilistic algorithms for sparse polynomials. In E. W. Ng, editor, *Symbolic and Algebraic Computation, EUROSAM '79, An International Symposiumon Symbolic and Algebraic Computation, Marseille, France, June 1979, Proceedings*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer, 1979.

Appendix A
Security Proofs

We begin by stating our multiparty shuffle ideal functionality. The functionality describes the more general k-server setting where we wish to achieve security against $k-1$ servers. In Section IV, we considered the special case where $k=3$ and the adversary is allowed to control at most 1 server. In this case, if the adversary controls the third server, called P_3, the adversary is not given the opportunity to see T' before it is output.

Definition A.1 (Multiparty shuffle ideal functionality)**.** The functionality interacts with N' clients $C_1, ..., C_{N'}$, some of which are honest and the rest are controlled by the adversary, and with k servers $P_1, ..., P_k$, of which at most $k-1$ are are controlled by the adversary. At any point, any adversary-controlled server can send the functionality an abort message, which causes the functionality to abort.

The functionality initiates an empty table T and waits for messages from clients. Each client sends a message (given to it as an input), with an optional additional tag malformed included by adversary-controlled clients. The functionality drops any message that contains the malformed tag. For the remaining messages, it polls the adversary-controlled servers (without sending them the message), and any adversary server can send the functionality a drop request, which will cause the functionality to drop that message. Any message that is not malformed or dropped will be added to T.

Once T contains N messages (where N is a parameter that specifies the minimum anonymity set size), the functionality samples a random permutation $\pi : \mathbb{Z}_N \to \mathbb{Z}_N$ and applies it to T, computing $T' \leftarrow \pi(T)$. It then sends T' to the adversary, who can respond with continue or abort. Before responding with continue, the adversary may choose to modify any message $T'_{\pi(i)}$, where i is the index of a message initially sent by an adversary-controlled client. Let T'' be the resulting table with any modifications made by the adversary. When the adversary sends continue, the functionality outputs T''. □

We now prove the security of our three server shuffle and its multi-server generalization. We will begin by proving the security of the secret-shared shuffle against honest-but-curious adversaries.

Theorem A.2 (Three server semihonest shuffle)**.** *Assuming that the inputs to P_1 and P_2 include a random 2-party shuffle correlation (Definition IV.1), Protocol IV.2 achieves the secret-shared shuffle ideal functionality described in Section IV-A against an honest-but-curious adversary.*

Proof. To simulate the view of P_1, the simulation produces a random message $\boldsymbol{z}_2 \xleftarrow{R} G^N$ from P_2 and otherwise follows the protocol honestly using P_1's inputs $[\boldsymbol{x}]_1, \boldsymbol{a}'_2, \boldsymbol{b}_2$, and π_1, and setting P_1's output to $\boldsymbol{b}_2$. This simulation is distributed identically to the view of P_1 in the real protocol because there $\boldsymbol{z}_2 \leftarrow [\boldsymbol{x}]_2 - \boldsymbol{a}_1$, where $\boldsymbol{a}_1$ is uniformly random by the definition of a shuffle correlation. The rest of the messages are identical to the messages sent by P_1 in the real protocol.

To simulate P_2, the simulation must simulate the message $\boldsymbol{z}_1$ sent by P_1. The simulation is given P_2's output $[\boldsymbol{s}]_2$, which is computed as $[\boldsymbol{s}]_2 \leftarrow \pi_2(\boldsymbol{z}_1) + \boldsymbol{\Delta}_2$, and it is also given P_2's inputs, including π_2 and $\boldsymbol{\Delta}_2$. Thus it can solve for the value $\boldsymbol{z}_1 = \pi_2^{-1}([\boldsymbol{s}]_2 - \boldsymbol{\Delta}_2)$. The rest of the simulation follows the protocol honestly using P_2's inputs. This simulation is distributed identically to the view of P_2 in the real protocol because all the inputs, outputs, and messages are exactly equal to the values that would be sent and received in the real protocol. □

The proof above generalizes directly to the k-server shuffle in Section V, so we omit the proof of the following theorem.

Theorem A.3 (k-server semihonest shuffle)**.** *Assuming that the inputs to $P_1 ... P_k$ include a random k-party shuffle correlation (Definition V.1), the k-party shuffle in Section V achieves the secret-shared shuffle ideal functionality against an honest-but-curious adversary.*

We now turn to proving the security of our anonymous broadcast scheme, which has the secret-shared shuffle at its core but also provides security against malicious adversaries. Throughout the rest of this section, we will assume that Beaver triple-based share multiplication achieves an ideal functionality $\mathcal{F}_{\mathsf{mult}}$ whose messages can be simulated by the simulator $\mathcal{S}_{\mathsf{mult}}$. The functionality takes as inputs shares of two values and returns shares of their product.

Our proof will model hash functions as random oracles when servers send hashes of their messages to each other before revealing them. We will also model our one-time semantically secure encryption scheme (Enc, Dec) as being built around another (distinct) random oracle $H : \mathbb{Z}_p \to \mathbb{Z}_p^\ell$ and defined as $\mathsf{Enc}(k, m) = m + H(k)$. This is necessary because the adversary should learn nothing from ciphertexts that are revealed to it, but at the end of an honest execution of the protocol, all the encryption keys and messages will be revealed.

Note that our proof does not invoke the security of the first blind MAC verification. This is because the first blind MAC verification is included only to protect against a malicious client who wishes to send disruptive messages, but we already allow a malicious adversary who controls a server to disrupt the protocol. We do not formalize a weaker adversary who only controls malicious clients, but the proof that such an adversary could not disrupt the protocol would simply rely on the first MAC verification and the correctness of the scheme. That is, we would show that when the servers are honest a malformed message whose MAC does not verify would be discarded prior to the shuffle, and a MAC that does verify would not fail to verify after the shuffle. To be clear, our model *does* capture malicious clients colluding with the adversary to violate others' privacy.

Theorem A.4 (Three server shuffle)**.** *Assuming that G is a secure PRG and that the beaver triple MPC achieves $\mathcal{F}_{\mathsf{mult}}$, then the three server anonymous broadcast scheme presented in Section IV achieves the three party shuffle ideal functionality (Definition A.1) in the random oracle model.*

Proof (sketch). We first describe and prove the security of the simulation for the third server and then move on to the shuffling servers. Due to space limitations, we only sketch the behavior of the simulators and their corresponding indistinguishability proofs.

Simulating the third server. The view of the third server in the honest protocol only consists of the random seeds it receives from the shuffling servers, the messages it sends to the shuffling servers, and the final output of the protocol. The adversary also sees the messages sent by any clients it controls.

Messages sent by the shuffling servers to the third server are simulated by random strings.

For an adversary that follows the protocol exactly, the simulator never sends drop, malformed, or abort messages to the ideal functionality, makes no changes to the table T' when given the opportunity, and receives the ideal functionality's output T''.

The adversary may also, acting as a malicious client, send messages that do not have correct MACs, but then send modified beaver triples and shuffle correlations to force the messages it sent to be accepted by the blind MAC verification. It knows the indexes of its own messages pre- and post-shuffle, so there is no barrier to doing this. For the rest of this proof, we do not consider beaver triples or shuffle correlations modified for this purpose to be malformed.

If the adversary sends a message with a MAC that does not verify, or sends a message with a correct MAC but sends malformed beaver triples for that message's initial blind MAC verification, the simulator sends the message with the malformed tag to the ideal functionality, which results in that message being excluded from the output.

If the adversary incorrectly generates the beaver triples for an honest client's message, then the simulator sends the message drop for that message, which results in that message being excluded from the output.

If the adversary sends a malformed shuffle correlation or malformed beaver triple for the second blind MAC verification, the simulation sends abort to the ideal functionality, resulting in the output $\perp$. This completes the description of the simulation of the third server.

Security of third server simulation. Note that any adversary in the real protocol that does not produce any malformed beaver triples or share correlations produces a transcript that is already identically distributed to the simulated transcript. To handle the remaining cases, we require two hybrids.

Hyb_1 : We modify the real protocol so that the MAC key shares $[\boldsymbol{k}]$ used by the shuffling servers for non-adversary controlled client messages are truly random instead of the output of a PRG. The adversary never sees these seeds or key shares, so this change is

indistinguishable from the previous hybrid by the security of the PRG G.

Hyb_2 : In this hybrid we have our protocol abort if either of the following occur:

- The adversary produces a malformed beaver triple for the first blind MAC verification, but the message the beaver triple is used to verify is included in the protocol output.
- The adversary produces a malformed shuffle correlation or beaver triple for the second blind MAC verification, but the output of the protocol is not $\perp$.

This hybrid is indistinguishable from the preceding one because of the events above occur with negligible probability by the DeMillo-Lipton-Schwartz-Zippel lemma [28, 63, 54]. Observe that our MAC has the form $t = \Sigma_i k_i \cdot m_i$, which means that it is an evaluation of a multilinear polynomial determined by the message blocks m_i being MACed, and it is evaluated at a random point corresponding to the choice of keys k_i which are random and not visible to the adversary. Thus the probability of a tampered MAC being accepted is at most negligible.

The transcript generated in Hyb_2 is distributed identically to that of our simulation, completing the proof.

Simulating the shuffling servers.

Sending client messages. The first part of the adversary's view is the messages it sends on behalf of clients it controls. The simulator recovers the plaintext message in each message sent by the adversary and sends that as a message to the ideal functionality. If any of the adversary's messages do not contain MACs that verify correctly, the simulation sends malformed to the ideal functionality for those messages so that they are excluded from the output of the protocol.

If at any point the adversary modifies shares of one of the messages it sent to produce a share of a new message that also has a legitimate MAC, the simulator modifies the message accordingly when given the chance to modify T' in the ideal functionality. Throughout the proof, we do not consider this a deviation from the protocol.

First blind MAC verification. Next, the adversary sees the messages it receives from honest clients. These messages are simulated by random strings. During the initial blind MAC verification for each message, the adversary is sent a random string for each component of a beaver triple, and the simulator $\mathcal{S}_{\mathsf{mult}}$ is used to simulate the beaver multiplication for all honest messages. For adversary-controlled messages, the simulation performs the role of the other servers in the real blind MAC verification protocol and sends their messages.

For the remaining messages, if the adversary performs the initial blind MAC verification honestly, the simulated message revealing the other server's share of $[d_i]$ is the negation of the adversary's share. If the adversary deviates from the protocol in computing the MAC verification, then the simulated message is a random element of $\mathbb{Z}_p$. In this case, the simulator sends the message drop to the ideal functionality, and this message is dropped from the output of the functionality.

Shuffling. The view of the adversary during the shuffling stage of the protocol is simulated as described in Theorem A.2. The adversary's share of the shuffle correlations are simulated by random strings.

Second blind MAC verification. After the shuffle, the simulator examines the table T' sent to it by the ideal functionality and identifies the entries where the adversary-produced messages landed after the shuffle. When the servers reveal their shares of the ciphertexts c_i, the simulator sends shares for the adversary-controlled messages that reconstruct the ciphertexts originally sent by the adversary. For the remaining messages, it chooses random keys $\mathsf{ek_i}$ and produces ciphertexts $c_i \leftarrow \mathsf{Enc}(\mathsf{ek}_i, T'_i)$. It then creates shares $[c_i]$ and sends the appropriate share to the adversary.

For the second blind MAC verification, the simulator gives the adversary random strings to simulate beaver triples, uses $\mathcal{S}_{\mathsf{mult}}$ as before to simulate beaver multiplications for honest shares, and follows the protocol honestly for shares of adversary-controlled messages.

When hashing and revealing the result of the difference d, the simulator uses the negation of the adversary's share if the adversary performs the MAC honestly. If the adversary deviates from any part of the protocol where it operates on honest client-generated messages, the simulation uses a random element of $\mathbb{Z}_p$. If the adversary deviates from the protocol but only deviates in parts of the protocol that touch adversary-controlled messages, the simulator performs the protocol honestly for only the adversary-controlled rows of the table and sends the resulting output. In any case where the adversary deviates from the protocol, the simulation sends abort to the ideal functionality, resulting in the output $\perp$.

If at any point the adversary sends a message whose hash does not match the hash it previously sent (of purportedly the same message), the simulation sends abort to the ideal functionality, resulting in the output $\perp$.

Output. To simulate the honest shuffling server's share of the output, the simulator produces a database DB of messages of the form $(\mathsf{k}_i, t_i, c_i, \mathsf{ek}_i)$, where the rows corresponding to adversary-controlled messages are exactly the messages produced by the adversary

(including any modifications it may have made after initially sending them). The remaining rows have the ek_i, c_i from the second blind MAC verification, random k_i, and a MAC $t_i \leftarrow \mathsf{k}_{i,\ell+1} \cdot \mathsf{ek}_i + \Sigma_{j=1}^{\ell} \mathsf{k}_{i,j} \cdot c_{i,j}$. Let $\mathsf{DB}_{\mathsf{Adv}}$ be the table of shares held by the adversary at the beginning of the output phase. The simulator computes $\mathsf{DB}_{\mathsf{Hon}} \leftarrow \mathsf{DB} - \mathsf{DB}_{\mathsf{Adv}}$, and uses $\mathsf{DB}_{\mathsf{Hon}}$ as the message the honest shuffling server hashes and reveals in the output phase.

If the adversary aborts before revealing its final output, or if the adversary's final output contains messages whose MACs do not verify, the simulation sends abort to the ideal functionality, resulting in the output $\perp$.

Security of shuffling server simulation. The proof that a real execution of the protocol produces a transcript indistinguishable from the simulated protocol proceeds by a series of hybrids. We sketch the proof below.

Hyb_0 : The transcript of a real execution of the protocol with the adversary controlling one of the shuffling servers.

Hyb_1 : In this hybrid we replace the real beaver triple multiplication messages with simulated messages. This step is indistinguishable from the real protocol by the security of $\mathcal{S}_{\mathsf{mult}}$.

Hyb_2 : We modify the real protocol so that the MAC key shares $[\boldsymbol{k}]$ used by the honest shuffling server for non-adversary produced client messages are truly random instead of the output of a PRG. The adversary never sees these seeds or key shares, so this change is indistinguishable from the previous hybrid by the security of the PRG G. Note that as long as one of the shares of the MAC key is truly random, the whole key is random. Moreover, if the adversary changes its share of a key, the result is a different key that is still random and unknown to the adversary.

Hyb_3 : We modify the protocol to abort if there is ever a situation where the protocol reaches the end of its execution without aborting, but the adversary queries the random oracle used for encryption at the same ek_i as an honest client message before the output phase. This hybrid is indistinguishable from the previous hybrid because this event occurs with negligible probability. If the protocol reaches the end without aborting, then the adversary never sees any client-generated key ek_i before the output phase, so the probability that it could guess a random ek_i is negligible.

Note that this change implies that any ciphertext component from an honest user that the adversary sees in the course of the blind MAC verifications, whether or not it complies with the protocol, will appear uniformly random and reveal nothing about the underlying messages. An adversary who deviates from the protocol and gets $d \neq 0$ can use the output d to learn an inner product of a combination of MAC keys, ciphertexts, and encryption keys. But observe that the MAC keys and encryption keys are uniformly random, meaning any inner product including them will be random. If instead the inner product only includes ciphertext blocks, then d will still appear random because the encryption key remains hidden and the encryption masks the blocks of the message with the output of the random oracle on the encryption key.

Hyb_4 : We replace the real view of the adversary during the shuffling step with a simulated shuffle. This is indistinguishable from the previous hybrid by the proof of Theorem A.2.

Hyb_5 : We modify the protocol to abort if the adversary deviates from the protocol before it receives the final message of the shuffle and then the protocol accepts the second blind MAC verification. This hybrid is indistinguishable from the previous hybrid because this event occurs with negligible probability. Observe that our MAC has the form $t = \Sigma_i k_i \cdot m_i$, which means that it is an evaluation of a multilinear polynomial determined by the message blocks m_i being MACed, and it is evaluated at a random point corresponding to the choice of keys k_i which are random and not visible to the adversary. Thus the probability of a tampered MAC being accepted, i.e., the servers reveal shares that sum to zero, is at most negligible by the DeMillo-Lipton-Schwartz-Zippel lemma [28, 63, 54].

Moreover, as mentioned above, whenever $d \neq 0$, it appears random to the adversary. Thus, knowledge of its share of $[d]$ gives the adversary no information about the share held by the honest party, meaning the probability that the adversary can guess the other party's share is negligible. Since the adversary cannot guess the other party's share, the hash of that share sent during the MAC verification also appears random to the adversary.

Hyb_6 : We modify the protocol to abort if the adversary reveals a $\mathsf{DB}_{\mathsf{Adv}}$ that it has not previously sent as a query to the random oracle, but the protocol does not abort and output $\perp$. This hybrid is indistinguishable from the preceding one because this event occurs with at most negligible probability. Since the hash of $\mathsf{DB}_{\mathsf{Adv}}$ must match a previously sent value, the probability that the adversary finds a second preimage for that hash is negligible.

Hyb_7 : We modify the protocol to abort if the final recovered DB does not contain the set of messages sent by honest clients in T''. This hybrid is indistinguishable from the previous hybrid by the security of our MAC, which follows directly from the DeMillo-Lipton-Schwartz-Zippel

lemma [28, 63, 54], as discussed above. For an honest client's message to be missing, the adversary must have modified the message in that position in T'', but this would require forging a MAC. Although the MAC keys have by now been revealed, the message $\mathsf{DB}_{\mathsf{Adv}}$ was produced independently of the keys, as it was hashed before the adversary sees $\mathsf{DB}_{\mathsf{Hon}}$, and the MAC security therefore still applies.

The transcript produced in Hyb_7 is distributed identically to a simulated transcript, completing the proof. □

The security of the preprocessing phase of the k-server protocol follows directly from the security of the tools used to build it. We state the following theorem for the online portion of the protocol.

Theorem A.5 (k-server shuffle)**.** *Assuming that G is a secure PRG, that the beaver triple MPC achieves $\mathcal{F}_{\mathsf{mult}}$, and assuming that the offline phase of the protocol securely generates random beaver triples and k-party shuffle correlations, then the k-server anonymous broadcast scheme presented in Section IV achieves the multiparty shuffle ideal functionality (Definition A.1) in the random oracle model.*

We omit a full proof of this theorem as it is almost identical to the proof for the three server scheme with one malicious server. The main reason the 3-server protocol was secure against only 1 adversary was the role of the third server in producing shuffle correlations and beaver triples. With the role of the third server pushed to the preprocessing phase, the remaining parts of the protocol are the shuffle and integrity checks. Both of these had 1-out-of-2 security in the 3-server protocol because two servers did the actual shuffling, and both generalize to (k-1)-out-of-k security. Thus the primary difference between the protocols is that the shuffling protocol used is the k-party shuffle to account for the shift from two to k shuffling servers. The other components of the protocol are the same except that multiplication of secret-shared values, whose security is used as a black box in the proof, is computed via beaver multiplications among k parties instead of only two parties. The similarities and differences between the protocols are described in more detail in Section V.

APPENDIX B
ALTERNATIVE MAC SCHEMES

Instead of separately multiplying each block of the message by a separate one-time MAC key, we could instead make use of an additively homomorphic collision resistant hash function to add together hashes of all the message blocks and then only multiply that sum of hashes by a single MAC key. This would require only one beaver triple for each blind MAC check, would halve the length of each entry in $\boldsymbol{x}$, and would reduce the communication costs of the third server by a factor of ℓ.

Concretely, we could use a hash function $H : Z_p^\ell \rightarrow G$ for a group G parameterized by public parameters $g_1, ..., g_\ell$ whose discrete logs are unknown. The hash is defined as $H(m) \leftarrow \Pi_{i=1}^{\ell} g_i^{m_i}$. Since instantiating this hash function would be more expensive than the multiplications in $\mathbb{Z}_p$ that we currently employ, we see this as a computation/communication tradeoff, although using this hash is strictly superior asymptotically.

We could also directly use a standard Carter-Wegman MAC [59] for our MAC, reducing the length of entries in $\boldsymbol{x}$ by a factor of 2, but this would require $\log \ell$ rounds to compute the MAC using Beaver triples.

APPENDIX C
PERFORMANCE OPTIMIZATIONS

Free permutation. Our three server shuffle requires the shuffling servers to share a permutation π_{12} which they use to permute their shares before running the secret-shared shuffle. We design our implementation so that this permutation is effectively computed "for free" while the servers receive messages from clients.

In our implementation, clients send all their messages to the first shuffling server, with shares meant for the second server encrypted under that server's public key. The servers wait until they have a predetermined number of messages before beginning their shuffles. This simplifies the process of ordering received messages among the servers, but it also means that the first shuffling server can simply pick π_{12} before receiving client messages, insert the ith client message into position $\pi_{12}(i)$ in its message storage, and send the other shuffling servers both the messages intended for them and the index $\pi_{12}(i)$ where they should be inserted. Using this approach to receiving client messages means that servers do not actually need to evaluate the permutation π_{12} on the messages.

This optimization does not harm security because the first server only helps route encrypted and authenticated messages to the second server and doesn't learn anything from the messages it passes on. Moreover, the permutation π_{12} is needed for security only in the case where the third server is malicious. In this case, since we are assuming only one of three servers is malicious, we know the shuffling servers are honest and the data will correctly be permuted according to π_{12}.

Concurrency. We interleave different parts of our protocol to reduce the time the shuffling servers spend waiting for inputs from the third server in the three server protocol. The shuffling servers' first action is to generate the seeds needed for shuffling, send them to the third server, and then expand those seeds into the vectors $\boldsymbol{a}_i, \boldsymbol{b}_i$, and $\boldsymbol{a}'_i$, so the shuffling servers don't wait idle while the third server prepares beaver triples for the blind MAC verification. The clients also generate the shares $[a]$ and $[b]$ for their beaver triples on their own from PRG seeds and send these seeds to the third server, who uses them to complete the triple, thus reducing beaver triple communication costs by a factor of three. After MAC verification, the servers immediately commence the shuffle without waiting for the second server to receive $\boldsymbol{\Delta}_2$, only waiting if they reach the last step of the shuffle (where $\boldsymbol{\Delta}_2$ is used) before it arrives.

The most time-consuming portion of our protocol is the blind MAC verification, but fortunately this part of the protocol can be parallelized among as many cores or even separate machines as are available.

Skipping final MAC verification. The final (non-blind) MAC verification and message decryption is left to clients, with servers simply sending the message along with the keys. This does not affect security because by this point the servers have already blindly verified the MACs both before and after shuffling, and the clients will not see anything that an adversary-controlled shuffling server has not already seen.

APPENDIX D
COMPARISON TO OTHER MPC-BASED APPROACHES

We now compare the online phase of our k-server protocol with Asynchromix [47], Blinder-CPU, and Blinder-GPU [2], three other works which use MPC techniques for anonymous communication. Here we compare to the performance numbers reported in each paper, so the comparison serves only to give a sense of order of magnitude differences between the systems' performance and does not compare them directly on the same hardware or parameters. When comparing to the largest message batch sizes each work was evaluated on, Clarion's online phase outperforms Asynchromix (4K 32B messages per batch, 4 servers), Blinder-CPU (100K 160B messages per batch, 5 servers), and Blinder-GPU (1M 160B messages per batch, 5 servers), by $26\times, 19\times$, and $2\times$ respectively. The performance improvement over Blinder-GPU holds despite the fact that Blinder was run on GPUs which would cost about $40\times$ more to run for the same amount of time as our setup (based on current Google cloud pricing).

We caution that this strong performance comparison does not imply that Clarion strictly dominates Asynchromix and Blinder. First, the evaluation includes only the online time to run Clarion. This means our evaluation corresponds well to the setting where low-latency anonymous broadcast is needed for a short period, e.g., during a live event, that can be planned ahead of time, as Blinder cannot support an offline preprocessing phase in the same way we can. Moreover, Asynchromix and Blinder both provide additional resilience properties not present in Clarion– the robustness, fairness, and censorship-resistance properties mentioned in Section II – that ensure the system can continue operating if some fraction of servers go down due to malicious interference. On the other hand, they also only provide security against a $k/3$ and $k/4$ fraction of malicious servers, respectively, whereas Clarion provides security against $k-1$ malicious servers. Thus our evaluation demonstrates that there are settings where our approach is a better choice than Asynchromix or Blinder, but the best system for a given scenario varies.

The differences in security properties between Clarion and Asynchromix/Blinder all arise from a fundamental difference in approach. Messages in these other systems use threshold secret sharing of messages, so not all servers are required to participate in order for protocol outputs to be correctly computed. In order for Clarion to achieve these same security properties, we would need to replace our secret-shared shuffle with a *threshold* secret-shared shuffle. This motivates an interesting question for future work: can the shuffling techniques of Clarion, or of Chase et al. [12], be extended to work in a setting where messages are threshold secret shared, e.g., using Shamir secret sharing [55], instead of being additively shared?