TEEMS: A Trusted Execution Environment based Metadata-protected Messaging System

Sajin Sasy
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
sasy@cispa.de

Aaron Johnson
U.S. Naval Research Laboratory
Washington, D.C., U.S.A.
aaron.m.johnson213.civ@us.navy.mil

Ian Goldberg
University of Waterloo
Waterloo, ON, Canada
iang@uwaterloo.ca

Abstract

Ensuring privacy of online messaging remains a challenge. While the contents or data of online communications are often protected by end-to-end encryption, the *metadata* of communications are not. Metadata such as who is communicating with whom, how much, and how often, are leaked by popular messaging systems today.

In the last four decades we have witnessed a rich literature of designs towards metadata-protecting communications systems (MPCS). While recent MPCS works often target metadata-protected messaging systems, no existing construction simultaneously attains four desirable properties for messaging systems, namely (i) low latency, (ii) high throughput, (iii) horizontal scalability, and (iv) asynchronicity. Existing designs often capture disjoint subsets of these properties. For example, PIR-based approaches achieve low latency and asynchronicity but have low throughput and lack horizontal scalability, mixnet-based approaches achieve high throughput and horizontal scalability but lack asynchronicity, and approaches based on trusted execution environments (TEEs) achieve high throughput and asynchronicity but lack horizontal scalability.

In this work, we present TEEMS, the first MPCS designed for metadata-protected messaging that simultaneously achieves all four desirable properties. Our distributed TEE-based system uses an oblivious mailbox design to provide metadata-protected messaging. TEEMS presents novel oblivious routing protocols that adapt prior work on oblivious distributed sorting. Moreover, we introduce the notion of ID and token channels to circumvent shortcomings of prior designs. We empirically demonstrate TEEMS' ability to support 2^{20} clients engaged in metadata-protected conversations in under 1 s, with 205 cores, achieving an 18× improvement over prior work for latency and throughput, while supporting significantly better scalability and asynchronicity properties.

Keywords

anonymous communications, metadata-protecting communication systems, oblivious algorithms

1 Introduction

In light of widespread digital surveillance [28], the Internet has adopted end-to-end encryption (E2EE) as a standard for all forms of online communications. Today, popular messengers like Signal or WhatsApp that are used by millions to billions of individuals, provide E2EE for all users' conversations by default. However, while E2EE protects the contents of communications, it offers no protections over the *metadata of communications*: metadata such as who is communicating with whom, when, and how much.

Metadata can have dire consequences [60]. The lack of metadata protections for our online communications curbs whistleblowing. Free and democratic societies depend on whistleblowers shedding light on corruption and misdeeds of their governments or employers, but whistleblowers are often rewarded with disheartening outcomes [10, 31, 59]. The same is true of individuals in oppressive regimes, whose lives are endangered by their mere sexual orientation or political stance [24, 64].

Research in this direction dates back decades [13, 14, 20, 53, 56, 61, 63, 65], and has received significant attention in the last several years alone [1, 15, 19, 21, 27, 40–42, 44, 51, 52, 73, 74, 77]. Yet a practical metadata-protecting communication system (MPCS) that protects against strong (global) adversaries still seems far from reality; systems like Tor [25] defend only against local network adversaries, and so cannot provide the protection we seek. A recent systematization of knowledge [66] provides a comprehensive view of the literature so far. One of their key observations is that none of the existing designs towards MPCS achieve three fundamental properties expected from a messaging system simultaneously, namely, (i) *low latency*, (ii) *horizontal scalability*, and (iii) *asynchronicity*. Additionally, any messaging system should be able to support a *high message throughput*; that is, it should be able to provide its low latency to a large number of messages simultaneously. In this work, we remedy the state of affairs by presenting a novel MPCS design TEEMS (Trusted Execution Environment based Metadata-protected Messaging System) with the objective of attaining these four fundamental properties together.

As suggested by its name, TEEMS leverages Trusted Execution Environments (TEEs) to achieve metadata-protected messaging. Existing MPCS designs predominantly reside in the distributed-trust model under non-collusion assumptions of servers; shifting the threat model to a TEE-based one has two immediate upshots that lead to our efficient system. First, all the TEE-supported servers can be run by a single service provider in one data center, resulting in low-latency and high-bandwidth links between servers of the system, while protecting metadata from even this service provider. Second, such a system does not face the deployment challenge of finding many cooperating but non-colluding entities. The challenge however with the TEE setting is to be secure against malicious service providers that may try to infer conversational metadata by exploiting TEE side channels. Hence, TEEMS focuses on the

Proceedings on Privacy Enhancing Technologies 2025(4), 56–75

https://doi.org/10.56553/popets-2025-0119

design of new oblivious protocols that compose together into a novel metadata-protected messaging system.

In the literature, two distinct classes of MPCS systems that hold promise are mixnet systems [37, 44, 73] and Reverse PIR (RPIR) systems [27, 74]. However, they have contrasting merits and shortcomings. Mixnet systems enable high throughput and horizontal scalability but lack support for asynchrony and incur higher latency overheads than the RPIR systems. On the other hand, RPIR-based mailbox systems provide asynchrony and low latency for message delivery, but they have poor horizontal scalability and throughput.

Another design approach, demonstrated in the Sparta system [29] uses TEEs to obliviously store and fetch messages. Sparta has several variants, but none is horizontally scalable, and none simultaneously provides low latency and high throughput. Sparta has several other security and performance deficiencies, as well (see Appendix A).

Our key observation is that mailbox-based MPCS can be horizontally scaled with distributed oblivious routing protocols. TEEMS, hence, uses a fully distributed *oblivious mailbox* architecture in which clients drop off messages, the system obliviously routes and stores them, and then clients pick up their waiting messages. Asynchrony in our design is facilitated by our novel *oblivious route-and-pad protocols*, TokenColumnRoute and IDColumnRoute. We note that while we construct such protocols using TEEs, our architecture and protocols could also be instantiated (less efficiently) through multi-party computation or homomorphic encryption.

While TEEMS supports asynchrony, internally the system operates in epochs. In each epoch, our protocols route messages to their correct destinations while simultaneously injecting padding messages such that each user in the system receives the same (tunable system parameter) number of messages at the end of each epoch. We present oblivious designs for our protocols such that even an adversary that controls the servers' operating systems, and observes all the communications among the servers and between servers and clients, cannot infer any information other than the set of users that interacted with the system in a given epoch.

Our contributions are:

1. We present the oblivious mailbox system TEEMS, which is the first MPCS to simultaneously achieve low latency, high throughput, horizontal scalability, and asynchronicity.

2. Our design introduces the notion of *ID* and *token channels*, and we present fully oblivious route-and-pad protocols for each. A token channel provides guaranteed delivery for messages from a limited number of "friends", while the ID channel has no limit on the number of potential senders but does not guarantee delivery. We further show how to bootstrap the token channel from the ID channel. This approach circumvents the rigid dialing protocols in the literature and lets users have multiple parallel conversations.

3. We implement a prototype of our design and provide benchmarks to demonstrate practicality.

2 Background

2.1 Trusted Execution Environments

Beginning with Intel TXT [33], TEEs have received significant academic and industry attention. TEEs enable execution of arbitrary programs on a remote adversarial server while ensuring confidentiality over the state and data of such programs, and integrity of the program being executed. Today, all major hardware manufacturers support different flavours of such TEEs, for instance Intel's SGX [3] and AMD's SEV [38].

Unfortunately, this grandiose promise of TEEs has some pitfalls in practice. Side-channel attacks plague even the most recent iteration of such TEEs. Broadly, there are two classes of side channels. Microarchitectural side channels such as speculative execution [16, 75] or voltage-scaling [50, 55] ones; these violate the TEE (and indeed the CPU) threat model, and consequently receive patches from hardware manufacturers [34–36]. Software side-channels like those based on memory access patterns [12, 47, 71, 76] and control flow [48, 54, 78] remain out of scope of the TEE threat model, and so are left to the authors of the programs meant to run inside the TEEs to ameliorate themselves. Consequently, leveraging TEEs while ensuring security and confidentiality of data requires the use of "fully oblivious" algorithms [58, 67]. Fully oblivious algorithms have control flow and memory access patterns independent of the private data they compute upon, and hence do not induce software side channels. TEEs can facilitate efficient privacy-preserving solutions (despite the overheads of running fully oblivious counterparts of the underlying programs) and have consequently earned several real-world deployments [11, 72, 79].

2.2 MPCS

Several MPCS have been proposed in the literature with varying degrees of privacy and functionality. In this work, we use terminology presented by a recent taxonomy of existing MPCS [66] to make precise the privacy and functionality goals of our system. Our work presents a novel design for a Communication Unobservable System (CUS). A CUS protects even the existence of a message from a global adversary. While the number of online users is observable, whether they are even communicating or not is protected from such an adversary. Moreover, the receiver anonymity set for any message in TEEMS is the set of *all users in the system whether or not they are currently online.*

In Section 1 we listed the fundamental desirable properties. In Appendix B we expound on these properties and an often overlooked aspect of MPCS: the setup involved in bootstrapping metadata-protected communications.

Among the existing MPCS designs, Karaoke [44], Groove [8], Boomerang [37], Sabre [74], and Sparta [29] are the closest to our work. **Karaoke** is a recent CUS from the line of MPCS constructions based on mixnets and differential privacy [73, 77]. In Karaoke all clients send and receive exactly one message in a round. Servers in Karaoke are distributed into input and output mixnet chains. Users engaged in a conversation establish a dead-drop address via a separate "dialing" protocol. The dead-drop address space is distributed evenly across all the output chains. Conversing users submit their messages destined for the same output chain, through any input chain. Each server mixes all messages before forwarding them to the next server on the chain. The last server in an output chain swaps messages that have the same dead-drop address, and then all messages are sent back in the reverse direction. Each server in the system also inserts noise messages to provide differential privacy guarantees on all the user observable variables of the system (such as the number of messages going from an input chain to an

output chain). While Karaoke's design supports high throughput and horizontal scalability, it does not support asynchronous clients nor low latency (see Appendix B).

Groove extends such differentially private MPCS with parallel conversations and partial support for asynchrony. In Groove clients perform an expensive circuit setup phase for an epoch (one day) in exchange for faster communication rounds. Setting up multiple circuits facilitates parallel conversations, however Groove requires that conversing clients already have a shared secret. Additionally, the setup phase fixes the clients' correspondents for the epoch. This assumption and model effectively allows Groove to side-step dialing protocols. Nonetheless adding new friends to setup circuits with still requires a dialing protocol to bootstrap a shared secret. Moreover, Groove only allows for partial asynchrony. It enables users to receive messages for a fixed number of future rounds of communications when they go offline. Clients precompute circuits for all the epochs they would be offline and provide their untrusted service provider with the circuit creation bundles. TEEMS on the other hand can enable clients to receive messages *for however long they may go offline without any additional client compute overheads*. We do have a limitation on the maximum number of real messages such users receive when offline, but this is tunable and can be handled *independent* of the actual communication protocol itself (see Section 4.7).

Boomerang leverages TEEs to present a system that has security under hardware trust without having to use differentially private guarantees the way Karaoke and its predecessors do. Each input and output mixnet chain is replaced by a single TEE-aided server that obliviously shuffles all the messages together without leaking any metadata information to the adversarial host via side channels. The output servers also run an oblivious algorithm to perform the matching and swapping of messages destined for the same dead-drop. In Boomerang the number of packets that each input node distributes to the output nodes is fixed by an upper bound that ensures minimal padding overheads while introducing an overflow failure probability. Boomerang hence does not require the differential privacy used by Karaoke and its predecessors, in exchange for trusted hardware and oblivious algorithms. Nonetheless, it still cannot support asynchronous clients nor low latency.

Sabre is the current state-of-the-art iteration of a Sender Unlinkable Messaging System (SUMS) [66] from the Reverse PIR family of MPCS [19, 27]. In a SUMS, all users have a dedicated mailbox. In Sabre, mailboxes are distributed over two non-colluding servers that facilitate a disruption-resistant and write-private database. Senders drop their message into their intended receiver's mailbox using PIR in reverse (private writing instead of private reading), preventing an adversary from linking messages with their destination mailboxes. Clients periodically retrieve messages directly from their mailbox, without having to hide which mailbox belongs to them. Sabre (and its predecessors) provide asynchrony and low latency, but cannot support horizontal scalability or high throughput.

Sparta [29] is a Receiver Unlinkable Messaging System (RUMS) based on TEEs. Sparta provides three designs: Sparta-LL is low latency, Sparta-SB is sorting-based for high throughput, and Sparta-D is distributed for more scalability. Running on a single server, Sparta-LL maintains each user's mailbox as a linked list in a single ORAM, while Sparta-SB uses a single sorted message vector for all mailboxes. Sparta-D assigns messages pseudorandomly to several sorted sub-vectors, which can be on different servers, and a coordinating server reads and writes messages to and from those sub-vectors in padded batches. Like TEEMS, Sparta is an oblivious mailbox design and thus provides asynchronous communication. However, all its designs fail to provide scalability, as even Sparta-D features a single server that processes all user queries and responses.

In comparison, TEEMS simultaneously achieves all four desirable MPCS properties. To do so, it includes several technical innovations over these prior systems. First, it introduces the notions of token and ID channels, where the token channel guarantees delivery to a limited number of user-designated friends, while only the ID of the recipient is needed to send on the ID channel, but delivery is not guaranteed. The ID channel in particular provides functionality not existing in prior work, enabling a user to receive messages without prior coordination beyond sharing their ID (a static string). Second, it allows users to manage friends using the ID channel, thereby boostrapping the token channel and avoiding the cumbersome dialing protocols of prior systems. Third, TEEMS includes novel oblivious algorithms to route messages from a set of servers that receive messages to a set that stores mailboxes, which provides the horizontal scalability that several prior works lack.

3 Overview

We introduce the components and parameters of TEEMS. The parameters and other notation are summarized in Appendix C.

3.1 System Design and Goals

TEEMS uses an oblivious mailbox MPCS design. That is, senders submit messages, which the system stores and then delivers to receivers while remaining oblivious to which users are communicating. Each user has a *mailbox* that will store messages until the user connects and retrieves them. TEEMS does not hide which mailbox belongs to which user, but it does hide if the mailbox receives or contains messages at all, as well as hiding the senders of any such messages. The system operates in synchronous *epochs*. However, the system provides *asynchronous* communication to users. Users may, at any time, retrieve their mailbox and submit messages to be stored in the receiver's mailbox by the next epoch.

TEEMS uses TEEs in its servers for metadata protection. The main process in each server runs in a TEE enclave, with any necessary calls outside the enclave (e.g., to send or receive messages) operating only on non-secret or encrypted data. Remote attestation to the server enclave processes is performed by other servers and clients, but it is done only once to exchange keys that are used from then on (see Section 4.2).

In contrast to distributed-trust based MPCS models, the use of TEEs enables TEEMS to run its servers in the same data center by the same operator while still achieving its security goals. Indeed, such a setup minimizes the number of parties that are in a position to perform the physical attacks that TEEs are vulnerable to. We thus assume such a setup, and therefore that the servers have high-bandwidth, low-latency connections among them.

All TEEs in the system share a symmetric key K that is generated by them and securely stored. This key enables bootstrapping of secure forward-secret enclave-to-enclave communication channels.

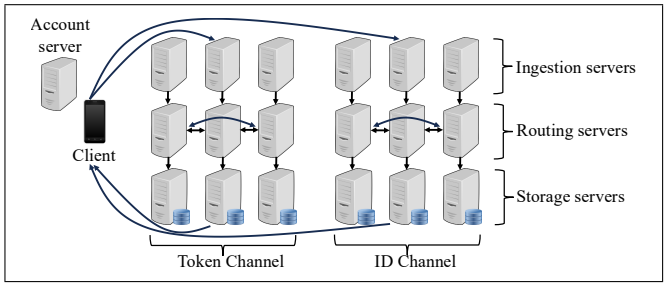

Figure 1: An example of the TEEMS architecture. Arrows indicate how messages are sent. Storage servers are shown holding mailboxes with stored messages. Just one client is shown, and it connects to the ingestion server and storage server that it is assigned to in each channel.

It is also used during account creation and other tasks as outlined below. Each server generates a public-private key pair, which is used to secure all communication among servers and between clients and servers. Details of key establishment appear in Section 4.2.

Threat Model. In our design, we consider an adversary that has OS-level control of the servers, controls network communication with clients and among servers, and controls some of its own clients. Following the TEE threat model, the adversary can perform software attacks on the TEE machines, including the observation of side channels intentionally left open by the TEE, such as timing and memory-access locations. The adversary can read, modify, delay, or drop packets arbitrarily. Our design (and implementation) algorithmically protects against all such adversaries. While the adversary may control some clients, the adversary must be a third party to the protected communication; i.e., the adversary cannot control the sender or the receiver for our security guarantees to apply.

Like other TEE-based systems [11, 23, 29, 37, 70], denial-of-service attacks are out of scope for this work. Microarchitectural attacks like those based on power consumption [17, 50, 55] or speculative execution [16, 75] are orthogonal to our work, and are mitigated by hardware manufacturers directly [34–36]. Physical attacks, such as invasively probing TEE chips to extract keys, are also outside of the threat model.

Goals. A main security goal of TEEMS is unobservable communication. The adversary should not be able to detect if a message is sent at all, let alone determine the sender, receiver, or message contents. The system should also ensure integrity of the delivered message, and a receiver should be able to determine who sent the received message and when.

The TEEMS performance and functionality goals are the four key MPCS properties outlined earlier. It should provide low latency to individual messages sufficient to support interactive conversations. It should provide high throughput by supporting that low latency for many simultaneous messages. It should be horizontally scalable to grow to large numbers of users by adding servers. Finally, it should support asynchronous communication where the senders can send messages when they wish, and receivers can receive them when they are online, without coordination between the two.

3.2 Token and ID Channels

The high-level system architecture is illustrated in Figure 1. In TEEMS, messages can be sent either through the *token channel* or *ID channel*. Each channel type is composed of a separate set of servers, and clients communicate with each set to receive messages on both channels. The token channel guarantees delivery within one epoch (subject to sufficient room in the receiver's mailbox), but its senders must be selected in advance to share the limited system resources. The ID channel does not guarantee delivery, but no prior coordination is required to use it. The ID channel thus can be used to send someone a message knowing only their user ID. Because a large number of users may try to send an ID-channel message to the same receiver at the same time, however, the ID channel may drop some messages to respect the resource limits imposed by the privacy requirement to hide how many messages a user receives.

For token-channel communications, each user u can select up to f other users as their *friends*. Each of those friends will be allowed to simultaneously send a token-channel message to u in the same epoch. Friend designations are persistent across epochs and need no further action from the user until they desire to change them. Friends are also symmetric in that if u is a friend of v, then v is a friend of u. The symmetric relationship will facilitate updating friend relations, as it allows such updates to be sent through the token channel. A client must obtain *tokens* before sending token-channel messages to authorize the use of the token channel for the message receivers. A token is a credential, valid for the current epoch, showing that a given sender and receiver are friends.

In contrast, no such authorization is required to send ID-channel messages. Each user is limited to sending at most one and receiving at most b ID-channel messages in a single epoch. An ID-channel message may be dropped, but, if not, it will be delivered within that epoch. Message drops in the ID-channel are similar to dialing failures in prior iterations of MPCS [2, 26]; in such systems when many users dial the same individual, dialing completes with one of these users at random. However, ID-channel messages have a priority attached to them by the system, based on how frequently a user has sent other ID-channel messages, and the system will prefer to drop lower-priority messages when dropping is necessary. Users have complete control over who can send them messages on the ID channel (see Section 4.1); i.e., malicious users cannot impede an honest client's ID channel.

The ID channel is also used to set up and modify friend relationships. Friend requests are sent through the ID channel, which can then be confirmed through the token channel. Friend revocations are sent through the token channel, which is possible because the friend relation is symmetric.

This two-channel design provides several improvements over existing MPCS. The ID channel circumvents the requirement of a dialing protocol that many MPCS schemes require. Any MPCS that require dialing incurs two limitations: (i) Dialing protocols add latency and rounds of overhead *before* real messages can be sent. (ii) Dialing only allows for online metadata protections; once clients go offline, they can no longer participate in a dialing protocol, and consequently they cannot receive messages asynchronously. To distinguish the ID channel from dialing protocols, observe that (i) the ID channel is a *self-contained* medium for metadata-protected

communications, while dialing protocols bootstrap a shared-secret for future conversation rounds. (ii) Clients can receive messages *asynchronously* via the ID channel; dialing protocols require both parties to be online. (iii) the ID channel can be run in tandem with (or completely independent of) the token channel. Dialing, however, always has to precede conversation protocols.

The token channel in TEEMS requires a one-time setup to establish a friendship, which can be accomplished asynchronously, and then afterwards requires no coordination at all. We note that either the token channel or the ID channel could be used by an MPCS in isolation. The ID channel is completely independent of the token channel, but the token channel would need some other method to set up friendships.

3.3 Clients

Clients in TEEMS do not have TEEs. Clients follow a fixed connection schedule to send and receive messages. The schedule can be randomized and may vary across clients, but it should only depend on information that the adversary already knows (or is allowed to know). In particular, it should not depend on if any message was actually sent or received during prior connections. Clients will send and receive *dummy* messages during a connection to hide how many real messages were sent and received. To provide low latency when the client is messaging, the connection schedule should have little time between connections. However, if a client is unavailable at a scheduled connection time, such as due to their device being off or disconnected from the network, this deviation from the schedule need not affect the security guarantees, as long as the unavailability is not related to their prior messaging behavior.

At each scheduled connection, the client retrieves its stored messages from each channel and sends any waiting messages from the user to each channel. Every epoch, each online client will receive exactly f messages on the token channel and b on the ID channel, and it will send exactly one message over each channel; any of those messages may be dummies.

3.4 Servers

There are four types of servers in TEEMS:

Account server: An account server handles the creation and update of user accounts. It takes requests for new accounts from users and assigns them ingestion and storage servers.

Ingestion server: Each client in TEEMS is assigned an ingestion server, and this assignment is not secret. An ingestion server collects messages from its clients, and at the start of each epoch, forwards their messages to the routing servers.

Routing server: The routing servers jointly perform an oblivious route and pad protocol. At the end of the protocol, the routing servers end up with the exact maximum number of messages (f and b respectively) designated for each client's token and ID mailbox, and forward these messages to the storage servers.

Storage server: Each mailbox in the system is hosted on a storage server, with each storage server having about the same number of mailboxes. The system does not hide the mapping between clients and their mailbox, including its storage server and its location on that server. Online clients receive all the messages from their mailbox every epoch, while offline clients will have their messages accumulate (up to a maximum size), and will receive their mailbox the next time they connect.

The servers are logical entities, and physical machines could run multiple servers; e.g., in an initial deployment, one physical machine could run all the servers, and servers could be migrated to multiple machines as the user base grows. Running multiple servers of the same or different type on a many-core machine is beneficial, as each server uses only a small number of cores (see Section 6).

4 Design Details

We assume that the underlying network provides asynchronous, message-oriented communication channels between each pair of servers and between each client-server pair. Let $\mathcal{I}$, $\mathcal{R}$, and $\mathcal{S}$ be the sets of ingestion servers, routing servers, and storage servers, respectively, for a given channel. When necessary, we will use superscripts to indicate the channel (e.g., $\mathcal{I}^{\text{tkn}}$ and $\mathcal{I}^{\text{id}}$).

Let $r = |\mathcal{R}|$ be the number of routing servers, and let $s = |\mathcal{S}|$ be the number of storage servers. Let the routing servers be ordered: $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$. For a given user u, let I_u and S_u be the ingestion server and storage server assigned to u, respectively. The routing servers each have a set of ingestion servers that forward messages to them and a set of storage servers that they forward messages to. No ingestion server forwards to more than one routing server, and no storage server is forwarded to by more than one routing server. For a given routing server R, let $\mathcal{I}_R$ be the set of ingestion servers that forward messages to R, and let $\mathcal{S}_R$ be the set of storage servers that R forwards messages to. We assume for simplicity that the same number of ingestion servers forwards to each routing server and that each routing server forwards to the same number of storage servers. Let the storage servers be ordered, $\mathcal{S} = \{S_1, S_2, \ldots, S_s\}$, such that, for all $S_h \in \mathcal{S}_{R_j}$ and $S_i \in \mathcal{S}_{R_k}$, if $j < k$ then $h < i$.

4.1 User Identifiers

Each TEEMS client has a *user identifier* (user ID). The user identifier is a value composed of the index of the storage server (under a canonical ordering) followed by a unique number for that server. Both of those components are of fixed sizes (e.g., 10 bits for the server index followed by 22 bits for the per-server user number). The largest value that can be represented in that fixed size is reserved for internal use. Each identifier is thus unique. Its structure also facilitates routing because sorting messages by the receiver's user ID simultaneously sorts them by the index of the receiver's storage server and by the index of the routing server that forwards to that storage server. The user identifier may be guessable, and so, to give the user some control over who can send them messages on the ID channel, the system maintains a *long* form of the user ID that also includes the MAC of the user ID under a key derived from the shared key K. The users must therefore distribute their long user IDs in order to receive ID-channel messages, which they may accomplish narrowly (e.g., only to select contacts) or broadly (e.g., publicly via a website). To prevent ambiguity, we will always specify when the long form of the user ID is the one being used.

4.2 Account Server

The account server coordinates key establishment for the system and facilitates user-account management. It maintains the following information about the other servers in the system: (1) their addresses; (2) their public keys; and (3) a count of the number of users assigned to each ingestion server and storage server. It also creates the system-wide key K. The other types of servers in the system use a one-time two-way remote attestation to securely connect to the account server and obtain K. That key is then used to secure a separate connection on which the server's public key is shared with the account server and the public keys of the other servers are obtained from the account server. Clients use a one-time remote attestation with the account server to securely obtain its public key, which they then use to secure communications with it.

The account server stores an account for each user in the system. Each account consists of the following: (1) the long user ID; (2) authenticating information, such as password hash or a public verification key; (3) the assigned ingestion server; and (4) the assigned storage server. This data is expected to be relatively small (less than 100 bytes) and infrequently accessed. The number of servers is expected to be many thousands of times less than number of users. We expect that a single account server can accommodate billions of users, although the account database could also simply be split across servers. It is not necessary to hide which account is being accessed because only the owner accesses it.

The account server A supports the queries Create, Update, and Delete. These queries are made by a user to create or modify their account. In Create, the client provides its authenticating data α. A picks the least-loaded ingestion servers I^{tkn} and I^{id} and storage servers S^{tkn} and S^{id}. It contacts S^{tkn} to obtain an available user ID u (users are assigned the same IDs in each channel). Recall that the high bits of u will be the index of S^{tkn}. A sends (u, α) to I^{tkn}, I^{id}, S^{tkn}, and S^{id}, which they all store. A computes a MAC tag t on u under the shared key K, and sends to the client: (u, t), the assigned servers $(I^{\text{tkn}}, I^{\text{id}}, S^{\text{tkn}}, S^{\text{id}})$, and those servers' public keys. In Update, user u submits new authenticating data α, which A updates locally and forwards to the assigned servers of u. In Delete, A deletes the account of the user u and forwards the query to the assigned servers of u for them to delete. These queries are not performed obliviously; the system does not hide if or how accounts are created or modified.

4.3 Messages

A message is a data structure containing metadata of the message as well as the content. Let $\mathcal{M}_{\text{src}}$ be the user ID of the source of a message, $\mathcal{M}_{\text{rec}}$ be the user ID of the receiver, $\mathcal{M}_{\text{pri}}$ be the message priority, and $\mathcal{M}_{\text{data}}$ be the content. A token-channel message is $\mathcal{M} = (\mathcal{M}_{\text{rec}}, \mathcal{M}_{\text{src}}, \mathcal{M}_{\text{data}})$, and a ID-channel message is $\mathcal{M} = (\mathcal{M}_{\text{rec}}, \mathcal{M}_{\text{pri}}, \mathcal{M}_{\text{src}}, \mathcal{M}_{\text{data}})$. The entire message has a fixed total length of ℓ (for both the token and ID channels) to hide the true length of $\mathcal{M}_{\text{data}}$. Note that $\mathcal{M}$ may be a *dummy* message, indicated by $\mathcal{M}_{\text{src}} = \perp$, or a *real* message, in which $\mathcal{M}_{\text{src}}$ is a user ID. Dummy messages are sent by clients and between servers as needed to hide the number of real messages being sent.

(1) Receive token-channel messages and tokens from S_u^{tkn}
(2) Receive ID-channel messages from S_u^{id}
(3) Send $\mathcal{M}_{\text{tkn}}$ with related token, or a dummy message and token if no such pair exists, to I_u^{tkn}
(4) Using long IDs for receivers, send $\mathcal{M}_{\text{id}}$, or dummy message if $\mathcal{M}_{\text{id}}$ is empty, to I_u^{id}

Figure 2: ClientInteract($\mathcal{M}_{\text{tkn}}, \mathcal{M}_{\text{id}}, u$): Interaction procedure executed by user u.

4.4 Clients

Clients send and receive messages according to an interaction schedule *when they are online*. The schedule may be unique to the client and need not be coordinated with any other client. However, the schedule must be independent of the actual client messaging behavior; that is, it should not depend on if a client has a real message to send or receive. For example, a client may choose to minimize latency by interacting every epoch they are awake and online, or they may trade off latency for communication cost by interacting less frequently (only every minute, hour, or day). Importantly, none of these choices affect the *security* of TEEMS. In fact, later in Section 5.2, we allow the adversary to *select* honest clients' interaction schedules when we prove TEEMS is communication unobservable. Clients in TEEMS can go offline arbitrarily, such as when their device is turned off or loses network connectivity.

During each interaction, the client sends and receives messages via both the ID and token channels. Some ordering among those actions is needed, for example retrieving a token before sending a token-channel message. Figure 2 shows the ClientInteract algorithm that a client uses during an interaction. Note that Steps 1 and 2 can be performed in parallel, as can Steps 3 and 4.

4.5 Ingestion

Ingestion servers in both channels first authenticate a connecting client using the stored user IDs and authentication data. Each epoch, the ingestion server receives one message from the user, which it validates by checking that the source user ID is correct and then stores until the next epoch starts. When the next epoch starts, each ingestion server forwards its stored messages to its routing server.

In the token channel, the client always sends one message and token; if the client has none, it sends a dummy message and token. The ingestion server validates the token by checking that it contains the same receiver ID as the message, that it is valid for the upcoming epoch, and that it has a valid MAC tag. Any invalid messages are replaced with dummy messages. The tokens are discarded and are not forwarded further.

In the ID channel, the client always sends one message, using a dummy message if necessary. The client includes the long user ID in the message. The ingestion server validates the MAC tag in the received long user IDs, turning the message into a dummy message if the check fails. Then it discards that MAC tag from the message, as it is no longer needed. An ID-channel ingestion server also maintains a record of the time since a real message was last sent by each of its assigned users to each other user within a short recent period (e.g., the last 5 minutes), which is used as the priority to determine which messages to drop, if necessary. That record

Round 1 (Scatter) After receiving the messages for this epoch from the ingestion servers, obliviously sort the messages by the user IDs of the receivers and send the messages in round-robin order to the routing servers. That is, given sorted messages M, send message $M[x]$ to R_j, where $j = (x \bmod r) + 1$.
Round 2 (Distribute) Obliviously append dummy messages such that the number of messages destined for each storage server is $y = \lfloor f\lceil n/s\rceil/r\rfloor + r$. Obliviously shuffle the messages, and send them to the routing server that forwards to the receiver's storage server.
Round 3 (Forward) Obliviously mark for deletion $yr - f\lceil n/s\rceil$ of the dummy messages for each storage server. Obliviously shuffle the messages, and then forward them to their receivers' storage servers.

Figure 3: TokenColumnRoute: Token-channel routing

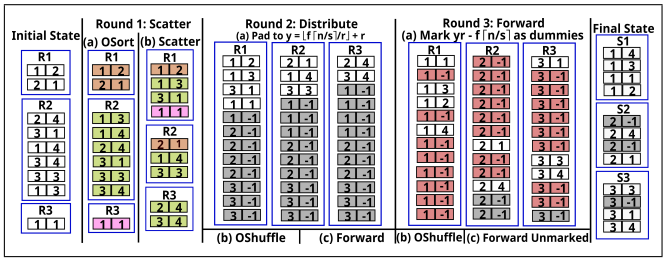

Figure 4: TokenColumnRoute example with $n = 12$, $r = 3$, $s = 3$. Message labels are the recipient user ID (storage server index | per-server user number). Message colors in Round 1 simply highlight how messages get scattered. Grey messages indicate dummies, and red ones indicate messages that were marked as dummies and do not get forwarded in round 3. $y = \lfloor f\lceil n/s\rceil/r\rfloor + r = 4$. The additional r messages dominates y in the figure, but in practice $n \gg r$.

is updated obliviously when a message is received from a client. When a message is received, the ingestion server copies into the $\mathcal{M}_{\text{pri}}$ component of the message how many epochs ago the user last sent an ID-channel message to the recipient. This value gives higher priority to messages from users that have not recently sent to that recipient via the ID channel.

4.6 Routing

We use similar routing algorithms for the ID and token channels. Our routing algorithms are derived from columnsort [49], a distributed sorting algorithm. Columnsort has been used *as is* in other TEE-based distributed systems [79] for oblivious distributed sorting. We observe that routing is a slightly easier problem than sorting because the values of the sorting "keys" (i.e., the server indices) directly imply the server the item should end up on. Hence, while columnsort could serve as an oblivious routing protocol, our oblivious routing protocols reduce the communication (by one round) and computation (from four oblivious sorts to one oblivious sort and two oblivious shuffles) overheads in comparison.

In the ID channel, however, we must also first determine if any receiver has been sent more messages than can be delivered in the epoch and, if so, drop the excess messages with the lowest priorities. We accomplish this with a distributed sort as well, this time using both the receiver user ID and the message priority as keys. Our protocols, however, encounter an additional challenge over columnsort; we need to add and manipulate dummy messages without revealing to the OS-level adversary which are dummies.

Let n be the total number of users. A description of the token-channel routing algorithm, TokenColumnRoute, is given in Figure 3. The ID-channel routing algorithm, IDColumnRoute, is described in Figure 5. The descriptions give the actions of a routing server by *round*, each of which consists of receiving messages, performing computation, and then sending messages. Additional details of these algorithms appear in Appendix D.

4.6.1 TokenColumnRoute. In Round 1 of the token-channel routing algorithm (Figure 3), routing server R_i will "scatter" the messages it receives from its ingestion servers $\mathcal{I}_{R_i}$. This step is accomplished by sorting the messages by the user IDs of the receivers, which simultaneously puts them in order of the storage servers of the receivers because each user ID begins with the index of that user's storage server, and then sending the sorted messages in "round-robin" order to the routing servers. The number of scattered messages is the same as the number of messages received from the ingestion servers. It is safe to reveal this number, because the adversary can observe the clients connecting to the ingestion servers. This round (corresponding to the "transpose" step of columnsort) has the effect of scattering the messages intended for each storage server nearly evenly across the routing servers.

In Round 2 of TokenColumnRoute, R_i will "distribute" the messages it received in the previous round to the routing servers that forward to the receivers' storage servers. To accomplish this, the server first obliviously counts how many messages have receivers at each storage server via a linear scan on the messages. The tokens have already enforced that no user receives more than f messages in an epoch, and therefore, due to the even scattering from the previous step, we can be sure that there are at most $y = \lfloor f\lceil n/s\rceil/r\rfloor + r$ messages for each storage server. The r additive term is due to the scatter step imperfectly distributing messages, possibly leading to two routing servers differing by at most r in the number of messages with receivers assigned to a given storage server, and ensures that each message can be routed to the correct routing server in this step. The server obliviously computes the number of dummy messages needed to reach the value y for each storage server, obliviously adds that number of dummies at the end, and then obliviously shuffles the messages. Since there are now guaranteed to be exactly y messages destined for each storage server, the shuffle allows us to take advantage of the Scramble-then-Compute [22] paradigm to non-obliviously send each message in the shuffled buffer to the routing server that forwards to the storage server for its receiver.

In Round 3 of TokenColumnRoute, the routing server will forward the messages to the storage servers of their receivers. The routing server first reduces the messages with the same receiver storage server by eliminating the r added in the previous round because at most $f\lceil n/s\rceil$ could be destined for each storage server. Next, the routing server obliviously shuffles the messages to hide which

Round 1 (Sort1) After receiving the messages for this epoch from the ingestion servers, obliviously sort the messages by the user IDs of the receivers and the priorities and send the messages in round-robin order to the routing servers. That is, given sorted messages M, send message $M[x]$ to R_j, where $j = (x \bmod r) + 1$.
Round 2 (Sort2) Obliviously sort the messages received in the last round by the receivers' user IDs and the message priorities. Let $w = \lceil b\lceil n/r\rceil/r\rceil$, and send the jth subsequence of w messages in the sorted list to R_j.
Round 3 (Sort3) Obliviously sort the messages received in the last round by the receivers' user IDs and the message priorities. Let $z = \max((r-1)^2, b)$, and M be the sorted array. If $i > 1$, send messages $M[1..z]$ to R_{i-1}. Also, send state $t = v \parallel c$, where v is the user ID of the recipient of message $M[z+1]$, and c is the number of messages in $M[z+1..z+b]$ with same receiver ID (extracted trivially via a linear scan of b messages).
Rounds 4 & 5 After receiving z messages M' and t from the next routing server R_{i+1} (if any), append M' to M. For $i < r$, obliviously sort the $2z$ messages $M[wr - z + 1..wr + z]$ together. Let $x = z + 1$ if $i > 1$; else let $x = 1$. Let $y = wr + z$ if $i < r$; else let $y = wr$. Obliviously count the messages to each user in $M[x..y]$ and convert to dummy messages the lowest priority messages to a given user exceeding b total, starting this process with the state t. Move all dummies in $M[x..y]$ to the end via oblivious compaction. Finally, proceed with Rounds 2 and 3 of TokenColumnRoute, using $M[x..y]$ as the message list and using b instead of f.

Figure 5: IDColumnRoute: ID-channel routing at server R_i

messages were removed in the previous step. Finally, it simply forwards each message to the storage server of its receiver. Figure 4 provides an example of the TokenColumnRoute protocol.

4.6.2 IDColumnRoute. The ID-channel routing algorithm (Figure 5) is more complex because a token is not required to send a message. As a result, a receiver might be sent more messages than the system can deliver to them in a given round. To handle this challenge, IDColumnRoute effectively performs the complete columnsort algorithm (Rounds 1–3) to determine if a receiver is being sent more than the b messages they can receive. If so, the lowest-priority excess messages are removed by converting them to dummy messages (Round 4). After that point, TokenColumnRoute can be applied directly (Rounds 4–5).

The distributed sort accomplished by IDColumnRoute in Rounds 1–3 is able to complete in one fewer round than given in columnsort because the distribution of the first and last $z = \max((r-1)^2, b)$ items (i.e., the "shift" and "unshift" steps in columnsort) can be simplified and parallelized in our case. Instead of each server R_i sending its first and last z items to R_{i-1} and R_{i+1}respectively, we have each R_i send (i) its first z items to R_{i-1}, and (ii) a state t with the receiver user ID of item $z+1$ at R_i, and the number of items destined for this user ID in the next b items (after the first z items).

Thus, in Round 4, after sorting its last z items with the received z items, R_{i-1} can remove any excess messages it holds for a given receiver by converting them to dummy messages. Because the sorting was performed on the receiver user ID and then the priority values, taking the excess messages from the first messages for a given

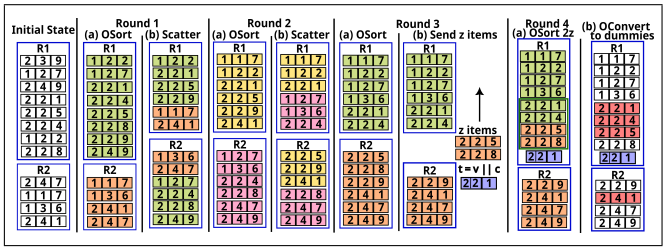

Figure 6: IDColumnRoute example with $n = 8$, $r = 2$, $s = 2$, $b = 2$. $z = \max((r-1)^2, b) = 2$. Message labels are the recipient user ID (storage server index | per-server user number) followed by the priority. Message colors in Round 1 and 2 simply highlight how messages get scattered. Blue messages in Round 3 and 4 correspond to the state message t. Red indicates low-priority messages that get dropped (i.e., marked as dummies). Otherwise a user would receive more than b messages this epoch. After performing the illustrated steps, obliviously compact the real messages and proceed with Rounds 2 and 3 of TokenColumnRoute.

user guarantees that the lowest-priority messages are removed. After the excess messages are converted to dummy messages, the receivers are guaranteed to have at most b messages in the system, and the steps of the token-channel routing algorithms are performed. Figure 6 provides an example of the IDColumnRoute protocol.

We note that in TEEMS, as presented, all routing is accomplished by the routing servers. However, we can reduce the rounds and communication by having ingestion and storage servers involved in the first and last rounds of routing, respectively, instead of merely forwarding messages to and from them. This variant can also reduce computation by eliminating the oblivious shuffle in the final round of routing, because it is made redundant by the initial message sort done by the storage server. We use this variant in our implementation because it is faster in the small deployments we consider experimentally. The original variant, on the other hand, is both simpler to describe and improves bandwidth utilization in larger deployments when throughput limits on the all-to-all communication pattern are reached.

4.7 Storage

Storage servers in each channel maintain a mailbox for each user assigned to them. The token-channel storage servers also maintain, for each of their users, a list of the user IDs of their friends. Each mailbox has a short-term and a long-term component, where messages are initially stored in the short-term mailbox and are eventually moved to the long-term mailbox if not yet sent to the client. Both mailboxes are simple lists of messages. However, the short-term mailbox is an append-only list that starts empty and increases to a maximum length of m_1 messages, while the long-term mailbox is fixed-size list of m_2 messages.

Both mailboxes may contain both real and dummy messages. The short-term mailbox is designed to provide efficient per-epoch updating via a simple append operation. However, to hide how many real messages are delivered in an epoch, the same number of messages (real or dummy messages) must be added in each

epoch. To avoid unlimited growth, the real messages in the short-term mailbox are periodically (i.e., less often than every epoch) transferred to the long-term mailbox via a compaction operation.

During an epoch, a storage server receives a list of messages from its routing server. The server attaches a current timestamp to each of the received messages, which enables clients to determine when the message was received, then obliviously sorts the messages by receiver user ID. Next, storage server S_j counts the number of messages intended for each user via a linear scan. It performs a reverse compaction so that the messages whose receiver is the ith user assigned to S_j appear in sequence starting in position $f \cdot i$ (or $b \cdot i$ for the ID channel), where the dummy messages will end up filling any gaps between those positions. Then, via another linear scan, S_j appends all of the messages for each user, including dummy messages, to their short-term mailbox. Note that this operation is inherently oblivious.

Because the short-term mailbox would otherwise grow without bound, the real messages it contains are periodically transferred into the long-term mailbox. If and when this transfer occurs depends on how recently the client has interacted with the system, which is observable to the adversary and thus does not reveal anything about the mailbox contents. A transfer occurs if the client has interacted more than m_1/f epochs ago. At that point, the storage server transfers the real messages between mailboxes obliviously by first appending the short-term mailbox to the long-term mailbox, next using order-preserving compaction to move any dummy messages to the end while maintaining existing messages in the long-term mailbox, and finally dropping the messages that appear after position m_2. m_1 should be set to hold messages for a short period, and m_2 should be set to hold the messages that might accumulate during a disconnection period; e.g., with 256-byte messages and 4-byte timestamps, $m_1 = 1200$ (312 KB) would hold 10 minutes of messages for one-second epochs and $f = b = 2$, and $m_2 = 100$ (26 KB) would hold 100 real messages.

When a client runs ClientInteract, the storage server sends the client a copy of the short-term mailbox and deletes all messages from it. If the short-term mailbox has been emptied into the long-term mailbox since the last client interaction, the server also sends a copy of the long-term mailbox and then overwrites the long-term mailbox with dummy messages. Note that these long-term mailboxes are completely independent of the actual routing protocols of TEEMS. Whether a client's mailbox needs to be compacted or not is not a secret; it is based on when the client last connected to TEEMS. Hence long-term mailbox storage and compaction can be handled by servers independent of our routing protocols. Additional details of these algorithms appear in Appendix D.

4.8 Friends

Friend relationships are symmetric, and the process to establish one begins with a friend request. The request must be accepted within e epochs of the request being issued, where e is set to create a relatively short expiration period (e.g., a day). Several of the steps involved in managing friend requests require extra communication or extra computation by the various parties involved in the process. Such communication and computation is performed (or simulated) every time such an action might be necessary to avoid leaking whether a friend relationship is being changed. However, the urgency of completing such changes is lower than delivering a message, and so we set a frequency ϕ at which friend-update messages or operations may be performed (e.g., $\phi = 10$ indicates such operations will happen once every 10 epochs).

The process for u to establish a new friend relationship with v is:

1. u reserves a friend slot for v at the token-channel storage server S_u^{tkn}, recording in that slot the current epoch and assigning it a nonce ζ. A slot reservation may be made in a free slot or one reserved more than e epochs prior. S_u^{tkn} creates and returns a friend request F, which includes the current epoch, the source u, the receiver v, the nonce ζ, and a MAC of those contents under key K.

2. u submits F to v via the ID channel as the contents of a normal message to v.

3. If the ID-channel storage server S_v^{id} receives a friend request F with a valid MAC and with the current epoch, it delivers the request to v's mailbox.

4. After v's client receives F, the user can manually approve it if it has a friend slot free. If the epoch in the request is within e of the current one, v's client sends the accept response G to u via the token channel. G contains the request F, indicates acceptance, and is sent in a normal token-channel message.

5. When token-channel ingestion server I_v^{tkn} receives a friend response G, it validates the MAC in F and checks that the request was sent less than e epochs in the past. If so, it forwards G as with a normal message with a valid token. At the same time, it directly contacts v's token storage server S_v^{tkn} to inform it that a friend relationship is being confirmed with u. S_v^{tkn} adds u as a friend of v and sends an updated list of friendships to v.

6. When G is received by the storage server S_u^{tkn}, the server detects that it is such a response, validates the MAC tag, and checks for the expiration of the contained request F. If either check fails, the response is ignored. Otherwise, S_u^{tkn} identifies the reserved friend slot with a matching nonce and records the confirmed friendship. S_u^{tkn} sends an updated list of friendships to u.

If user u wants to revoke an existing friendship with user v:

1. u creates a revocation request F and sends it on the token channel included in a normal message. Since u and v are friends, u can send a token-channel message to v.

2. After receiving F, the ingestion server I_u^{tkn} directly contacts the storage server S_u^{tkn} to tell it to remove v as a friend of u.

3. When F arrives at the storage server S_v^{tkn}, the server detects the revocation request, removes u from v's friend list, and sends an updated list of friendships to v.

As noted, the maintenance of friendships requires some computation and communication beyond what is required for normal messaging, and the system must perform these obliviously to hide not just which users have friendships but also if a user is updating their friendships at all. The added (possibly dummy) actions are (1) u sends a slot reservation command to S_u^{tkn}, and S_u^{tkn} obliviously processes the request and returns a friend request; (2) S_v^{id} obliviously identifies valid friend requests; (3) I_u^{tkn} obliviously identifies valid friend responses and revocations, it sends a friend update to S_u^{tkn}, and S_u^{tkn} obliviously updates the friend list of u; (4) S_u^{tkn} obliviously identifies valid friend responses and revocations and updates the associated friend list; and (5) S_u^{tkn} sends an updated friend list to u. The system performs these additional actions only once every ϕ

epochs, due to their lower urgency and frequency. The storage and ingestion servers limit each user to performing one friend request, response, and revocation during such an epoch. The system thus makes friendship operations as unobservable to the adversary as it makes sending messages.

5 Analytical Evaluation

5.1 Efficiency

We focus on messaging costs, as the costs of the maintenance procedures for accounts and friends are much lower. A detailed analysis is presented in Appendix F. There is no asymptotic difference in the runtimes between the token and ID channel, and asymptotically TEEMS incurs an online cost of $O(\ell(n/r + rs)\log(n/r + rs))$, and an offline cost of $O((n/r + rs)\log^3(n/r + rs))$

In the token channel, communication is dominated by the Distribute round (Round 2), in which each server sends and receives $O(n/r + sr)$ messages, where the sr term is due to the r messages added for each storage server to ensure enough space to send messages to the appropriate routing server. In the ID channel, total communication is also dominated by the Distribute round (Round 4), as all servers must send the maximum possible to each storage server. Before that round, no dummy messages are added. Storage servers send and receive $O(n/s)$ messages because mailboxes are evenly distributed across storage servers.

To consider how the system as a whole scales with the number of users, assume for simplicity that $s = r$ and that n/s is integral. Then in the token channel the total communication and computation per routing server is proportional (up to logarithmic factors) to the total number of messages that server sent or received, which is $(2f + 1)n/r + r^2$. To minimize this value, we set r as a function of n: $r = ((2f + 1)n/2)^{1/3}$. At this value, the number of users per routing server becomes $(2n^2/(2f + 1))^{1/3} = O(n^{2/3})$, and similarly for the ID channel with b in the place of f (since the load on the ID channel is roughly a constant $2\times$ that of the token channel).

This analysis indicates that the load per router grows with the number of users. Ingestion and storage servers are easy to keep at constant load, as an arbitrary number of them can connect to a single routing server. Although the increasing load on routing servers does limit system scalability, our experimental results (Section 6) indicate that the system can scale to at least tens of millions of users. Those results show that we can obtain message latencies of less than our target 3 seconds when each (single-core) server has up to 190,000 users. Following the above analysis, we can stay at or below this load per routing server with as many as $n = 101\,\text{M}$ users, which requires $r = 532$. At this size, TEEMS could host the estimated 40–100 million users of the Signal messaging app [32], which is popular among the privacy-conscious.

Scaling TEEMS to even larger sizes may be possible by changing its routing algorithm. The routing servers essentially sort the messages by the storage server of the recipient, and networks of constant node degree exist for such distributed sorting, including butterfly-routing and sorting networks [7, 9]. However, such networks require at least $\log_2 n$ rounds compared to the two rounds of TokenColumnRoute.

5.2 Security

The definition of Communication Unobservability (CU) for a system like TEEMS appears in Appendix E (Definition E.1). The underlying security game (Figure 9) is an instance of the Communication Unobservability game as defined by Kuhn et al. [39]. That definition is generic, which we make specific for our adversary model. In our game, the adversary can run some malicious clients, chooses the interaction and messaging behavior of all clients, and designates one challenge message, which is not sent if the challenge bit is zero. As with all prior MPCS, compromised-friend attacks are outside of our threat model [5], which we realize by requiring the challenge message to be between honest users. The challenger executes the system and returns to the adversary the observables in our threat model: messages received by malicious users, network messages, and the internal execution traces of the servers. The server execution traces include the intructions and memory locations accessed but do not include memory contents, which models a TEE threat model in which memory is encrypted but side channels can reveal access information. The adversary is then challenged to guess if the challenge message was sent or not.

We prove the security of the token and ID channels individually. These channels can be used on their own, or, as described in Section 4.8, composed for friend management. Theorem 1 shows that each TEEMS channel is Communication Unobservable. A proof of this theorem appears in Appendix E.

Theorem 1. *The token and ID channels in TEEMS are each CU.*

While the definition of CU security we have adapted from Kuhn et al. [39] addresses a passive adversary only, we informally claim several security properties of TEEMS with respect to an active adversary. Foremost, the system is still CU against an active network adversary, as the messages in an epoch use authenticated encryption and are dropped if the integrity check fails. Dropped messages, which an active adversary can also accomplish directly, cause the receiving server to stop executing for the epoch, which is predictable to the adversary and thus leaks no information. Modified or dropped messages from or to clients cause the sent or received messages to be dropped without any consequent halting of the system in that epoch. The integrity of TEE enclaves and the attestation process prevents any server from performing its prescribed calculation incorrectly due to an active adversary. Malicious clients may send malformed messages, but they are easily obliviously recognized by the ingestion server and replaced by dummy messages.

We further claim that TEEMS provides message integrity against an active adversary, including for the message content, sender, and timestamp. To defend this claim, we observe that the ingestion servers authenticate users and ensure that the authenticated identity matches the senders in the submitted messages. TEE correctness and the use of secure channels between enclaves ensures that the sender and content stay correct until delivered to the user. Messages are delivered in the epoch after they are received (if they are delivered at all), and so the TEEs at the storage servers attach the correct timestamp before delivery to the client.

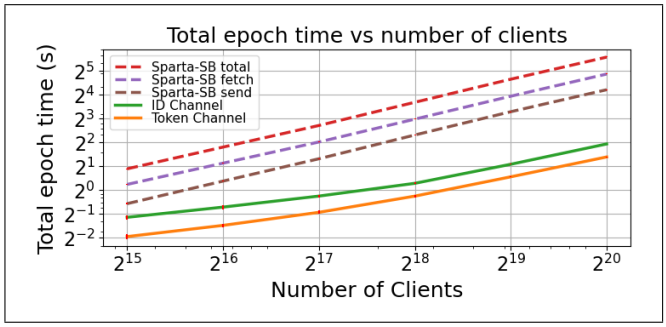

Figure 7: Measuring ID and token channel epoch times as the number of clients increases with a fixed set of 4 four-core servers. Both axes are log scale. Messages are 256 B in TEEMS, while they are 128 B in Sparta. At 2^{20} clients, TEEMS improves latency and throughput by 18× over Sparta.

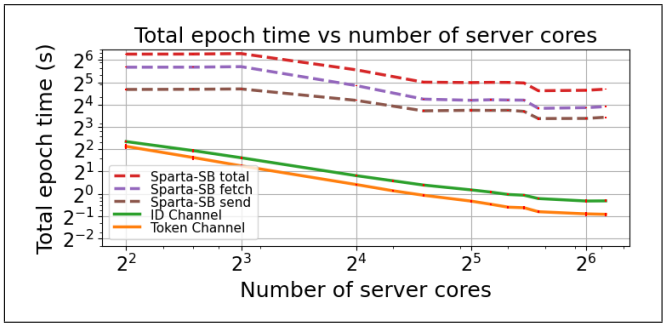

Figure 8: Measuring ID and token channel epoch times for TEEMS as the number of single-core servers increases with a fixed set of 2^{20} clients. Both axes are log scale. Messages are 256 B in TEEMS, while they are 128 B in Sparta.

6 Experimental Evaluation

We implemented the token and ID channel protocols of TEEMS, and present benchmarks in this section. Our implementation is written in C++, is open source, and is publicly available at https://git-crysp.uwaterloo.ca/iang/teems. Our evaluations evince TEEMS' ability to support low-latency and high-throughput metadata-protected communications, while being able to scale horizontally as more users join the system.

Experiment Setup. All our experiments are performed using an Intel SGX machine with two Xeon 8380 CPUs @ 2.30 GHz with 40 cores each. We implement multiple servers by instantiating each logical server (that runs all three server processes of ingestion, routing, and storage) on our machine with a single core assigned to it; we set aside 8 cores for simulating clients on the same machine. Consequently, the largest of our experiments present instantiations of TEEMS with up to 72 servers. For a direct comparison, we also run Sparta's implementation [30] on our setup.

We envision a real-world instantiation of TEEMS would have all servers in a data center and assume 13 Gbps network bandwidth between servers (like prior work [23, 37]). Our reported timings account for latency overheads of this network bandwidth, even though all our servers are situated on the same machine. All messages (even those between servers on the same machine) in our implementation are sent over TCP sockets, with forward-secret enclave-to-enclave encryption. This running of multiple servers on the same machine is not just an artifact of our experiment; a real deployment of TEEMS could also run multiple TEEMS servers on a single many-core machine. As we run all of our benchmarks on the same server, larger experiments incur paging overheads. This implies our results are conservative and in practice TEEMS would yield better performance when instantiated as a distributed system.

In our implementation, we use ORCompact [67] for oblivious compaction. For reverse compaction, we design and implement ORExpand, which runs the ORCompact algorithm in the reverse direction. For efficiency, TEEMS leverages the recent offline/online oblivious shuffling (WaksShuffle) and sorting (WaksShuffle+QS) algorithms of Sasy et al. [69], and we build TEEMS with their oblivious algorithm library [68]. Our timing results presented only incur the online phase of these oblivious shuffles and sorts. In practice, as long as the server has a few additional cores available, they can be used in parallel with each epoch of ID/token route to perform the precomputation required for the next epoch. For instance, with 2^{20} clients, we can use 24 single-core server processes for routing in the token channel and 40 for the ID channel, so that the latencies are similar in the two channels at just under one second. Then each server would need around 2.4 and 2.1 additional cores for precomputation in the token and ID channels, for a total of 64 server processes and 205 cores (for both channels together). Since we have a limited number of cores in our experimentation setup, we perform these precomputations up front before an epoch to measure TEEMS' scalability.

In our experiments, we set the message size to 256 bytes, and users send and receive exactly one message in every epoch for both token (f=1) and ID channel (b=1) cases. This configuration of TEEMS matches prior MPCS [26, 37, 42, 44] where users send and receive exactly one message every epoch (see Table 6 in Appendix G for performance comparisons). These parameters can be tuned to allow for a clients to receive more messages at the expense of more server computation. In our experiments, all TEEMS servers perform all three roles of ingestion, routing, and storage. Allocating servers designated roles also presents another axis that can be tuned to improve performance and scale horizontally.

Latency and throughput with fixed servers. In Figure 7, we observe the latency overheads of TEEMS with 4 four-core servers, as the number of clients in the system increases. Even in this small-scale experiment, TEEMS can enable metadata-protected messaging for 2^{16} clients in under 0.39 s/0.66 s (token/ID channel), and a throughput of 168K/99K messages per second. As the number of users in the system increases, epoch latency grows linearly with a constant < 1 in the client range we evaluate; i.e., doubling the number of clients with the same number of servers yields slightly less than 2× the epoch latency. At 2^{20} clients with 4 servers, TEEMS performs one epoch in 2.6 s/3.8 s, exhibiting a throughput of ≈400K/274K messages per second. In contrast, Sparta with the same amount of server resources takes 47.5 s for one full round, exhibiting a throughput of ≈22 K messages per second. TEEMS' token channel presents over 18× improvement in end-to-end latency and throughput over Sparta.

Table 1: Comparing MPCS for metadata-protected messaging. 'O' implies the system has a receiver anonymity set of currently online users. 'A' indicates that all users (irrespective of whether they are online or offline) are part of the receiver anonymity set. The ◐ for Loopix's asynchronous property reflects its trusted service provider assumption (see Section 7, and for Groove since it only allows for partial asynchrony (see Section 2.2). Additionally, in Appendix G we report performance numbers that demonstrate TEEMS' improvement in latency and throughput over these systems.

System	Low Latency	High Thrput	Horiz Scalable	Async	Receiver Anon Set	CUS
Clarion [26]	○	○	○	○	O	●
XRD [42]	○	○	○	○	O	●
Karaoke [44]	○	●	●	○	O	●
Groove [8]	○	●	●	◐	O	●
Loopix [62]	●	○	●	◐	A	○
SealPIR [4]	●	○	○	●	A	●
Sabre [74]	●	○	○	●	A	○
Boomerang [37]	○	●	●	○	O	●
Sparta-LL [29]	●	○	○	●	A	○
Sparta-SB/D [29]	○	●	○	●	A	○
TEEMS	●	●	●	●	A	●

Horizontal Scalability. Next, we analyze how well TEEMS scales horizontally. In Figure 8, we observe the total time taken to route token and ID channel messages of 2^{20} clients, as a function of the number of servers. We observe that with fixed number of clients, doubling the number of servers consistently results in ≈30–40% reductions in the epoch latency. Routing messages from 2^{20} clients, gradually drops down from 4.4 s and 5.1 s (throughput of ≈240K/206K messages per second) for token and ID channels with just four (single-core) servers, all the way down to 0.53 s/0.80 s (throughput of ≈2.0M/1.3M messages per second) with 72 (single-core) servers, exhibiting TEEMS' ability to scale horizontally to maintain low latency and high throughput as the user base grows.

In contrast, Sparta's total time reduces from 76.9 s to 25.9 s over the same range of server cores. However, Sparta is by design not horizontally scalable (see Appendix A.3). The added server cores can only parallelize their bottleneck oblivious sorts. While our results demonstrate the ability of TEEMS to deliver messages in under one second, in practice a latency in the order of a few seconds, about the time it takes to actually type a message, would suffice.

7 Related Work

Over the last two decades several novel MPCS designs have been proposed [66]. However, a substantial fraction of these constructions [1, 19, 40, 41, 43, 51, 57] are broadcast systems, where clients publish a message to all other users of the system. In contrast, TEEMS aims to provide pairwise messaging for its clients. Moreover, TEEMS targets the strong metadata-protection goal of a Communication Unobservable System (CUS). As we observe in Appendix A.1, the weaker goals of sender or receiver unlinkability can be vulnerable to traffic analysis, as it is common for the members of a communicating pair of clients to repeatedly swap the roles of sender and receiver during a single conversation. Table 1 presents an overview of how state-of-the-art MPCS satisfy the goals of TEEMS.

Clarion [26] is the latest iteration of a multiparty computation based CUS [2], but it incurs high latency and lacks scalability in comparison to mixnet-based designs. Mixnet-based designs have produced many high-throughput constructions, and some of them have low latency and are horizontally scalable as well, but they all lack asynchronicity. XRD [42] presents a mixnet-based CUS that is more efficient than prior pure mixnet based approaches [15, 41]. Mixnet-based designs drastically improved latency and throughput by incorporating differential privacy to hide the metadata of adversary-observable parameters of the system. This line of works started with Vuvuzela [77], and was followed by its horizontally scalable variants Stadium [73], Karaoke [44], Groove [8], and Yodel [45]. Yodel is tailored for metadata-protected voice calls; each round incurs an expensive one-time circuit establishment phase, to enable long-duration low-latency voice calls; this is not appropriate for messaging. In Table 1, Karaoke and Groove represent this line of works.

The improvements in throughput and latency, however, are at the expense of shifting the underlying privacy guarantee to a differentially private one rather than a cryptographic one. A fundamental shortcoming of such MPCS is that the privacy guarantees have a finite duration that they are configured for. The faster such differential privacy systems become, the quicker they expend their allocated privacy budget (for meaningful privacy guarantees) requiring either a system reset or users attaining weaker and unclear privacy guarantees. Boomerang [37], inspired by Karaoke, presents an MPCS that leverages TEEs to avoid using differential privacy. However, just like all the other aforementioned mixnet designs, they cannot support asynchronous clients, and they all have to incur the additional overheads of a dialing protocol before any actual communication. Loopix [62] represents an outlier among mixnet-based MPCS designs as it can support asynchronous clients, but at the expense of trusting special "service provider" nodes in the system, and adversarial providers can subvert clients' metadata privacy. Moreover, Loopix has the weakest form of the manytrust assumption due its underlying stratified network structure [66].

Most designs that can support asynchronous clients have stemmed from PIR-based techniques [4, 6, 18, 65, 74]. Pung [6] and its successor SealPIR [4] present a CUS through single-server computational PIR, and consequently incur significant computational overheads. Express [27] presents a Reverse PIR based mailbox system for end-to-end metadata protected communications, and its successor Sabre [74] improves the performance of this style of mailbox system. In Table 1, we present Sabre and SealPIR as representatives for these two PIR-based approaches. Ultimately, such schemes can offer low latency for message delivery and retrieval, but they have to handle each message individually and consequently have poor throughput. Moreover, both latency and throughput degrade as more users join the system, as they cannot efficiently scale horizontally.

Sparta [29] is a TEE-based design that provides asynchronous communication. Sparta is a Receiver Unobservable System (RUS), but because it reveals who sends messages and when, it can reveal when a pair of users is communicating back and forth, thus falling short of its stated goal to resist traffic analysis and showing the advantage of communication unobservability (provided by a

CUS like TEEMS). Sparta provides variable-size mailboxes with the goal of never dropping messages. It has three variants: Sparta-LL provides low latency but not high throughput, and both Sparta-SB and Sparta-D provide high throughput but not low latency. Moreover, no Sparta variant scales horizontally, including the partially distributed Sparta-D design due to its centralized query processor. In addition, Sparta-SB and Sparta-D possess a critical information leak when the system hits its storage capacity, which both violates the receiver unobservability goal and provides a denial-of-service attack on all users simultaneously. Those designs also do not scale over time because their message storage grows with each message sent. For details on these issues, see Appendix A.

8 Conclusion

TEEMS is the first MPCS that targets metadata-protected messaging and supports the four fundamental properties of (i) low latency, (ii) high throughput, (iii) horizontal scalability, and (iv) asynchronicity required for any messaging service to be deployed in practice. In comparison to prior work, TEEMS presents a 18× improvement in latency and throughput, requires fewer servers for scaling, can scale to 100M users with a 1 s epoch time, and affords more desirable properties. We hope that TEEMS brings metadata-protected messaging closer to practical mass adoption.

Acknowledgments

This work was supported by the Office of Naval Research. We thank the Ontario Graduate Scholarships program, NSERC (CRDPJ-534381 and RGPIN-2023-03260), and the Royal Bank of Canada for supporting this work. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program, award CRC-2018-00135. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References

[1] Ittai Abraham, Benny Pinkas, and Avishay Yanai. 2020. Blinder – Scalable, Robust Anonymous Committed Broadcast. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[2] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. 2017. MCMix: Anonymous Messaging via Secure Multiparty Computation. In *26th USENIX Security Symposium*.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. https://software.intel.com/content/www/us/en/develop/articles/innovative-technology-for-cpu-based-attestation-and-sealing.html. Accessed August 2024.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy (S&P)*.

[5] Sebastian Angel, David Lazar, and Ioanna Tzialla. 2018. What's a Little Leakage Between Friends?. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*.

[6] Sebastian Angel and Srinath Setty. 2016. Unobservable Communication Over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[7] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket Oblivious Sort: An Extremely Simple Oblivious Sort. In *Symposium on Simplicity in Algorithms (SOSA)*.

[8] Ludovic Barman, Moshe Kol, David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2022. Groove: Flexible Metadata-Private Messaging. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[9] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of American Federation of Information Processing Societies (AFIPS)*.

[10] BBC News. 2021. Putin critic Navalny jailed in Russia despite protests. https://www.bbc.com/news/world-europe-55910974. Accessed August 2024.

[11] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. 2017. Prochlo: Strong Privacy for Analytics in the Crowd. In *Symposium on Operating Systems Principles (SOSP)*.

[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[13] David Chaum. 1988. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *Journal of Cryptology* (1988).

[14] David Chaum. 2003. Untraceable Electronic mail, Return Addresses and Digital Pseudonyms. In *Secure Electronic Voting*.

[15] David Chaum, Debajyoti Das, Farid Javani, Aniket Kate, Anna Krasnova, Joeri De Ruiter, and Alan T Sherman. 2017. cMix: Mixing with Minimal Real-Time Asymmetric Cryptographic Operations. In *International Conference on Applied Cryptography and Network Security (ACNS)*.

[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. https://doi.org/10.1109/EuroSP.2019.00020

[17] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai

[18] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. 2020. Talek: Private Group Messaging with Hidden Access Patterns. In *Annual Computer Security Applications Conference (ACSAC)*.

[19] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. 2015. Riposte: An Anonymous Messaging System Handling Millions of Users. In *2015 IEEE Symposium on Security and Privacy (S&P)*.

[20] Henry Corrigan-Gibbs and Bryan Ford. 2010. Dissent: Accountable Anonymous Group Messaging. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*. ACM.

[21] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. 2013. Proactively Accountable Anonymous Messaging in Verdict. In *22th USENIX Security Symposium*.

[22] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. 2017. Privacy-Preserving Computation with Trusted Computing via Scramble-then-Compute. *Proceedings on Privacy Enhancing Technologies (PoPETs)* (2017).

[23] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*.

[24] Deutsche Welle. 2019. Iran defends execution of gay people. https://www.dw.com/en/iran-defends-execution-of-gay-people/a-49144899. Accessed August 2024.

[25] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*.

[26] Saba Eskandarian and Dan Boneh. 2022. Clarion: Anonymous Communication from Multiparty Shuffling Protocols. In *29th Network and Distributed System Security Symposium (NDSS)*.

[27] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. 2021. Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy. In *30th USENIX Security Symposium*.

[28] Stephen Farrell and Hannes Tschofenig. 2014. Pervasive Monitoring Is an Attack. RFC 7258, https://www.rfc-editor.org/rfc/rfc7258.txt.

[29] Kyle Fredrickson, Ioannis Demertzis, James Hughes, and Darrell Long. 2024. Sparta: Practical Anonymity with Long-Term Resistance to Traffic Analysis. In *2025 IEEE Symposium on Security and Privacy (SP)*.

[30] Kyle Fredrickson, Ioannis Demertzis, James Hughes, and Darrell Long. 2024. Sparta: Practical Anonymity with Long-Term Resistance to Traffic Analysis. https://github.com/ucsc-anonymity/sparta-experiments. Software artifact.

[31] The Guardian. 2013. Bradley Manning: A sentence both unjust and unfair. https://www.theguardian.com/commentisfree/2013/aug/21/bradley-manning-sentence-unjust. Accessed August 2024.

[32] Justin Hendrix, Caroline Sinders, Cooper Quintin, Leila Wagner, Tim Bernard, and Ami Mehta. 2023. What is secure? Analysis of popular messaging apps. *Tech Policy Press* (June 2023).

[33] Intel. 2012. Intel Trusted Execution Technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf. Accessed August 2024.

[34] Intel. 2018. Q3 2018 Speculative Execution Side Channel Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html. Accessed August 2024.

[35] Intel. 2019. Intel Processors Voltage Settings Modification Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html. Accessed August 2024.

[36] Intel. 2020. 2020.2 IPU - Intel RAPL Interface Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00389.html. Accessed August 2024.
[37] Peipei Jiang, Qian Wang, Jianhao Cheng, Cong Wang, Lei Xu, Xinyu Wang, Yihao Wu, Xiaoyuan Li, and Kui Ren. 2023. Boomerang: Metadata-Private Messaging under Hardware Trust. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*.
[38] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD Memory Encryption. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf. Accessed August 2024.
[39] Christiane Kuhn, Martin Beck, Stefan Schiffner, Eduard Jorswieck, and Thorsten Strufe. 2019. On Privacy Notions in Anonymous Communication. *Proceedings on Privacy Enhancing Technologies (PoPETs)* 2 (2019).
[40] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. 2017. Atom: Horizontally Scaling Strong Anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.
[41] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. 2016. Riffle. *Proceedings on Privacy Enhancing Technologies (PoPETs)* (2016).
[42] Albert Kwon, David Lu, and Srinivas Devadas. 2020. XRD: Scalable Messaging System with Cryptographic Privacy. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation (NSDI)*.
[43] Simon Langowski, Sacha Servan-Schreiber, and Srinivas Devadas. 2023. Trellis: Robust and Scalable Metadata-private Anonymous Broadcast. In *Network and Distributed System Security Symposium (NDSS)*.
[44] David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2018. Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[45] David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2019. Yodel: Strong Metadata Security for Voice Calls. In *ACM Symposium on Operating Systems Principles (SOSP)*.
[46] David Lazar and Nickolai Zeldovich. 2016. Alpenhorn: Bootstrapping Secure Communication Without Leaking Metadata. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[47] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX Security Symposium*.
[48] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*.
[49] Tom Leighton. 1984. Tight bounds on the Complexity of Parallel Sorting. In *Proceedings of the sixteenth annual ACM Symposium on Theory of Computing*.
[50] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (S&P)*.
[51] Donghang Lu and Aniket Kate. 2023. RPM: Robust Anonymity at Scale.
[52] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and Its Application to Anonymous Communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[53] Prateek Mittal and Nikita Borisov. 2009. ShadowWalker: Peer-to-peer Anonymous Communication using Redundant Structured Topologies. In *16th ACM Conference on Computer and Communications Security (CCS)*.
[54] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *USENIX Security Symposium*.
[55] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks Against Intel SGX. In *IEEE Symposium on Security and Privacy (S&P)*.
[56] Arjun Nambiar and Matthew Wright. 2006. Salsa: A Structured Approach to Large-Scale Anonymity. In *Proceedings of the 13th ACM conference on Computer and Communications Security (CCS)*.
[57] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. 2022. Spectrum: High-Bandwidth Anonymous Broadcast with Malicious Security. In *Proceedings of the 19th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.
[58] Nicholas Ngai, Ioannis Demertzis, Javad Ghareh Chamani, and Dimitrios Papadopoulos. 2024. Distributed & Scalable Oblivious Sorting and Shuffling. In *2024 IEEE Symposium on Security and Privacy (S&P)*.
[59] NPR. 2021. Tech workers recount the cost of speaking out, as tensions rise inside companies. https://www.npr.org/2021/10/21/1048038154/fired-apple-facebook-netflix-google-workers. Accessed August 2024.
[60] NYR Daily. 2014. We Kill People Based on Metadata. https://www.nybooks.com/daily/2014/05/10/we-kill-people-based-metadata/. Accessed August 2024.
[61] Andriy Panchenko, Stefan Richter, and Arne Rache. 2009. NISAN: Network Information Service for Anonymization Networks. In *16th ACM Conference on Computer and Communications Security (CCS)*.
[62] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. 2017. The Loopix Anonymity System. In *26th USENIX Security Symposium*.
[63] Marc Rennhard and Bernhard Plattner. 2002. Introducing MorphMix: peer-to-peer based anonymous Internet usage with collusion detection. In *Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society (WPES)*.
[64] Reuters. 2018. UN Experts says Egypt systematically targets rights activists. https://www.reuters.com/article/us-egypt-rights/u-n-experts-says-egypt-systematically-targets-rights-activists-idUSKCN1M82EB/. Accessed August 2024.
[65] Len Sassaman, Bram Cohen, and Nick Mathewson. 2005. The Pynchon Gate: A Secure Method of Pseudonymous Mail Retrieval. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society (WPES)*.
[66] Sajin Sasy and Ian Goldberg. 2024. SoK: Metadata-Protecting Communication Systems. *Proceedings on Privacy Enhancing Technologies (PoPETs)* (2024).
[67] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[68] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2023. Fully Oblivious Algorithms. https://crysp.uwaterloo.ca/software/obliv/. Software artifact.
[69] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2023. Waks-On/Waks-Off: Fast Oblivious Offline/Online Shuffling and Sorting with Waksman Networks. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[70] Sajin Sasy and Olga Ohrimenko. 2019. Oblivious Sampling Algorithms for Private Data Analysis. In *Advances in Neural Information Processing Systems (NeurIPS)*.
[71] Florian Sieck, Zhiyuan Zhang, Sebastian Berndt, Chitchanok Chuengsatiansup, Thomas Eisenbarth, and Yuval Yarom. 2023. TeeJam: Sub-Cache-Line Leakages Strike Back. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)* 2024, 1 (2023).
[72] Signal. 2017. Private Contact Discovery for Signal. https://signal.org/blog/private-contact-discovery/. Accessed August 2024.
[73] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. 2017. Stadium: A Distributed Metadata-Private Messaging System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.
[74] Adithya Vadapalli, Kyle Storrier, and Ryan Henry. 2022. Sabre: Sender-Anonymous Messaging with Fast Audits. In *IEEE Symposium on Security and Privacy (S&P)* .
[75] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.
[76] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*.
[77] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM.
[78] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (S&P)*.
[79] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

A Comparison to Sparta

The Sparta system [29] provides three different designs: Sparta-LL (for low latency), Sparta-SB (a sorting-based design for high throughput), and Sparta-D (designed to scale to larger numbers of users). Sparta is similar in several ways to TEEMS. It also uses TEEs, aims to provide resistance to traffic analysis, and provides low-latency asynchronous messaging.

However, Sparta has several significant deficiencies that TEEMS avoids: (1) it reveals sending behavior and is thus susceptible to traffic analysis, (2) it shares limited storage resources across users and thereby leaks information about receiving behavior, (3) it requires a single server to perform computation and communication over all clients and is thus non-scalable, and (4) the system allows malicious users to perform denial-of-service on other users. We

discuss these problems in some detail and describe how TEEMS avoids them.

A.1 Traffic Analysis Resistance

Sparta is explicitly designed to resist traffic analysis. Similar to TEEMS, it hides which users receive messages (if any) by requiring users to fetch messages (or dummies) on a schedule independent of their true messaging behavior. However, Sparta does not provide the same protection to sending behavior. In Sparta, a client connects to the system and submits a "send" request if and only if the user has a genuine message to send. Fredrickson et al. acknowledge this leakage (which they denote as S_t, indicating the set of users sending a message at a given time t).

However, this leakage does not strongly resist traffic analysis, as claimed. For example, a messaging pattern in which a pair of users A and B send messages to each other back and forth repeatedly would be apparent in the leaked information, as a message sent by A is repeatedly followed by a message sent by B, and a message sent by B is repeatedly followed by a message sent by A. Thus a global passive adversary, which is in the threat model of Sparta, could perform an attack similar to an intersection attack, looking for statistically unlikely sequences of alternating message sending by pairs of users.

TEEMS solves this problem by applying the same protections to sending messages as to receiving messages. Users in TEEMS must send messages (or dummies) on a schedule that is unrelated to their actual sending behavior. This protection does come at a latency and bandwidth cost, but those costs are already being paid on the receiving action, and so including them during sending at most doubles them.

A.2 Information Leakage via Shared Resources

Sparta uses variable-size mailboxes to avoid dropping messages. However, this design fails to recognize that the storage of such messages is ultimately limited in several ways: (1) the ORAM used to store them (in Sparta-LL) has a size fixed at initialization, (2) the TEE has a maximum amount of memory in its Enclave Page Cache, and (3) the system has a maximum amount of storage. Whenever the lowest of these limits is reached, the system cannot store any additional messages. An adversary can detect that the capacity has been reached by determining when messages between accounts it controls begin to drop messages. Thus the system at capacity leaks how many messages are being stored.

A malicious client can cause the system to reach capacity by sending messages to its accounts but never fetching them. Moreover, because Sparta reveals how many genuine messages are sent, leaking the number of stored messages reveals information about the number of genuine messages that have been fetched, which the system is supposed to hide. For example, if the adversary observes that a message between its accounts is not delivered at one time but a similar message is delivered at a later time, it can conclude one of the fetches of an honest user between those times contained a genuine message. In the worst case, there is one such honest recipient, identifying them completely. Since Sparta does not hide when users send messages, this leakage can allow a sender and recipient to be linked, violating the primary privacy goal of Sparta.

Messages can also be dropped in TEEMS, but it avoids any resulting information leakage in each of its two channel types. First, in the token channel, the number of senders for a recipient is limited to f, allowing the system to provide sufficient routing capacity for each message. Messages may still be dropped due to fixed-size mailboxes, but only due to existing messages to the same recipient as the dropped messages and thus not leaking information about messages to honest users. Second, in the ID channel, the case of too many messages for a given recipient in a round is handled by enforcing a similar *per-recipient* routing limit. Because the limit of b messages received is applied during routing to each recipient independently, the dropped messages for a malicious receiver will be dropped the same regardless of the current or past activity of the other users. Similar to the token channel, any subsequent dropping due to limited mailbox size is only due to existing messages, and which messages for an honest receiver get dropped is only observable to the honest user (and not to the adversary).

A.3 Scalability

Sparta-D is designed to scale to larger numbers of users. However, it centralizes query processing at a single "queue maintainer" that also maintains all of the user account information. In a given round, the following operations are performed: (1) users send their queries to the queue maintainer, (2) the queue maintainer sorts the queries with per-user metadata, (3) the queries are batched and padded and sent to submaps, (4) the submap responses are returned to the queue maintainer, (5) the responses are shuffled together, and (6) the users are sent their messages (including dummies as necessary).

The queue maintainer thus creates a significant bottleneck as the numbers of users and queries grow. If there are n users in the system and m queries in a given round, the oblivious sort operation (implemented with bitonic sort) requires roughly $(n+m)\log_2^2(n+m)/4$ compare-and-swap operations, each of which operates over an item the size of a message (128 B in the Sparta experiments). This computational cost becomes significant when the number of users grows large, such as the tens of millions of users that Signal currently has [32]. Moreover, the queue maintainer must receive all queries and send all responses, which presents a significant communication cost. For example, suppose the system had 128B messages, 50 million users, each user received two messages in a round (i.e., $k_i = 2$), and all users fetched in a round (all users should fetch frequently to avoid delay in receiving a message). Then the queue maintainer would need to send 12.8 GB per round just responding to users.

Another scalability issue, which affects both Sparta-D and Sparta-SB, is that the data structures storing messages in both systems constantly grow and at any time store the total number of messages *ever* sent into the system. Messages in these systems are stored in a simple vector M (Sparta-D divides this vector across its submaps). In a given round, for each user u_i, k_i dummy messages are initially added into M, where k_i is a public parameter controlling how many messages (real or dummy) u_i will receive during a fetch operation. For each user u_i, the system ultimately only removes from M the k_i messages that u_i receives during the fetch operation. Therefore, the size of M does not change as a result of fetch operations. However, send operations simply add the new messages to M, and so the size

of M at any point is the total number of messages ever sent into the system (or, equivalently, the total number of send queries ever performed by users).

This scalability issue is fatal to the sustained operation of Sparta-SB and Sparta-D over time. Eventually, as the message vector M grows, it will exceed the capacity of the system to store it. For example, if there are 10 million users and they send an average of 10 messages a day (each stored in 128 B internally), then each month the message store M will grow by an additional 389 GB. Moreover, as M grows, the computational cost of processing queries grows because it involves sorting M, and so the system runs increasingly slowly over time.

TEEMS is designed to avoid such issues and provide true scalability in the number of users and over time. TEEMS distributes all parts of sending and receiving messages. In particular, no single server must communicate with all users or process all messages sent or received in a given round. The data structures in TEEMS also maintain a constant size over time, given a fixed number of users.

A.4 Denial of Service

All the Sparta designs also suffer from severe denial-of-service attacks. Because mailboxes are allowed to vary in size, malicious users can send messages between themselves but never fetch them and thereby cause the system to reach its capacity to store messages. At that point, no new messages can be sent into the system, and it is rendered useless. Sparta describes no limit on the rate at which messages are sent into the system, and so malicious users can cause this to happen rapidly. Even if per-user sending limits were instituted, an adversary could still quickly fill up the available storage capacity by controlling many accounts.

Similarly, an adversary can perform a targeted denial-of-service attack on a given user u_i by sending many messages to u_i. Users fetch at most once a round, and a user u_i retrieves k_i messages in a single fetch query. As a result, if the adversary sends more than k_i messages to u_i every round, u_i's mailbox will continually grow, and u_i will experience an increasing delay in retrieving its messages from honest users. For example, if $k_i = 5$ and a round occurs every 10 seconds, then by sending just 10 messages to u_i per round for 24 hours, an adversary cause subsequent messages from honest users to u_i to take 24 hours to be delivered.

TEEMS avoids denial-of-service attacks entirely in the token channel because routing and storage capacity is reserved for users holding tokens. In the ID channel, a malicious user who obtains the ID of a target user u_i may cause messages to u_i to be dropped by simultaneously sending many messages to that user. However, such an attack requires the adversary to obtain the long user ID, which user may choose to share only with trusted contacts. Moreover, malicious flooding in a given round can only cause message drops in that round, and the priority mechanism causes the messages from malicious accounts repeatedly performing the attack to be steadily de-prioritized and thus dropped instead of messages from accounts sending to u_i at lower rates.

B MPCS Properties

Here we elaborate on the different properties we discussed in Section 1 and also quantify the thresholds we set for the properties of low latency and high throughput.

Low Latency: A MPCS for messaging needs to have an acceptable messaging latency. For messaging, the time to deliver a message should be (no worse than) about the time it takes to type a message. We therefore consider 3 s to be an acceptable messaging latency.

Throughput: The number of messages the system can process in a fixed time frame. We use messages per second as the unit, and require a throughput of at minimum 100 K messages per second to be considered high throughput.

Horizontal Scalability: The system should be able to scale as more users enter the system, by adding more servers into the system, while maintaining metadata protections and without partitioning the anonymity set.

Asynchronicity: Users should be able to send messages to receivers that are not currently online, with such messages being retrievable the next time the receiver connects to the system. Most existing MPCS constructions assume a synchronous system, where all users are online at all times.

Setup: Existing MPCS typically require some form of setup. In particular, they require some form of out-of-band exchange, and in most cases an additional dialing protocol is required to establish a channel for actual conversations [2, 26, 37, 42, 44, 73, 77].

C Summary of Parameters and Notation

Table 2 lists the parameters used to configure an instance of TEEMS. Each parameter is a number that is set system-wide at initialization and is constant during system operation. Table 3 summarizes the remaining notation used in the paper. These variables are just used by the paper to describe the system and its operation.

Table 2: The parameters in TEEMS

Parameter	Description
b	The maximum number of ID-channel messages a user can send or receive
e	Friend request expiration period
f	Maximum size of a user's friend list
g	Number of ingestion servers in a given channel
ℓ	The length of a message
m_1	Max messages in a short-term mailbox
m_2	Max messages in a long-term mailbox
r	Number of routing servers in a given channel
s	Number of storage servers in a given channel
ϕ	Frequency of friend operations

D Oblivious Algorithm Details

Here we provide further details of key oblivious algorithms in TEEMS.

Table 3: Key notation used in this paper

Notation	Description	Notation	Description
a	Fraction of users controlled by adversary	$\mathcal{M}$	A message
A	An account server	n	Total number of users in the system
C	A client	$\mathcal{R}$	The set of routing servers in a given channel
c_I	Number of client interactions this epoch at ingestion server I	R_i	The ith routing server in a given channel
d	The number of servers done sending inputs this round	$\mathcal{S}$	The set of storage servers in a given channel
F	A friend request	S_i	The ith storage server in a given channel
G	A friend response	$\mathcal{S}_R$	The set of storage servers in a given channel that routing server R forwards messages to
h_R	Fraction of users sending to accounts assigned to $\mathcal{S}^R$	S_u	The storage server in a given channel assigned to user ID u
$\mathcal{I}$	The set of ingestion servers in a given channel	t	The MAC tag in a long user ID
I_i	The ith ingestion server in a given channel	u, v	User IDs
$\mathcal{I}_R$	The set of ingestion servers in a given channel that forward messages to routing server R	ζ	A nonce
I_u	The ingestion server assigned to user ID u in a given channel	η	Current epoch
K	The global symmetric key	μ	Length of MAC tag in long user ID
M	A list of messages	π	Boolean indicating if the channel is token
		ρ	Current round
		τ	Length of a token

D.1 TokenColumnRoute

In Round 2, dummy messages are appended to achieve $y = \lfloor f\lceil n/s\rceil/r\rfloor + r$ total messages destined for each of the s storage servers. The number of messages x_i held by routing server R_i in this round are known given the (adversarially observable) number of messages received from clients by each ingestion server. However, the number of those messages that are both real and destined for a given storage server are hidden, and the algorithm must be oblivious to it.

To add the dummy messages obliviously, therefore, R_i first performs a linear scan of the first x_i messages to obtain a count of the number of messages destined for each storage server. This count can be maintained in a list L of s numbers which is itself linearly scanned and obliviously updated for each of the x_i entries. Next, $sy - x_i$ dummy messages are appended to the message list M. Finally, via a linear scan through those appended dummy messages, they are assigned dummy recipients such that y total messages are destined for each storage server. At a given dummy message, this assignment is made by a linear scan through L and an oblivious test for the first non-zero entry, followed by an oblivious decrement of that entry and an update of the next dummy recipient to assign. Note that in large deployments where s is large, the data structure L could instead be implemented as an ORAM to achieve $O(\log s)$ performance for each update or lookup operation in L.

D.2 IDColumnRoute

In Round 4, the lowest-priority messages to a given user are converted to dummy messages if the total to that user exceed b. To accomplish this, routing server R_i stores and updates the state $t = v \parallel c$ containing a user ID v and a count c, initialized as the value t received from R_{i+1}. R_i performs a linear scan of $M[x..y]$ in reverse order, obliviously updating t with a new user ID if the current recipient differs from v, obliviously incrementing the count c or setting it to one if v has changed, and obliviously converting the current message to a dummy message if $c > b$.

D.3 Storage

In a given epoch, the storage server appends messages to the short-term mailboxes of the users. The same number of messages is appended to each short-term mailbox, where some may be dummy messages. To perform this append, the storage server S_i first sorts the messages by recipient ID. Note that this positions all real messages before all dummy messages. Then S_i performs a linear scan of the messages to create the position list L used for expansion (i.e., reverse compaction). To create L, a count c is maintained during the linear scan of M of the number of messages seen to recipient of the current message being examined. Position L_j is obliviously set to be $L_{j-1} + 1$ if the last recipient matches the current one, or to $L_{j-1} + f - c + 1$ (or $L_{j-1} + b - c + 1$ for the ID channel) otherwise, with $L_1 = 1$. c is obliviously updated with each new message examined either by incrementing it or setting it to one if the last recipient differs from the current one.

E Security Proof

The game defining Communication Unobservability (CU) is given in Figure 9. A system is said to satisfy Communication Unobservability, as given in Definition E.1, if its advantage in the CU game is negligible.

Definition E.1. Messaging system $\mathcal{S}$ is Communication Unobservable if $\left|\Pr\left[\mathsf{CU}_{\mathcal{S}}^{0,\lambda} = 0\right] - \Pr\left[\mathsf{CU}_{\mathcal{S}}^{1,\lambda} = 0\right]\right|$ is negligible in λ for all polynomial-time adversaries.

Theorem 1, first stated in Section 5.2, proves that each channel in TEEMS is Communication Unobservable.

Theorem 1. *The token and ID channels in TEEMS are each CU.*

Proof. There are two distributions of the adversary's observations, one for each possible value of the challenge bit β of the security game (Figure 9). We argue that the two distributions of

The parameters of the game are:

- $\mathcal{A}$: A polynomial-time adversary
- C: The challenger
- E: The number of epochs
- n: The number of users
- $\mathcal{S}$: The messaging system
- β: The challenge bit
- λ: The security parameter

$\mathcal{A}$ and C interactively execute the game:

(1) $\mathcal{A}$ sends C the following:
   - For each client, a subset of epochs during which the client will interact with $\mathcal{S}$
   - For each client, for each epoch it interacts with $\mathcal{S}$, a set of messages to send during that epoch, and their recipients.
   - A subset of users who are malicious
   - A challenge user, epoch, and message. The user must be honest, the epoch must be one during which the user interacts with $\mathcal{S}$, the message must be one of the messages the user sends in that epoch, and the message must be to an honest recipient.

(2) If $\beta = 0$, C removes the challenge message from the messages to be sent by users.

(3) C executes $\mathcal{S}$ for E epochs.

(4) C sends to $\mathcal{A}$ the following:
   - The messages received by malicious users
   - Communication traces between all parties
   - Execution traces of all servers, each of which is a sequence of instructions and the memory locations read and written by those instructions

(5) $\mathcal{A}$ outputs its guess at β.

Figure 9: $\mathsf{CU}_{\mathcal{S}}^{\beta,\lambda}$: Communication Unobservability game

observables are indistinguishable. The observables are the communication traces between all parties, the execution traces of all servers, and the messages received by malicious users.

We first make the standard argument that all communications are encrypted and thus obtain confidentiality with respect to the adversary who does not possess the secret keys. Therefore we implicitly move to games in which only the lengths of the messages between servers and between honest clients and servers are sent to the adversary.

For the clients, the epochs in which they interact, while adversarially chosen, are the same regardless of β. The size of the client messages while sending messages is constant regardless of the number of real messages sent or received. When receiving messages, the size of the messages depends only on the client interaction history, as the size of the short-term mailbox is proportional to the length of time since the last interaction and the constant-sized long-term mailbox is sent depending on the time of the last interaction. Therefore the clients' communication traces are identical regardless of β.

For the ingestion servers, the lengths of its messages to routing servers depend only on the number of client interactions (and not on the number of real messages they contain). Therefore, the lengths of those messages do not depend on β. The algorithms implementing the ingestion-server operations (e.g., to validate tokens or MAC tags) are oblivious, which in this context means that their sequences of instructions and memory-location accesses do not depend on more than the size of their inputs. The lengths of those inputs depend only on the number of clients that have interacted, which does not depend on β, and therefore the execution sequences are the same in the two distributions.

For the routing servers, we consider the token and ID channel separately.

For the token channel, we consider each round of TokenColumnRoute. A routing server in Round 1 sorts the received messages (real and dummy) and then sends them round-robin to the other routing servers. The sorting is performed obliviously, with execution traces (i.e., instruction and memory-location access sequences) depending only on the number of received messages. That number has already been argued to be independent of β because it depends only on the number of client interactions, making the execution traces similarly independent. The number of messages sent to the other servers also depends only on the number received from clients in this epoch and is thus independent of β.

In Round 2 of TokenColumnRoute, a routing server appends messages to those it received in the previous round, shuffles them, and then sends them to the routing server forwarding to the storage server of each message. The number of messages appended depends only on the number received, which has already been argued to be independent of β as it only depends on the number of client interactions in this epoch. That the desired number of messages for a given storage server can be reached by appending additional messages follows from the following facts: (1) tokens limit the number of messages in the entire system intended for any user to fn; (2) there are at most $\lfloor f\lceil n/s\rceil$ users with mailboxes at a given storage server due to the even division of users across storage servers; and (3) after Round 1 the routing server has at most $\lfloor f\lceil n/s\rceil/r\rfloor + r$ messages destined for any given storage server, as it would have at most $\lfloor f\lceil n/s\rceil/r\rfloor$ messages if the messages were globally round-robin scattered but each routing server locally scatters the messages, adding at most one excess messages from each of the r routing servers. The appending process is oblivious in that it only depends on the number of appended messages, and thus its execution trace is independent of β. Similarly, the shuffling process is oblivious and only depends on the number of messages, which after appending is constant, and so the execution traces are independent of β. Finally, the number of messages destined for each storage server is the same, as a result of the appending process, and those messages are in a uniformly random order after the shuffling process. Therefore, the distribution of the execution and communication traces is the same regardless of β, where each routing server ultimately receives the same number of messages but in a random order.

In Round 3 of TokenColumnRoute, the number of dummy messages marked for deletion is the same for each of the storage servers the messages are destined for, and so the number remaining for each storage server is the same regardless of β. The subsequent shuffle depends only on this number of undeleted messages, and the execution trace is thus independent of β because an oblivious algorithm is used that depends only that number. Finally, forwarding

messages to the storage servers yields execution and communication traces independent of β because the same number of messages is sent to each storage server, albeit in a uniformly random order.

Now turning our attention to IDColumnRoute, Round 1 is the same as in TokenColumnRoute except the messages are sorted using priorities as well. Therefore, for the same reasons already given, the communication and execution traces of Round 1 are independent of β.

In Round 2 of IDColumnRoute, the number of messages sent to each other routing server is a deterministic function of the number of messages received in Round 1, making it also independent of β. The sorting is done obliviously, where the execution trace depends only on the number of messages being sorted, and so the execution trace is also independent of β.

In Round 3 of IDColumnRoute, the number of messages sent is a deterministic function of the number received in Round 2 and is therefore independent of β. The sorting is oblivious, depending only on the number of messages received, and therefore produces execution traces independent of β. The computation of the state is also performed obliviously depending only on b and thus produces execution traces independent of β.

In Rounds 4 and 5 of IDColumnRoute, the appending and sorting algorithms are oblivious given the number of messages received previously, and so their execution traces do not depend on β. The dummy conversion is performed via an oblivious linear scan that similarly only depends on the number of messages received, making the execution traces also independent of β. The oblivious compaction depends only on the number of messages being compacted, which does not depend on β, and so their execution traces are independent of β. The subsequent execution of two rounds of TokenColumnRoute produces communication and execution traces independent of β, using the arguments already given for those rounds of TokenColumnRoute, given that they operate on a number of messages that does not depend on β.

For the storage servers, the number of messages (real and dummy) they each receive from the routing servers is f (or b for the ID channel) times the number of mailboxes they store. That number is the same regardless of β, and so the size of that communication is the same. The algorithm to append to the short-term mailbox is oblivious, and its execution traces depend only on the number of mailboxes and the number of message received. Both of those are independent of β, making the execution traces also independent of β. The algorithm to transfer messages to the long-term mailbox is executed based on how frequently the client interacts, which does not depend on β. Moreover, it is oblivious in that its execution trace depends only on the size of the short-term and long-term mailboxes. The size of the short-term mailbox depends only on the client interaction history, and the long-term mailbox has constant size. Therefore, the execution traces of the mailbox transfers are independent of β. Furthermore, when, during a client interaction, the short-term mailbox is sent and overwritten, doing so is oblivious process that depends only on the mailbox size. Similarly, the long-term mailbox is sent at times that only depend on the client interaction history, and overwriting it is an oblivious operation depending only on its size, which is constant. Therefore, the execution traces of sending mailboxes to clients are independent of β.

Table 4: Computation per epoch for a client or server

Entity	Runtime
C	$O(\ell)$
I	$O(\ell c_I)$
R (offline)	$O((n/r + rs)\log^3(n/r + rs))$
R (online)	$O(\ell(n/r + rs)\log(n/r + rs))$
S (offline)	$O((n/s)\log^3(n/s))$
S (online)	$O(\ell(n/s)\log(n/s))$

Table 5: Maximum communication per epoch per client or server

Entity	Bytes received	Bytes sent
C	$\ell(f + b)$	$2\ell + \mu + \tau$
I^{tkn}	$c_I(\ell + \tau)$	$c_I\ell$
I^{id}	$c_I(\ell + \mu)$	$c_I\ell$
R^{tkn}	$(c_{I_R}\ell)+$ $\ell s(\lfloor f\lceil n/s\rceil/r\rfloor + r)$	$\ell s(\lfloor f\lceil n/s\rceil/r\rfloor + r)+$ $\ell(s/r)f\lceil n/s\rceil$
R^{id}	$\ell(c_{I_R})+$ $\ell(\sum_i\lceil c_{I_{R_i}}/r\rceil)+$ $\ell(\lceil n/r\rceil + r)+$ $\ell(r-1)^2+$ $\ell(\lceil n/r\rceil + r)+$ $\ell s(\lfloor b\lceil n/s\rceil/r\rfloor + r)$	$\ell(c_{I_R})+$ $\ell(\sum_i\lceil c_{I_{R_i}}/r\rceil)+$ $\ell(r-1)^2+$ $\ell(\lceil n/r\rceil + r)+$ $\ell s(\lfloor b\lceil n/s\rceil/r\rfloor + r)+$ $\ell(b\lceil n/s\rceil)(s/r)$
S^{tkn}	$\ell f\lceil n/s\rceil$	$c_S f\ell$
S^{id}	$\ell b\lceil n/s\rceil$	$c_S b\ell$

Finally, we consider the messages that a malicious user receives in a given interaction. None of those messages is the challenge message because it must be to an honest user. In the token channel, all messages sent to the malicious user get routed to the storage server. Therefore, because the challenge message is not to the malicious user, the state of the malicious user's mailbox, which depends only on the messages it gets routed and the interactions of the malicious client, is independent of β, and so are the messages delivered from it to the malicious user. A similar argument applies to the ID channel, except that messages to a malicious user may be dropped during routing. However, in IDColumnRoute a message is only dropped in favor of another message to the same user with the same or higher priority. Because the challenge message is to an honest user, its presence does not change the messages dropped for the malicious users, and therefore the messages received by the malicious users are the same regardless of β. □

F Analytical Efficiency

Let C be a client, and let c_x be the number of clients that have interacted with server(s) x in a given epoch. Let μ be the length of a MAC tag used in long user IDs, and let τ be the length of a token. We analyze runtimes asymptotically in terms of the message length and the numbers of users and servers. We use the runtimes of the oblivious subroutines used in our implementation: for y items of length z, the runtime for oblivious compaction is $O(zy\log y)$ [67], and for oblivious shuffles and sorts it is $O(zy\log y)$ online and $O(y\log^3 y)$ offline [69].

Table 6: Experimental results of related work. Clarion, XRD, Karaoke, and Boomerang require setup in the form of a dialing protocol. The experiment numbers below do not account for dialing. In Clarion, dialing incurs about the same overhead as its communication protocol. The dialing protocol used by Boomerang and Karaoke [46] takes ≈16 s for 1 million users. XRD requires dialing, but no existing dialing protocol fits the system. Dialing has to be run every few communication rounds before any actual message data can be transmitted. Groove circumvents dialing by assuming shared secrets between clients. However, prior to any communication Groove require a circuit setup process that takes ≈300 s for 1 million users. Gray entries are results as reported in the related work, and black entries are results from experiments we ran on our server.

System	Experiment Reported				
	Clients	Servers (Total cores)	Msg size (bytes)	Latency	Throughput (msgs/sec)
Clarion [26]	10^6	3 (48)	160	80 s	12.5 K
XRD [42]	10^6	200 (7200)	256	128 s	8 K
Karaoke [44]	10^6	100 (3600)	256	6 s	166 K
Groove [8]	2^{20}	100 (3200)	256	32.4 s	1.5 M
Loopix [62]	3×10^4	5 (80)	224	1.9 s	16 K
SealPIR [4]	2.6×10^5	1 (16)	288	0.51 s	2
Sabre [74]	2^{18}	3 (48)	1000	0.05 s	20
Boomerang [37]	2^{20}	32 (256)	256	7.8 s	134 K
Sparta-SB [29]	2^{20}	1 (72)	128	25.89 s	41 K
TEEMS	2^{20}	64 (205)	256	0.98 s	2.14 M

The runtimes of individual clients and servers per epoch are given in Table 4. For clients, the runtime is linear in the message size because for each channel they send and receive a fixed-size batch of messages. For ingestion servers, we see that the runtime is linear in the number of client interactions and the message length. For routing servers, the costs are dominated by the message processing in the Distribute round (i.e., token-channel Round 2 and ID-channel Round 4), where there are $O(n/(sr) + r)$ messages for each of the s storage servers, and the most costly operation on them is the oblivious shuffle. For storage servers, there are $O(n/s)$ messages to process, and that processing is dominated by an oblivious sort of them. The offline operations are not on the messages themselves (but just on message indices), and so the ℓ factor only appears in the online runtimes.

The total communication of each type of server per epoch is given in Table 5. Clients send and receive fixed-size batches of messages. In the token channel, tokens are included with the messages from the clients, while in the ID channel, the MAC tags in the long user IDs are included instead. Ingestion servers send and receive a single message for each of their client interactions, and so their load increases with the number of online clients rather than the number of assigned clients.

In the token channel, communication is dominated by the Distribute round (Round 2), in which each server sends and receives $O(n/r + sr)$ messages, where the sr term is due to the r messages added for each storage server to ensure enough space to send messages to the appropriate routing server. A factor of f also appears in the communication for the last two rounds, as a user might receive a message from each friend.

In the ID channel, total communication is also dominated by the Distribute round (Round 5), as all servers must send the maximum possible to each storage server. Before that round, no dummy messages are added. ID routing does involve three additional rounds with communication, and the last two rounds of communication include a factor of b as each user might receive that many messages.

Storage servers send and receive $O(n/s)$ messages because mailboxes are evenly distributed across storage servers.

G Latency and Throughput Comparisons

In Table 6, we revisit the systems presented in Table 1 and show the sizes (numbers of clients, servers, and cores) and results (latency and throughput) of experiments reported in those works. We also include TEEMS to compare to those systems. We include this comparison in an appendix because we did not re-run all the comparator systems on common hardware. However, we can still do order-of-magnitude comparisons of the latency and throughput achievable by TEEMS and the other systems.

We see that Clarion and XRD yield latencies far too large for our target application of interactive messaging. Karaoke's and Boomerang's latencies are only slightly larger than one would like, but their throughputs are each more than an order of magnitude smaller than that of TEEMS. Loopix, SealPIR, and Sabre have acceptable (and even impressive) latencies, but their throughputs range from small (Loopix) to completely impractically tiny (SealPIR, Sabre). All of this also completely ignores, as described in the caption to Table 6, the additional cost of the dialing protocol required by Clarion, Boomerang, Karaoke, and XRD, but not the others, including TEEMS.