Abuse Reporting for Metadata-Hiding Communication Based on Secret Sharing

Saba Eskandarian, *University of North Carolina at Chapel Hill*

https://www.usenix.org/conference/usenixsecurity24/presentation/eskandarian

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Abuse Reporting for Metadata-Hiding Communication Based on Secret Sharing

Saba Eskandarian
University of North Carolina at Chapel Hill
saba@cs.unc.edu

Abstract

As interest in metadata-hiding communication grows in both research and practice, a need exists for stronger abuse reporting features on metadata-hiding platforms. While message franking has been deployed on major end-to-end encrypted platforms as a lightweight and effective abuse reporting feature, there is no comparable technique for metadata-hiding platforms. Existing efforts to support abuse reporting in this setting, such as asymmetric message franking or the Hecate scheme, require order of magnitude increases in client and server computation or fundamental changes to the architecture of messaging systems. As a result, while metadata-hiding communication inches closer to practice, critical content moderation concerns remain unaddressed.

This paper demonstrates that, for broad classes of metadata-hiding schemes, lightweight abuse reporting can be deployed with minimal changes to the overall architecture of the system. Our insight is that much of the structure needed to support abuse reporting already exists in these schemes. By taking a non-generic approach, we can reuse this structure to achieve abuse reporting with minimal overhead. In particular, we show how to modify schemes based on secret sharing user inputs to support a message franking-style protocol. Compared to prior work, our *shared franking* technique more than halves the time to prepare a franked message and gives order of magnitude reductions in server-side message processing times, as well as in the time to decrypt a message and verify a report.

1 Introduction

Public discussions on abuse reporting in messaging platforms primarily focus on content moderation policy questions regarding what kinds of messages should and should not be allowed. However, when it comes to reporting abuse on private messaging platforms that provide end-to-end encryption or hide metadata, important technical problems must be addressed before moderation policy decisions can be enforced.

In an unencrypted platform, a moderator can see the contents of all messages, as well as the identities of their senders, and can use this information to make decisions when a user reports a message. When messages are end-to-end encrypted, the moderator no longer sees the message contents, and a user making a report must be able to demonstrate to the moderator that the reported message actually corresponds to content sent through the platform. The metadata-hiding setting poses an even greater challenge, as the moderator sees neither message contents nor the identities of message senders.

Message franking, first introduced by Facebook and used in both WhatsApp and Messenger Secret Conversations, provides a lightweight solution to the problem of abuse reporting for end-to-end encrypted messages [29, 32, 25]. Unfortunately, message franking schemes used in practice do not work in the metadata-hiding setting. Proposals to handle abuse reporting for metadata-hiding platforms include asymmetric message franking (AMF) [44] and the Hecate scheme [33]. While these schemes provide strong security guarantees for abuse reporting in the challenging metadata-hiding setting, they come with performance overheads orders of magnitude higher than the message franking used in practice.

This work takes a new approach to abuse reporting for metadata-hiding communication. While previous solutions have been designed to work generically for any metadata-hiding platform, we take a non-generic approach that applies to a specific class of private messaging schemes: schemes based on additive secret sharing techniques. Given the large number of proposed messaging schemes that use secret sharing, as well as incipient deployments of other privacy preserving technologies based on secret sharing [19, 26, 5, 1, 2], this family of approaches merits special attention. We show how these schemes can use a message franking-style protocol with minimal performance overhead.

At a high level, our scheme involves secret sharing the message franking process. Message franking involves the platform MACing a compactly committing encryption of a message [29, 32], so we need to reproduce this efficiently in a setting where the server serving as the moderator does not have access to the whole message at the time a message is processed by the platform. Our solution has the moderator

compute a MAC over hashes of shares of the message, ensuring the moderator cannot recover the message itself, and has the message sender include the necessary information in the message to regenerate these shares at report verification time. Starting from this idea, we build a *shared franking* protocol that allows a moderator to authenticate messages sent through a platform and tie them to their senders, despite only ever seeing secret shares of the messages when they are initially sent.

In addition to designing the protocol itself, we introduce appropriate security definitions for abuse reporting in this setting and prove that our scheme satisfies security under these definitions. Our threat model allows all the servers except the moderator to be malicious, and ensures that misbehaving servers and users cannot violate confidentiality of messages, forge reports, or evade the reporting mechanism.

We show how to integrate our shared franking protocol with several families of existing metadata-hiding communication systems in the research literature. Our protocol is directly compatible with schemes based on computation over additive secret shares of messages [8, 35, 16, 27] and, in a limited sense, with schemes using threshold secret sharing [42, 6, 36]. We also describe how to adapt our scheme to support shared franking in schemes based on DC-nets [17, 21, 47], including works that augment a DC-net approach with distributed point functions [30, 14, 15, 20, 28, 39].

We implement our shared franking scheme and compare its performance to AMF, Hecate, and conventional message franking. We find that shared franking results in comparable or lower communication overhead than AMF and Hecate while dramatically lowering computation costs. In particular, for handling 1 KB messages in the two server setting and comparing to Hecate, we reduce the computation cost of sending a message by 2.6×, the overall computation for processing a message on the server(s) by 22.6×, the cost to read a message by 19×, and the cost to verify reported messages by 31×. Comparing to AMF, we achieve 1-2 order of magnitude performance improvements.

Overall, the performance of our scheme resembles that of deployed message franking more than it does previous abuse reporting schemes for metadata-hiding communication. These significant performance improvements are enabled by the fact that shared franking requires no public key cryptography, eliminating the need for signatures or zero knowledge proofs found in prior work. Our implementation and raw evaluation data are freely available and open source at https://github.com/SabaEskandarian/Shared_Franking.

In summary, we make the following contributions.

- Introduce the notion of shared franking and develop appropriate security definitions.
- Build a shared franking scheme compatible with metadata-hiding communication platforms based on additive secret sharing.

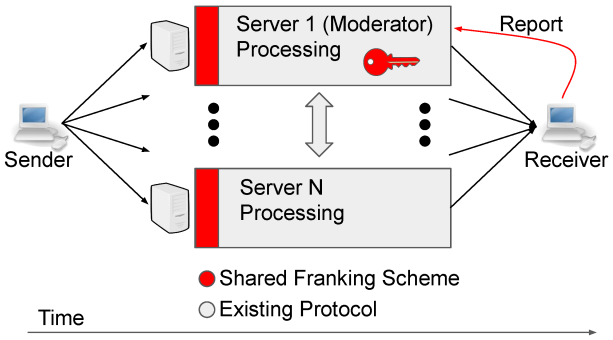

Figure 1: Shared franking augments a metadata-hiding communication scheme based on secret sharing. After receiving secret shares of a client's message, servers do some additional preprocessing before running the underlying protocol. This allows the moderator to later verify any messages reported by users. Only the moderator holds any new secrets, and security holds even if other servers collude with each other or with malicious users.

- Show how to integrate shared franking with existing metadata-hiding communication systems.
- Implement/evaluate our scheme, showing order of magnitude performance improvements and approaching the performance of conventional message franking used in practice.

2 Design Goals

A *shared franking* scheme builds on top of a metadata-hiding communication scheme based on secret sharing and allows users of the scheme to verifiably report abusive messages. This is achieved by augmenting the process of sending, processing, and receiving messages with additional information that can be sent to a moderator to report a message. The moderator runs a new verify() algorithm to authenticate reported messages and recover relevant metadata that it can use to make moderation decisions. In addition to satisfying various security requirements relating to authentication and verification of reports, a shared franking scheme must not compromise the privacy properties of the underlying communication system that it augments. Figure 1 abstractly illustrates how a shared franking scheme fits in with an existing metadata-hiding communication system.

We now briefly summarize the security properties we expect of a shared franking scheme before discussing them in greater detail below.

- **Confidentiality**: The shared franking scheme reveals nothing about who sent a message or what messages are sent. This property ensures that the security of the underlying scheme is not compromised by the addition of shared franking.

- **Accountability**: Every message delivered through the platform must be able to be reported to the moderator.

- **Unforgeability**: Users cannot be framed for sending messages they did not send.

- **Deniability**: Only the moderator can verify the authenticity of reported messages. If the contents of a report are leaked, it would be impossible for a third party to verify them.

As is the case in many metadata-hiding communication schemes, we will assume that the servers operating the communication system are trusted for *availability* but not for *security*. That is, correctness of outputs will assume that the servers cooperate with the protocol, but all our security definitions will require security against actively malicious servers. All our definitions additionally aim to protect against misbehavior by malicious or disruptive clients. In defining and achieving security, we will assume pairwise secure connections (encrypted and authenticated) between servers via TLS, which are generally already in use by the underlying communication protocol.

Accountability and anonymity. Before we move on, we emphasize that there are inherent tradeoffs between anonymity and accountability, and that, in addition to the technical considerations of how to build a shared franking scheme, it is also important to consider *whether* and *where* such schemes should be used. While our confidentiality definition will ensure that unreported messages enjoy the same privacy as in a scheme that does not support abuse reports, adding the ability to report a message to a moderator means that the moderator can learn the identity of a message sender and the contents of that message. Some metadata-hiding communication systems provide anonymity for the sender *even against the message recipient*. In these cases, shared franking would enable a message recipient to reveal information to the moderator that the recipient themself does not know, resulting in a greater cost to privacy than in standard message franking for end to end encrypted messaging. While this kind of disclosure may be desirable in certain limited settings, it does not seem suitable as a feature for general-purpose communication platforms. As such, examples of metadata-hiding schemes discussed in this paper provide *unlinkability* of message senders and receivers, but do not necessarily hide the identity of message senders from the receivers. This is not a technical limitation, but a reflection of our priorities in building better private communication technology. However, a relevant technical limitation of our scheme in the setting where the sender is anonymous to the receiver is that a malicious receiver could potentially collude with a malicious non-moderator server to reveal the sender's identity. This is not an issue in cases where the sender and receiver already know each other's identities.

2.1 Formalizing Shared Franking

Notation. Before formally stating the requirements of a shared franking scheme, we briefly summarize some notation used throughout the paper. We use $y \leftarrow f(x)$ to denote the assignment of the output of $f(x)$ to the variable y, and we use $x \xleftarrow{R} S$ to denote assigning to x a uniformly random element from the set S. The notation $\{\}$ represents the empty set. We use $\mathcal{A}^O$ to indicate that $\mathcal{A}$ has oracle access to O. A function $\mathsf{negl}(\lambda)$ is *negligible* if for all $c > 0$, there is an x_0 such that for all $x > x_0$, $\mathsf{negl}(x) < \frac{1}{x^c}$. We sometimes omit the security parameter λ when it is implicit from context.

We use $[x]$ to denote an additive secret sharing of x. For a message x split into N shares, we use $[x]_i, i \in \{1,...,N\}$ to denote each share. In particular, for an additive secret sharing $[x] = ([x]_1,...,[x]_N)$ it holds that $x = \Sigma_{i=1}^{N}[x]_i$. Finally, we use $\perp$ as a special character to indicate failure.

Throughout the paper, we use standard definitions of MACs, PRGs, CPA-secure encryption, collision-resistance, random oracles, and other widely used cryptographic primitives [13].

Syntax. A shared franking scheme consists of Send and Receive algorithms to be run by clients sending or receiving messages, a processing protocol run by servers $S_1,...,S_N$ who operate a metadata-hiding communication scheme, one of which also serves a content moderator, and a Verify algorithm used by the moderator to verify reports of abusive messages sent through the platform. For our scheme, we model the processing protocol with two functions, Process and ModProcess. The Process algorithm is the algorithm run by most of the servers participating in the protocol, and the ModProcess algorithm is a variant of Process run by the server serving as the platform's moderator.

We assume that the users in a shared franking scheme have access to a shared key k_U produced according to the protocol used in the underlying communication system, which they will use to encrypt messages. During the processing protocol, the moderator can include message *context* ctx with a message, to be revealed again if a user later reports the message. This is where the moderator can include information about the sender of the message or a timestamp indicating when the message was sent. Deployed message franking schemes use a similar mechanism to give moderators relevant information needed to judge reported messages [29].

We define the syntax of a shared franking scheme as follows.

Definition 2.1 (Shared Franking Scheme)**.** A *Shared Franking Scheme* consists of five algorithms $(\mathsf{Send}, \mathsf{Process}, \mathsf{ModProcess}, \mathsf{Read}, \mathsf{Verify})$. The syntax of a shared franking scheme is defined with respect to message space $\mathcal{M}$, *context space* $\mathcal{C}$, tag space $\mathcal{T}$, user key space $\mathcal{K}_U$, and server key space $\mathcal{K}_S$ as follows.

- $\mathsf{Send}(k_U, m, N) \rightarrow (w_1,...,w_N)$: This function takes a user secret $k_U \in \mathcal{K}_U$, a message $m \in \mathcal{M}$, and an integer

N indicating the number of servers to which the message will be sent. The function returns a series of *write requests* $w_1, ..., w_N$, each of which is sent to one of the servers.

- $\mathsf{Process}(N, i, w_i) \to (v_i, w_i')$: This function takes the total number of servers in the protocol, as well as the index of the server running the function and the input for that server. It returns the server's output value v_i as well as an output w_i' for the moderator server.
- $\mathsf{ModProcess}(N, k_S, w_1, \mathsf{ctx}, (w_2', ..., w_N')) \to v_1$: This function is a variant of the Process function that is run by the server serving as moderator. By convention, we set the moderator to be S_1. In addition to the number of users and the server's input, this function additionally takes the moderator's secret key $k_S \in \mathcal{K}_S$, the values w_i' for $i \in \{2, ..., N\}$ from the other servers, and a *context* input $\mathsf{ctx} \in \mathcal{C}$.
- $\mathsf{Read}(k_U, N, (v_1, ..., v_N)) \to (m, t)/\perp$: This function takes a user secret $k_U \in \mathcal{K}_U$, an integer N indicating the number of servers with which a message is shared, and a series of server output values $v_1, ..., v_N$. The function either recovers a message $m \in \mathcal{M}$ and tag $t \in \mathcal{T}$, or it outputs $\perp$ to indicate malformed server outputs.
- $\mathsf{Verify}(k_S, N, m, t) \to \mathsf{ctx}/\perp$: This function takes a server secret $k_S \in \mathcal{K}_S$, an integer N, a message $m \in \mathcal{M}$, and tag $t \in \mathcal{T}$. It outputs a context $\mathsf{ctx} \in \mathcal{C}$, or it outputs $\perp$ to indicate failed verification of the message/tag.

When a system uses shared franking, the user sending a message runs the Send algorithm to prepare messages rather than simply secret sharing – or encrypting and then secret sharing – their message. Once the servers receive the shares produced by Send, along with any other protocol-dependent information, they run the Process or ModProcess algorithms before doing any of the message processing used to deliver messages in the underlying communication system. When message shares are processed according to the design of the underlying system and delivered to a receiver, the receiver runs the Read algorithm to recover the message sent by the sender. If a message receiver decides to report a message, it sends the message m and tag t to the moderator, who runs Verify to check that the reported message is authentic and to recover relevant metadata.

A communication system compatible with shared franking is one that takes shares of (potentially encrypted) messages as inputs and then runs an interactive protocol between the servers offering the service to deliver messages. This protocol (called Deliver below), involves somehow shuffling the shares of messages in a way that the recipient of a message can identify it but the servers cannot link message senders and receivers. A simple example of such a scheme would be an MPC-based shuffle followed by anonymous broadcast of messages, where receivers do a linear scan of results to find their intended message. More sophisticated systems, such as MCMix [8], include mechanisms for senders and receivers to download only their intended messages after a shuffle, dramatically reducing communication and eliminating the need for trial decryptions. DC-net or DPF-based schemes [17, 30] can also rely on either anonymous broadcast or a mechanism for clients to privately determine where in a list of messages a given message will be delivered. In all cases, the correctness property provided by the underlying communication scheme must be that shares output by the scheme are shares of the same messages as the input shares, at least for the parts of the shares that correspond to the messages themselves. It is possible for additional protocol-dependent information to be appended by the servers during processing to aid in retrieval, but this is not relevant for the purposes of shared franking.

Correctness. The correctness definition for a shared franking scheme requires that messages sent via the scheme can be successfully read, and that the context the moderator retrieves when verifying a report matches the context used when the message was sent. Our definition of correctness, when restricted to the single-server case where $N = 1$, corresponds to a correctness definition for a conventional message franking scheme.

Our correctness definition also accounts for the possibility that the platform to which we add shared franking may do some computation on the shared franking outputs before they are returned to the user. For example, if the server outputs are secret shares of some messages and the underlying platform uses MPC to shuffle the messages before delivery, the final messages may be re-randomized versions of the shares output by the shared franking scheme.

We model the message delivery mechanism of the underlying protocol as an interactive protocol $\mathsf{Deliver}\langle\rangle$ between N servers, where each server S_i has a two-part input consisting of a share $[m]_i$ of the message being sent and some extra protocol-dependent data e_i,

$$\mathsf{Deliver}\langle S_1([m]_1, e_1), ..., S_N([m]_N, e_N)\rangle \to (([m']_1, e_1'), ..., ([m']_N, e_N')),$$

subject to the condition that the shared message m is preserved, i.e., $m = m'$. Although a real scheme acts on multiple messages at once, this description abstracts away other messages and routing logic but captures the impact on an individual message, which is the scope at which franking operations take place. A shared franking scheme modifies the shares $[m]$ of the message, replacing them with the processed outputs v_i of the shared franking protocol.

Definition 2.2 (Correctness)**.** We say that a shared franking scheme satisfies *correctness* if for any $\lambda, N \in \mathbb{N}, m \in \mathcal{M}, \mathsf{ctx} \in$

$\mathcal{C}, k_U \in \mathcal{K}_U, k_S \in \mathcal{K}_S$ and for some $e_1, ...e_N$, when we compute

$(w_1, ..., w_N) \leftarrow \mathsf{Send}(k_U, m, N)$
$v_i, w_i' \leftarrow \mathsf{Process}(N, i, w_i)$ for $i \in \{2, ..., N\}$
$v_1 \leftarrow \mathsf{ModProcess}(N, k_S, w_1, \mathsf{ctx}, (w_2', ..., w_N'))$
$((v_1', e_1'), ..., (v_N', e_N')) \leftarrow \mathsf{Deliver}\langle S_1(v_1, e_1), ..., S_N(v_N, e_N)\rangle$

we have that

$$(m', t) \leftarrow \mathsf{Read}(k_U, N, (v_1', ..., v_N'))$$

where $m' = m$, and

$$\mathsf{ctx}' \leftarrow \mathsf{Verify}(k_S, m, t)$$

where $\mathsf{ctx}' = \mathsf{ctx}$.

2.2 Defining Security

This section describes the various security properties we expect of a shared franking scheme. Following the discussion here, formal definitions for each property appear in Appendix A.

Confidentiality. The primary confidentiality property required of a shared franking scheme is that it does not compromise the privacy properties of the underlying metadata-hiding communication platform, except when revealing metadata stored in the context attached to a reported message. We use *metadata-hiding* to refer to systems that render message senders and receivers unlinkable to the platform. In secret sharing based schemes, message senders start by directly connecting to the servers to upload message shares, at which point the servers can identify the senders, as there is no need for anonymity in that connection. Only after receiving the messages do the servers process them in a way that breaks the link between message senders and message recipients. Our confidentiality definition is designed to ensure that the security of the message processing cannot be compromised by the addition of a tag to enable abuse reporting before messages are processed. This means that no server operating the platform, and no user other than the intended recipient of a message, can learn anything about the contents of a message sent through the platform or about who sent a given message through the platform. Since our work focuses on the case of secret-shared data, we will capture this requirement via an adversary who controls some strict subset of the N servers operating the service.

Our security definition allows an adversary to pick a set M of corrupted servers to control. In the security experiment, the adversary is allowed to adaptively ask an honest user to send the servers write requests for one of two messages m_0, m_1 of the adversary's choosing, and the adversary must determine which message is sent after seeing the resulting write requests w_i. We say that a scheme has confidentiality if no efficient adversary can distinguish between the two messages. This effectively means that the view of the adversary while processing messages for shared franking does not depend on the actual messages sent in any way. After the shared franking process is over, the security properties of the underlying messaging scheme will ensure that an adversary cannot see the contents of delivered messages or identify their senders. Our definition only requires that the additional message preprocessing required for shared franking does not expose information that could compromise users' privacy.

One could imagine other supplemental confidentiality requirements, such as requiring that a user's message remains confidential even against an adversary who controls all the servers, or that the context string ctx does not leak to servers other than the moderator. We do not formalize security definitions for these notions, but our scheme trivially satisfies both. To protect message contents against compromise of all servers, messages are encrypted via a CPA-secure encryption scheme before being secret shared. Our syntax prevents the message context ctx from being sent directly from the moderator to the other servers, and our scheme masks ctx with pseudorandom bits before the moderator outputs it to the underlying messaging protocol.

Accountability. Our accountability definition ensures that every message sent through the system can be successfully verified if reported, even in the presence of malicious clients and servers. In particular, an adversarial client or group of clients, even if they collude with all the servers except the moderator, cannot send a set of write requests where a message will be successfully decrypted by the recipient but verification of that message by the moderator will fail.

In our definition, the adversary is allowed to play the role of any number of malicious clients and servers, except for the moderator. The adversary controls the write requests, intermediate outputs, and context received by the moderator and gets the moderator's output share v_1 in response to inputs of the adversary's choosing. The adversary wins the game if, after its interactions with the moderator, it can produce a user key k_U^*, a context ctx^*, and write request w_1^*, and server outputs $v_2^*, ..., v_N^*, w_2', ...w_N'$, such that, after processing via $\mathsf{ModProcess}$, the Read algorithm can recover a non-$\perp$ message, but Verify either fails to verify or returns an incorrect $\mathsf{ctx}^{**} \neq \mathsf{ctx}^*$.

Unforgeability. Unforgeability requires that a malicious reporter cannot report messages or message contexts that were not actually sent through the platform. This property differs from accountability in that accountability protects against malicious *senders* who wish to send unreportable messages and abuse the platform without repercussions, whereas unforgeability protects against malicious *receivers* who wish to produce false reports that frame honest senders.

In this definition, similar to the accountability definition,

the adversary plays the role of any number of malicious clients and servers, except the moderator. After being allowed to interact with the moderator and receiving the moderator outputs for any successfully delivered/verified message, the adversary produces a new message and tag pair. The definition requires that the adversary cannot produce successfully verifying tags on new messages, and, moreover, the adversary cannot produce tags on already-sent messages that verify but return a context different from the context used when the message was originally sent.

Deniability. Deniability requires that only the moderator can verify the authenticity of reported messages. This property helps maintain the expectation that messages sent through private channels are largely ephemeral and cannot verifiably be exposed to third parties other than the moderator.

We do not formalize our own deniability definitions here, as notions of deniability from prior work extend to our setting in a straightforward way [44, 33, 40]. Instead, we briefly discuss some forms of deniability that a shared franking scheme must achieve and return to these in the security analysis of our scheme. See the asymmetric message franking work of Tyagi et al. [44] for an extensive discussion of the potential space of security definitions for deniability.

Following Tyagi et al., we consider three forms of deniability, all three of which should be satisfied: *receiver compromise deniability*, *moderator compromise deniability*, and *universal deniability*. Each definition requires that there exist a forgery algorithm that produces message, report pairs that are indistinguishable from real ones, thus allowing users to deny that they really sent or received a given message. The difference between the three definitions lies in what information the forgery algorithm can use in producing its forgery and in what information the distinguisher is allowed to see. Universal deniability gives the forger and distinguisher access to any public parameters of the system and the secret keys of all users uninvolved in the message being denied. In receiver compromise deniability, the forger and distinguisher additionally receive the user secret key that encrypts and decrypts the message being denied. In moderator compromise deniability, the forger and distinguisher additionally receive the moderator's secret key.

3 Franking for Secret-Shared Messages

This section introduces our shared franking scheme. We begin with background on conventional message franking. Our scheme can be seen as a way to lift standard, single-server message franking into the secret shared setting.

We will consider the message franking scheme used by Facebook [29] using the abstractions introduced by Grubbs et al. [32]. Roughly speaking, the scheme consists of two components: a *compactly committing* encryption scheme and a MAC. At its core, it consists of the platform MACing part of an appropriately formatted ciphertext from the user sending a message. This ciphertext, along with associated context information, is revealed later in the event that the receiving user reports the message. Verifying the MAC when a user reports a message allows the platform to record the context for each message without the need for extensive metadata collection and storage.

In this work, we will take advantage of the fact that, for metadata-hiding communication schemes based on secret sharing, the servers involved in the protocol can identify the sender of a message *at the time a message is sent*, even though they cannot identify the recipient or link senders and recipients for any message. In secret sharing based schemes, message senders start by directly connecting to the servers to upload message shares (without need for any intermediate anonymity layer), at which point the servers can identify the senders. Only *after* receiving shares of messages do the servers process them in a way that breaks the link between message senders and message recipients. This allows the server acting as the moderator to attach the necessary context, e.g., sender identity, at the time a message is sent.

Our approach is fundamentally different from prior work that extends content moderation to the metadata-hiding setting (e.g., [44]). Whereas prior works use cryptographic techniques to remove the involvement of the moderator from the message delivery process, we find ways to render the moderator's involvement harmless. This results in increased performance at the cost of only being applicable to schemes based on secret sharing.

3.1 Background: Message Franking and Compactly Committing Encryption

As mentioned above, a message franking scheme requires compactly committing encryption. This is an encryption where the ciphertext $c = (c_1, c_2)$ can be thought of as consisting of two components. Component c_1 is a conventional ciphertext, and c_2 functions as a commitment to the underlying message, which a moderator can verify in the event the message is reported. *Compactness* means that the size $|c_2|$ of the committing component of the ciphertext must be independent of the length $|m|$ of the underlying encrypted message m.

We restate the syntax for a compactly committing encryption scheme below. This is the same syntax introduced by Grubbs et al. [32], except whereas they presented their definitions in terms of compactly committing AEAD schemes, we present a syntax for compactly committing AE only. An AEAD scheme allows for additional authenticated, unencrypted data to be associated with a ciphertext [41], but we will not need this feature for our scheme. That said, although we do not use the AEAD syntax for simplicity of presentation, our scheme can easily be modified to accommodate additional associated data should it be necessary.

Definition 3.1 (Committing AE [32])**.** A committing AE scheme $\mathsf{CE} = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ver})$ is a four-tuple of algorithms with the syntax described below. Associated to a scheme is a key space $\mathcal{K}$, message space $\mathcal{M}$, ciphertext space $\mathcal{C}$, opening space $\mathcal{K}_o$, and franking tag space $\mathcal{T}$.

- $\mathsf{Kg}(1^\lambda) \to k$: This function takes a security parameter and generates a key for use in the ccAE. We omit calls to this function when the key is provided by the calling context.
- $\mathsf{Enc}(k, m) \to (c_1, c_2)$: This function takes in a key $k \in \mathcal{K}$ and message $m \in \mathcal{M}$, and it returns a two part ciphertext (c_1, c_2) where $c_1 \in \mathcal{C}$ and $c_2 \in \mathcal{T}$.
- $\mathsf{Dec}(k, (c_1, c_2)) \to (m, \mathsf{fo})/\perp$: This function takes a key $k \in \mathcal{K}$ and a two part ciphertext $(c_1, c_2) \in \mathcal{C} \times \mathcal{T}$, and it returns either $\perp$ or a message $m \in \mathcal{M}$ and a *franking opening* $\mathsf{fo} \in \mathcal{K}_o$.
- $\mathsf{Ver}(m, \mathsf{fo}, c_2) : 0/1$ This function takes a message $m \in \mathcal{M}$, an opening $\mathsf{fo} \in \mathcal{K}_o$, and a tag $c_2 \in \mathcal{T}$, and it outputs 1 or 0 to indicate acceptance or rejection of a message, respectively.

Our implementation uses the CtE1 scheme of Grubbs et al., which can be built entirely from standard primitives.

Grubbs et al. introduce sender and receiver binding security definitions for committing AE that go beyond the standard CPA-security and ciphertext integrity properties of authenticated encryption and ensure the security of the committing component of the ciphertext. See their work for a detailed description of these definitions [32]. In this work, we use SBadv and RBadv to denote the sender binding and receiver binding advantage of an adversary against a ccAE scheme.

3.2 Our Shared Franking Protocol

This section describes the intuition and design choices for our shared franking scheme. As a starting point, consider a scheme where the client computes a ccAE ciphertext (c_1, c_2) of the message it wishes to send, and secret shares it among the N servers. As an optimization, we set the shares w_i for $i \in \{2, ..., N\}$ to be short seeds which the servers can expand into message shares, and only the share sent to the moderator is a full-length message share. We now show how to build up from this starting point to a shared franking scheme.

Secret sharing a MAC. In order to apply the intuition of message franking to the secret shared setting, we need to compute a MAC on a ccAE ciphertext in a way that is amenable to computation on secret shared data.

One option for building a sharing-friendly MAC would be to adopt a Carter-Wegman MAC [46] or a homomorphic MAC [7]. Unlike widely used standard MACs like HMAC [12], the structure of these MACs allows efficient distributed computation of MACs over secret-shared data using MPC. Unfortunately, this is only the case if every party involved in the MAC computation holds the secret key for the MAC. In the shared franking setting, only the moderator holds the key, so a MAC cannot be computed by simply aggregating computations done by the servers processing a message. We can get around this by using a multiparty computation protocol over an arithmetic circuit that computes the MAC [31, 9, 24, 23], but this adds both communication and computation costs that we wish to avoid.

Instead, we have the moderator compute a MAC on the secret shares of the message rather than directly on the message itself. Since the shares, taken together, allow for reconstruction of the messages, this is equivalent to MACing the message directly. To avoid having the moderator see all the message shares directly, we have each server send the moderator a hash of its input share $w'_i \leftarrow H(w_i)$. The moderator computes a MAC over its own share $[c_2]_1$ of c_2, all the hashes it receives, and the context string ctx for the message being processed. Formally, the moderator computes

$$h \leftarrow (w'_2, ..., w'_N)$$
$$\sigma \leftarrow \mathsf{MAC.Sign}(k_S, ([c_2]_1, h, \mathsf{ctx})).$$

Note that, as is the case in standard message franking, appending a MAC means messages sent through the shared franking scheme will be longer than user-submitted ciphertexts. To accommodate this, other servers generate sufficiently long pseudorandom outputs from their write request seeds to match the length of the moderator's output. The moderator also masks these outputs with randomness generated from a seed provided by the user sending the message.

Reproducing shares. While MACing shares of a message suffices to serve as a MAC on that message, MACing *hashes* of shares of a message poses a problem for the correctness of our scheme. In order to verify this MAC, the recipient of a report needs the exact same shares that were used to generate the MAC tag. But the shares of a message delivered to the message recipient after passing through an MPC-based metadata-hiding communication scheme will almost certainly differ from the ones produced when a sender produces the message.

We get around this problem by ensuring the recipient of a message can reproduce the original shares used by the sender. When sending a message, the randomness for producing each share is generated from a seed r, and the sender includes r in the ciphertext delivered to the platform. Specifically, the sender computes

$$r \xleftarrow{R} \{0,1\}^\lambda$$
$$c = (c_1, c_2) \leftarrow \mathsf{ccAE.Enc}(k_U, (m, r))$$
$$\{s_i\}_{i \in \{1,...,N\}} \leftarrow G(r)$$
$$[c]_1 \leftarrow c \oplus G(s_2) \oplus ... \oplus G(s_N),$$

where G is a PRG with suitable output lengths. Since the message recipient can recover r, it can reproduce the original shares used when the message was sent. These can in turn be used to send the moderator the correct inputs to the MAC produced when it was processing the message, fixing the correctness problem introduced by MACing hashes of message shares instead of the message itself.

Adding accountability. The scheme as sketched thus far suffices to provide confidentiality and unforgeability (via the secret sharing and the MAC+ccAE combination), but accountability can easily be circumvented by a malicious user or server:

- A malicious user could include an incorrect value r' instead of r in the ccAE ciphertext.
- A malicious server could output a partially incorrect secret share so that the MAC σ is corrupted and fails to verify.

In order to achieve accountability, we need to ensure that these kinds of misbehavior can be caught by the receiving user in the Read algorithm, rather than only becoming apparent when Verify fails after a user reads and reports a message.

This problem can be resolved via additional integrity checks by receiving users to ensure messages can be verified by the moderator. In particular, we have the moderator compute a hash σ_c on the same message as the MAC σ, but with σ itself included at the end of the hash input. That is,

$$\sigma_c \leftarrow H'(k_r, [c_2]_1, h, \mathsf{ctx}, \sigma)$$

where the hash function H' is modeled as a random oracle. This serves as a checksum on the values that will be verified if a message is later reported.

The hash σ_c is masked and appended to the moderator's output, and the receiving user verifies σ_c when running Read on a received message. The fact that messages are secret shared means that σ_c is not visible to any party besides the moderator until a message is read.

Observe that when we instantiate this scheme with a MAC that also serves as a PRF, e.g., HMAC [12, 10, 11], the function H' has a pseudorandom input σ known only to the moderator. Since the values of σ and σ_c are secret shared among the servers, an adversary can only introduce additive offsets to them between the time σ_c is generated and checked. Since H' is modeled as a random oracle, an attacker has a negligible chance of offsetting both σ_c and its inputs in such a way that the checksum still verifies. Thus a receiving user can recompute σ_c to verify for itself that σ will be accepted during verification.

3.3 Formal Protocol Description

We formalize our protocol in Construction 3.2 below, including details on message lengths and masking/unmasking that were omitted in the explanation of the protocol above. We present our scheme assuming the servers operate over bit strings using the XOR operation, but the scheme generalizes to additive secret sharing in an arbitrary abelian group.

Construction 3.2. Our N party shared franking protocol Π appears in Figure 2 makes use of the following primitives to send messages of length ℓ.

- A ccAE scheme $\mathsf{ccAE}(\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ver})$ where $\mathsf{Enc} : \mathcal{K} \times \mathcal{M} \to \{0,1\}^{\ell+v_1} \times \{0,1\}^{v_2}$.
- A PRG $G : \{0,1\}^{\lambda} \to \{0,1\}^*$. We abuse notation and truncate $\{0,1\}^*$ to the needed length of the PRG output in each case where the PRG is used. We explicitly specify the length of the PRG output unless it's clear from context.
- Hash functions $H : \{0,1\}^{\lambda'} \to \{0,1\}^{\lambda''}$ and $H' : \{0,1\}^* \to \{0,1\}^{\lambda''}$ modeled as random oracles. The parameters $\lambda', \lambda'' = \mathsf{poly}(\lambda)$ are derived from the security parameter λ.
- A MAC scheme $(\mathsf{Sign}, \mathsf{Verify})$ with tag space $\{0,1\}^{v_3}$ where the Sign algorithm is also a PRF.

We assume that the input ctx to ModProcess has a fixed, publicly-known length denoted by $|\mathsf{ctx}|$.

The correctness of this scheme follows from the correctness of the underlying cryptographic tools used to build it, namely the ccAE encryption scheme and the MAC scheme, as well as the correctness of the underlying metadata-hiding communication scheme. Recall that correctness is defined (Definition 2.2) with respect to a scheme-dependent Deliver$\langle\rangle$ protocol. In many schemes, this will involve some kind of re-randomization or shuffling of the ciphertext shares sent to the servers [8, 36, 27]. Our scheme remains correct regardless of how this re-randomization happens because as long as the correct message is reconstructed at the end, the moderator can get the "original" shares the sender sent to the system by deriving them from r. This means that the correctness of the shared franking scheme only relies on the correctness of the underlying metadata-hiding communication scheme, not on the details of how it manipulates shares.

3.4 Security

We now briefly discuss the various security properties of our scheme. Due to space limitations, proofs for the theorems stated here appear in the full version of this paper.

Confidentiality. The confidentiality of our scheme relies on the fact that the view of any subset of the servers simply consists of secret shares of ciphertexts that the adversary, who does not control all the servers, cannot reconstruct. Since the secret shares are generated by several invocations of the PRG

$\underline{\mathsf{Send}(k_U, m, N):}$
$r \xleftarrow{R} \{0,1\}^\lambda$
$c = (c_1, c_2) \leftarrow \mathsf{ccAE.Enc}(k_U, (m, r))$
$\{s_i\}_{i\in\{1,...,N\}} \leftarrow G(r)$
where $|s_i| = \lambda$ for $i \in \{1,...,N\}$
$[c]_1 \leftarrow c \oplus G(s_2) \oplus ... \oplus G(s_N)$
$w_1 \leftarrow ([c]_1, s_1)$
$w_i \leftarrow s_i$ for $i \in \{2,...,N\}$
output $w_1, ..., w_N$

$\underline{\mathsf{Process}(N, i, w_i)}$
$v_i \leftarrow G(w_i)$
where $|v_i| = \ell + \nu_1 + \nu_2$
$\quad + |\mathsf{ctx}| + \nu_3 + \lambda''$
$w_i' \leftarrow H(w_i)$
output v_i, w_i'

$\underline{\mathsf{ModProcess}(N, k_S, w_1, \mathsf{ctx}, (w_2', ..., w_N')):}$
$([c]_1, s_1) \leftarrow w_1$
$([c_1]_1, [c_2]_1) \leftarrow [c]_1$
$h \leftarrow (w_2', ..., w_N')$
$\sigma \leftarrow \mathsf{MAC.Sign}(k_S, ([c_2]_1, h, \mathsf{ctx}))$
$u_1' \leftarrow G(s_1),\ |u_1'| = |\mathsf{ctx}| + \nu_3 + \lambda''$
$\sigma_c \leftarrow H'([c_2]_1, h, \mathsf{ctx}, \sigma)$
output $([c]_1, (u_1' \oplus (\mathsf{ctx}, \sigma, \sigma_c))$

$\underline{\mathsf{Read}(k_U, N, (v_1, ..., v_N)):}$
$v \leftarrow v_1 \oplus ... \oplus v_N$
$(c_1, c_2, c_3) \leftarrow v$
where $|c_1| = \ell + \nu_1, |c_2| = \nu_2,$
and $|c_3| = |\mathsf{ctx}| + \nu_3 + \lambda''$
$(m, r, \mathsf{fo}) \leftarrow \mathsf{ccAE.Dec}(k_U, (c_1, c_2))$
if $(m, r, \mathsf{fo}) = \perp$, output $\perp$
$\{s_i\}_{i\in\{1,...,N\}} \leftarrow G(r),\ |s_i| = \lambda$ for $i \in \{1,...,N\}$
$(u_i, u_i') \leftarrow G(s_i)$ for $i \in \{2,...,N\}$
where $|u_i| = \ell + \nu_1, |u_i'| = \nu_2 + |\mathsf{ctx}| + \nu_3 + \lambda''$
$u_1' \leftarrow G(s_1),\ |u_1'| = |\mathsf{ctx}| + \nu_3 + \lambda''$
$([c_2]_1, \mathsf{ctx}, \sigma, \sigma_c) \leftarrow (c_2, c_3) \oplus (0^{\nu_2}, u_1') \oplus u_2' ... \oplus u_N'$
$h \leftarrow (H(s_2), ..., H(s_N))$
if $\sigma_c \neq H'([c_2]_1, h, \mathsf{ctx}, \sigma)$, output $\perp$
else output $m, (r, \mathsf{fo}, [c_2]_1, \mathsf{ctx}, \sigma)$

$\underline{\mathsf{Verify}(k_S, N, m, t):}$
$(r, \mathsf{fo}, [c_2]_1, \mathsf{ctx}, \sigma) \leftarrow t$
$\{s_i\}_{i\in\{1,...,N\}} \leftarrow G(r),\ |s_i| = \lambda$ for $i \in \{1,...,N\}$
$h \leftarrow (H(s_2), ..., H(s_N))$
$\mathsf{ver} \leftarrow \mathsf{MAC.Verify}(k_S, ([c_2]_1, h, \mathsf{ctx}), \sigma)$
$(u_i, u_i') \leftarrow G(s_i)$ for $i \in \{2,...,N\}$
where $|u_i| = \ell + \nu_1, |u_i'| = \nu_2$
$c_2 \leftarrow [c_2]_1 \oplus u_2' \oplus ... \oplus u_N'$
$\mathsf{ver}' \leftarrow \mathsf{ccAE.Ver}((m, r), \mathsf{fo}, c_2)$
if $\mathsf{ver} = 0 \ \vee\ \mathsf{ver}' = 0$, output $\perp$; else output ctx

Figure 2: Our shared franking scheme (Construction 3.2).

G, security primarily reduces to the security of G. The only messages sent in the confidentiality experiment that are not secret shared are outputs of queries to H, so we additionally bound the probability that the adversary queries the random oracle at any point that overlaps with a query made by an honest server, ensuring that these random oracle outputs cannot help the adversary.

Theorem 3.3. *If we assume that G is a secure PRG and model H as a random oracle, then our shared franking scheme (Construction 3.2) satisfies sharing confidentiality (Definition A.1).*

In particular, for every confidentiality adversary $\mathcal{A}$ that attacks Π and makes at most Q_{RO} random oracle queries to H, there exists a PRG distinguishing adversary $\mathcal{B}$ such that for every λ, N, Q, and $N_M < N$,

$$\mathsf{CONFadv}(\mathcal{A}, \Pi, \lambda, N, Q, N_M) \leq 4Q \cdot \mathsf{PRGadv}(\mathcal{B}, G, \lambda) + 2Q \cdot (N - N_M) \cdot \frac{Q_{\mathsf{RO}}}{2^{\lambda'}}.$$

Accountability and unforgeability. The accountability of our scheme follows from the sender binding of the ccAE scheme and the hardness of forging a correct tag σ_c. Sender binding requires that an adversary cannot send a message (m, r) that successfully decrypts but does not pass ccAE.Ver. Modeling H' as a random oracle, an information-theoretic argument can be made that an adversary forges a valid σ_c with at most negligible probability in λ.

Unforgeability is ensured by the MAC σ and the ccAE verification checks in Verify. In order for a forged message to pass MAC verification, the adversary must either forge a new MAC tag, or find a collision in H, because an output of H is part of the message being MACed. If the adversary instead finds a new opening for the ccAE ciphertext, this would break the receiver binding of the ccAE scheme.

Theorem 3.4. *Assuming we model H' as a random oracle, that* MAC *is a correct MAC where* MAC.Sign *is also a PRF, and that* ccAE *satisfies sender binding, our shared*

franking scheme Π *(Construction 3.2) has accountability (Definition A.2).*

In particular, for every accountability adversary $\mathcal{A}$ *that attacks our protocol* Π*, there exists collision-finding adversary* $\mathcal{B}$*, a PRF adversary* $\mathcal{C}$*, and sender binding adversary* $\mathcal{D}$ *such that for every* λ, N, Q

$$\begin{aligned}&\mathsf{ACCTadv}(\mathcal{A},\Pi,\lambda,N,Q)\\&\quad\leq \mathsf{CRadv}(\mathcal{B},H',\lambda)+\mathsf{PRFadv}(\mathcal{C},\mathsf{MAC.Sign},\lambda)\\&\quad+\mathsf{SBadv}(\mathcal{D},\mathsf{ccAE},\lambda)+\mathsf{negl}(\lambda).\end{aligned}$$

Theorem 3.5. *Assuming that* MAC *is an existentially unforgeable MAC, that H is a collision-resistant hash function, and that* ccAE *satisfies receiver binding, our shared franking scheme* Π *(Construction 3.2) has unforgeability (Definition A.3).*

In particular, for every unforgeability adversary $\mathcal{A}$ *that attacks our protocol* Π*, there exist unforgeability, collision-finding, and receiver binding adversaries* $\mathcal{B}$*,* $\mathcal{C}$*, and* $\mathcal{D}$ *such that for every* λ, N

$$\begin{aligned}&\mathsf{FORGadv}(\mathcal{A},\Pi,\lambda,N)\leq\\&\mathsf{MACadv}(\mathcal{B},\mathsf{MAC},\lambda)+\mathsf{CRadv}(\mathcal{C},H,\lambda)+\mathsf{RBadv}(\mathcal{D},\mathsf{ccAE},\lambda).\end{aligned}$$

Deniability. As mentioned in Section 2.2, our scheme must achieve three kinds of deniability: universal deniability, receiver compromise deniability, and moderator compromise deniability. Since we do not formalize these notions in our setting, we sketch the forgery algorithms required to satisfy each definition. All three forms of deniability rely only on the deniability of the underlying ccAE encryption scheme and MAC to an adversary who does not know their secret keys. In particular, a formal proof would rely on an anonymity property of our encryption and MAC schemes – that two ciphertexts or MACs produced by different random keys should appear indistinguishable to an adversary who does not know the keys. Our implementation uses schemes (AES-GCM and HMAC-SHA256) that satisfy a pseudorandomness property that implies the required anonymity definition.

To produce a universal forgery – one that appears valid to a distinguisher who does not have the key k_U used to encrypt/read a message or the moderator key k_S – the forger can sample random keys k'_U and k'_S and use them to run the Send and Process, and ModProcess operations. The outputs of these operations will appear indistinguishable from the results of the same process using the real keys k_U and k_S to an adversary who does not know the keys. Receiver compromise deniability requires a forgery algorithm that has access to the key k_U used to send/receive a message but not the moderator key k_S. The forger for this definition behaves just like the universal forgery forger, but it uses the keys k_U, k'_S instead of generating a random k_U. Finally, the forger for moderator compromise deniability, who has access to k_S but not k_U, uses a random k'_U but the real k_S in generating its forgery.

4 Integration with Metadata-Hiding Communication Schemes

This section discusses how our protocol from Section 3 can be integrated into various kinds of metadata-hiding communication platforms that rely on secret sharing.

4.1 Schemes based on MPC

Our shared franking protocol integrates directly with metadata-hiding communication schemes based on MPC that use additive secret sharing. In an MPC-based scheme, users send shares of messages to the servers. The servers use MPC to shuffle the shares and prepare them for delivery to their intended recipients. To integrate shared franking into these schemes, users send messages to the servers via the Send algorithm of the shared franking scheme, incorporating any client-side preprocessing of the messages before secret sharing. Then the servers run the shared franking protocol *before* they run the underlying message processing and delivery algorithm. After a message's shares are made available to users, a message recipient retrieves the relevant shares according to the underlying messaging protocol and then recovers the message via our read algorithm. The report algorithm does not make use of the metadata-hiding communication functionality and therefore does not need to be integrated directly into the underlying messaging system. Recent schemes in this category that use additive secret sharing include MCMix [8], Ruffle [4], and Clarion [27], which is based on the secret shared shuffle work of Chase et al. [16].

Schemes based on threshold secret sharing. Our approach can be integrated with schemes that use threshold secret sharing [42], such as Asynchromix [36] or RPM [35], in a limited way. If a fixed subset of the servers processes each batch of messages, and this subset is known to the moderator and the message receiver, our scheme can be used. However, this means using shared franking would require giving up some of the security and robustness properties of these schemes. For example, Asynchromix aims to achieve robustness against malicious servers who may stop responding to messages or go offline during protocol execution. On the other hand, we assume that servers are trusted for availability but not for privacy. This means that while shared franking as described here can be useful for protocols that use threshold sharing to allow a subset of a potentially large number of servers to process user messages, further work is needed to strengthen the security guarantees of shared franking to handle robustness against malicious servers for availability in addition to privacy.

4.2 Schemes Based on DC-nets

A number of recent schemes combine a DC-net approach [17, 21, 47, 22], or a DC-net augmented with distributed point func-

tions (DPFs) [30, 14, 15] to improve scalability [20, 28, 39, 6]. Our shared franking scheme does not apply to these schemes directly, but we can adapt the scheme to work in this setting too. This results in higher computation and communication costs between servers, but not between users and the servers.

Background: Private writing with DC-nets and DPFs. Before describing our scheme, we briefly summarize aspects of DC-nets and DPFs that will be relevant for our purposes. We will be focusing on the setting where the work of the DC-net is outsourced to a small number of servers rather than the original DC-net setting where a number of clients run the DC-net among themselves [17]. Since the most widely-used and efficient DPF construction only applies for two servers, we focus on the two-server setting.

This family of schemes generally has servers maintain shares D_A and D_B of a database $D = D_A + D_B$, where the database is a vector of n elements of some finite abelian group $\mathbb{G}$, i.e., $D \in \mathbb{G}^n$. To privately write a message m to an entry j in the database, a client prepares secret shared vectors x_A, x_B such that $x = x_A + x_B = m \cdot e_j$, where e_j represents the jth standard basis vector: a length-n vector of all zeros except for a 1 in the jth position. To process a write, the servers compute

$$D_A \leftarrow D_A + x_A \qquad D_B \leftarrow D_B + x_B.$$

Since the shares x_A, x_B appear random to the servers, they cannot tell which entry of the shared database D is modified by a given write. After processing many messages, the servers merge their shares D_A and D_B to recover the modified database. This allows a number of clients to write messages into the database without the servers learning who wrote which message. To protect against denial of service attacks by malicious users, clients generally include lightweight proofs alongside their shares x_A, x_B to convince the servers that they are sending shares of a vector with only one non-zero entry [22, 20, 28].

The approach above is not efficient because it requires a client to send n group elements to each server to write a message. DPFs are a mechanism that "compress" the vectors x_A and x_B into a much more concise representation that takes advantage of the fact that most entries of these vectors are really shares of zero. A DPF allows a client to create two small *function shares* f_A and $f_B : \{1,...,n\} \to \mathbb{G}$ such that $f_A(j) = x_{A,j}$ and $f_B(j) = x_{B,j}$. Now clients can send a small function share to each server, and the servers can recover x_A, x_B on their own by evaluating $f_A(j), f_B(j)$ for $j \in \{1,...,n\}$. Importantly, the representations of the function shares f_A and f_B are concise, i.e., they require fewer bits to represent than x_A and x_B. Observe that this operation slightly *increases* computation costs on the servers, but it dramatically lowers the communication cost on the client, which is more likely to be a bottleneck in a communication-constrained setting.

Integration challenges. Let us see why we cannot integrate shared franking into DC-net schemes the same way we did with MPC. Shared franking involves a step where the servers append a share of additional data c_3 to the messages sent by users. But in a DC-net scheme, the servers do not know which entry of a database a user has modified. If the servers append shares of c_3 to every entry in the database every time a user sends a message, they will end up increasing the length of each entry until it is linear in the number of messages sent. This is clearly impermissible from a performance perspective, as it dramatically increases server storage costs and requires clients to download gigantic message shares when recovering a message.

In order to make shared franking compatible with a DC-net scheme, we need a way for the servers to *blindly* append shares of c_3 to the correct entry in D, while adding shares of 0 to other entries. We achieve this with a slight modification to the structure of messages sent by users and with the addition of a small MPC. In particular, the servers will compute a multiplication on secret-shared values for each entry in D.

Shared franking for DC-nets. First, if we are to use shared franking in the DC-net setting, we need to modify the behavior of Send so that instead of creating additive shares of c_1, c_2, it creates shares of an n-entry vector x such that every entry of x has the form (c_1, c_2, c_3) where $c_1 \in \{0,1\}^{\ell+\nu_1}$, $c_2 \in \{0,1\}^{\nu_2}$, and $c_3 \in \mathbb{F}_q$ where $\mathbb{F}_q$ is a prime order finite field. When the user wishes to write a message into the j^*th entry on the servers, the jth entry of x contains

$$(c_1, c_2, 1 \in \mathbb{F}_q) \text{ if } j = j^*, \text{ and}$$

$$(0^{\ell+\nu_1}, 0^{\nu_2}, 0 \in \mathbb{F}_q) \text{ otherwise.}$$

Note that c_1, c_2 remain bit strings, whereas the $0/1$ values appended to each row are interpreted as elements of a finite field and are secret shared as elements of the field, not bit strings. Thus the shares of the jth entry in the secret shared vector x that are sent to the ith server have the form $[c_{1,j}]_i, [c_{2,j}]_i, [b_j]_i$, where $([c_{1,j}]_i, [c_{2,j}]_i) \in \{0,1\}^{\ell+\nu_1+\nu_2}$ and $[b_j]_i \in \mathbb{F}_q$ is a share of zero or one. As is the case in the standard shared franking scheme, all servers except the moderator can receive a single seed s_i. Seed s_i can in turn be expanded into seeds s_{ij}, $j \in \{1,...,n\}$, one for each entry $x_j \in x$.

Putting aside distinctions between bit strings and field elements for a moment, when servers process messages, they run Process and ModProcess separately for each entry x_j, $j \in \{1,...,n\}$. That is, ignoring the role of $[b_j]$ for now, the non-moderator servers run

$$\mathsf{Process}(N, i, s_{ij}) \text{ for each } j \in \{1,...,n\},$$

and the moderator runs

$$\mathsf{ModProcess}(N, k_S, s_{1j}, \mathsf{ctx}, (w'_{2j}, ..., w'_{Nj})) \text{ for each } j \in \{1,...,n\}.$$

The only change we need to make in the computation of these functions is that we expect the components the servers

append to the end of the shares of the ccAE ciphertext, which merge to form c_3 in Read, are an element of $\mathbb{F}_q$ instead of $\{0,1\}^{|\mathsf{ctx}|+\nu_3+\lambda''}$. This means that some portion of the output of the PRG G needs to be interpreted as an element of $\mathbb{F}_q$, as does the tuple $(\mathsf{ctx},\sigma,\sigma_c)$. This is easily accomplished if we set q close to a power of 2, as these values are random or random-looking values (ctx is not necessarily random but can be encrypted by the platform). This means both the relevant PRG outputs and $(\mathsf{ctx},\sigma,\sigma_c)$ will already be a valid representation of an integer mod q with all but negligible probability. If the string $(\mathsf{ctx},\sigma,\sigma_c)$ is too long for a single element of $\mathbb{F}_q$, we can either use a larger choice of q or, for performance reasons, represent it as multiple elements of $\mathbb{F}_q$.

Blindly appending shares. After processing each row in x as described above, the servers will hold shares

$$([c_{1,j}],[c_{2,j}],[c_{3,j}]) \text{ for each } j \in \{1,\ldots,n\}.$$

Since most of the entries of x are all zeros, we will have that

$$(c_{1,j},c_{2,j}) = 0^{\ell+\nu_1+\nu_2} \text{ for } j \neq j^*.$$

In order to append the correct tag c_{3,j^*} at the end of the j^*th entry and to not interfere with the tag at the end of other rows when updating the database D, we need to replace $[c_{3,j}]$ with $[c'_{3,j}]$, where $c'_{3,j} = c_{3,j}$ when $j = j^*$ and $c'_{3,j} = 0 \in \mathbb{F}_q$ otherwise. This is where we will use the shares $[b_j]$.

For each entry j, the servers will do a small MPC to compute the product $[c'_{3,j}] \leftarrow [b_j] \cdot [c_{3,j}]$, using *Beaver triples* [9] to multiply secret-shared values. Since $b_j = 1$ if and only if $j = j^*$, the products $c'_{3,j}$ will satisfy the requirement that $c'_{3,j} = c_{3,j}$ when $j = j^*$ and $c'_{3,j} = 0$ otherwise. After a message-independent preprocessing phase among the servers, each Beaver multiplication requires a small amount of communication and computation among the servers. The communication and computation for all the multiplications can be done in parallel. This increases costs on the servers but has no effect on client-side performance. We do not describe the widely-used Beaver multiplication technique here, but it is summarized in many excellent online resources, e.g., [43, 48].

5 Implementation and Evaluation

We implemented the shared franking construction described in Section 3. Our implementation is written in C and uses OpenSSL for standard cryptographic primitives.

We instantiate our PRG G with AES in CTR mode, and our MAC scheme with HMAC-SHA256. Our ccAE scheme, following the CtE1 scheme of Grubbs et al. [32], requires an AEAD scheme and a commitment scheme. We use AES-GCM for the AEAD scheme and HMAC-SHA256 for the commitment, where the commitment randomness serves as the HMAC key. Field operations are computed modulo $2^{256} - 189$, the largest 256-bit prime [3]. OpenSSL supports hardware acceleration for evaluating AES, so our implementation benefits from this feature by default. Finally, we instantiate our hash functions H, H' with SHA256. Since all the metadata-hiding communication systems we have discussed require sending fixed-length messages to avoid leaking size information about messages, SHA256 will be indifferentiable from a random oracle in this restricted setting [37, 18].

We evaluated our implementation using a machine with a 11th Gen Intel(R) Core(TM) i7-11700K @ 3.60G processor running Ubuntu 20.04. We evaluated our scheme on message lengths of up to 1,020 Bytes in 20 Byte increments, and we varied the number of servers from 2 to 10 for each message length. We fixed the length of the ctx string used by the moderator at 32 bytes for all our experiments. All reported results for our scheme are averages taken over 1,000 runs.

Appendix B briefly reports the results from repeating our evaluation on GCP to get performance numbers for a lower-end server. There, we also use GCP to confirm that the performance characteristics of our scheme remain similar when run in a networked setting.

Evaluation results. Figure 3 shows the running time of each operation in our shared franking scheme as the message size increases from 40 Bytes to 1,020 Bytes. All operations take under ten microseconds, with the Send and Read exhibiting the most rapid increase in computation time as the message length increases. Their cost comes primarily from producing the ccAE encryption of the message, which includes both encrypting the message and computing a commitment over it, both of which depend on the entire length of the message. Verification of messages also depends on the entire message length because the verifier needs to verify the ccAE tag on the message. Finally, the cost of processing a message, either for the moderator or for any other server, is extremely low and does not require any cryptographic or arithmetic operations that depend on the length of the message, other than expanding a PRG seed to the required length, which benefits from hardware acceleration.

Increasing the number of servers results in behavior similar to increasing the message length, as seen in Figure 4. Once again, Send and Read exhibit the highest costs and increase in costs because they primarily consist of operations that depend on the message length as well as the number of servers. Since processing a message on a non-moderating server is identical as the number of servers increases and processing a message as the moderator is almost identical (it only involves MACing a message 32 bytes longer for each server), these operations are largely unaffected by an increase in the number of servers.

We conclude that shared franking adds very little computational overhead to metadata-hiding communication schemes, with the majority of the cost incurred by clients or in the report verification process, neither of which affect the critical path of message delivery on the server side. The communication overhead is also small, increasing the data sent from the client to the moderator server by 124 Bytes, and requiring only 16

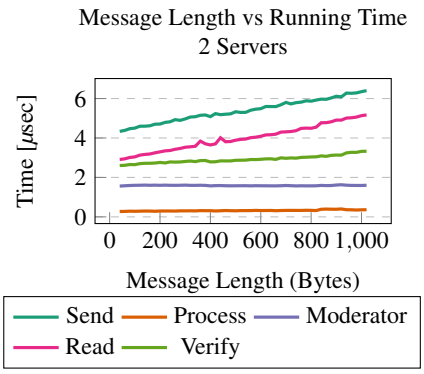

Figure 3: Running time as message length increases. Server-side operations on the critical path for message delivery run the fastest and do not noticeably increase in cost as message lengths increase.

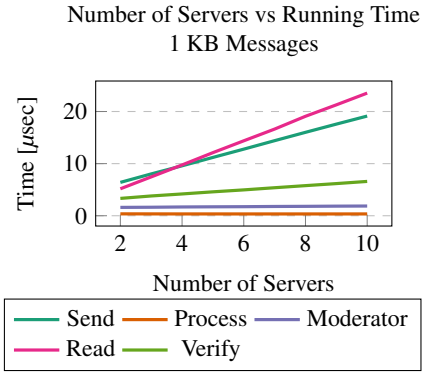

Figure 4: Running time of each operation as number of servers increases. Server-side operations on the critical path for message delivery run the fastest and do not noticeably increase in cost with the number of servers.

	Send	Read	Verify
AMF	489B	489B	489B
Hecate	380B	484B	380B
Our Work	124B	204B	144B
Plain Franking	92B	156B	128B

Figure 5: Comparison of communication costs for clients in AMF [44], Hecate [33], shared franking (our work), and plain message franking [29]. Communication costs are the overhead beyond the cost of sending the message itself. Costs for send/read in shared franking are the cost per server and do not include the additional 32 Byte hash sent from each processing server to the moderator. The additional communication cost for Send for each additional server in shared franking is 16B.

bytes sent to each other server, regardless of message length. The data retrieved from each server to read a message is the message length plus 204 Bytes, and a report consists of 144 Bytes in addition to the message itself.

We conclude the evaluation of our scheme by estimating the cost of the extension to DC-nets discussed in Section 4.2. This scheme adds a Beaver multiplication for each of the n elements of the DC-net. The Beaver multiplication adds one communication round to the protocol, as well as a preprocessing phase to generate the Beaver triples to aid in fast online multiplication. On our test machine, online evaluation of a single Beaver multiplication for 128 bits of data, which is more than sufficient for our scheme, takes 0.75μs (average of 1M runs, written in Go). Our scheme requires the cost of the Beaver multiplication and the normal processing protocol to be repeated n times to process each message. The small additional per-multiplication cost means that while shared franking remains an excellent choice for DC-net deployments with small anonymity sets – e.g., anonymous chat for participants in a class, seminar, or online community of limited size – large, general-purpose messaging platforms may opt for a different solution.

Comparison to prior work. We are aware of two prior works that extend message franking to the metadata-hiding setting: the asymmetric message franking (AMF) of Tyagi et al. [44], and the Hecate scheme of Issa et al. [33]. AMF can be added as a drop-in solution on top of *any* metadata-hiding messaging scheme where message delivery/retrieval only requires a single message from the client to the server(s), but it uses expensive proofs of knowledge to achieve this. Hecate achieves significantly improved performance, but requires an additional message-independent round trip between a user and the moderator before sending a message. AMF and Hecate both also allow for third-party moderation, which our scheme and plain message franking do not support. In the setting of secret-sharing based schemes, our solution is a drop-in approach similar to AMF, but it also outperforms both AMF and Hecate. We view the three works as exploring different areas in the design space of message franking, and quantitative comparisons between them are not necessarily apples-to-apples. AMF is the most general and the most costly. Hecate matches the generality of AMF and significantly improves performance while changing the model to add another round trip (which can be run in a message-independent preprocessing phase). Our performance improves on both prior schemes, but it is restricted to schemes based on secret sharing. We believe this is a valuable point in the design space given the large body of work that uses secret sharing approaches to hide metadata.

	Hecate	Shared Franking	Plain Franking
Preprocess	29.5μs	N/A	N/A
Send	16.4μs	6.4μs	3.7μs
Process	15.7μs	1.6μs or 0.4μs	1.1μs
Read	100.1μs	5.2μs	2.1μs
Verify	102.8μs	3.3μs	2.6μs

Figure 6: Computation time for various operations in Hecate [33], shared franking (our work), and plain message franking [29] on 1 KB messages. We report numbers for shared franking with two servers, and include the times for both the moderator and non-moderator servers to process messages.

To measure how shared franking compares quantitatively to prior work, we re-ran the benchmarks for Hecate and AMF (using the original Hecate implementation and a faster Rust implementation of AMF [38]) on our evaluation machine. We also implemented an instantiation of Facebook-style "plain" message franking [29] using the same cryptographic primitives and implementations as our scheme so we can compare to techniques deployed in practice.

Our instantiation of shared franking reduces the client communication overhead to process a message and report it compared to both AMF and Hecate, and approaches the report size of messages reported in a plain franking scheme. Figure 5 summarizes the communication costs of each scheme.

Figure 6 compares the performance of shared franking with Hecate and plain franking. Compared to Hecate, shared franking reduces the computation cost to send a message by 2.6× and the cost to read a message or verify a reported message by 19× and 31.0× respectively. The reduction in total time to process a message on the server, including time on both the servers in our implementation, is 22.6×. If we instantiate Hecate with a separate message-independent preprocessing phase per message, the online performance improvement of using our scheme is 7.9×. Our performance numbers are also more than an order of magnitude faster than those reported for AMF, which took about 230μs for each of sending, reading, and verifying a message.

Our performance improvements come from the fact that shared franking does not use any public key cryptography, whereas Hecate makes use of signatures and AMF uses zero knowledge proofs of knowledge. We can avoid these more expensive tools by taking advantage of the fact that, in the setting of secret-shared messages, the servers know the identity of message senders when they send a shared message. Thus we can use lightweight techniques more akin to those of standard message franking for E2EE messaging rather than the public key tools previously used in metadata-hiding schemes.

6 Conclusion and Future Work

We have shown how to add lightweight abuse reporting on top of metadata-hiding communication platforms based on secret sharing. Our scheme only requires symmetric cryptographic primitives and can be adapted to a broad family of communication platforms that use diverse underlying techniques. Our results show that relying on some existing structure provided by a communication platform – in this case the presence of secret sharing – can dramatically reduce the cost of adding support for abuse reporting functionality.

While this work focuses on building shared franking schemes to enable reporting abusive messages, the same technique can potentially be applied to build related private moderation capabilities as well. Recent works aimed at combating misinformation and disinformation on private messaging platforms have studied the problem of message *traceback* or *source tracking* [45, 40, 34, 33]. In these works, the goal of a report is not to identify the immediate sender of a message, but rather to find the the user who originated a piece of widely forwarded misinformation. An interesting opportunity for future work lies in extending our ideas to these kinds of schemes. In principle, our approach seems applicable to a broad class of protocols where the platform does not need to know the identity of the message recipient at the time it is processing a message. The path traceback scheme of Tyagi et al. [45] and the tree-linkable source tracking of Peale et al. [40] fall into this category, so it may be possible to efficiently extend these schemes to work in the secret shared setting as well.

Acknowledgments

I would like to thank the anonymous reviewers and shepherd for their helpful comments and suggestions that strengthened the results in this paper.

This material is based upon work supported by the National Science Foundation under Grant No. 2234408, as well as gifts from Google and Cisco. GCP provided credits used to perform part of the evaluation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Analytics in exposure notifications express: Faq. https://github.com/google/exposure-notifications-android/blob/master/doc/enexpress-analytics-faq.md, 2021. Accessed 5/1/2023.

[2] Exposure notification privacy-preserving analytics (enpa) white paper. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf, 2021. Accessed 5/1/2023.

[3] Primes just less than a power of two. https://t5k.org/lists/2small/, 2021.

[4] Pranav Shriram A, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Ruffle: Rapid 3-party shuffle protocols. *PoPETs*, 2023.

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

[6] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder: MPC based scalable and robust anonymous committed broadcast. 2020.

[7] Shweta Agrawal and Dan Boneh. Homomorphic macs: Mac-based integrity for network coding. In *Applied Cryptography and Network Security, ACNS*, 2009.

[8] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. Mcmix: Anonymous messaging via secure multiparty computation. In *USENIX Security*, 2017.

[9] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[10] Mihir Bellare. New proofs for NMAC and HMAC: security without collision-resistance. In *CRYPTO*, 2006.

[11] Mihir Bellare. New proofs for NMAC and HMAC: security without collision resistance. *J. Cryptol.*, 28(4):844–878, 2015.

[12] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, 1996.

[13] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography (version 0.5, Chapter 9)*. 2017. https://cryptobook.us.

[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, 2015.

[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS*, 2016.

[16] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret shared shuffle. 2020.

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

[18] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-damgård revisited: How to construct a hash function. In *CRYPTO*, 2005.

[19] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, 2017.

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

[21] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *ACM CCS*, 2010.

[22] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in verdict. In *USENIX Security*, 2013.

[23] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, 2013.

[24] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[25] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In *CRYPTO*, 2018.

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

[27] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. 2022.

[28] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. 2021.

[29] Facebook. Messenger secret conversations technical whitepaper. https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf, July 2016.

[30] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, 2014.

[31] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, 1987.

[32] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *CRYPTO*, 2017.

[33] Rawane Issa, Nicolas Alhaddad, and Mayank Varia. Hecate: Abuse reporting in secure messengers with sealed sender. *IACR Cryptol. ePrint Arch.*, 2021.

[34] Linsheng Liu, Daniel S. Roche, Austin Theriault, and Arkady Yerukhimovich. Fighting fake news in encrypted messaging with the fuzzy anonymous complaint tally system (FACTS). *IACR Cryptol. ePrint Arch.*, 2021.

[35] Donghang Lu and Aniket Kate. RPM: Robust anonymity at scale. *IACR Cryptol. ePrint Arch.*, 2022.

[36] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew K. Miller. Honeybadgermpc and asynchromix: Practical asynchronous MPC and its application to anonymous communication. In *ACM CCS*, 2019.

[37] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *TCC*, 2004.

[38] Sanketh Menda and Michael Rosenberg. amaze. https://github.com/sgmenda/amaze, 2022.

[39] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-bandwidth anonymous broadcast. In *NSDI*, 2022.

[40] Charlotte Peale, Saba Eskandarian, and Dan Boneh. Secure complaint-enabled source-tracking for encrypted messaging. In *ACM CCS*, 2021.

[41] Phillip Rogaway. Authenticated-encryption with associated-data. In *ACM CCS*, 2002.

[42] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[43] Nigel Smart and Peter Scholl. FHE-MPC notes. https://homes.esat.kuleuven.be/~nsmart/FHE-MPC/Lecture8.pdf, November 2011.

[44] Nirvan Tyagi, Paul Grubbs, Julia Len, Ian Miers, and Thomas Ristenpart. Asymmetric message franking: Content moderation for metadata-private end-to-end encryption. In *CRYPTO*, 2019.

[45] Nirvan Tyagi, Ian Miers, and Thomas Ristenpart. Traceback for end-to-end encrypted messaging. In *ACM CCS*, 2019.

[46] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.

[47] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *OSDI*, 2012.

[48] David Wu. Lecture notes on secret sharing and Beaver triples. https://crypto.stanford.edu/cs355/18sp/lec7.pdf, 2018.

A Formal Security Definitions

Definition A.1 (Sharing Confidentiality)**.** We define the shared franking confidentiality experiment

$$\mathsf{CONF}[\mathcal{A},\Pi,\lambda,N,Q,N_M,b]$$

with respect to a stateful adversary $\mathcal{A}$ who makes at most Q queries to an oracle $\mathcal{O}_{\mathsf{honProcess}}$, shared franking scheme Π, security parameter λ, number of servers $N \in \mathbb{N}$, number of malicious servers N_M such that $N_M < N$, and a bit $b \in \{0,1\}$. The experiment proceeds as follows. We will use $\mathcal{M}$ to denote the message space for Send.

$\underline{\mathsf{CONF}[\mathcal{A},\Pi,\lambda,N,Q,N_M,b]:}$
$M \leftarrow \mathcal{A}(\lambda,N,N_M)$
where $M \subset \{1,...,N\}, |M| = N_M$(else output 0)
$b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{honProcess}}}(\lambda,N,M)$
if $b' = b$: output 1; else : output 0
$\underline{\mathcal{O}_{\mathsf{honProcess}}(k_U,m_0,m_1):}$
if $b = 0$: $(w_1,...,w_N) \leftarrow \mathsf{Send}(k_U,m_0,N)$
else : $(w_1,...,w_N) \leftarrow \mathsf{Send}(k_U,m_1,N)$
$\mathsf{malw} \leftarrow (w_i)_{i\in M}$
if $1 \notin \mathsf{M}$: output $(\mathsf{malw},\perp)$
else :
  $v_i, w_i' \leftarrow \mathsf{Process}(N,i,w_i)$ for $i \in \{2,...,N\} \setminus M$
  $\mathsf{honw}' \leftarrow (w_i')_{i\in\{2,...,N\}\setminus M}$
  output $(\mathsf{malw},\mathsf{honw}')$

We define the *confidentiality advantage* of $\mathcal{A}$ as

$$\begin{aligned}&\mathsf{CONFadv}(\mathcal{A},\Pi,\lambda,N,Q,N_M)\\ &\quad= \Big|\Pr\big[\mathsf{CONF}[\mathcal{A},\Pi,\lambda,N,Q,N_M,0]=1\big]\\ &\qquad\quad - \Pr\big[\mathsf{CONF}[\mathcal{A},\Pi,\lambda,N,Q,N_M,1]=1\big]\Big|.\end{aligned}$$

We say that a shared franking scheme Π satisfies $\varepsilon_{\mathsf{conf}}$-*confidentiality* if for all efficient adversaries $\mathcal{A}$, security parameters $\lambda \in \mathbb{N}$,and $N \in \mathbb{N}$, it holds that

$$\mathsf{CONFadv}(\mathcal{A},\Pi,\lambda,N,Q,N_M) \le \varepsilon_{\mathsf{conf}}(\lambda).$$

We say that Π has *confidentiality* if $\varepsilon_{\mathsf{conf}}(\lambda) \le \mathsf{negl}(\lambda)$.

Definition A.2 (Accountability)**.** We define the accountability experiment $\mathsf{ACCT}[\mathcal{A},\Pi,\lambda,N,Q]$ with respect to an adversary $\mathcal{A}$ who makes at most Q queries to an oracle $\mathcal{O}_{\mathsf{verification}}$, shared franking scheme Π, security parameter λ, and number of servers $N \in \mathbb{N}$. The experiment is defined as follows.

$\underline{\mathsf{ACCT}[\mathcal{A},\Pi,\lambda,N,Q]:}$
$\mathsf{win} \leftarrow 0$
$k_S \xleftarrow{R} \mathcal{K}_S$
$\mathcal{A}^{\mathcal{O}_{\mathsf{frank}},\mathcal{O}_{\mathsf{verification}}}(\lambda,N)$
output win
$\underline{\mathcal{O}_{\mathsf{frank}}(w_1,w_2',...,w_N',\mathsf{ctx}):}$
$v_1 \leftarrow \mathsf{ModProcess}(N,k_S,w_1,\mathsf{ctx},(w_2',...,w_N'))$
output v_1
$\underline{\mathcal{O}_{\mathsf{verification}}(k_U^*,w_1^*,v_2^*,...,v_N^*,w_2',...,w_N',\mathsf{ctx}^*):}$
$v_1^* \leftarrow \mathsf{ModProcess}(N,k_S,w_1^*,\mathsf{ctx}^*,(w_2',...,w_N'))$
$\mathsf{res} \leftarrow \mathsf{Read}(k_U^*,N,(v_1^*,...,v_N^*))$
output 0 if $\mathsf{res} = \perp$
$(m^*,t^*) \leftarrow \mathsf{res}$
$\mathsf{ctx}^{**} \leftarrow \mathsf{Verify}(k_S,m^*,t^*)$
$\mathsf{win} \leftarrow 1$ if $\mathsf{ctx}^{**} = \perp \vee \mathsf{ctx}^{**} \neq \mathsf{ctx}^*$
output win, res

We define the *accountability advantage* of $\mathcal{A}$ as

$$\mathsf{ACCTadv}(\mathcal{A},\Pi,\lambda,N,Q) = \Pr\big[\mathsf{ACCT}[\mathcal{A},\Pi,\lambda,N,Q]=1\big],$$

and we say that the shared franking scheme Π satisfies $\varepsilon_{\mathsf{acct}}-$*accountability* if for all efficient adversaries $\mathcal{A}$, security parameters $\lambda \in \mathbb{N}$, and $N \in \mathbb{N}$, it holds that

$$\mathsf{ACCTadv}(\mathcal{A},\Pi,\lambda,N,Q) \le \varepsilon_{\mathsf{acct}}(\lambda).$$

We say that Π has *accountability* if $\varepsilon_{\mathsf{acct}}(\lambda) \le \mathsf{negl}(\lambda)$.

Definition A.3 (Unforgeability)**.** We define the unforgeability experiment $\mathsf{FORG}[\mathcal{A},\Pi,\lambda,N]$ with respect to an adversary $\mathcal{A}$, shared franking scheme Π, security parameter λ, and number of servers $N \in \mathbb{N}$. The experiment is defined as follows.

$\underline{\mathsf{FORG}[\mathcal{A},\Pi,\lambda,N]:}$
$\mathsf{win} \leftarrow 0$
$k_S \xleftarrow{R} \mathcal{K}_S$
$T \leftarrow \{\}$
$\mathcal{A}^{\mathcal{O}_{\mathsf{send}},\mathcal{O}_{\mathsf{verification}}}(\lambda,N)$
output win
$\underline{\mathcal{O}_{\mathsf{send}}(k_U,w_1,v_2,...,v_N,w_2',...,w_N',\mathsf{ctx}):}$
$v_1 \leftarrow \mathsf{ModProcess}(N,k_S,w_1,\mathsf{ctx},(w_2',...,w_N'))$
$(m',t) \leftarrow \mathsf{Read}(k_U,N,(v_1,...,v_N))$
if $(m',t) = \perp$: output $\perp$
$\mathsf{ctx}' \leftarrow \mathsf{Verify}(k_S,m',t))$
if $\mathsf{ctx}' = \perp$: output $\perp$
$T \leftarrow T \cup \{(m',\mathsf{ctx}',t)\}$
output v_1
$\underline{\mathcal{O}_{\mathsf{verification}}(m^*,t^*):}$
$\mathsf{ctx}^* \leftarrow \mathsf{Verify}(k_S,m^*,t^*)$

	Hecate	Shared Franking	Plain Franking
Preprocess	61.3μs	N/A	N/A
Send	36.1μs	30.1μs	21.6μs
Process	32.0μs	15.7μs or 2.2μs	12.9μs
Read	215.8μs	27.7μs	17.6μs
Verify	222.9μs	31.2μs	27.8μs

Figure 7: Computation time for various operations in Hecate [33], shared franking (our work), and plain message franking [29] on 1 KB messages. We report numbers for shared franking with two servers, and include the times for both the moderator and non-moderator servers to process messages.

$$\mathsf{win} \leftarrow 1 \text{ if } \mathsf{ctx}^* \neq \perp \wedge (m^*, \mathsf{ctx}^*, t^*) \notin T$$
$$\text{output } \mathsf{win}, \mathsf{ctx}^*$$

We define the *unforgeability advantage* of $\mathcal{A}$ as

$$\mathsf{FORGadv}(\mathcal{A}, \Pi, \lambda, N) = \mathsf{Pr}\big[\mathsf{FORG}[\mathcal{A}, \Pi, \lambda, N] = 1\big],$$

and we say that the shared franking scheme Π satisfies $\varepsilon_{\mathsf{forg}}$ – *unforgeability* if for all efficient adversaries $\mathcal{A}$, security parameters $\lambda \in \mathbb{N}$, and $N \in \mathbb{N}$, it holds that

$$\mathsf{FORGadv}(\mathcal{A}, \Pi, \lambda, N) \leq \varepsilon_{\mathsf{forg}}(\lambda).$$

We say that Π has *unforgeability* if $\varepsilon_{\mathsf{forg}}(\lambda) \leq \mathsf{negl}(\lambda)$.

B Additional Evaluation Data

We repeated our evaluation on GCP, and Figure 7 shows a version of Figure 6 with data from the GCP evaluation. The GCP evaluation was run on a e2-standard-4 instance using Ubuntu 22.04.

On this instance, the cost of shared franking approaches that of plain franking, with all aspects of the protocol taking less than 1.6× the time of the corresponding plain franking operations. Repeating the evaluation on a lower-end server shows that, while the performance gap between the systems is reduced, a large gap remains between the time it takes for our scheme and Hecate to (pre-)process a message and verify reports to the moderator – the main server-side operations. Client-side sending time is much closer between the two schemes, but the gap in time to read messages remains large as well.

To verify that the performance characteristics of our scheme remain consistent when deployed in a networked setting, we also ran the processing phase of our scheme between servers located on the east and west coast. The server code for this evaluation is written in Go but uses our C implementation for the shared franking operations.

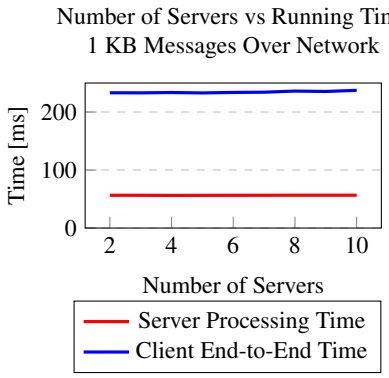

Figure 8: Processing time, including network costs, as number of servers increases. Since the primary latency bottleneck in this setting is the network, the server processing time is mostly flat around the ping time between the servers. The overall processing time from the client's perspective – including time spent setting up a TLS connection to the moderator, waiting for the servers to receive and process the message, and reading the result – are about 4.1-4.2× the ping time, reflecting the number of east/west network round trips in our evaluation setup. Times are averages of 10 runs of the protocol.

Our setup consists of one server on the east coast behaving as the moderator, a number of servers on the west coast (simulated by a single larger server) playing the role of the other servers, and another server on the west coast playing the role of the client. All messages from or to the client are sent to the moderator with the message encrypted under the receiving party's public key. This ensures that all the servers can receive their shares in the same order and also maximizes the communication cost between servers because all messages are sent between the east and west servers. There is no direct communication between the servers and client simulated on the west coast.

We had our e2-standard-4 GCP instance running in the us-east-4b zone simulating the moderator, an e2-standard-16 GCP instance running in the us-west1-c zone playing the role of the other servers, and an e2-standard-2 GCP instance in the us-west1-c zone playing the role of the client. The ping time between the east and west servers was 55.7ms. To isolate the performance characteristics of the shared franking scheme itself, our evaluation does not shuffle, mix, or otherwise run an anonymous message delivery process on the message shares sent to the servers. The servers only compute the processing stage of the shared franking protocol and send back the results to the moderator, who then runs the modProcess algorithm and returns the resulting shares to the client.

The results of this experiment, shown in Figure 8, demonstrate that, for unloaded servers, our scheme does not incur notable additional costs as the number of servers increases. The cost to the client increases very slightly due to the requirement that the client decrypt the messages it has received from each of the servers.