Onion Franking: Abuse Reports for Mix-Based Private Messaging

Matthew Gregoire, Margaret Pierce, Saba Eskandarian
University of North Carolina at Chapel Hill
{mattyg, mapierce, saba}@cs.unc.edu

***Abstract*—The fast-paced development and deployment of private messaging applications demands mechanisms to protect against the concomitant potential for abuse. While widely used end-to-end encrypted (E2EE) messaging systems have deployed mechanisms for users to verifiably report abusive messages without compromising the privacy of unreported messages, abuse reporting schemes for systems that additionally protect message metadata are still in their infancy. Existing solutions either focus on a relatively small portion of the design space or incur much higher communication and computation costs than their E2EE brethren.**

This paper introduces new abuse reporting mechanisms that work for any private messaging system based on onion encryption. This includes low-latency systems that employ heuristic or opportunistic mixing of user traffic, as well as schemes based on mixnets. Along the way, we show that design decisions and abstractions that are well-suited to the E2EE setting may actually impede security and performance improvements in the metadata-hiding setting. We also explore stronger threat models for abuse reporting and moderation not explored in prior work, showing where prior work falls short and how to strengthen both our scheme and others' – including deployed E2EE messaging platforms – to achieve higher levels of security.

We implement a prototype of our scheme and find that it outperforms the best known solutions in this setting by well over an order of magnitude for each step of the message delivery and reporting process, with overheads almost matching those of message franking techniques used by E2EE encrypted messaging apps today.

I. Introduction

Today's widespread deployment of end-to-end encrypted (E2EE) messaging applications is the fruit of decades of efforts in cryptographic protocol development and science communication on the part of the security and privacy community. As E2EE becomes a standard feature for popular messaging apps, protecting metadata looms as the next goal on the horizon. This heightened level of security, where the messaging platform sees neither the contents of messages nor the identities of users' conversation partners, is itself the subject of a decades-long research effort originating in the early works of Chaum [9], [10], and recent years have seen large strides in this space, with a plethora of increasingly performant academic proposals.

While techniques for hiding conversation metadata have seen limited deployment in widespread messaging apps – with the notable exception of Signal's efforts to hide metadata at the application layer via Sealed Sender [1], [37] – anonymity systems focused on web browsing have enjoyed much more success. The most popular approaches for these systems, including those used in Tor [19] and Apple's iCloud Private Relay [2], rely on onion encryption, where two or more non-colluding servers remove layers of encryption from a client-provided ciphertext, such that no server learns both the identity of the client and the destination of the client's message. This is also the foundation of many proposed metadata-hiding messaging systems, originating with mixnets [9] and including recent systems that target a diverse array of security goals [45].

This rapid development of private communication technology in academia and practice also creates a need for mechanisms to address abuse of the protections provided by these systems by bullies, harassers, etc. In an unencrypted messaging platform, users can directly report abuse to a platform moderator who has access to all relevant messages. When a platform is E2EE or hides metadata, moderators do not have access to conversation data (or metadata), so user reports need to additionally prove that the reported content corresponds to actual conversations that occurred on the platform. In response to this need, Meta has developed a technique called Message Franking [39], [27], [20] that allows users to verifiably report abusive messages to the platform without compromising the privacy of unreported messages.

Unfortunately, message franking does not work in the metadata-hiding setting because it relies on the platform having access to conversation metadata when processing messages. Proposals to support message franking for metadata-hiding communication systems include Asymmetric Message Franking (AMF) [50], Hecate [28], and Shared Franking [22]. Unfortunately, these schemes either incur 2-3 orders of magnitude in computational overhead, or only apply to a narrow subset of schemes, rendering them less than ideal for deployment alongside many of the most promising approaches to metadata-hiding messaging.

This paper shows how metadata-hiding schemes based on onion encryption can support verifiable abuse reports while maintaining performance akin to standard message franking. Our scheme, which we refer to as *onion franking*, applies to any system where multiple servers take turns processing onion-encrypted messages, regardless of the network-layer techniques used to sever the connection between sender and receiver. That is, we equally support Tor-style solutions that rely on heuristic mixing of traffic and true mixnets that shuffle messages in batches, as well as options that fall between these two. We believe that our techniques may be more broadly applicable in other settings, which we describe in Appendix D.

Our approach takes advantage of the fact that these systems share a common structure where users send their message to one server who then passes it on to other server(s) during the mixing process. We have the first server, who has a direct connection with the message sender and can therefore learn its identity, perform the role of the moderator. We present two versions of our main scheme, both of which abstract away the details of the underlying onion encryption packet format, but in different ways. One uses the underlying onion-encrypted ciphertext in a fully black-box way, allowing for compatibility with messaging schemes that use variants of typical onion-encryption approach, e.g., re-encryption mixnets. The other modifies the onion encryption process in order to get even better performance and further reduce communication overhead.

Along the way, we find that abstractions used for reasoning about message franking in the E2EE setting may not be the best fit for the metadata-hiding setting. We show that breaking with abstractions used in prior work and relying directly on the underlying cryptographic tools enables savings in communication costs and allows for stronger confidentiality goals than those considered in some prior works. Specifically, we show that breaking the compactly committing authenticated encryption (ccAE) abstraction [27], which very cleanly and intuitively captures the properties required of a standard message franking scheme, can allow us to support anonymous abuse reports, whereas instantiating our scheme using ccAE would necessarily compromise anonymity.

We also explore stronger notions of accountability, the property that prevents a malicious message sender from evading abuse reporting mechanisms. In addition to a standard accountability definition that focuses only on a malicious user, we develop a notion of *strong accountability* that captures the compromise of a large portion of the message moderation and delivery infrastructure. We show how to modify our scheme, as well as prior work, to meet this definition. Our discussion of strong accountability, as well as the constructions we present to provide this property, apply to both the metadata-hiding and plain E2EE settings, so our strong accountability schemes may be of interest for strengthening the accountability of deployed E2EE messaging platforms too.

We implement and evaluate our onion franking scheme, as well as the extensions for strong accountability, and compare their performance to abuse reporting mechanisms from prior work, both for E2EE and metadata-hiding systems. We find that onion franking achieves order of magnitude computation overhead reductions compared to the most efficient prior work that is applicable to onion encryption-based systems.

In summary, this paper makes the following contributions:

- Introduces the notion of onion franking, a broad abstraction that provides verifiable abuse reporting features for any private communication platform based on onion encryption.
- Presents two versions of a lightweight onion franking scheme built from standard and widely-deployed cryptographic tools.
- Explores the notion of *strong accountability* against partially-compromised moderation infrastructure and shows how to augment our scheme and other prior work to be secure in this stronger threat model.
- Implements and evaluates onion franking, showing order of magnitude performance improvements over prior schemes applicable to onion-encrypted messaging, as well as performance comparable to that of message franking in the E2EE setting.

II. Background: E2EE Message Franking

Before moving on, we briefly summarize standard message franking as it is deployed today by Facebook Messenger [39]. We refer to this scheme as E2EE message franking in this paper.

E2EE message franking consists of the message sender producing not only a ciphertext c_1 on their message, but also a commitment c_2 to the same message. More concretely, to send a message m, the message sender samples commitment randomness k_f and produces the ciphertext $\mathsf{Enc}(\mathsf{k}, (\mathsf{m}, \mathsf{k_f}))$. Additionally, the sender produces the commitment $c_2 \leftarrow \mathsf{Commit}(k_f, m)$. Grubbs et al. [27] introduce the abstraction of *compactly committing authenticated encryption* (ccAE) to elegantly merge the required message sending functionality and security properties into a single primitive. Although we follow this same pattern in our onion franking scheme, we do not use the ccAE primitive, for reasons explained later.

When a message is sent through the messaging platform, which also serves as the moderator, the moderator computes a MAC tag $\sigma \leftarrow \mathsf{MAC.Sign}(k_m, (c_2, \mathsf{ctx}))$, MACing the commitment c_2 and a *context string* ctx using a moderator MAC key. The context string includes any information that will be relevant to the moderator should the message be reported later, e.g., sender identity and message timestamp. The tag σ is delivered to the message recipient alongside c_1, c_2.

After decrypting and reading an abusive message, a message recipient can report the message by sending the moderator $m, \mathsf{ctx}, c_2, k_f, \sigma$. The moderator checks the MAC and verifies the commitment to make sure the report is legitimate before adjudicating the user complaint according to platform policies.

Shortcomings of E2EE message franking. Unfortunately, the E2EE message franking approach is designed for E2EE settings and is not compatible with metadata-hiding settings. When looking at metadata-hiding platforms generically, the platform does not have access to the information needed to attach the context string ctx to a user message. Indeed, preventing this is exactly the stated security goal of metadata-hiding sytems.

A path forward. Looking ahead, we will show how to use an E2EE message franking-style approach to build abuse reporting schemes for any metadata-hiding messaging system based on onion encryption. We use the broad term "onion encryption" to mean the set of approaches where a message is protected by many layers of encryption as it passes through the system. This is inclusive of decryption and re-encryption mixnets, onion routing, etc.

Crucially, a common characteristic of onion encryption-based schemes is that the first server to receive the onion-encrypted ciphertext can learn the identity of the message sender. The system always breaks the connection between a message sender and their message after this initial contact. This key observation gives us hope for using an E2EE message franking-like protocol, because the first server has an opportunity to know the necessary context to assemble a useful context string ctx.

III. Design Goals

An *onion franking* scheme augments an onion-encrypted ciphertext to support abuse reporting, allowing any recipient of the encrypted message to verifiably report it to the moderator running the communication platform. While any number of different applications of onion encryption could benefit from reporting features in principle, we will focus on the setting where the underlying application is metadata-hiding messaging, and the onion-encrypted message is a piece of text or media sent from one user of a messaging platform to another.

Augmenting onion encryption. For our purposes, we model onion encryption as a process consisting of two algorithms: `Onion.Encrypt` produces an onion-encrypted ciphertext, which is then decrypted in layers by a series of calls to `Onion.Peel`. In practice `Onion.Encrypt` consists of repeatedly encrypting a message under the public keys of a number of servers, and each server runs `Onion.Peel` to remove one layer of encryption while the message moves from the sender to the recipient. This approach abstracts away details like the specifics of the onion encrypted packet format and separates encryption considerations from network-level mixing strategies.

Onion franking augments this interface in a way that incorporates additional information that a message recipient can send to a moderator in the case that the recipient takes issue with the message content. We add `Send` and `Read` functions to be called by the sender and receiver of a message, as well as a `Process` function to be called by each server in parallel with running `Onion.Peel`. The first server to receive an onion encrypted ciphertext serves as the moderator, and this server runs an additional `ModProcess` function to attach necessary metadata before processing the ciphertext. Finally, the moderator uses a `Moderate` function to verify reports and extract relevant information once a user reports a message. Figure 1 shows how onion franking can work alongside onion encryption, using the inputs and outputs of the formal syntax introduced in Section IV. This is one of two ways, discussed in Section V, to integrate onion franking into an existing messaging scheme.

Security goals. An onion franking scheme must satisfy a number of security requirements that ensure all messages can be correctly reported, that malicious users cannot frame others, and that the scheme cannot be abused to publicly shame users for expressing their opinions. Most obviously, the onion franking scheme must in no way compromise the confidentiality and metadata protections that the unmodified messaging platform provides to its users for unreported messages. These security requirements can be summarized by the following list of required properties.

- **Unforgeability**. No malicious client may submit a false report that is then verified by the moderator. This guarantees that no malicious client, even if it colludes with malicious servers, can blame another client for a message it did not send.
- **Accountability**. Every message sent through the onion franking scheme can be verified if reported, even if an adversarial sender maliciously deviates from the protocol.
- **Deniability**. No server, client, or third party may independently verify a message report, except the moderator.
- **Confidentiality**. The addition of onion franking functionality must not compromise the confidentiality of the underlying messaging system. More specifically, the franking scheme must not affect the confidentiality and metadata-hiding properties of the platform, except for reported messages.

We discuss how to apply these high-level security goals to onion franking in Section IV. While these security goals have been considered by a number of works on abuse reporting for private messaging, we additionally explore two previously unaddressed aspects of security: accountability in the presence of potentially compromised moderation infrastructure, and the privacy provided to users by a franking scheme after a report has been made.

Threat modeling assumptions. Our work focuses on assuring the necessary security properties of users' messages, but not on the robustness of the platform against disruptive servers. That is, we trust that servers will remain available to process messages, even if they may maliciously deviate from the specified protocol when doing so. Ultimately, honest clients can simply discard messages that arrive without onion franking tags due to non-cooperative servers.

We make no assumption about the behavior of clients, who may be arbitrarily malicious and can collude with malicious servers to compromise our various security goals. Our security goals focus on security at the application layer, so we assume the existence of pairwise secure connections between servers, e.g., via TLS, which are often already in use by the underlying communication protocol.

IV. Formalizing Onion Franking

This section formalizes the syntax for onion franking and discusses the various security properties required of an onion franking scheme in greater detail. We present formal definitions in the appendices in cases where a clear and self-contained security goal emerges, and we give a precise but informal description of our goals in cases where the security definitions depend on those of the underlying messaging system.

Notation. Before we continue, we pause here to briefly summarize the notation used throughout this paper. By $x \leftarrow f(y)$ we denote assignment to x of the value $f(y)$, and by $x \xleftarrow{R} S$ we denote assignment to x of a value chosen uniformly at

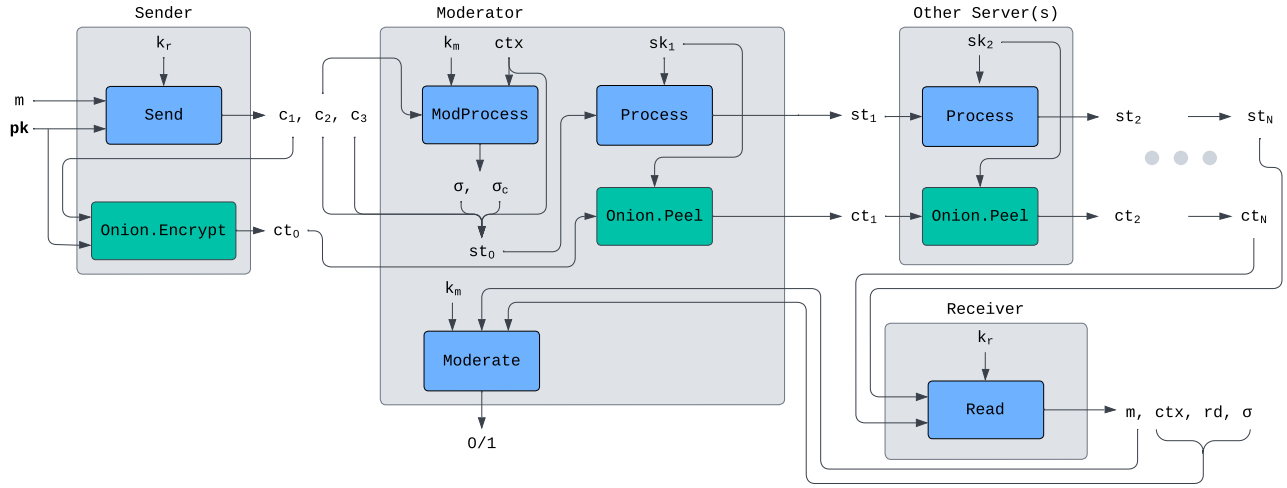


Fig. 1: Illustration of how to integrate onion franking with a messaging system that uses onion encryption, e.g., mixnets, onion routers, etc.

random from a set S. Tables, denoted with capital letters and initialized at $T \leftarrow \{\}$, act as key-value stores, where values can be accessed by $T[\mathsf{key}]$. Vectors, denoted $\mathbf{v}$ and initialized as $\mathbf{v} \leftarrow []$, are ordered lists of values. Elements can be added to vectors via $\mathbf{v}.\mathsf{add}(\mathsf{val})$. A function $\mathsf{negl}(x)$ is *negligible* if, for all $c > 0$, there exists an x_0 such that, for all $x > x_0$, $\mathsf{negl}(x) < \frac{1}{x^c}$. We use $\perp$ as a special character indicating protocol failure.

A. A formal syntax for onion franking

An *onion franking scheme* consists of the following algorithms.

- $\mathsf{ServerSetup}(1^\lambda) \rightarrow (\mathsf{sk}, \mathsf{pk})$: This function takes in a security parameter λ and generates a key pair. It is run by each server, and the public keys $\mathsf{pk}_1, ..., \mathsf{pk}_n$ form the vector $\mathbf{pk}$.
- $\mathsf{Send}(\mathbf{pk}, k_r, m) \rightarrow c_1, c_2, c_3$: This function takes a vector of n server public keys $\mathbf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$, the receiver's symmetric key $k_r \in \mathcal{K}_r$, and a message m. It returns a symmetric ciphertext c_1, a franking tag c_2, and a mixnet franking packet c_3.
- $\mathsf{ModProcess}(k_m, c_2, \mathsf{ctx}) \rightarrow \sigma, \sigma_c$: This function, performed by the moderator server, takes the moderator's MAC key $k_m \in \mathcal{K}_m$, franking tag c_2, and message context ctx. It returns a reporting tag σ, as well as a checksum value σ_c used by the client to validate the integrity of σ.
- $\mathsf{Process}(\mathsf{sk}_i, \mathsf{st}_{i-1}) \rightarrow \mathsf{st}_i$: This function is performed by each server S_i. It takes S_i's secret key sk_i and masked report state st_{i-1}. It then outputs a re-masked report state st_i. In correct usage, the tuple $(c_3, c_2, \mathsf{ctx}, \sigma, \sigma_c)$ is the initial st_0 used by this function.
- $\mathsf{Read}(k_r, c_1, \mathsf{st}_N) \rightarrow m, \mathsf{ctx}, \mathsf{rd}, \sigma$ or $\perp$: This function takes the receiver's symmetric key k_r, symmetric ciphertext c_1, and masked report state st_N. It returns the message m, the context ctx, additional reporting data rd, and a tag σ. If reading fails, this function returns $\perp$.
- $\mathsf{Moderate}(k_m, m, \mathsf{ctx}, \mathsf{rd}, \sigma) \rightarrow 0/1$: this function is used by the moderator at reporting time. It takes the moderator's MAC key k_m, a reported message m, associated context ctx, report data rd, and a tag σ. The function outputs 1 if the report is valid, and 0 otherwise.

Correctness. Correctness for onion franking simply requires that an honestly generated onion franking message can be successfully read and reported after processing by the moderator and other servers. Specifically, for all $N = |\mathbf{pk}|$, all keys $k_r \in \mathcal{K}_r, k_m \in \mathcal{K}_m, \lambda \in \mathbb{N}, i \in \{0, ..., N-1\}$, and for all choices of m, ctx, after running

$$\mathsf{sk}_i, \mathsf{pk}_i \leftarrow \mathsf{ServerSetup}(1^\lambda)$$

for each server $i \in \{1, ..., N\}$, and then computing

$$c_1, c_2, c_3 \leftarrow \mathsf{Send}(\mathbf{pk}, k_r, m)$$

$$\sigma, \sigma_c \leftarrow \mathsf{ModProcess}(k_m, c_2, \mathsf{ctx})$$

$$\mathsf{st}_1 \leftarrow \mathsf{Process}(\mathsf{sk}_0, (c_3, c_2, \mathsf{ctx}, \sigma, \sigma_c))$$

followed by $N-1$ iterated calls to

$$\mathsf{st}_{i+1} \leftarrow \mathsf{Process}(\mathsf{sk}_i, \mathsf{st}_i) \text{ for } i \in \{1, ..., N-1\},$$

it must hold that

$$m', \mathsf{ctx}', \mathsf{rd}, \sigma' \leftarrow \mathsf{Read}(k_r, c_1, \mathsf{st}_N),$$

where $m' = m$, $\mathsf{ctx}' = \mathsf{ctx}$, and

$$\mathsf{Moderate}(k_m, m', \mathsf{ctx}', \mathsf{rd}, \sigma') = 1.$$

The remainder of this section addresses each of the security requirements of an onion franking scheme – unforgeability, accountabililty, deniability, and confidentiality.

B. Unforgeability

A key requirement of an abuse reporting scheme is that it should not itself become a vector for abuse. In particular, a malicious user should not be able to forge abuse reports to harm someone else. We capture this requirement in our unforgeability definition, which allows a malicious user(s) colluding with all the servers except the moderator to send a number of messages of its choosing through the moderator and receive its outputs.

The experiment includes an OnionFrank oracle that allows the adversary to send potentially malformed ciphertexts to the moderator. Since the adversary controls all the other servers, the experiment does not need to explicitly model them, allowing the adversary to choose the output of the final server. However, the experiment does distinguish between messages that are moderated and never delivered versus messages that are successfully delivered after passing through the moderator.

Whenever the adversary wishes, it can send a forged report to the Verify oracle. The adversary wins if this forged report is accepted and the corresponding report tag σ has not already been produced by OnionFrank, or if the tag σ was produced by the moderator, but was delivered with a different message m, context ctx, or report data rd than the ones being reported. This second condition is why we need to keep track of delivered messages and not just the ciphertexts that pass through the moderator. This definition captures our unforgeability goal because only messages that have been sent through OnionFrank have been processed by the moderator, so messages not in this list must be forgeries.

We formalize our unforgeability definition in Appendix A.

C. Accountability

Accountability is the property that prevents malicious users from evading the abuse reporting mechanism and sending messages that can be read by the recipient but fail verification by the moderator. This is a slightly different property than unforgeability: unforgeability prevents fake messages that are accepted as valid, and accountabililty prevents real messages that are not accepted as valid.

Our accountabililty definition allows a malicious user to send ciphertexts of its choice through the system via an OnionFrank protocol. In this protocol, the experiment processes messages as the moderator, subsequent servers, eventual message recipient, and report moderator. If the message is successfully read but fails to be accepted by the Moderate function, or if the message is successfully read but produces a different context than the one used by the moderator, the adversary wins.

In the interest of keeping the onion franking security definitions self-contained and independent of the properties of the underlying messaging scheme, we avoid including calls to Onion.Peel in the security definition by allowing the adversary to directly specify the ciphertext ct_N. This is a conservative modeling of reality, where the adversary may only have partial control over this value.

We state our accountabililty definition formally in Appendix A. Our standard accountabililty definition assumes that the moderator and other platform servers follow the protocol honestly. In Section VI we present a *strong accountabililty* definition where any subset of the moderator and platform servers can be malicious at message delivery time. This captures the strong threat model where the message delivery infrastructure, and even part of the moderation infrastructure, are compromised by an attacker.

D. Deniability

Deniability preserves the ephemeral nature of real-world communication. It requires that only the moderator is able to verify reports, and that other parties cannot conclusively show that a given user sent a given message. There are many ways to define deniability, each of which comes with differing tradeoffs between deniability and accountabililty [50]. Recent works on abuse reporting focus on a similar family of security goals [50], [28], [22] where a scheme must satisfy a handful of deniability properties, each of which requires that there exists a report forgery algorithm whose outputs are computationally indistinguishable from those of the real reports generated by the scheme. The differences between the properties arise from the choice of what secrets are given to the forgery algorithm and the distinguisher. Since the actual reports generated by our onion franking scheme will be identical to reports generated by conventional E2EE message franking (or its variants [27]), we do not repeat an analysis of the deniability of this scheme, which follows directly from the security properties of the underlying tools [39].

Note that deniability definitions apply only to the abuse reporting scheme itself, and that the remaining components of a larger messaging system must also be deniable in order for deniability to hold for the whole system. For example, it doesn't help for onion franking reports to be deniable if the onion encryption ciphertexts or key exchange mechanisms themselves are not.

E. Confidentiality

Finally, the most important and fundamental requirement of an onion franking scheme is that it in no way compromise the security properties of the underlying messaging scheme, except when a user reports a message to the moderator. This includes protecting the confidentiality of the message contents themselves, as well as message metadata. Onion franking aims to augment any scheme that uses onion encryption, rather than targeting a given family of confidentiality goals. Thus, rather than targeting one or another class of metadata-hiding goals, we require that a scheme that has been augmented with onion franking has the same confidentiality properties (for unreported messages) as the underlying scheme. In practice, the confidentiality of our schemes will depend almost entirely on that of the encryption schemes employed. How best to define privacy guarantees for onion encryption is itself the subject of active research [15], [44], [17], [47], so we do not introduce a new confidentiality definition here except to state that the encryption scheme used for transmitting additional data as part of our scheme must satisfy the necessary properties to

preserve any anonymity, in addition to the confidentiality and integrity properties typically expected of encryption [25], [43].

Confidentiality for reported messages. Once a message has been reported, its contents are no longer confidential because the message receiver has voluntarily reported them to the platform. In an E2EE platform, this is the end of the story. However, hiding metadata introduces the possibility of retaining some privacy requirements even after a message's contents have been reported. We consider two such cases.

- In some metadata-hiding platforms, e.g., in anonymous broadcast systems, the sender of a message can be anonymous even to the message receiver. In this case, a platform may wish to maintain sender anonymity from the receiver, even if a message is reported. We say that a platform that satisfies this property has *post-report anonymity*.
- If a messaging platform does not link message senders and receivers, then message recipients may wish to make anonymous reports. That is, they might want to reveal to the platform that some user sent a particular abusive message without revealing to the platform that they were speaking to that user. We call this property *anonymous reporting*.

Post-report anonymity and anonymous reporting are not properties that can be provided by an onion franking scheme alone, as they also depend on design and deployment choices in the underlying messaging platform. However, it is possible for an onion franking scheme to render a messaging scheme incompatible with one or both of these properties, so we will study the compatibility of our proposed schemes with these properties. Interestingly, we will show in Section V that abstractions used to elegantly and concisely capture the properties required for message franking in the E2EE setting actually impede the ability of an onion franking scheme to support anonymous reporting.

V. Onion Franking Construction

This section introduces our main constructions of onion franking. We adapt the tools used for E2EE message franking, with additional techniques to preserve metadata-hiding properties while messages are being routed, and to prevent malicious users from using the stronger privacy properties to evade accountability.

A. Scheme Description

As mentioned in Section II, onion encryption is a setting where the first server can learn the necessary context ctx to make useful abuse reports. Thus E2EE message franking is a plausible starting point for our scheme. Unfortunately, message franking requires a commitment c_2 to the message to be visible to the moderator, and the moderator produces a MAC tag σ that must be attached to the ciphertext when it is delivered. Simply appending $(c_2, \mathsf{ctx}, \sigma)$ to a ciphertext as it enters an onion encryption-based messaging system will make the ciphertext clearly identifiable and traceable to other servers in the system, violating the confidentiality of the messaging system.

To maintain the anonymity of the message, we need to mask $(c_2, \mathsf{ctx}, \sigma)$ from the view of other servers. We achieve this by giving each server a random mask value r_i that it XORs into these values. Since each server XORs in a random value into this bit string, which we refer to as the *state* st, the state will look independently random to each server. To ensure that the message receiver can still read $(c_2, \mathsf{ctx}, \sigma)$, we generate the r_i values from a PRG G on a sender-selected input s that is encrypted alongside the message m in the ciphertext c_1. We send the r_i values to the various servers in an onion-encrypted ciphertext c_3, the i^{th} layer of which holds the value r_i and the ciphertext for the next server to decrypt. Thus, the state actually consists of two sets of values. A masked tuple $(c_2, \mathsf{ctx}, \sigma)$ and an onion-encrypted ciphertext c_3, the layers of which are gradually removed to reveal random masks r_i for each server.

The scheme as described so far works if message senders follow the protocol honestly, but a malicious sender seeking to evade the accountability of the scheme could simply pick $r_1, ..., r_N$ however it wants and choose s unrelated to these values. Then, when the message recipient decrypts c_1 to recover (m, s) and uses s to unmask $(c_2, \mathsf{ctx}, \sigma)$, the "unmasked" values will be incorrect, resulting in the moderator rejecting the user's reports as inauthentic. We avoid this by including a checksum value in the state st to detect when $r_1, ..., r_N \neq G(s)$. After computing the MAC tag σ, the moderator also computes a separate tag $\sigma_c \leftarrow H(\sigma, c_2, \mathsf{ctx})$ and includes σ_c in the string that forms the masked state. Thus, when a message is read, the recipient recovers $(c_2, \mathsf{ctx}, \sigma, \sigma_c)$ and can check that the appropriate relationship between these four variables is maintained. We show in our security analysis, that this rules out accountability attacks by a malicious message sender.

Anonymous reporting. Throughout this paper, we use the c_1, c_2 notation for message franking that is used in compactly committing authenticated encryption (ccAE) [27]. Abstractions like ccAE and encryptment [20] provide concise and elegant ways to capture the core requirements for message franking ciphertexts. However, we do not adopt these abstractions in their entirety because, while they very effectively capture the requirements of E2EE message franking, they actually introduce inefficiencies and weaken the privacy afforded to reporters in the metadata-hiding setting.

To see why, suppose we had directly used the ccAE abstraction. This abstraction takes in a message and produces c_1, c_2 that are an encryption and commitment to the message, respectively, very similar to what our scheme does. The message we encrypt here is (m, s), where m is the actual message, and s is the seed used to generate $r_1, ..., r_N$. The randomness r_f used to produce and open the commitment c_2 is incorporated into the ciphertext c_1 and returned to the user when the ciphertext is decrypted.

Avoiding using ccAE gives us a performance benefit because the moderator doesn't need to see s, but it would have to be sent s in order to verify the commitment c_2 to (m, s). Moreover, if we're already using $G(s)$ to generate random masks, we can

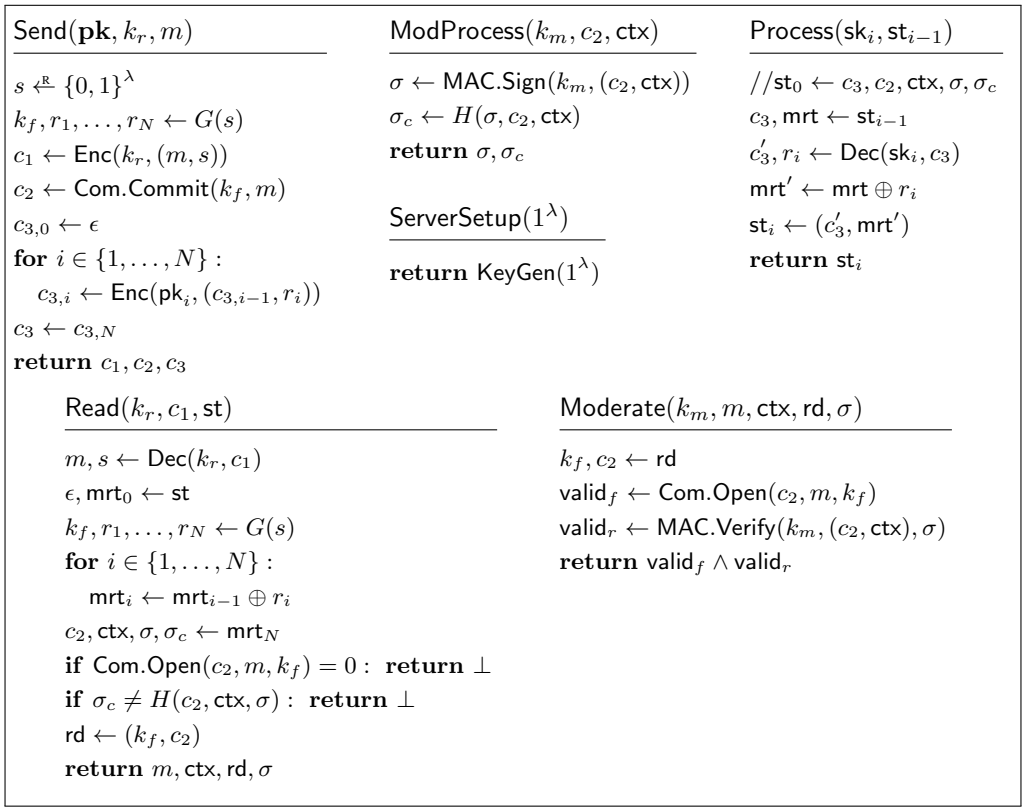


Fig. 2: Our onion franking scheme Π (Construction V.1).

also use it to generate r_f, reducing the size of the ciphertext. In practice, these changes save 16-32 bytes of communication for each message and each report.

Much more importantly, while sending s to the moderator may seem like just a minor inefficiency, it also has negative ramifications for anonymous reporting. If a platform supports anonymous reports, i.e. the reporting user can use the metadata-hiding system itself to send reports to the moderator, then a malicious moderator can use s to unmask the identity of the recipient, circumventing all the protections of the anonymous reporting feature. If the malicious moderator has s, it can compute $r_1, ..., r_N$, meaning it has all the randomness that was used to mask $(c_2, \mathsf{ctx}, \sigma, \sigma_c)$ as it was passed from server to server. By collecting traffic sent from the platform to users (or colluding with the last server), the moderator can now trace the path of these values from sender to receiver, revealing who received and then reported this message. Thus while ccAE provides excellent intuition for E2EE message franking, its formalization does not quite fit with the new security goals of the metadata-hiding setting.

B. Formal Description

We now formally describe the onion franking scheme outlined above.

Construction V.1 (Onion Franking)**.** Our N-server onion franking scheme Π with security parameter λ appears in Figure 2 and makes use of the following primitives:

- A PRG $G : \{0,1\}^\lambda \to \{0,1\}^{(N+1)\lambda}$.
- A hash function $H : \{0,1\}^* \to \{0,1\}^{\lambda'}$ modeled as a random oracle. The parameter $\lambda' = \mathsf{poly}(\lambda)$ is derived from the security parameter.
- A MAC scheme $\mathsf{MAC} = (\mathsf{Sign}, \mathsf{Verify})$ where the Sign algorithm is also a PRF.
- A commitment scheme $\mathsf{Com} = (\mathsf{Commit}, \mathsf{Open})$
- A public-key encryption scheme $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$

Before moving on to the security analysis, we briefly discuss two potential optimizations that can be applied to our scheme.

Preprocessing client values. Very little of the computation in Send depends on the message m in any way. The client only needs to encrypt and commit to the message, the same operations required in plain E2EE message franking. The additional operations introduced for onion franking – computing $r_1, ..., r_N$ and encrypting them in the nested ciphertext c_3 – can be preprocessed before a user decides to send a message. Thus the online computational cost of onion franking is identical to that of standard message franking.

C. An Optimized Scheme

As an alternative to preprocessing client values, observe that the ciphertext c_3 is produced in parallel with the onion encrypted ciphertext holding the message, and these two values are sent into the messaging system alongside each other. This design allows us to be fully agnostic with respect to the design

$\mathsf{Send}(\mathbf{pk}, k_r, m)$	$\mathsf{Process}(\mathsf{sk}_i, \mathsf{ct}_{i-1}, \mathsf{st}_{i-1})$
$s \xleftarrow{R} \{0,1\}^\lambda$	$//\mathsf{st}_0 \leftarrow c_2, \mathsf{ctx}, \sigma, \sigma_c$
$k_f, r_1, \ldots, r_N \leftarrow G(s)$	$\mathsf{ct}_i, r_i \leftarrow \mathsf{Dec}(\mathsf{sk}_i, \mathsf{ct}_{i-1})$
$\mathsf{ct}_0 \leftarrow \mathsf{Enc}(k_r, (m, s))$	$\mathsf{st}_i \leftarrow \mathsf{st}_{i-1} \oplus r_i$
$c_2 \leftarrow \mathsf{Com.Commit}(k_f, m)$	**return** $\mathsf{ct}_i, \mathsf{st}_i$
for $i \in \{1, \ldots, N\}$:	
$\quad \mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}_i, (\mathsf{ct}_{i-1}, r_i))$	
return ct_N, c_2	

Fig. 3: Onion franking scheme, optimized to integrate with decryption mixes. Changes within each algorithm are shown in blue. Algorithms not shown are unchanged.

of the onion encryption scheme, but we can merge the two into one ciphertext by making slightly stronger assumptions about the underlying messaging scheme.

If we assume that messages are encrypted by nesting layers of encryption that are then decrypted by each subsequent server, we can put each server's value of r_i inside the message ciphertext that server decrypts, instead of producing another nested ciphertext that is decrypted alongside it. This optimization saves a great deal of computation and communication by only incurring the overhead of encrypting and decrypting a ciphertext once. It significantly reduces the portion of the scheme that can be preprocessed (just generating $r_1, ..., r_N$), but in return the overhead of sending messages and processing them drops precipitously.

We show the changes that would be made to construction V.1 to merge the ciphertexts for the message and masks in Figure 3. This optimization is compatible with many messaging systems, but it can not be used with, e.g., systems that rely on re-encryption mixnets because the optimization adds a small plaintext in between the different layers that are being decrypted.

Throughout this paper, we formalize, implement, and evaluate all our other variations of onion franking on top of the general scheme formalized in construction V.1, but all the variations can just as easily be implemented on top of this optimized scheme. Looking ahead to our evaluation in Section VII, the optimized scheme has the overall strongest performance, only being beaten only by the online Send time in the general scheme and plain E2EE message franking.

D. Security Analysis

We now analyze the security properties of our scheme.

Unforgeability. The unforgeabilty of the scheme follows from the binding property of the commitment and the unforgeability of the MAC. Intuitively, the moderator will only accept MACs that have been passed through the moderator, which means that any report sent to the moderator will have authentic values of $(\sigma, c_2, \mathsf{ctx})$. But since c_2 is a binding commitment to the message, this means that the moderator will only accept messages whose ciphertexts have honestly passed through ModProcess. This intuition is formalized in the following theorem, which we prove in Appendix B.

Theorem V.2 (Unforgeability)**.** *Assuming that* MAC *is an existentially unforgeable MAC scheme, and that* Com *is a binding commitment scheme, our onion franking scheme* Π *(Construction V.1) satisfies unforgeability (Definition A.1).*

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

$$\mathsf{FORGAdv}(\mathcal{A}, \Pi, N, \lambda) \leq \mathsf{MACAdv}(\mathcal{B}, \mathsf{MAC}, \lambda) + \mathsf{BINDAdv}(\mathcal{C}, \mathsf{Com}, \lambda) + \mathsf{negl}(\lambda).$$

Accountability. Accountability follows from the pseudorandomness of the MAC outputs and the fact that we model H as a random oracle. Since MAC.Sign is modeled as a PRF, each value of (c_2, ctx) is associated with an independent and uniformly random value σ. But since σ is an input to H, each $(c_2, \mathsf{ctx}, \sigma)$ is associated with an independent and uniformly random value σ_c. A malicious user breaks accountability by causing a reader to accept the check of σ_c while the moderator rejects σ. A malicious user has the power to maliciously generate $r_1, ..., r_n$ in an attempt to cause this bad event, resulting in the moderator's outputs being XORed by values of the attacker's choice. Thus the ability of the attacker to break accountability is tied to the probability that it can guess the offsets between the random values mentioned above and XOR in values to cancel them out, a task that is only achieved with negligible probability. We prove this formally in Appendix B, where we prove the following theorem.

Theorem V.3 (Accountability)**.** *Assuming that we model* H *as a random oracle, and that* MAC *is a correct MAC scheme where* MAC.Sign *is also a PRF, our onion franking scheme* Π *(Construction V.1) satisfies accountability (Definition A.2).*

In particular, for every accountability adversary $\mathcal{A}$ *that attacks our protocol* Π*, there exists an accountability adversary* $\mathcal{B}$ *such that for every* λ, N*,*

$$\mathsf{ACCTAdv}(\mathcal{A}, \Pi, N, \lambda) \leq \mathsf{PRFAdv}(\mathcal{B}, \mathsf{MAC.Sign}, \lambda) + \mathsf{negl}(\lambda).$$

Confidentiality. The confidentiality of our scheme follows from that of the encryption scheme used to encrypt the message and r_i values, as well as the hiding property of the commitment scheme Com. Consider an adversary who controls all but the i^{th} server. The input to this server's Process function is a ciphertext and a bit string masked with the previous server's random r_{i-1}. The output is another ciphertext and the same bit string now masked with a new random r_i. As long as the encryption scheme (Enc, Dec) provides the necessary semantic security and anonymity properties, confidentiality is preserved. The exception to this pattern is the moderator, who receives c_2, unmasked and unencrypted, from the user, but here the hiding property of the commitment scheme ensures that the moderator learns nothing about the message from c_2.

This scheme is compatible with post-report anonymity because the only server who receives any information about the sender identity is also the moderator, so as long as the context string ctx does not reveal information about the sender identity, e.g., if the sender identity is encrypted, the receiver cannot learn anything about the sender identity from the onion franking scheme, even if it colludes with other non-moderator servers. This is true of any scheme using a layered mixing structure, but it is not necessarily true for all abuse reporting schemes. For example, schemes that rely on a third party moderator may leak information to the receiver that allows it to collude with the first server to reveal the sender's identity, or schemes that rely on secret-sharing may allow the receiver to collude with non-moderator servers to do the same.

Finally, as already discussed, this scheme is compatible with anonymous reporting because the secrets used to preserve confidentiality remain hidden from the moderator, even after a report is made. In this way, although onion franking breaks the confidentiality of message contents and sender context, other aspects of confidentiality remain intact post-reporting.

VI. Resilience Against Compromised Moderation Infrastructure

Having presented our main constructions of onion franking, we now discuss a stronger threat model under which our scheme and prior works may wish to achieve security: the case of partially compromised moderation infrastructure. Although we present our contributions in this setting as an extension to onion franking, the techniques are equally applicable to standard end-to-end encryption tools deployed today.

All abuse reporting schemes ultimately rely on the subjective judgment of a moderator to make decisions regarding the admissibility of content and appropriate ramifications for those who violate platform content rules. Since our work (and other work in this space [50], [28], [22]) aims for strong deniability properties, this means that the moderator's decisions about the validity of a report and the objectionability of its contents cannot be questioned. Thus it does not make sense to discuss security against a moderator who behaves maliciously and lies about the outputs of the `Moderate` function.

However, the moderator also needs to be involved in the processing of messages, i.e., when running the `ModProcess` and `Process` functions. It is possible that these two moderator functions – message processing and message moderation – take place on different computing infrastructure because message processing and content moderation are logically distinct tasks. This raises the possibility that a compromised moderator server behaves maliciously at message delivery time, causing an honest moderator to get an incorrect output from the `Moderate` function. The definition of accountability does not rule out this possibility because it explicitly assumes the moderator to be honest.

Stronger accountability. We consider a stronger notion of accountability, named *server accountability*, which captures an attacker who attempts to break accountability via compromised moderation infrastructure at message delivery time, as well as a *strong accountability* definition that captures both standard accountability and server accountability. Unlike our standard accountability definition, which did not allow the adversary to control any of the platform servers, strong accountability (and server accountability) allows the adversary to control any subset of servers it chooses, including the moderator, during message processing/delivery only. The strong accountability security game is similar to the standard accountability one, except that the adversary is allowed to pick which servers it controls at the beginning of the experiment, and while running the OnionFrank protocol, the adversary receives the inputs to servers it controls and can choose their outputs. The definition also includes a relaxed variant, $1/\ell$*-strong accountability* (or $1/\ell$*-server accountability*), which allows for schemes that catch abuse with some constant but non-negligible probability $\frac{\ell-1}{\ell}$. We formalize our strong accountability definition, the stronger of the two notions, in Appendix A.

A. Vulnerability of Existing Schemes to Moderator Compromise

Almost all existing schemes for abuse reporting in both the E2EE and metadata-hiding settings, including our own schemes in Section V, fail to achieve server accountability, and therefore strong accountability. The only exception is asymmetric message franking (AMF) [50], the most computationally expensive of the known schemes, which avoids the problem entirely because the moderator plays no role in the message delivery process prior to receiving reports. This section briefly describes server/strong accountability attacks on prior work. We will then show how to strengthen onion franking to defend against strong accountability attackers.

At a high level, the issue is that existing schemes never "check the work" of the moderator during message processing. In plain E2EE franking [39], shared franking [22], and onion franking (this work), the moderator at some point computes a MAC σ on values c_2 a ctx, so it can later identify messages that have passed through the platform and detect forged reports. However, since this is a symmetric key primitive, only the moderator knows if the MAC has been computed correctly. A message with a malformed MAC can be delivered with no apparent problems until someone decides to report it.

Hecate [28] uses a signature scheme instead of a MAC in the corresponding phase of its protocol, which can be done during a preprocessing phase. This means that anyone can verify the signature and detect if it is malformed. While at first glance this might suggest that Hecate does not fall prey to attacks based on moderator compromise, there is still an issue. The Hecate moderator signs an encryption of a user identifier corresponding to the user sending a message. That is, it signs $\mathsf{Enc}(\mathsf{sk}_{\mathsf{mod}}, \mathsf{id}_{\mathsf{src}})$, an encryption under a moderator-held key of the identifier of the message sender. So while the signature can be checked, the ciphertext it contains may still be undetectably malformed.

$\mathsf{ModProcess}(k_m, c_2, \mathsf{ctx})$

$\sigma \leftarrow \mathsf{MAC.Sign}(k_m, (c_2, \mathsf{ctx}))$
$\pi \leftarrow \mathsf{ZK.Prove}(\sigma, c_2, \mathsf{ctx})$
$\mathsf{rt} \leftarrow (\sigma, \mathsf{ctx})$
return rt, π

$\mathsf{Read}(k_r, c_1, \mathsf{st}, \sigma_k)$

$m, s, c_2 \leftarrow \mathsf{Dec}(k_r, c_1)$
$\mathsf{mrt}_0, \epsilon \leftarrow \mathsf{st}$
$k_f, r_1, \ldots, r_N \leftarrow G(s)$
if $\mathsf{Com.Open}(c_2, m, k_f) = 0$: **return** $\perp$
for $i \in \{1, \ldots, N\}$:
  $\mathsf{mrt}_i \leftarrow \mathsf{mrt}_{i-1} \oplus r_i$
$\mathsf{rt}, \pi \leftarrow \mathsf{mrt}_N$
$\sigma, \mathsf{ctx} \leftarrow \mathsf{rt}$
if $\mathsf{ZK.Verify}(\sigma, c_2, \mathsf{ctx}, \pi) = 0$: **return** $\perp$
$\mathsf{rd} \leftarrow (c_2, k_f)$
return $m, \mathsf{rt}, \mathsf{rd}$

Fig. 4: High-level changes to our onion franking scheme to enable strong accountability using zero-knowledge proofs. Omitted functions are unchanged from Π (Figure 2). Changes are shown in blue.

B. Achieving Stronger Accountability

The rest of this section explores two high-level approaches to achieving strong accountability and server accountability, respectively: proving moderator honesty in zero knowledge and catching malicious behavior with trap messages. Although we formalize these schemes in the context of onion franking, the exact same techniques can be applied to add strong accountability or server accountability to plain E2EE message franking, as they in no way depend on the fact that the scheme is built on onion encryption.

Proving honesty. The most straightforward way to protect against a malicious moderator is to have the moderator prove that it is behaving fully honestly while doing its job, i.e., while running ModProcess. Generically, non-interactive zero-knowledge proofs provide the tools needed to produce a proof π that the moderator has correctly computed the MAC σ relative to a publicly posted commitment to the MAC key k_m [26], [6]. Assuming the proof system used is non-malleable (which comes for free in proofs made non-interactive via the Fiat-Shamir transform [24], [23]), no servers between the moderator and the message recipient can tamper with the proof to produce a π' that successfully verifies. The soundness of the proof system means it won't verify with incorrect inputs, so the proof π can also replace the checksum σ_c because the proof will fail to verify if the values of $c_2, \mathsf{ctx}, \sigma$ differ from the ones seen at moderation time. We summarize the changes needed to prove and verify moderator honesty in the ModProcess and Read algorithms in Figure 4.

Unfortunately, while zero-knowledge provides a ready-made and drop-in solution for strong accountability, generically applying zero-knowledge proofs can be quite expensive. Thus the challenge in using such well-known tools is to identify the right instantiation of the relevant cryptographic primitives to attain reasonable performance. We commit to the key k_m using a multi-commitment scheme based on Pedersen commitments [41] and use a MAC scheme proven secure under the DDH assumption [18] by Dodis et al. [21]. Given these building blocks, our proof π consists of the moderator proving that it knows the two-part MAC key (x_0, x_1) and commitment randomness used to commit to the key, and that the same (x_0, x_1) has been used to produce the MAC σ on (c_2, ctx). The proof is constructed as a sigma protocol for generic linear relations [13], [7] made non-interactive via the Fiat-Shamir transform [24]. Formal details of the commitment scheme, MAC scheme, and the statement proved in zero knowledge appear in Appendix C.

Trap messages. Our second approach to strengthening accountability achieves a degree of server accountability, but not strong accountability. That is, it does not provide accountability if message senders and servers collude, but it does separately protect against malicious senders (via standard accountability) and against malicious servers (via server accountability). To do this, we use trap messages: empty messages sent with the sole purpose of being reported in order to detect reporting failures. Trap messages have the benefit of not using any cryptographic tools beyond those already present in onion franking, and they allow for a range performance/security tradeoffs. As such, trap messages achieve $1/\ell$-server accountability, for a configurable parameter ℓ. We begin by describing a naïve trap message scheme to convey the intuition of the approach before showing how we optimize it to significantly reduce the cost of this technique.

A naïve trap message scheme would simply have each message sender send ℓ messages in place of each message they intend to send. One of the ℓ messages is the real message m, and the others contain a fixed message z: a string of length $|m|$ consisting only of zeros. When the receiver gets the ℓ messages, it immediately reports the $\ell - 1$ messages containing z to the moderator. The moderator runs Moderate on each trap message, and it returns 0 to the user if any fail, 1 otherwise, indicating whether or not the user should discard the message m as being malformed. Since accountability is the property that an adversary cannot cause a message to be successfully read but not moderated, trap messages only fail if the adversary guesses which of the ℓ messages sent contains m and only attacks that one. Thus choosing larger values of ℓ results in stronger accountability, at the cost of sending more messages.

Such a trap message scheme would work in the E2EE setting, but a metadata-hiding platform must also support anonymous reporting. Otherwise, the moderator would be able to link the message sender and receiver through the trap message context strings, thereby breaking the metadata-privacy of all messages, regardless of whether they are really reported as abuse. Fortunately, onion franking is compatible

with anonymous reporting, as discussed in Section V.

Note that while we need the underlying platform to allow for anonymous reporting to hide the identity of the message sender, it is fine (and in fact necessary) for the moderator to learn whether reports are traps or real reports at reporting time. Trap messages are there to protect against malicious tampering during message sending/delivery, not to protect anonymity while reporting. In this way, trap messages are distinct from noise or cover traffic techniques used in some works to hide when users are sending real messages, e.g., [53], which could be a feature of the underlying messaging system but has nothing to do with the abuse reporting functionality.

Optimizing trap messages. Observe that the largest costs of trap messages are the communication cost of sending $\ell - 1$ copies of z and the computation cost of hashing $\ell - 1$ copies of z, e.g., to perform the commitment or MAC. Both of these costs are in fact redundant because all parties involved know the value of z, but trap messages, as described thus far, need to send the message every time to hide which one of ℓ messages is the real message m.

Our insight is that we don't need to produce ℓ actual trap messages. We only need to generate ℓ trap *reports* for the moderator to check. We return to sending only a single message, but now the message is accompanied by tags ℓ distinct tags $c_2[1], ...c_2[\ell]$, setting up ℓ potential trap reports. One of these is a real commitment to m, and the rest are commitments to z. Since the length of c_2 is independent of the message, this significantly reduces the overhead of the scheme. Moreover, since we're sending all these together with one message, we can use the same seed s to generate the randomness needed to open all the commitments, further reducing communication costs. The seed s is also used to pick the index r_{swap} of the real message, allowing the message recipient to separate the real report data from the trap reports. When the moderator runs ModProcess, it separately MACs each version of c_2. Since ModProcess does not depend on c_1, the actual encryption of the message, it does not matter that the moderator does not have access to any encryptions of z.

We formalize this trap message design for onion franking in Figure 5. The same changes, shown in blue in the figure, can convert an E2EE message franking scheme into one that uses trap messages for $1/\ell$-server accountability against a potentially compromised message delivery server. Since the actual franking being done is identical to our previous onion franking scheme, the security analysis of the scheme is identical for unforgeability, deniability, confidentiality, and accountability adversaries who do not control malicious servers. For accountability adversaries who do control malicious servers, their violations of accountability are caught with probability $1/\ell$, as described above.

VII. Implementation and Evaluation

We implemented our general scheme and its optimized variant as described in Section V, as well as the zero knowledge and trap message schemes described in Section VI. Our implementation is in Rust, and we implement the group operations necessary for the ZK proof scheme using the `curve25519_dalek` Ristretto group [16]. Our implementation, evaluation, and results can be found at https://github.com/MatthewGregoire42/message_franking_crypto.

We instantiate our PRG with Rust's `StdRng`, based on the ChaCha stream cipher. We use HMAC-SHA256 for our MAC and commitment schemes, AES256-GCM for symmetric encryption, and Rust's `crypto_box` abstraction for public key encryption. Finally, we implement our hash function modeled as a random oracle using SHA3-256.

A. Evaluation Results

We evaluated our implementation on an 11th Gen Intel Core i7-11700K @ 3.6GHz processor running Ubuntu 22.04. We varied message lengths from 0 to 1,000 bytes in 100-byte increments, while varying the number of servers from 2 to 10. In addition, for the trap message scheme, we ran each of these experiments while varying the number of trap messages from 1 to 5 ($\ell = 2$ to $\ell = 6$), and we averaged each result over 1,000 trials. For all Send operations, where applicable, we separate the operations into offline and online computations.

$\mathsf{Send}(\mathbf{pk}, k_r, m)$

$s \xleftarrow{R} \{0,1\}^\lambda$
$k_{f,1}, \ldots, k_{f,\ell},$
$\quad r_1, \ldots, r_N, r_{\mathsf{swap}} \leftarrow \mathsf{G}(s)$
$c_1 \leftarrow \mathsf{Enc}(k_r, (m, s))$
$c_2 \leftarrow []$
$c_2.\mathsf{append}(\mathsf{Com}(k_{f,1}, m))$
for $i \in \{2, \ldots, \ell\}$:
$\quad c_2.\mathsf{append}(\mathsf{Com}(k_{f,i}, 0))$
$r_{\mathsf{sw}} \leftarrow r_{\mathsf{swap}} \mod \ell$
$\mathsf{swap}(c_2[1], c_2[r_{\mathsf{sw}}])$
$c_{3,0} \leftarrow \epsilon$
for $i \in \{1, \ldots, N\}$:
$\quad c_{3,i} \leftarrow \mathsf{Enc}(\mathsf{pk}_i, (c_{3,i-1}, r_i))$
$c_3 \leftarrow c_{3,N}$
return c_1, c_2, c_3

$\mathsf{ModProcess}(k_m, c_2, \mathsf{ctx})$

$c_{2,1}, \ldots, c_{2,\ell} \leftarrow c_2$
$\sigma \leftarrow []$
for $i \in \{1, \ldots, \ell\}$:
$\quad \sigma_i \leftarrow \mathsf{MAC.Sign}(k_m, c_{2,i}, \mathsf{ctx})$
$\quad \sigma.\mathsf{append}(\sigma_i)$
$\sigma_c \leftarrow H(\sigma, c_2, \mathsf{ctx})$
return σ, σ_c

$\mathsf{Read}(k_r, c_1, \mathsf{st})$

$m, s \leftarrow \mathsf{Dec}(k_r, c_1)$
$\epsilon, \mathsf{mrt}_0 \leftarrow \mathsf{st}$
$k_{f,1}, \ldots, k_{f,\ell},$
$\quad r_1, \ldots, r_N, r_{\mathsf{swap}} \leftarrow \mathsf{G}(s)$
for $i \in \{1, \ldots, N\}$:
$\quad \mathsf{mrt}_i \leftarrow \mathsf{mrt}_{i-1} \oplus r_i$
$c_2, \mathsf{ctx}, \sigma, \sigma_c \leftarrow \mathsf{mrt}_N$
if $\sigma_c \neq H(\sigma, c_2, \mathsf{ctx})$:
$\quad$ **return** $\perp$
$r_{\mathsf{sw}} \leftarrow r_{\mathsf{swap}} \mod \ell$
$\mathsf{swap}(c_2[1], c_2[r_{\mathsf{sw}}])$
$c_{2,1}, \ldots, c_{2,\ell} \leftarrow c_2$
$\mathsf{swap}(\sigma[1], \sigma[r_{\mathsf{sw}}])$
$\sigma_1, \ldots, \sigma_\ell \leftarrow \sigma$
if $!\mathsf{Com.Open}(c_{2,1}, m, k_{f,1})$:
$\quad$ **return** $\perp$
$\mathsf{rd}_1 \leftarrow (k_{f,1}, c_{2,1})$
$\mathsf{output}_1 \leftarrow (m, \mathsf{ctx}, \mathsf{rd}_1, \sigma_1)$
for $i \in \{2, \ldots, \ell\}$:
$\quad$ **if** $!\mathsf{Com.Open}(c_2, 0, k_{fi})$:
$\quad\quad$ **return** $\perp$
$\quad \mathsf{rd}_i \leftarrow (k_{f,i}, c_{2,i})$
$\quad \mathsf{report}_i \leftarrow (0, \mathsf{ctx}, \mathsf{rd}_i, \sigma_i,)$
return report

Fig. 5: Changes to the mixnet franking scheme with trap messages included. Here each message is accompanied by $\ell - 1$ trap messages. Changes from Construction V.1 are shown in blue. Functions not shown remain unchanged from Figure 2.

	Preprocessing	Send	ModProcess	Process	Read	Moderate
Onion-General	117.2μs	0.9μs	0.5μs	58.4μs	1.5μs	0.4μs
Onion-Optimized	—	1.6μs	0.6μs	1.6μs	1.5μs	0.5μs
Onion-zk	117.5μs	0.9μs	208.4μs	58.7μs	172.8μs	37.1μs
Onion-Trap	117.9μs	1.3μs	1.3μs	59.1μs	2.6μs	1.2μs
Hecate [28]	28.6μs	16.2μs	—	15.4μs	100.1μs	101.8μs
AMF [50]	—	233.1μs	—	—	225.1μs	225.2μs
Shared Franking [22]	—	8.6μs	6.3μs	0.8μs	8.6μs	11.3μs
E2EE Franking	—	0.9μs	0.2μs	—	0.8μs	0.4μs

TABLE I: Measured computation times for each onion franking operation, compared to prior work. All measurements are with 100 byte messages. For shared franking and all onion franking schemes, reported times are for two servers. Times for the trap message scheme are for two trap messages ($\ell = 3$). The reported preprocessing time for Hecate is the measured time for token generation (TGen) by the moderator.

	Send	Read	Report
Onion-General	$202 \cdot N + 138$B	138B	96B
Onion-zk	$338 \cdot N + 274$B	274B	128B
Onion-Trap	$138 \cdot N + 72 \cdot (N+1) \cdot \ell + 210$B	$210 + 72 \cdot \ell$B	96B
Hecate [28]	380B	484B	380B
AMF [50]	489B	489B	489B
Shared Franking [22]	124B	204B	144B
E2EE Franking	92B	156B	128B

TABLE II: Communication overhead to implement each scheme. The communication-reducing optimization described in the text can reduce all terms dependent on N to $80 \cdot N$: 64 bytes of overhead for each layer of public key encryption, and 16 bytes for each PRG seed. Thus onion franking can achieve communication costs comparable to, and sometimes even smaller than, prior works that add abuse reporting features to metadata-hiding communication systems.

The exception is our optimized scheme (Figure 3), for which this separation is not possible. For the rest of this section, reported costs only refer to the additional overhead incurred by adding onion franking to an existing system, and don't include the overhead of the onion encryption itself.

Computation overhead. Table I shows the concrete computation overhead for each of our schemes when sending 100 Byte messages through 2 servers, and using $\ell = 3$ for trap messages. See Appendix E for additional evaluation data on a range of parameters. Asymptotically, we find that almost all operations in our scheme are constant or linear in number of servers, message length, and number of trap messages. Moderator processing times are all constant with the exception of the trap message scheme, which needs to compute reporting tags for each trap message. Because the additional moderation ciphertexts ct_i contain one random mask per server, Process requires $\mathcal{O}(N)$ work to decrypt these ciphertexts. In addition, by moving work to the preprocessing phase, the Send operation has complexity independent from the number of servers. The exception is for the optimized scheme, which bundles this work within Send. Since the trap message scheme requires making $\mathcal{O}(\ell)$ reports for each message, we record this variant's Moderate complexity as $\mathcal{O}(\ell)$. For a more detailed view of asymptotic behavior in our schemes and prior work, see Table III in Appendix E.

The majority of the computational costs of our scheme are incurred by public key encryption or decryption. This includes the preprocessing stage where c_3 is prepared with server masks, as well as the Process algorithm, where servers remove a layer of encryption from c_3. Since our optimized scheme puts the additional data held by c_3 inside of a ciphertext already produced by the underlying scheme, the additional computation incurred in this scheme is simply the overhead of computing a few more blocks of a symmetric encryption or MAC, not that of introducing additional public-key operations. This makes our optimized scheme by far the most efficient overall. We note that the same optimization can just as easily be applied to the zero knowledge and trap schemes.

Communication overhead. We present the communication costs of our schemes in Table II. Overheads are presented abstractly in terms of N, the number of servers, and ℓ, the trap message parameter, where applicable. Since Send produces a mask r_i for each server, the communication overhead of this function depends on the number of servers. Concrete communication costs are shown in Appendix E.

As presented in this paper, our scheme requires a few hundred additional bytes of masking material r_i for each server, resulting in higher than necessary communication costs. We can avoid this with a simple mask-shrinking optimization. Instead of using r_i directly as the mask, servers receive a short, 16-Byte r_i which they use as a seed in a PRG which generates the longer mask.

B. Comparison to Prior Work

Tables I and II also compare the performance of onion franking to prior work. These schemes are described in more detail in Section VIII. We note that these comparisons are not necessarily all apples to apples, as onion franking is specifically tailored to schemes that use onion encryption, whereas our other

points of comparison are applicable in slightly different settings. Hecate [28] and Asymmetric Message Franking (AMF) [50] are generically applicable to any scheme and also support third party moderation, shared franking [22] only applies to schemes based on secret sharing, and E2EE franking is an instantiation of standard message franking techniques for E2EE messaging [39], [27]. Nonetheless we compare to these works to show how onion franking compares in those settings where they share applicability.

Hecate is the most relevant point of comparison for onion franking because it is the most performant scheme that can also be applied to systems that use onion encryption. Ignoring Hecate's additional preprocessing cost, which is over $10\times$ larger than any cost incurred by optimized onion franking, optimized onion franking outperforms Hecate by $10\times$ to send messages or process them through a server (or $7\times$ for the moderator server), by $67\times$ when reading a message, and by $204\times$ when moderating one. Optimized onion franking also comes within $2\times$ the cost of message franking for E2EE when sending and receiving messages, and more or less matches the time required for moderating them.

When considering our schemes that achieve stronger accountability, AMF becomes the most relevant point of comparison because it is the only prior work to achieve strong accountability, in either the E2EE or metadata-hiding settings. We use a Rust implementation of AMF as our point of comparison [38]. We find that using trap messages to provide $1/\ell$-server accountability significantly outperforms AMF for small to moderate values of ℓ, and the zero knowledge version of onion franking provides the same strong accountability as AMF with significantly reduced client costs, albeit at the cost of additional overhead for the moderator's message processing. The time to moderate reports, however, is reduced by $6\times$.

Onion franking achieves large performance improvements over prior work because it avoids the expensive signatures in Hecate and the zero knowledge proofs in AMF. In fact, the optimized onion franking scheme – as well as using trap messages for server accountability – only adds symmetric cryptographic operations on top of the underlying messaging system. When we do augment onion franking with zero knowledge proofs for strong accountability, we see performance much closer to that of prior work, although there are still improvements. By taking advantage of the fact that many metadata-hiding messaging system designs make use of onion encryption, we are able to achieve our goals using cryptographic tools much more akin to E2EE message franking than prior, more generic abuse reporting techniques like Hecate or AMF.

VIII. Related Work

This section discusses prior work on abuse reporting for private messaging as well as a small sampling of the myriad systems that use onion encryption or mixnets to achieve metadata-hiding messaging.

Abuse reporting for private messaging. Recent works exploring how to support abuse reporting for private messaging begin with Meta's message franking scheme [39] and subsequent analysis of this scheme [27], [20], [5]. Most relevant to this work are extensions that support abuse reporting for metadata-hiding communication settings, including AMF [50], Hecate [28], and shared franking [22], to which we have compared onion franking. AMF relies on ideas from designated verifier signature schemes [29] to achieve message franking in this setting for arbitrary messaging platforms, at the cost of expensive, heavyweight zero-knowledge proofs. Hecate improves upon the performance of AMF by allowing the platform to create and distribute consumable tokens to clients. Each message sent requires the use of one token and, if necessary, the token can be passed back to the moderator at moderation time. Shared franking, on the other hand, provides a message franking solution for metadata-hiding messaging systems based on secret sharing techniques [48]. Shared franking achieves performance gains by taking advantage of the messaging system architecture, similar in approach to the onion franking systems we present in this paper.

Another related problem is that of reporting and identifying the sources of misinformation. A body of work on message traceback or source tracking uses sometimes overlapping techniques to find the originator of misinformation messages rather than the direct sender of such a message [51], [40], [36], [4]. The Hecate scheme [28], examined here in the context of abuse reporting for metadata-hiding messaging, can also be used in the context of identifying the sources of misinformation. Scheffler and Mayer include discussions of these and other forms of content moderation for end-to-end encrypted platforms in a recent SoK [46].

Metadata-hiding messaging via onion encryption. Onion franking is designed to be integrated into messaging schemes that make use of onion encryption. An enormous body of work, beginning with Chaum's pioneering mixnets [9], use some form of onion encryption combined with mixing of ciphertexts to privately communicate. These range from systems that use low-latency mixing strategies like Loopix [42] to those that provide differential privacy guarantees like Vuvuzela [53], Alpenhorn [35], or Yodel [34], and to those that provide k-anonymity type guarantees [49], [31], [32], [33]. A number of systematization of knowledge papers [52], [45] explore various aspects of the design of metadata-hiding messaging schemes using mixnet-style approaches as well as other techniques, e.g., DC-nets [10]. The concept of trap messages used in our scheme has precedent in trap messages used as part of prior metadata-hiding communication schemes [30], [31]. In those works, however, the goal of trap messages is to provide metadata-privacy in the presence of malicious servers, not to support abuse reporting.

Independent from the work on metadata-hiding messaging are a number of works that explore the design and analysis of onion routing schemes, such as Tor [19]. These include both works that attempt to design more effective onion routing systems for various use-cases [11], [12], [14] and analysis of the cryptographic properties of onion-encrypted ciphertexts

intended to be used in such systems [15], [44], [17], [47]. These schemes are not directly related to messaging per se, but they rely on the same fundamental approach to protect anonymity in web browsing and other applications.

IX. Conclusion

We have presented onion franking, a mechanism by which metadata-hiding messaging systems based on onion encryption can support lightweight, verifiable abuse reports while maintaining relevant anonymity properties for message senders and receivers, even for reported messages. In addition to showing two variants of an onion franking construction, we have shown extensions that satisfy stronger accountability notions than those achieved by most prior works, and which apply to E2EE and metadata-hiding schemes alike. For each security level targeted, our scheme is the most efficient in terms of both computation and communication, making it an excellent choice for incorporation into metadata-hiding messaging schemes seeking to support abuse reporting features.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful comments and feedback to improve this paper.

This material is based upon work supported by the National Science Foundation under Grant No. 2234408, as well as a gift from Cisco. 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] "Technology preview: Sealed sender for signal," https://signal.org/blog/sealed-sender/, 2018, accessed 6/21/2024.

[2] "icloud private relay overview," https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf, 2021, accessed 6/21/2024. [Online]. Available: https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf

[3] "I2P Anonymous Network — geti2p.net," https://geti2p.net/en/, 2024.

[4] C. Bell and S. Eskandarian, "Anonymous complaint aggregation for secure messaging," *Proc. Priv. Enhancing Technol.*, 2024.

[5] M. Bellare and V. T. Hoang, "Efficient schemes for committing authenticated encryption," *EUROCRYPT*, 2022.

[6] M. Blum, P. Feldman, and S. Micali, "Non-interactive zero-knowledge and its applications (extended abstract)," in *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, J. Simon, Ed. ACM, 1988, pp. 103–112.

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

[8] J. Camenisch and M. Stadler, "Efficient group signature schemes for large groups (extended abstract)," in *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, 1997, pp. 410–424.

[9] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, vol. 24, no. 2, pp. 84–88, 1981.

[10] ——, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *J. Cryptology*, vol. 1, no. 1, pp. 65–75, 1988.

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

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

[13] R. Cramer, "Modular design of secure yet practical cryptographic protocols," Ph.D. dissertation, Jan. 1997.

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

[15] G. Danezis and I. Goldberg, "Sphinx: A compact and provably secure mix format," in *IEEE Symposium on Security and Privacy*, 2009.

[16] H. de Valence and I. A. Lovecruft, "curve25519-dalek (version 3.2.1)," 2023. [Online]. Available: https://docs.rs/curve25519-dalek/latest/curve25519_dalek/ristretto/index.html

[17] J. P. Degabriele and M. Stam, "Untagging tor: A formal treatment of onion encryption," in *EUROCRYPT*, J. B. Nielsen and V. Rijmen, Eds., 2018.

[18] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theory*, vol. 22, no. 6, pp. 644–654, 1976.

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

[20] Y. Dodis, P. Grubbs, T. Ristenpart, and J. Woodage, "Fast message franking: From invisible salamanders to encryptment," in *CRYPTO*, 2018.

[21] Y. Dodis, E. Kiltz, K. Pietrzak, and D. Wichs, "Message authentication, revisited," in *EUROCRYPT*, 2012.

[22] S. Eskandarian, "Abuse reporting for metadata-hiding communication based on secret sharing," *USENIX Security*, 2024.

[23] S. Faust, M. Kohlweiss, G. A. Marson, and D. Venturi, "On the non-malleability of the fiat-shamir transform," in *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, S. D. Galbraith and M. Nandi, Eds., vol. 7668. Springer, 2012, pp. 60–79.

[24] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO*, 1986.

[25] S. Goldwasser and S. Micali, "Probabilistic encryption," *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 270–299, 1984.

[26] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM J. Comput.*, vol. 18, no. 1, pp. 186–208, 1989.

[27] P. Grubbs, J. Lu, and T. Ristenpart, "Message franking via committing authenticated encryption," in *CRYPTO*, 2017.

[28] R. Issa, N. Alhaddad, and M. Varia, "Hecate: Abuse reporting in secure messengers with sealed sender," *USENIX Security*, 2022.

[29] M. Jakobsson, K. Sako, and R. Impagliazzo, "Designated verifier proofs and their applications," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1996, pp. 143–154.

[30] S. Khazaei, T. Moran, and D. Wikström, "A mix-net from any CCA2 secure cryptosystem," in *ASIACRYPT*, 2012.

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

[32] A. Kwon, D. Lu, and S. Devadas, "XRD: scalable messaging system with cryptographic privacy," 2020.

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

[34] ——, "Yodel: strong metadata security for voice calls," in *SOSP*, 2019.

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

[36] L. Liu, D. S. Roche, A. Theriault, and A. Yerukhimovich, "Fighting fake news in encrypted messaging with the fuzzy anonymous complaint tally system (FACTS)," in *NDSS*, 2022.

[37] I. Martiny, G. Kaptchuk, A. J. Aviv, D. S. Roche, and E. Wustrow, "Improving signal's sealed sender," in *NDSS*, 2021.

[38] S. Menda and M. Rosenberg, "amaze," https://github.com/sgmenda/amaze, 2022.

[39] Meta, "Messenger end-to-end encryption overview," https://engineering.fb.com/wp-content/uploads/2023/12/MessengerEnd-to-EndEncryptionOverview_12-6-2023.pdf, December 2023.

[40] C. Peale, S. Eskandarian, and D. Boneh, "Secure complaint-enabled source-tracking for encrypted messaging," in *ACM CCS*, 2021.

[41] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, ser. Lecture Notes in Computer Science, J. Feigenbaum, Ed., vol. 576. Springer, 1991, pp. 129–140.

[42] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The loopix anonymity system," in *USENIX Security*, 2017.

[43] P. Rogaway, "Authenticated-encryption with associated-data," in *ACM CCS*, 2002.

[44] P. Rogaway and Y. Zhang, "Onion-ae: Foundations of nested encryption," *Proc. Priv. Enhancing Technol.*, vol. 2018, no. 2, pp. 85–104, 2018.
[45] S. Sasy and I. Goldberg, "Sok: Metadata-protecting communication systems," *Proc. Priv. Enhancing Technol.*, vol. 2024, no. 1, pp. 509–524, 2024.
[46] S. Scheffler and J. R. Mayer, "Sok: Content moderation for end-to-end encryption," *Proc. Priv. Enhancing Technol.*, vol. 2023, no. 2, pp. 403–429, 2023.
[47] P. Scherer, C. Weis, and T. Strufe, "A framework for provably secure onion routing against a global adversary," *Proc. Priv. Enhancing Technol.*, vol. 2024, no. 2, pp. 141–159, 2024.
[48] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.
[49] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich, "Stadium: A distributed metadata-private messaging system," in *SOSP*, 2017.
[50] N. Tyagi, P. Grubbs, J. Len, I. Miers, and T. Ristenpart, "Asymmetric message franking: Content moderation for metadata-private end-to-end encryption," in *CRYPTO*, 2019.
[51] N. Tyagi, I. Miers, and T. Ristenpart, "Traceback for end-to-end encrypted messaging," in *ACM CCS*, 2019.
[52] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith, "Sok: Secure messaging," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015.
[53] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich, "Vuvuzela: scalable private messaging resistant to traffic analysis," in *SOSP*, 2015.

APPENDIX A
DEFERRED DEFINITIONS

Definition A.1 (Unforgeability)**.** We define the onion franking unforgeability experiment $\mathsf{FORG}[\mathcal{A}, \mathcal{F}, N, \lambda]$ in Figure 6 with respect to an efficient adversary $\mathcal{A}$, an onion franking scheme $\mathcal{F}$, a number of servers N, and a security parameter λ.

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

$$\mathsf{FORGAdv}(\mathcal{A}, \mathcal{F}, N, \lambda) = \Pr\big[\mathsf{FORG}[\mathcal{A}, \mathcal{F}, N, \lambda] = 1\big].$$

We say that a scheme $\mathcal{F}$ has *unforgeability* if, for all efficient adversaries $\mathcal{A}$, and all $N \in \mathbb{N}$,

$$\mathsf{FORGAdv}(\mathcal{A}, \mathcal{F}, N, \lambda) \leq \mathsf{negl}(\lambda).$$

Definition A.2 (Accountability)**.** We define the onion franking accountability experiment $\mathsf{ACCT}[\mathcal{A}, \mathcal{F}, N, \lambda]$ in Figure 7 with respect to an efficient adversary $\mathcal{A}$, an onion franking scheme $\mathcal{F}$, a number of servers N, and a security parameter λ.

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

$$\mathsf{ACCTAdv}(\mathcal{A}, \mathcal{F}, N, \lambda) = \Pr\big[\mathsf{ACCT}[\mathcal{A}, \mathcal{F}, N, \lambda] = 1\big]$$

We say that a scheme $\mathcal{F}$ has *accountability* if, for all efficient adversaries $\mathcal{A}$, and all $N \in \mathbb{N}$,

$$\mathsf{ACCTAdv}(\mathcal{A}, \mathcal{F}, N, \lambda) \leq \mathsf{negl}(\lambda).$$

Definition A.3 (Strong Accountability)**.** We define the onion franking strong accountability experiment $\mathsf{STRACCT}[\mathcal{A}, \mathcal{F}, N, \lambda]$ in Figure 7 with respect to an efficient adversary $\mathcal{A}$, an onion franking scheme $\mathcal{F}$, a number of servers N, and a security parameter λ.

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

$$\mathsf{STRACCTAdv}(\mathcal{A}, \mathcal{F}, N, \lambda) = \Pr\big[\mathsf{STRACCT}[\mathcal{A}, \mathcal{F}, N, \lambda] = 1\big]$$

We say that a scheme $\mathcal{F}$ has $1/\ell$*-strong accountability* if, for all efficient adversaries $\mathcal{A}$, and all $N \in \mathbb{N}$,

$$\mathsf{STRACCTAdv}(\mathcal{A}, \mathcal{F}, N, \lambda) \leq 1/k + \mathsf{negl}(\lambda).$$

Moreover, we say that a scheme $\mathcal{F}$ has *strong accountability* if, for all efficient adversaries $\mathcal{A}$, and all $N \in \mathbb{N}$,

$$\mathsf{STRACCTAdv}(\mathcal{A}, \mathcal{F}, N, \lambda) \leq \mathsf{negl}(\lambda).$$

APPENDIX B
DEFERRED PROOFS

Proof of Theorem V.2 (Unforgeability).

Proof. The proof proceeds through a short series of hybrids.

- Hyb_0 : This hybrid corresponds to the unforgeability experiment $\mathsf{FORG}[\mathcal{A}, \Pi, N, \lambda]$.
- Hyb_1 : This hybrid is identical to the preceding one, except we add an additional abort criterion to the experiment. The experiment will abort and output 0 if, in a call to $\mathcal{O}_{\mathsf{Verify}}$, $\mathsf{MAC.Verify}(k_m, (c_2, \mathsf{ctx}), \sigma) = 1$, and at least one of the following conditions are met:
  - $(\mathsf{ctx}, \sigma) \notin T_{\mathsf{Frank}}$
  - $(\cdot, \cdot, \cdot, \sigma) \in T_{\mathsf{Read}} \wedge (\cdot, \mathsf{ctx}, (\cdot, c_2), \sigma) \notin T_{\mathsf{Read}}$

  In Lemma B.1, we show that this hybrid is indistinguishable from the preceding one by the existential unforgeability of MAC.
- Hyb_2 : This hybrid is identical to the preceding one, except we add an additional abort criterion to the experiment. The experiment will abort and output 0 if, in a call to $\mathcal{O}_{\mathsf{Verify}}$, $\mathsf{Com.Open}(c_2, m, k_f) = 1$, and there exists an element $(m', \cdot, (k'_f, c_2), \cdot) \in T_{\mathsf{Read}}$ where $m' \neq m$ or $k'_f \neq k_f$. In Lemma B.2, we show that this hybrid is indistinguishable from the preceding one by the binding property of Com.

We now show that in Hyb_2, the adversary has unforgeability advantage 0. Suppose toward contradiction that at some point in the experiment, $\mathcal{O}_{\mathsf{Verify}}$ outputs 1. This means that $\mathsf{Moderate}(k_m, m, \mathsf{ctx}, (k_f, c_2), \sigma) = 1$ and that one of the following conditions is met:

(1) $(\mathsf{ctx}, \sigma) \notin T_{\mathsf{Frank}}$
(2) $(\cdot, \cdot, \cdot, \sigma) \in T_{\mathsf{Read}} \wedge (m, \mathsf{ctx}, (k_f, c_2), \sigma) \notin T_{\mathsf{Read}}$

Suppose $\mathcal{O}_{\mathsf{Verify}}$ outputs 1 because condition (1) is met. In order for Moderate to output 1, it must be that $\mathsf{MAC.Verify}(k_m, (c_2, \mathsf{ctx}), \sigma) = 1$. But then the experiment will abort and output 0 by the abort criterion introduced in Hyb_1. This is a contradiction, so the adversary cannot win by meeting condition (1).

Now suppose $\mathcal{O}_{\mathsf{Verify}}$ outputs 1 because condition (2) is met. In order for Moderate to output 1, it must be that $\mathsf{MAC.Verify}(k_m, (c_2, \mathsf{ctx}), \sigma) = 1$, which means that $(\cdot, \mathsf{ctx}, (\cdot, c_2), \sigma) \in T_{\mathsf{Read}}$ or else there is a contradiction of the abort criterion introduced in Hyb_1. In order for Moderate to output 1, it must also hold that $\mathsf{Com.Open}(c_2, m, k_f) = 1$. By the abort criterion introduced in Hyb_2, this implies that there is no element $(m', \cdot, (k'_f, c_2), \cdot) \in T_{\mathsf{Read}}$ where $m' \neq m$ or $k'_f \neq k_f$. Since we know there is an element in $(\cdot, \mathsf{ctx}, (\cdot, c_2), \sigma) \in T_{\mathsf{Read}}$, this means that the same entry must contain m and k_f. Ergo $(m, \mathsf{ctx}, (k_f, c_2), \sigma) \in T_{\mathsf{Read}}$. But this contradicts condition (2).

Since neither condition for $\mathcal{O}_{\mathsf{Verify}}$ outputting 1 can be met, the adversary must have no advantage. As we have shown

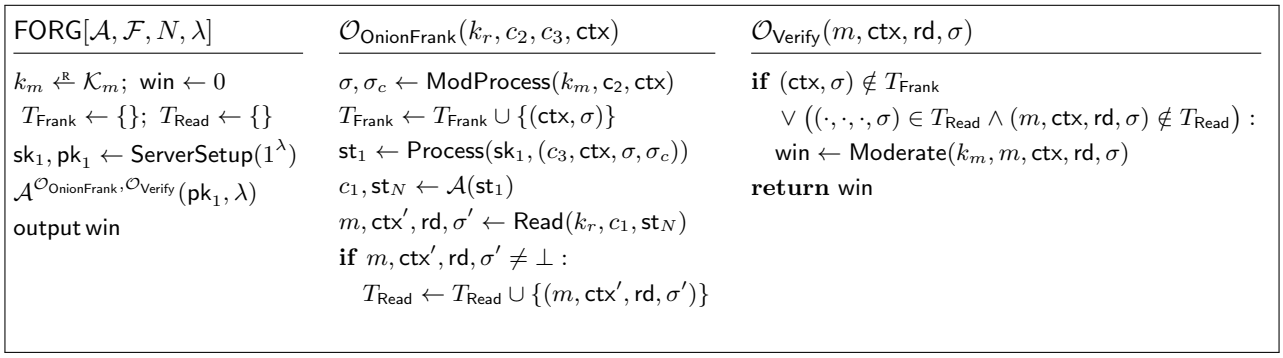


Fig. 6: Onion franking unforgeability security experiment (Definition A.1).

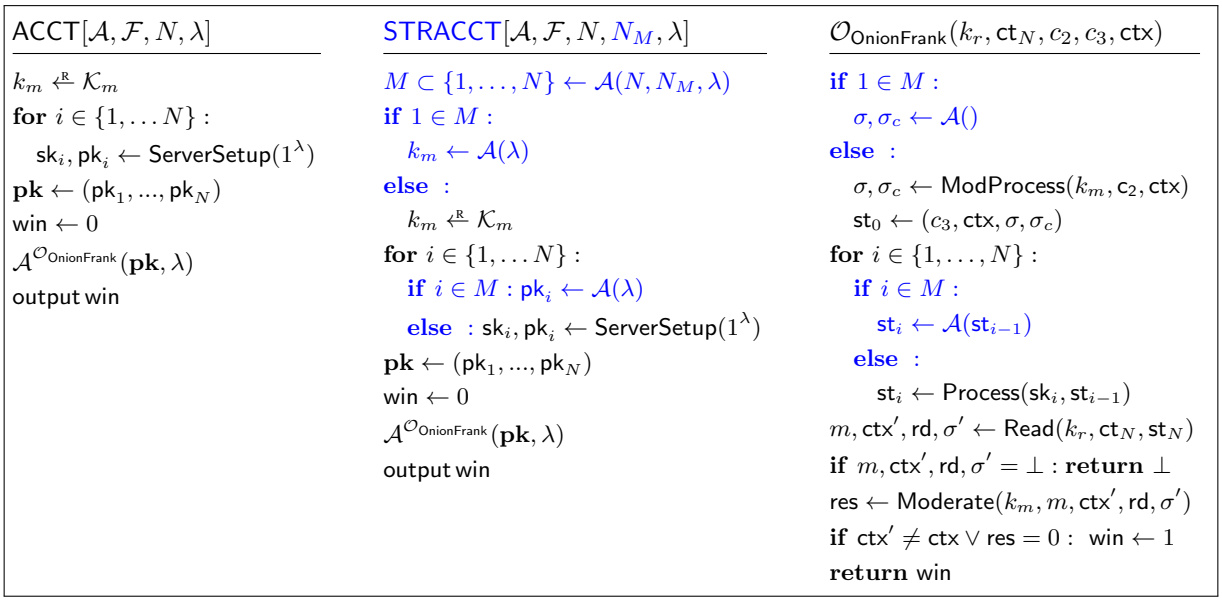


Fig. 7: The onion franking accountability security experiment (Definition A.2), and corresponding OnionFrank oracle, is shown in black. Additions and changes necessary to achieve strong accountability (Definition A.3) are shown in blue.

that the adversary has no advantage in Hyb_2, the proof of the theorem follows from the proofs of the lemmas and the triangle inequality.

Lemma B.1. *Suppose that for every adversary $\mathcal{B}$ attacking* MAC, *the advantage of $\mathcal{B}$ in breaking the existential unforgeability of* MAC *is at most* $\mathsf{MACAdv}(\mathcal{B}, \mathsf{MAC}, \lambda) \leq \mathsf{negl}(\lambda)$. *Then for all adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]\right| \leq \mathsf{MACAdv}(\mathcal{B}, \mathsf{MAC}, \lambda) \leq \mathsf{negl}(\lambda).$$

Proof. We show how to use an adversary $\mathcal{A}$ who distinguishes between the hybrids to build an adversary $\mathcal{B}$ who breaks the unforgeability of the MAC scheme. For simplicity, we use the (equivalent) variant of the standard MAC definition where the adversary is given access to verification queries [7]. We note that this choice asymptotically makes no difference, but it does make a difference in terms of the resulting concrete security bound because the version of the definition with verification queries adds a factor of Q_v, the number of verification queries, to the adversary's advantage in the security analysis. Since our reduction requires $O(Q_{\mathsf{OF}}^2)$ MAC verification queries, where Q_{OF} is the number of queries to the $\mathcal{O}_{\mathsf{OnionFrank}}$ oracle, the choice of definition hides a $O(Q_{\mathsf{OF}}^2)$ factor in the concrete security analysis.

Adversary $\mathcal{B}$ plays the role of the adversary in the MAC scheme, and the role of the challenger in the unforgeability experiment, perfectly simulating expriment Hyb_1 except whenever a call is made to MAC.Sign or MAC.Verify, $\mathcal{B}$ forwards the inputs to the MAC challenger to get the tag σ or verification response $1/0$. Moreover, $\mathcal{B}$ keeps track of the list $(\sigma_1, ..., \sigma_{Q_{\mathsf{OF}}})$ of tags returned by the MAC challenger, and before requesting a tag on the i^{th} MAC input $(c_2, \mathsf{ctx})_i$, it queries the MAC verification oracle with inputs $((c_2, \mathsf{ctx})_i, \sigma_1), ..., ((c_2, \mathsf{ctx})_i, \sigma_{i-1})$, which we call the *collision queries*. If there is ever a case where one of the collision queries results in acceptance (assuming without loss of generality that MAC signing queries are distinct), $\mathcal{B}$ has successfully produced a MAC forgery. If at any point in the experiment, the abort criterion introduced in Hyb_1 is met,

$\mathcal{B}$ forwards the values $(c_2, \mathsf{ctx}), \sigma$ that triggered the criterion as a MAC forgery.

We will prove that $\mathcal{B}$ breaks the existential unforgeability of MAC with at least the same probability that $\mathcal{A}$ triggers the abort criterion. Since the abort criterion is the only difference between hybrids Hyb_0 and Hyb_1, this completes the proof of the lemma.

Suppose that the abort criterion is met because $(\mathsf{ctx}, \sigma) \notin T_{\mathsf{Frank}}$. Since elements are added to T_{Frank} immediately after ModProcess, the only function where MAC tags are produced, this means that for each entry $((\cdot, \mathsf{ctx}), \sigma) \in T_{\mathsf{MAC}}$ in the table of MAC tags produced by the MAC challenger, there is an entry $(\mathsf{ctx}, \sigma) \in T_{\mathsf{Frank}}$. The contrapositive of this statement is that if $(\mathsf{ctx}, \sigma) \notin T_{\mathsf{Frank}}$, then $((\cdot, \mathsf{ctx}), \sigma) \notin T_{\mathsf{MAC}}$. But then if Moderate outputs 1 for $(\mathsf{ctx}, \sigma) \notin T_{\mathsf{Frank}}$, this implies that MAC.Verify accepts a tuple $((\cdot, \mathsf{ctx}), \sigma) \notin T_{\mathsf{MAC}}$, meaning that $\mathcal{B}$ wins the MAC experiment.

Alternatively, suppose that the abort criterion is met because $(\cdot, \cdot, \cdot, \sigma) \in T_{\mathsf{Read}} \wedge (\cdot, \mathsf{ctx}, (\cdot, c_2), \sigma) \notin T_{\mathsf{Read}}$. Since $(\cdot, \cdot, \cdot, \sigma) \in T_{\mathsf{Read}}$, there exists some $(\cdot, \mathsf{ctx}'', (\cdot, c_2''), \sigma) \in T_{\mathsf{Read}}$. If $((c_2'', \mathsf{ctx}''), \sigma) \notin T_{\mathsf{MAC}}$, this means that $\mathcal{B}$ wins the MAC experiment. On the other hand, if $((c_2'', \mathsf{ctx}''), \sigma) \in T_{\mathsf{MAC}}$, then we have two distinct MAC inputs (c_2, ctx) and (c_2'', ctx'') with the same σ, so $\mathcal{B}$ would have won the MAC experiment with one of the collision queries. Note that $(c_2'', \mathsf{ctx}'') \neq (c_2, \mathsf{ctx})$ because one pair appears in T_{Read} while the other does not.

Thus whenever $\mathcal{A}$ triggers the abort criterion, $\mathcal{B}$ wins the MAC security experiment. Since we assume that the advantage of any $\mathcal{B}$ in breaking MAC security is at most negligible, the same applies to the advantage of $\mathcal{A}$ in distinguishing between the hybrids. □

Lemma B.2. *Suppose that for every adversary $\mathcal{B}$ attacking* COM, *the advantage of $\mathcal{B}$ in breaking the binding property of* Com *is at most* $\mathsf{BINDAdv}(\mathcal{B}, \mathsf{Com}, \lambda) \leq \mathsf{negl}(\lambda)$. *Then for all adversaries $\mathcal{A}$,*

$$\big|\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1]\big| \leq \mathsf{BINDAdv}(\mathcal{B}, \mathsf{Com}, \lambda) \leq \mathsf{negl}(\lambda).$$

Proof. We show how to use an adversary $\mathcal{A}$ who distinguishes between the hybrids to build an adversary $\mathcal{B}$ who breaks the binding property of the commitment scheme. Adversary $\mathcal{B}$ plays the role of the binding adversary for the commitment scheme while playing the role of the challenger in the Hyb_2 unforgeability experiment. If, during the course of the experiment, the abort criterion introduced in Hyb_2 is triggered, $\mathcal{B}$ submits $c_2, (m, k_f), (m', k_f')$ to the binding challenger.

We prove that $\mathcal{B}$ breaks the binding of the commitment scheme with the same probability that $\mathcal{A}$ triggers the Hyb_2 abort criterion. Since this abort criterion is the only difference between the two hybrids, this suffices to prove the lemma. The criterion requires that in a call to $\mathcal{O}_{\mathsf{Verify}}$ where $\mathsf{Com.Open}(c_2, m, k_f) = 1$,, there exists an element $(m', \cdot, (k_f', c_2), \cdot) \in T_{\mathsf{Read}}$ with $m' \neq m$ or $k_f' \neq k_f$. Since $(m', \cdot, (k_f', c_2), \cdot) \in T_{\mathsf{Read}}$, this means that m', k_f', c_2 were the outputs of a call to Read() in the $\mathcal{O}_{\mathsf{OnionFrank}}$ oracle, which requires that $\mathsf{Com.Open}(c_2, m', k_f') = 1$. But this means that we have $c_2, (m, k_f), (m', k_f')$ where $(m, k_f) \neq (m', k_f')$, $\mathsf{Com.Open}(c_2, m, k_f) = 1$, and $\mathsf{Com.Open}(c_2, m', k_f') = 1$, which is exactly the criteria for breaking the binding of Com.

Thus whenever $\mathcal{A}$ triggers the abort criterion, $\mathcal{B}$ wins the binding security experiment. Since we assume that the advantage of any $\mathcal{B}$ in breaking binding is at most negligible, the same applies to the advantage of $\mathcal{A}$ in distinguishing between the hybrids.

□

□

Proof of Theorem V.3 (Accountability).

Proof. The proof proceeds through a short series of hybrids.

- Hyb_0 : This hybrid corresponds to the accountability experiment $\mathsf{ACCT}[\mathcal{A}, \Pi, N, \lambda]$.
- Hyb_1 : This hybrid is identical to the preceding one, except that we replace the outputs of MAC.Sign with uniformly random strings of the same length. The correctness of the MAC is maintained by keeping a table T_{MAC} of MAC inputs/outputs, and MAC.Verify accepts all inputs that appear in T_{MAC}.
  Recall that we built our scheme with a MAC scheme where MAC.Sign also functions as a PRF. In Lemma B.3, we show that this hybrid is indistinguishable from the preceding one by the PRF security of MAC.Sign.
- Hyb_2 : This hybrid is identical to the preceding one, except we add an additional abort condition to the execution of the experiment. The experiment aborts and outputs 0 if during a call to $\mathcal{O}_{\mathsf{OnionFrank}}$, the outputs $c_2', \mathsf{ctx}', \sigma'$ of Read (if they are not $\perp$) differ from the values $c_2, \mathsf{ctx}, \sigma$ that are inputs/outputs of ModProcess.
  In Lemma B.4, we show that this hybrid is indistinguishable from the preceding one, relying on the fact that the function H is modeled as a random oracle (without programming the random oracle).

We now show that the adversary has no advantage in Hyb_2. Combined with the proofs of the lemmas below and the triangle inequality, this completes the proof of the theorem.

Observe that since Read and Moderate make calls to Com.Open on the exact same inputs, this call will always have the same output in the two functions. Moreover, since $(c_2, \mathsf{ctx}, \sigma) = (c_2', \mathsf{ctx}', \sigma')$, or else the experiment aborts, and $\sigma = \mathsf{MAC.Sign}(k_m, (c_2, \mathsf{ctx}))$, the correctness of MAC ensures that $\mathsf{MAC.Verify}(k_m, (c_2', \mathsf{ctx}'), \sigma') = 1$. Thus whenever Read outputs something other than $\perp$, it follows that Moderate will output 1, so $\mathsf{res} = 1$. Finally, since $(c_2, \mathsf{ctx}, \sigma) = (c_2', \mathsf{ctx}', \sigma')$, it follows that $\mathsf{ctx} = \mathsf{ctx}'$. But this means that it is never possible for $\mathsf{win} \leftarrow 1$ to be reached.

Lemma B.3. *Suppose that* MAC.Sign *is a correct MAC, and that for every adversary $\mathcal{B}$ attacking* MAC.Sign, *the advantage of $\mathcal{B}$ in breaking the PRF security of* MAC.Sign

is at most $\mathsf{PRFAdv}(\mathcal{B}, \mathsf{MAC.Sign}, \lambda) \leq \mathsf{negl}(\lambda)$. *Then for all adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]\right| \leq \mathsf{PRFAdv}(\mathcal{B}, \mathsf{MAC.Sign}, \lambda) \leq \mathsf{negl}(\lambda).$$

Proof. We show how to use $\mathcal{A}$ to build an adversary $\mathcal{B}$ who breaks the PRF security of MAC.Sign with the same advantage that $\mathcal{A}$ distinguishes between the two hybrids.

Adversary $\mathcal{B}$ plays the role of the PRF adversary for MAC.Sign and simulates the Hyb_1 challenger for $\mathcal{A}$. It provides a perfect simulation of Hyb_1, except that whenever a call is made to MAC.Sign, it forwards the input to the PRF challenger and passes on the challenger's response as σ. At the end of the experiment, $\mathcal{B}$ passes on the output of the accountability experiment as its own output.

Observe that if the PRF challenger is providing $\mathcal{B}$ with evaluations of a PRF, then $\mathcal{B}$ is providing $\mathcal{A}$ a perfect simulation of Hyb_0. The additional bookkeeping used to maintain T_{MAC} has no impact on the view of the adversary because the correctness of MAC ensures that replacing verification of known MAC inputs with table lookups has the same output behavior. On the other hand, if the PRF challenger is providing $\mathcal{B}$ with random strings, then $\mathcal{B}$ is providing $\mathcal{A}$ with a perfect simulation of Hyb_1. Thus $\mathcal{B}$ distinguishes between PRF outputs and random strings with the same advantage that the outputs of $\mathcal{A}$ distinguish between Hyb_0 and Hyb_1.

□

Lemma B.4. *Assuming that the hash function* H *is modeled as a random oracle, then for all adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1]\right| \leq \mathsf{negl}(\lambda).$$

Proof. Observe that the values $c_2, \mathsf{ctx}, \sigma, \sigma_c$ that are inputs and outputs for ModProcess are passed through a number of calls to Process, wherein they are XORed with values r_i chosen by the adversary. This means that the values $c_2', \mathsf{ctx}', \sigma', \sigma_c'$ output by Read are the results of XORing some adversary chosen values a, b with the values $\sigma_c, (c_2, \mathsf{ctx}, \sigma)$. Note that $\sigma_c = H(c_2, \mathsf{ctx}, \sigma)$ and that σ is a uniformly random value for each (c_2, ctx) tuple. Since H is modeled as a random oracle, this means that for each input (c_2, ctx), there is an independent and uniformly random (σ, σ_c). While the adversary may make multiple queries to $\mathcal{O}_{\mathsf{OnionFrank}}$ with the same (c_2, ctx), it is never shown the corresponding values of (σ, σ_c).

In order to trigger the abort condition introduced in Hyb_2, the only case where the behavior of the two experiments differs, the adversary must find $(c_2, \mathsf{ctx}, a, b)$ with nonzero (a, b) for which $\sigma_c \oplus a = H((c_2, \mathsf{ctx}, \sigma) \oplus b)$. Define event E_x to be the event that the adversary triggers the condition with a query where $a = x$. Then $\Pr[E_x]$ is the probability that the adversary makes a query with $(c_2, \mathsf{ctx}, x, b)$ where $H(c_2, \mathsf{ctx}, \sigma) \oplus x = H((c_2, \mathsf{ctx}, \sigma) \oplus b)$. This is at most the probability of the adversary finding a collision in H and is therefore a negligible value ϵ_{coll}. Now an adversary that makes at most Q_{OF} calls to $\mathcal{O}_{\mathsf{OnionFrank}}$ can only try Q_{OF} distinct values of a. We call the i^{th} value queried a_i, so the adversary's probability of success in triggering the abort is, by union bound, $\Sigma_{i=1}^{Q_{\mathsf{OF}}} \Pr[E_{a_i}] = Q_{\mathsf{OF}} \cdot \epsilon_{\mathsf{coll}} \leq \mathsf{negl}(\lambda)$. □

□

Appendix C
Strong Accountability Zero Knowledge Details

Figure 8 presents the formal details of the commitment scheme, MAC scheme, and zero knowledge proof used to achieve strong accountability by proving honesty. We use the Camenisch-Stadler notation [8] to describe the statement to be proved. The commitment is a variation of the Pedersen commitment [41], and the MAC scheme is due to Dodis et al. [21]. The figure describes the statement to be proved and directly assigns that value to π as a shorthand for the generic linear Σ-protocol used in our implementation.

Appendix D
Other Applications of Onion Franking

We propose onion franking as a mechanism for supporting verifiable abuse reporting on metadata-hiding messaging platforms. However, these same techniques can potentially be applied in other areas where there is anonymous communication accompanied by potential for abuse. For example, many distributed systems rely on onion encryption to anonymize network traffic, such as Tor [19] and i2p [3]. For these systems, our "message sender" is an end user, and the "message receiver" is a web server or other service provider. These service providers have no recourse against users spamming the system, since the users' identities are hidden. Onion franking would enable such systems to allow websites receiving anonymized traffic to verifiably report abusive content to servers providing the anonymity service. This can be most easily envisioned in a more centralized scheme like Apple's iCloud Private Relay [2]. This application of onion franking fundamentally changes the nature of the security properties the anonymity service provides for users, so we urge caution in considering the consequences of a broader deployment in this setting. Nonetheless, this serves as one other example of where onion franking could be used.

Moving further away from the messaging use case, applications in a number of scenarios can be thought of as special cases of the private messaging problem. For example, private transactions in payment apps or accumulation of data from sensor networks, with privacy constraints on where each piece of data comes from, can both be thought of as cases of a platform passing messages between various parties, where the messages must satisfy certain syntactical and semantic constraints in addition to the privacy requirements of private messaging. Whenever potential for abuse arises in these settings (e.g., fraudulent charges, maliciously misreported data), an abuse reporting mechanism like onion franking could be deployed to allow an authority operating the system to intervene.

	Preprocessing	Send	ModProcess	Process	Read	Moderate
Onion-General	$\mathcal{O}(N)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(N+m)$	$\mathcal{O}(1)$
Onion-Optimized	—	$\mathcal{O}(N \cdot m)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(N+m)$	$\mathcal{O}(1)$
Onion-zk	$\mathcal{O}(N)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(N+m)$	$\mathcal{O}(1)$
Onion-Trap	$\mathcal{O}(N \cdot \ell)$	$\mathcal{O}(\ell+m)$	$\mathcal{O}(\ell)$	$\mathcal{O}(N \cdot \ell)$	$\mathcal{O}(N \cdot \ell + m)$	$\mathcal{O}(\ell)$
Hecate [28]	$\mathcal{O}(1)$	$\mathcal{O}(m)$	—	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$
AMF [50]	—	$\mathcal{O}(m)$	—	—	$\mathcal{O}(m)$	$\mathcal{O}(1)$
Shared Franking [22]	—	$\mathcal{O}(N \cdot m)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(N \cdot m)$	$\mathcal{O}(N \cdot m)$
E2EE Franking	—	$\mathcal{O}(m)$	$\mathcal{O}(1)$	—	$\mathcal{O}(m)$	$\mathcal{O}(1)$

TABLE III: Asymptotic performance overhead. N is the number of servers, ℓ is the number of trap messages, and m is the message length.

$\mathsf{Com.Commit}(r, m)$

$x_0, x_1 \leftarrow m$
return $g_0^{x_0} g_1^{x_1} h^r$

$\mathsf{Com.Open}(c, m, r)$

$x_0, x_1 \leftarrow m$
if $c = g_0^{x_0} g_1^{x_1} h^r$:
  return 1
else :
  return 0

$\mathsf{MAC.KeyGen}(\lambda)$

$x_0, x_1 \xleftarrow{R} \mathbb{F}_p^2$
$k \leftarrow (x_0, x_1)$
return k

$\mathsf{MAC.Sign}(k, m)$

$x_0, x_1 \leftarrow k$
$u \xleftarrow{R} \mathbb{G} \setminus \{1\}$
$u' \leftarrow u^{x_0 + H(m) x_1}$
$\sigma \leftarrow (u, u')$
return σ

$\mathsf{MAC.Verify}(k, m, \sigma)$

$u, u' \leftarrow \sigma$
if $u \neq 1 \wedge u' = u^{x_0 + H(m) x_1}$:
  return 1
else :
  return 0

$\mathsf{ZK.Prove}(\sigma, c_2, \mathsf{ctx})$

$x_0, x_1 \leftarrow k_m$
$u, u' \leftarrow \sigma$
$v \leftarrow u^{H(c_2, \mathsf{ctx})}$
$\pi \leftarrow \{(x_0, x_1, r) :$
  $\sigma_k = g_0^{x_0} g_1^{x_1} h^r \wedge u' = u^{x_0} v^{x_1}\}$
return π

Fig. 8: Implementation details of the MAC, commitment, and zero-knowledge proof schemes used in our zero knowledge construction for strong accountability. We omit the ZK.Verify procedure which verifies the proof π output by ZK.Prove. Here σ_k, g_0, g_1, and h are public parameters of the scheme published beforehand. $\mathbb{G}$ is a cyclic group with prime order p, and H is a hash function which takes arbitrary strings from $\{0,1\}^*$ to $\mathbb{F}_p$.

APPENDIX E
ADDITIONAL EVALUATION DATA

In this appendix, we include a variety of graphs to demonstrate relationships between our schemes' overheads and various parameters: message length, number of servers, and number of trap messages. We also include a complete analysis of each algorithm's asymptotic complexity, as well as the complexity of operations in prior work, reported in Table III.

Our evaluation shows that increasing message length gradually increases computation costs as we encrypt or MAC longer strings, as can be seen in Figures 9i and 9h, but has little effect on communication overhead, as seen in Figure 9g. We can see that message length only has a modest effect on our schemes' performance. For example, our optimized scheme with 10 servers requires 19.4μs to send and 1.8μs to receive a 100 byte message, as opposed to 22.8μs to send and 3.5μs to receive a 1000 byte message.

Increasing the number of servers increases the number of public key operations invoked in Send preprocessing, as well as in Process. As these public key operations are relatively expensive, increasing the number of servers has the largest impact on our schemes' performance. This can be seen in Figures 9a, 9b, and 9c. Each additional server adds approximately 60μs of client-side preprocessing overhead for all but our optimized scheme.

Additionally, incrementing ℓ by 1 requires more bits of random mask, incurring an extra 0.5μs in pre-processing time. It also linearly increases the number of symmetric primitives computed in Send, ModProcess, Read, and Moderate, resulting in extra running time of approximately 0.2μs, 0.4μs, 0.6μs, and 0.4μs, respectively. This can be seen in Figures 9e, 9d, and 9f. In all other schemes, ModProcess runs in constant time.

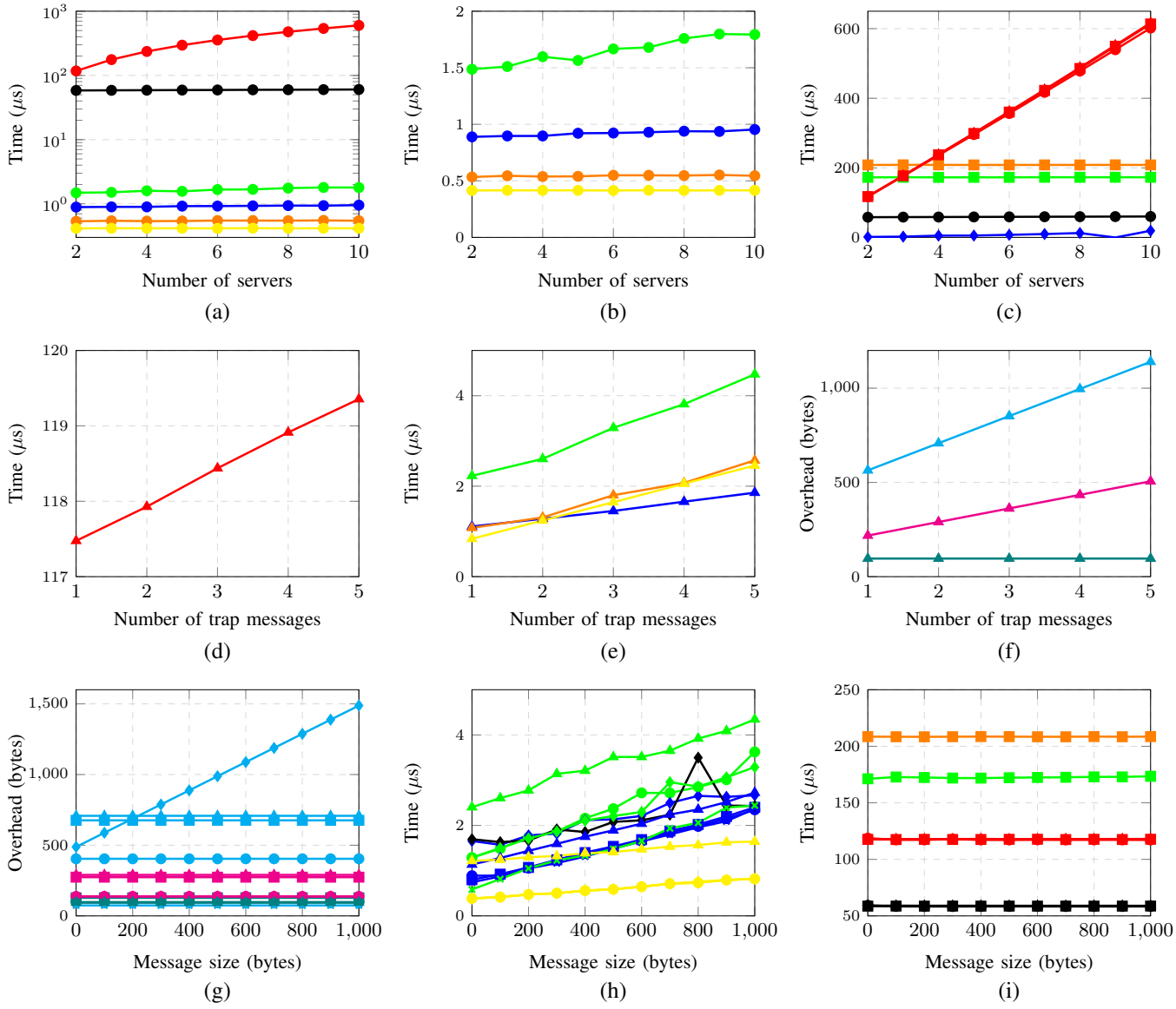
APPENDIX F
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: Our artifact can be accessed at DOI https://doi.org/10.5281/zenodo.14225977. Prior work which we compare to can be accessed as follows:

- Asymmetric message franking: https://github.com/initsecret/amaze
- Hecate: https://github.com/Ra1issa/hecate
- Shared franking: https://github.com/SabaEskandarian/Shared_Franking/tree/main

2) Hardware dependencies: None.



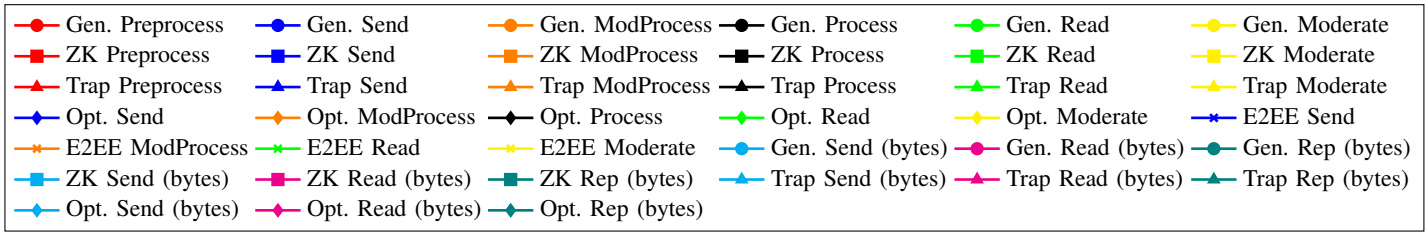


Fig. 9: Combined performance analysis showing the impact of different parameters on our scheme. The first row shows the effect of varying server numbers, the second row shows the impact of trap messages, and the third row shows the effect of message size on computation and communication overheads. Here "Gen" refers to our general onion franking scheme (Figure 2), and "Opt" refers to the optimized scheme (Figure 3).

3) Software dependencies: The artifact requires Rust to be installed. All Rust crate dependencies are handled automatically by cargo.

4) Benchmarks: None.

B. Artifact Installation & Configuration

This artifact requires no configuration. Once the repository is cloned, navigate to the project directory and run `cargo run --release` to install and execute the artifact. To run comparisons to prior work, follow their respective installation and experiment instructions.

C. Experiment Workflow

Please see section F-E.

D. Major Claims

- (C1): Compared to prior work, onion franking and its variants (with the exception of Onion-zk) provide one to two orders of magnitude of speed improvement for various message franking operations. Specifically, all operations except Process on Onion-General, Onion-ZK, and Onion-Trap This is proven by experiment (E1) whose results are demonstrated in Table I.
- (C2): Onion franking has communication overhead that's comparable to prior work for message reading and reporting operations, and higher message sending overhead. These overheads increase as a linear function of the number of servers (and the number of trap messages in the trap message scheme). This is proven by experiment (E1) and demonstrated by Table II.

E. Evaluation

1) Experiment (E1): [20 minutes compute]: Obtain computation and communication overheads for onion franking ran with different message sizes and differing numbers of servers and trap messages.

[How to] Run the code in the execution section below.

[Preparation] To run the included unit tests on the code, run `cargo test`. You should see output saying that all tests pass.

[Execution] Run the command `cargo run --release`. The results will be printed to stdout, or can be redirected into a file.

[Results] Each line of the output will show computation overheads (in nanoseconds) and communication overheads (in bytes) for a unique choice of (scheme variant, number of servers, message size, number of trap messages) parameters, averaged over (by default) 1000 trials. Numbers of trap messages are only varied for the trap message scheme. And because plain E2EE message franking does not have any parameters other than message size, no parameters other than message size are varied.

For most operations, for a given message size, the onion franking computation overhead is very similar to that of plain franking, and much lower than for prior work. This supports claim (C1) and data in Table I comes directly from running (E1).

In addition, we can see that communication size overhead follows the formulas as described in Table II, with size increasing as functions of the number of servers (N) and the number of trap messages (ℓ), supporting claim (C2). Visualizations of the data supporting claims (C1) and (C2) can be seen in Figure 9.

F. Customization

To increase performance or, alternatively, to increase the amount of detail provided, the constants on lines 17-19 in `main.rs` can be modified. This enables varying the number of trials each line of output performs, the maximum number of servers, and the maximum number of trap messages.