TOWARD SAFER AND MORE FEATUREFUL ENCRYPTED PLATFORMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Armin Namavari

August 2025

TOWARD SAFER AND MORE FEATUREFUL ENCRYPTED PLATFORMS

Armin Namavari, Ph.D.

Cornell University 2025

End-to-end encrypted (E2EE) messaging is a modern success story for user privacy. Even if the servers of iMessage, WhatsApp, or Signal were breached, the content exchanged over those platforms would not be available to attackers. This threat model, though strong, does not account for harms that may arise from the content itself. Hate, harassment, and misinformation are several examples of such harms. Platforms protect users from abuse via content moderation. Yet such mitigations often require visibility of the content, which is at odds with end-to-end encryption. This tension has led to calls for "private" scanning of E2EE content, which have been sharply criticized by privacy advocates.

This dissertation presents novel techniques for balancing user safety with the strong privacy guarantees of end-to-end encryption. In particular, I investigate how providing enhanced tools for community moderation and content reporting can protect users without compromising on their privacy or agency. First, I introduce a novel framework for private hierarchical governance, which enables rich governance features for encrypted messaging groups while allowing for platform-driven moderation. I then go on to propose a new cryptographic primitive called transcript franking, which extends prior work on reporting encrypted messages to handle sequences of messages with integrity over their causal ordering. Finally, I propose shared encrypted state franking, which enables the maintenance and reporting of shared state in E2EE platforms.

BIOGRAPHICAL SKETCH

Armin Namavari was born in Los Angeles, California. He moved to New York City in 2003 and moved again to the San Francisco Bay Area in 2005. He stayed in the Bay Area to pursue his undergraduate studies at Stanford University, graduating with a B.S. in computer science in 2019 and with an M.S. in computer science in 2020. His studies introduced him to the uniquely interdisciplinary nature of security and cryptography, as well as the societal implications of these fields. This led him to pursue his PhD at Cornell, which he began in August 2020, at the height of the COVID-19 pandemic. He spent his first year in Ithaca and later moved to New York City in the summer before his second year, where he continued at the Cornell Tech campus. Throughout his PhD, he completed two industry research internships: one at Zoom and another at Microsoft Research in Redmond. He will start a research scientist position on an infrastructure security team at Meta in September.

To my parents, who have always supported and encouraged me during every step of my education.

ACKNOWLEDGEMENTS

It has been an immense privilege and joy to have worked with my advisor, Tom Ristenpart, these past five years. When I was applying to PhD programs and asking about potential advisors, I heard nothing but praise for Tom, not only for the quality of his academic work, but also for being a good human. This praise is well-deserved. I thank Tom for his guidance, encouragement, and mentorship, which made this thesis possible. I would also like to thank Ari and Mor for guiding this work as part of my committee. Over the course of my PhD, I have been fortunate to work with so many great people across various projects and internships. Thank you to all my collaborators, from whom I have learned so much. I thank my friends for providing me support, company, and joy over these past five years. Community is incredibly important to me, and so it feels fitting that a big portion of my work examines how we can better support communities. Throughout my PhD I have been part of many wonderful communities for which I am thankful: the dear friends I have lived with, my fellow PhD students, and RRG, to name a few. Finally, I thank my parents, to whom I dedicate this thesis. Thank you for supporting and believing in me.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With the rise of the Internet came many new digital platforms allowing users to generate and exchange content. Early examples include email and instant messaging. Traditionally, these platforms operate directly on user data. A platform server would act as an intermediary between users, storing, processing, and transmitting their data as needed. When Alice sends a message to Bob, her client device (phone, tablet, laptop, etc.) contacts a platform server, conveying the message along with the intended recipient. When Bob comes online and establishes a connection to the server, he receives the message sent by Alice. Although this arrangement is simple and straightforward, it makes the server a valuable target for hackers attempting to obtain access to user data, which can often contain sensitive information. Although client-server encryption via TLS, which protects data in transit, somewhat addresses this issue, user data at rest is still visible to the server. In addition to external bad actors, malicious insiders can directly obtain user data. Users may also feel uncomfortable with this setup, given the possibility of the platform surveilling their communications or surreptitiously using their data in ways that can violate their privacy.

End-to-end encryption (E2EE) empowers users by giving them exclusive access to the keys used to encrypt their data. Platform servers in the E2EE setting simply ferry ciphertexts decryptable only by the endpoints of the communication. In addition to strengthening user privacy, E2EE provides defense in depth: even compromised or malicious platform servers cannot breach user communications. Popular messaging platforms such as WhatsApp, iMessage, and Signal, support end-to-end encryption for billions of users. Meanwhile, there is a rapid

growth of E2EE platforms beyond the realm of messaging: password managers, file-sharing, document editing, as well as other forms of social media [19].

The emergence of digital platforms brought not only security and privacy concerns, but also new worries surrounding user safety. In response to these worries, platforms had to rapidly develop content moderation policies [75], which provided systematic ways for them to handle abuse of their services. This work was often led by Trust and Safety (T&S) teams. The traditional centralized model of platform access to user data facilitated the work of moderation, providing the platform with a direct view into potentially harmful content and enabling it to operate proactively. End-to-end encryption presents a barrier to T&S in this regard, since the platform cannot directly access user content.

A line of work has emerged examining content moderation approaches in encrypted messaging. The Facebook secret conversations whitepaper [1] was among the earliest such proposals: it introduced the notion of message franking, which enables reporting of E2EE messages to the platform. Following work has expanded message franking to metadata-private and third-party moderation settings [113], tracing forwarded content [115], and notifying users when they receive known harmful content [77]. More controversially, some proposals have advocated for privacy-preserving scanning of illegal content. One such proposal was put forth by Apple in 2021 [34]. Privacy experts denounced such approaches as backdoors vulnerable to government and law-enforcement overreach [23]. Designing content moderation mechanisms for E2EE platforms that do not compromise on user privacy or agency poses a unique challenge.

My dissertation addresses this challenge by developing novel protocols for community moderation and improved reporting in E2EE platforms. In partic-

ular, I propose a novel approach for private hierarchical governance [96] in encrypted messaging and expand message franking to sequences of messages and shared encrypted state.

Chapter 2 details private hierarchical governance, which provides mechanisms for community moderation within encrypted messaging groups while maintaining channels for reporting to the platform. In doing so, we achieve user agency while allowing the platform to set and enforce its own policies. We emphasize that community moderation is highly aligned with the encrypted messaging setting as community members have direct access to content exchanged within groups while platform moderators do not.

Chapter 3 proposes transcript franking, which allows reporting sequences of messages sent through an E2EE channel with causality guarantees. This builds on top of prior approaches that handle single-message franking and require interaction from both parties in order to disclose full transcripts of messages. We provide novel security definitions and constructions, along with an extension to the group messaging setting.

Chapter 4 discusses shared encrypted state for encrypted platforms, in particular, how this state can be maintained in order to allow reporting to platform moderators. Such state arises in private hierarchical governance and in settings such as file-sharing and broader E2EE social media. We provide security definitions and a construction for shared encrypted state franking, along with benchmarks to demonstrate practicality.

Chapter 5 provides a discussion of the societal implications of this work and Chapter 6 concludes.

CHAPTER 2

PRIVATE HIERARCHICAL GOVERNANCE FOR ENCRYPTED MESSAGING

2.1 Introduction

Today, end-to-end encryption (E2EE) helps protect the private communications of billions of people [124] from data breaches, overzealous advertisers, and snooping governments. At the same time, surveys [110, 119] show that people are being harmed by online abuse: almost half of respondents report experiencing abuse online, the fraction of those reporting experiencing abuse is growing each year, and much of this abuse is carried out over messaging apps.

As a result, many large E2EE messaging platforms have dedicated trust and safety teams that develop and execute moderation policies to mitigate abuse. But E2EE makes moderation hard with existing techniques [105]. Content-oblivious approaches [100] do not handle most types of abuse, and those that allow reporting content [1, 50, 63, 67, 113, 126] rely on a centralized platform-operated moderation service that, in turn, relies on a combination of automated tools and large groups of human moderators [47, 60, 61, 75]. Such centralized moderation infrastructure represents a dangerous accumulation of power [109, 123] and struggles with nuanced, contextualized, or community-specific abuse [53].

Some plaintext platforms, like Reddit [8] and Discord [5, 70] instead employ a hierarchical governance structure. Certain community members, known as *community moderators*, define and enforce community policies, while modera-

tors at the platform level oversee communities and enforce platform policies. This leads to a separation of powers in which most moderation of user activity happens at the community level. Community moderators also benefit from tools that automate parts of the moderation process [68]. For instance, Subreddit moderators can automatically flag posts based on keywords with the AutoModerator tool, lessening the burden of moderation. Decentralizing and distributing power in this way can better respect user agency and enables a diversity of community approaches to governance [69], and much work has explored the design of tools to support or enable community governance [58,68,74,127] for plaintext systems.

In this paper, we explore for the first time *private hierarchical governance* for encrypted group messaging. This entails a platform design that provides rich community-level governance features, while achieving strong E2EE privacy, integrity, and accountability guarantees for both content and governance-related tasks. For example, even a malicious platform should not be able to infer who are a community's moderators, nor undetectably interfere with governance actions moderators have taken. As in the plaintext setting, we aim to provide community moderators with tools that help automate moderation tasks, like Reddit's AutoModerator. We give users the choice of reporting content to community moderators and platform moderators. Given the private nature of the E2EE setting, the platform will mainly rely on user reports to inform its moderation.

Realizing private hierarchical governance in the E2EE setting requires overcoming a number of challenges, chief among them that the platform is assumed to be malicious and must not learn anything about content exchanged within

groups. This complicates enabling automated moderation policies, similar to those available to community moderators on Reddit and Discord, in which the platform handles policy execution. One could enable such policies by adding a centralized governance service as a trusted endpoint, however this is undesirable from a usability and security standpoint. Users should not be burdened with setting up the infrastructure for such an endpoint nor should they have to trust a third-party platform for managing one.

Our approach instead shifts community governance to be client-side logic. But in so doing, we must determine how to manage governance actions in a distributed, asynchronous network setting where the messaging platform is potentially untrustworthy. One straw approach would be to just use standard techniques for consensus [97,99] or state-machine replication [106] to all community content and actions, but this would not be practical. A key enabling insight is that our desired governance functions can be achieved while only requiring clients to consistently agree on a small portion of community state related to governance tasks, allowing more expensive consensus mechanisms to only be needed rarely.

Given this insight, we detail a modular approach to private, hierarchical governance. We suggest extending E2EE group messaging systems to support ordered application messages (OAMs) for which the group achieves consensus on the content and order of these messages. Existing E2EE protocols such as message layer security (MLS) [28, 32] already have suitable consensus mechanisms to support our OAM extension; our suggested extension to MLS can be viewed as a generalization of a recent mechanism proposed in [27]. We expect our extension to be of broad utility; here we show how we can build a gover-

nance layer distributed across clients in a group that agree on state via OAMs.

Our governance layer provides a full role-based access control (RBAC) [57] mechanism and a policy engine, inspired by prior work [127], that allows execution of expressive policies securely using OAMs. To demonstrate the generality of our governance framework, we use it to implement a policy for voting on changing the group name. We analyze how our approach provides strong cryptographic guarantees of governance privacy, integrity, and accountability that prevents adversarial platforms from monitoring or interfering with community messages or governance actions, and prevents adversarial clients from preventing abuse reports or reporting messages they did not receive. We also support reporting abuse to platform moderators to enable hierarchical governance; unreported messages stay private from them. Note that our focus is on providing the technical infrastructure for realizing a broad range of automated governance policies, and leave to future work how to guide the design of good policies and prevent abusive policies.

In order to provide a concrete instantiation of private hierarchical governance, we build a full prototype E2EE messaging platform called MlsGov on top of a suitably modified MLS implementation, and show via extensive performance analysis that our governance framework is practical. To encourage further work on building governance for encrypted messaging, we release MlsGov as an open-source project[1].

We summarize our contributions below:

- Our paper proposes the goal of private, hierarchical governance, which

[1] `https://github.com/AME2E/MLSGov`

System	E2EE Content	Gov. Confidentiality	Gov. Integrity	RBAC	Generic Policies	Gov. Enforcement
Matrix	Yes	No	No	Multi-role	No	Server
Signal	Yes	Yes	No	Two-role	No	Server
Whatsapp	Yes	No*	No*	Two-role	No	Server*
Discord	No	No	No	Multi-role	Yes	Server
A-GCKA	Yes	No	Yes	Two-role	No	Client
This Work	**Yes**	**Yes**	**Yes**	**Multi-role**	**Yes**	**Client**

Figure 2.1: A comparison between our work and other messaging platforms that support communities. * - based on our assessment (no public documentation).

shares moderation and other tasks across communities and platforms, while retaining the security benefits of E2EE.

- We present a design that allows for governance policies while limiting the amount of consensus needed across clients. An extension to MLS that we propose allows for shared governance state. We analyze the security of our design by reasoning through possible attacks.
- To demonstrate practicality, we build a prototype E2EE messaging platform called MlsGov that is the first to support a wide range of governance features previously only available in plaintext settings, and measure its performance.

2.2 Background and Related Work

Community governance. Social media platforms have developed a wide range of governance strategies to counter abusive content, including both industrial moderation approaches as well as community-reliant approaches [37,59]. In industrial approaches, governance is primarily centralized at the platform level,

with policies set by trust and safety teams and carried out by commercial content moderators [75, 102], whereas in community-reliant platforms, communities have broader latitude to define their own governance. In these cases, each community has volunteer moderators who can designate and enforce community-specific rules [92,107]. However, the platform still retains the power to step in and enforce its rules, including banning accounts and removing or quarantining entire communities [38, 39], forming a two-level hierarchical governance structure [69].

Some community-reliant platforms provide a powerful set of tools to communities to automatically carry out policies [107]. For instance, subreddit moderators can use the Automoderator tool to automatically filter and act on posts based on their content [68]. Discord has a strong ecosystem of third-party tools that community moderators can use to customize their community [74]. More recently, academic work has explored richer governance approaches, where communities can define and execute arbitrarily complex governance procedures written in code [127], enabling communities to vote on new moderators, enforce content filters, or institute reputation systems.

Abuse mitigations in E2EE settings. In E2EE messaging platforms, governance is complicated by the fact that moderators cannot by default verify the plaintext contents of communications. A body of work has explored how to enable cryptographically secure reporting of individual encrypted messages, starting with Facebook's message franking feature [1]. Subsequent academic work formalized message franking and characterized (in)secure methods for achieving it [50, 63, 67, 113]. In addition to reporting abusive messages, moderators may benefit from the ability to trace the spread and identify the origin of forwarded

messages. Recent work has explored how to realize this feature for E2EE platforms [67, 98, 115].

Another abuse mitigation approach is automated content detection. Given the scale at which content is exchanged on platforms, many plaintext platforms have turned to automated content detection that alerts platforms when particular content is sent or received [82]. Perceptual hashing [56] is a popular tool used in these approaches, particularly for the detection of child sexual abuse media (CSAM). These techniques are not immediately applicable to E2EE platforms as platform servers cannot observe the plaintext contents of messages and client devices might not be allowed to see the plaintext contents of the harmful content list (especially in the case of CSAM). Recent work has explored how multiparty computation can be used to navigate these challenges while respecting user privacy [34, 77]. Such proposals have been met with criticism owing to their potential to undermine the privacy goals of E2EE [23] because they automate reporting to platform moderators without user consent.

Existing E2EE governance tools. Given the difficulty of conducting effective moderation at the platform level without violating privacy, some attention has been paid toward strengthening governance tooling at the community level. Matrix [3], a protocol for federated E2EE messaging, provides some basic community moderation features via role-based access control (RBAC) [6]. However, information about user roles is directly visible to the homeserver (platform server). Mjolnir [4] provides server-level moderation features such as banning users, taking down content, and managing user accounts. However, it does not enable arbitrary programmable governance and cannot process content in encrypted rooms.

WhatsApp's recent launch of Communities makes it easier for groups of over a thousand people to use the platform [125]. Whatsapp groups allow for two-level access control (between users and moderators). Although no public documentation describes how or where Whatsapp enforces governance, it bears mentioning that group metadata such as the profile picture and topic associated with a group are visible to Whatsapp. As such, we suspect that governance metadata such as the roles and permissions within a group are visible to the platform and that the platform enforces these roles.

Signal has explored private group management for E2EE chats, using a combination of anonymous credentials and a server-managed encrypted list of members [42]. Their design allows the server to authenticate community moderators in the sealed sender setting. This achieves governance confidentiality, but the reliance on platform-enforced access control weakens governance integrity. There is also some leakage of which encrypted entries perform actions on the governance state. Furthermore, their design does not handle other aspects of governance, such as allowing for rich and programmable moderation policies, such as those enabled via Discord moderation bots, the Reddit Automoderator tool, and PolicyKit [127].

The recently proposed A-CGKA construction [27] suggests embellishments to the message layer security (MLS) protocol involving cryptographically enforced roles, such as administrators, for group encrypted messaging. Their solution, however, does not describe a mechanism for keeping administrator roles confidential. They achieve governance integrity, but do not tackle governance confidentiality as a goal. While they do not provide the rich governance features our solution provides, our modifications to MLS are similar to theirs; we

compare and contrast in more detail in Section 2.4.

These solutions all stop short of the rich governance features that many communities have available to them on non-E2EE community-reliant platforms. In addition, there has been little work examining how community-level governance intersects with governance happening at the platform level, such as banning accounts and communities from the platform. We summarize the governance capabilities of popular messaging platforms in Figure 2.1. We define governance privacy and integrity in detail Section 2.3. Role based access control (RBAC) allows users to define permissions and roles within their community. The "Generic Policies" column refers to whether the system supports programmable governance logic. We also indicate whether governance is enforced on the platform server or on client devices. Asterisks indicate aspects that are not publicly documented and the values we provide in those entries are our conjectures based on the available information we have. In summary, no prior work tackles the design of rich governance tools in E2EE that takes into account both community and platform levels. Our work fills this gap.

2.3 Overview and Goals

Our paper presents a framework for building and testing governance for E2EE online communities. In this section we provide an overview of our goals and approach, and detail various components further in subsequent sections.

Private, hierarchical governance. We seek to enable governance frameworks for E2EE social media, in particular private direct and group messaging like that offered by Signal and WhatsApp. But unlike today's messaging solutions,

our system should allow groups of users to form self-governed communities with the support of a rich suite of governance features. Going forward, we use the term "group" and "community" interchangeably. A group should be able to elect one or more moderators who retain special privileges within the group to perform various actions, including removing users, adding users, and blocking messages. These moderators can change over time. Group members should be able to send abuse reports to community moderators to help inform moderation decisions. We further aim to provide group members with features that help automate parts of governance, such as handling votes for new moderators or enforcing a moderator-specified word filter.

Consider the following scenario, which we will use as a running example to illustrate the types of governance issues that our framework can handle. A community consisting of N users $U_1, \ldots, U_N$ has a single moderator U_1. Group member U_2 proposes to change the community guidelines so that profanity is not allowed within the group. The other users vote to pass this change. Later, U_3 sends an abusive direct message (DM) to U_2 for proposing this change. After receiving the message, U_2 reports the message to the moderator U_1, who decides to remove U_3 from the community as a result. As U_1 knows sending abusive messages is against platform guidelines, U_1 forwards the report to the platform moderation endpoint M. A platform moderator receives the report and decides to ban U_3 for one week for violating platform guidelines. As a result U_3 is temporarily unable to participate in any community hosted on the platform. A diagram summarizing this scenario appears in Figure 2.2.

Looking ahead to our threat model, we want governance actions like those in the scenario above to be *private* by default. That means that governance ac-

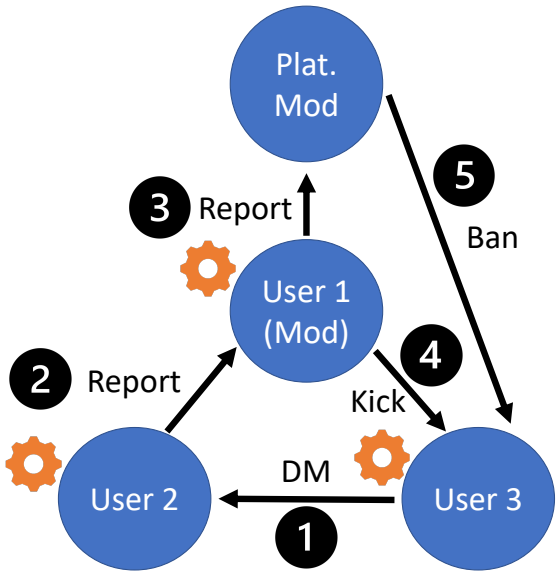

Figure 2.2: Diagram of an example governance scenario in which a user U_2 reports another user U_3 to a community moderator U_1, who kicks that user from the community and escalates by reporting the abuse to the platform's moderation systems. This results in a temporary ban of U_3 from the platform. Execution of community governance is handled on clients (denoted by the gears) and remains private to the community. Only when a user chooses to report to the platform does the latter become involved.

tions should be cryptographically secure and opaque to the messaging platform provider (from now on, simply the platform), so that, for example, the platform should not need to know that U_1 is the moderator, that U_2 proposed a new policy, that it was adopted and who voted for it, or that U_3 violated it. Designing privacy as a default for governance is conservative: not all users and communities require such privacy, but it can be critical for those that do—e.g., political activists, journalists, or community organizers from marginalized groups, any of whom may be targeted by nation-state or other powerful adversaries. In particular, governance metadata can be sensitive information; for instance, being a moderator of an activist group may also indicate being a leader within that group.

But not all governance actions can be handled within a single community. For example, U_1 decided in the scenario that the in-community problems caused by U_3 warrants further action, possibly to help protect other communities from U_3 misbehavior. Thus, we want to balance privacy with accountability and support what we call *hierarchical governance*. In addition to group-level governance, users can choose to report in-group misbehavior, as done by U_1 in the example. Even abusive behaviors of community moderators should be reportable. Because the platform ultimately retains control over who is allowed to use it, governance forms a hierarchy where platform moderators can overrule community moderators.

Such hierarchical governance has already been explored in plaintext systems such as subreddits and Facebook Groups [69], but adding in privacy surfaces new challenges, since by default platform moderators will not have access to group messages or community moderator identities. A key challenge addressed by our work is providing a framework to explore the design space of private, hierarchical governance policies and supporting mechanisms.

Platform architecture and layered design. To achieve E2EE communities with private, hierarchical governance, we will use an architecture consisting of three services: a *delivery service* (DS), an *authentication service* (AS), and a *platform moderation service* (MS).

The DS and AS form what we call the *messaging layer*. The DS provides API endpoints to route messages and service metadata between clients. The AS provides bindings between identities and public keys. These are similar to existing messaging service solutions and serve to ensure confidential and authenticated delivery of messages between users intermediated by a platform.

In particular, the DS and AS are analogous to services of the same name in the ongoing MLS [32] standard. However, governance needs cryptographic support from the messaging layer, in particular the ability of a group of clients to asynchronously agree on an ordered sequence of governance messages (in the presence of potentially adversarial clients). We show how a relatively straightforward modification to MLS provides the features necessary to build our governance layer in a modular way (see Section 2.4).

Logically sitting above the messaging layer we have a *governance layer*. This layer consists of logic within the clients to enact governance actions taken within the community, such as group members electing a new moderator or a group moderator kicking a member. One approach here would be to hard-code particular governance mechanisms, but we instead adopt the viewpoint underlying PolicyKit [127] that governance should be easily customizable. PolicyKit allows on-the-fly customization of the software used by individual communities on a platform. We build a similarly expressive governance layer for the E2EE setting. To address our security goals (detailed below), the policy engine execution is handled by clients that must agree on the current state of the group. Managing this state in the presence of adversarial clients introduces complexity compared to insecure plaintext approaches, but our layered design approach means that the complexity can be largely ignored by those developing applications on top of our governance layer. We describe the governance layer design in Section 2.5.

To provide hierarchical governance, we include the MS to receive abuse reports from community members. For simplicity, we realize the MS via a distinguished, virtual user (operated by the platform or someone to which it delegates) to which report messages can be sent—this ensures a level of simplicity

and flexibility that will be beneficial in deployment. We emphasize that the MS only becomes involved when a user affirmatively chooses to submit a report to the platform moderator: in our example above the MS only observes the community moderator's report about U_3.

Lastly, all of this enables the *application layer* in which applications that support community governance may be built. As a showcase, we build a group messaging application using MlsGov that supports expressive governance policies.

Threat model and security goals. Our system is designed to achieve our governance and privacy goals even in the presence of malicious parties. We use the term malicious to refer to parties that deviate from the protocol, and use the term abusive to refer to clients that send legitimate protocol messages but with an intent to cause harm. We consider both malicious users within the community and a malicious platform-operated DS and MS. Our design assumes that the AS is honest, which can be enforced using separate mechanisms such as public key infrastructure (PKI) transparency [41, 94, 95, 112]. In accordance with the MLS Architecture Document [32] §2.3.3, we expect the DS to aid with message ordering when needed and to allow for eventual delivery of client messages sent over successful network connections. Nonetheless, when the DS violates these expectations, we will have mechanisms for the eventual detection of such misbehavior.

Integrity: Recall that we introduce a governance layer that includes client-side state, called the governance state. State will include information like the permissions of group members such as who is a moderator, what policies are being enforced, etc. As such governance state evolves over time and the current

governance state determines how actions performed by users produces future versions of the governance state. A system achieves *governance integrity* if all honest, online clients in a group agree on the same sequence of governance states. This requires that clients observe a consistent sequence of governance state updates and that they apply these updates according to their governance functionality.

We target achieving governance integrity even in the presence of one or more malicious clients in the group that collude with a malicious DS and MS. A malicious DS can drop and reorder encrypted messages, including governance-related messages. This means that denial of service by a malicious DS is always possible. A malicious DS can partition the group into subgroups, by not delivering messages across their maliciously chosen partition. This can fork the governance state across the two (or more) subgroups, but such an attack should be detected as soon as the first cross-partition message is delivered. We note that this is the same security level achieved by MLS (c.f., [32]), and absent further infrastructure there is no way to avoid partitioning.

Accountability: We target *user and reporter accountability*. This means that abusive users, even ones that are malicious (i.e., using compromised, adversarial client software) should be reportable to honest community moderators and to the MS. Reporting an abusive user to an abusive community moderator may not lead to desired outcomes. So user accountability extends to moderators, meaning any user should be able to report abusive moderators to the MS. On the flipside, reporter accountability means that community moderators and the MS must reject malicious reports, such as reporting messages that were never sent. In the cryptographic literature [113], user and reporter accountability are

sometimes referred to as sender and receiver binding, respectively.

An abusive group is one in which all members are abusive. For example, a group that is trading child-sexual abuse material (CSAM) or other abusive content. In this work, we have as a non-goal automated detection of abusive communities, such as considered in recent proposals by Apple [34] and others [77]. While automated detection might be necessary to detect abusive groups, privacy advocates raised serious concerns about their deployment (c.f., [23]). Our governance mechanisms will all be user initiated, which makes them less useful for mitigating abusive groups but still critical in mitigating all other types of abuse. We believe our results would be compatible with future automated mechanisms, but leave detailed exploration to future work.

Confidentiality: In addition to traditional message confidentiality, we target *governance confidentiality*. This means that even in the face of a malicious DS and MS, a group's governance remains private to the group and the amount of information leaked about the governance state is minimized. So for example, the DS and MS should not be able to infer a group's current governance state, including who are the moderators, what policies are being enforced, and whether a user has been reported to a community moderator.

Complete governance confidentiality should last until the first report is made to the MS. At this point, the report may reveal some information about governance state within the group, e.g., that an abusive moderator exists. But this revelation should be minimal, meaning that governance state and actions not involved in a particular report remain undisclosed to the platform (e.g., who else is a moderator remains hidden). If a community moderator adds someone to the group, that will be revealed to the platform (a limitation of the underlying

MLS protocol we utilize), but removing members from the group will not be.

Similarly, and like in other prior works on content-based moderation in encrypted messaging [1,63,113], we will also ensure confidentiality of unreported messages. In our hierarchical reporting structure, this surfaces as confidentiality from community moderators and the MS in the case of direct messages (DMs) and from the MS in the case of group messages (GMs).

Further non-goals: In addition to our non-goal of automated detection of fully abusive groups and denial of service, we also do not focus on resisting traffic analysis attacks. It could be that the pattern of encrypted messages sent reveals, for example, information about governance actions, such as who is a moderator. Whether such attacks work and, if so, what obfuscation strategies can be used is an interesting question for future work.

We also do not target prevention of bad policies: their design and the criteria that define what makes a robust policy are beyond the scope of this work. Poorly or maliciously designed polices could have loopholes that enable abuse or circumvention, much in the same way a badly specified multi-party computation functionality can directly reveal secret inputs. Other work addresses questions of policy design and specification [121]. As new insights arise in the literature on policy design, our system will enable rapid prototyping of these ideas in an encrypted messaging setting.

We also do not specifically target private governance while achieving metadata privacy, i.e., sender or receiver anonymity [46, 49, 85, 86, 118]. That said, our layered approach means that if one can build a metadata private encrypted messaging system that provides an ordered message primitive, then our tech-

niques would also work in that context. That said, we are not aware of any metadata-private systems that can easily provide ordered messaging primitives for groups, and so this remains an open problem.

Finally, we do not target deniability [36] and, relatedly, coercion resistance [71]. MLS does not provide deniability. But just like metadata privacy our system should be adaptable to a deniable encryption layer (e.g., [36,90,101,116]), by replacing our use of non-deniable digital signatures with deniable message franking tools [113]. Policies that include voting are subject to coercion, and again asymmetric message franking would help make the system more coercion resistant since they would not be as useful as proof to a coercer of how someone voted.

2.4 The Messaging Layer

In this section, we describe how to construct an E2EE messaging layer that supports our governance layer. Our encrypted messaging layer will also be useful more broadly for privacy applications that require synchronized, distributed state. We start from the messaging layer security (MLS) draft standard [28, 32] which provides encrypted group messaging with strong security properties and scales to large group sizes. Then we describe a crucial—but in hindsight straightforward—modification that endows MLS clients with the ability to maintain synchronized application state. We will use this capability for governance, but note that our extension to MLS is likely of broader interest.

MLS Overview. MLS relies on an authentication service (AS) for maintaining user credentials and on a delivery service (DS) for transmitting messages. The

MLS protocol [28] supports efficient encrypted group messaging with strong guarantees including confidentiality, authenticity, integrity, forward secrecy, and post-compromise security. We provide more background on MLS in Appendix 2.9. Content messages in MLS are not expected to be broadcast in a consistent order, hence we refer to them as unordered application messages (UAMs). On the other hand, messages that update cryptographic state must be consistently ordered.

Consensus in MLS. MLS uses a simple consensus mechanism for shared cryptographic state: updates to this state are conveyed through *proposals* and *commits*. Proposals convey an intention to carry out an action, such as adding someone to the group. The MLS protocol currently supports eight proposal types, most of which have to do with group membership. Add, Remove, and Update are examples of MLS proposal types that allow adding a group member to the bhargavanTreekem18, removing one from it, or allowing a user to update their public KEM key (e.g., for forward secrecy). A proposal message contains information relevant to carrying out its associated operation. For instance, an Add proposal contains the cryptographic information of a user to be added to the group. By default, proposal messages are private messages.

A commit message contains one or more proposals, and when a client processes a commit, they carry out the actions specified in those proposals, advancing to a new *epoch*, which is the fundamental and atomic unit of shared group state in MLS. The history of group state evolution in MLS proceeds in epochs with commit messages referencing the epochs directly before them. Commit messages are likewise by default private messages. In our implementation, both proposal and commit messages are private.

MLS requires that all group members agree on a total ordering over all commit messages, in order to ensure a linear progression of epochs. Relatedly, MLS clients must have an established way for dealing with the scenario in which two or more clients attempt to submit conflicting commits building off of the same prior epoch. The MLS protocol does not specify a concrete way for clients to agree on ordering or handle conflicting commits; these are left as implementation choices for MLS clients and servers. For our design, we assume a strongly consistent DS that provides a consistent message ordering to clients, which handle conflicts by merging commits that arrive first according to the server-provided ordering.

Ordered application messages. A key insight underlying our approach to governance is developing a method for maintaining a consistent, shared view of governance state within MLS groups. Looking ahead, governance state will include information on user roles, group topic, information regarding ongoing governance processes such as votes. This requires that we extend MLS to allow transmitting arbitrary application-defined data in a way that is atomically updatable; this is already solved within MLS to support shared cryptographic material. We therefore can use the same consensus mechanism (proposals and commits) to maintain arbitrary shared group state.

We extend MLS to include a new proposal type called an *ordered application message (OAM) proposal*. It contains an arbitrary sequence of bytes, similarly to application messages. The difference is that by introducing this as a proposal type, we force the sequence of bytes to be committed to: clients must agree on this string and when it was committed to. Committing an OAM proposal triggers an epoch change. In addition to ordered application messages, we devise

a mechanism for bootstrapping existing governance state for new group members and eventual detection of incorrectly supplied initial state. We call this mechanism group state announcement and describe it in detail in Section 2.5.

Our design includes a DS that provides a total ordering on commit messages. When a client sends a commit message to the DS, the DS includes in its response a list of messages intended for the client that arrived before its commit message, according to the server's ordering. If this list contains no other commit messages that attempt to build on the same previous epoch as this client's commit message, the client proceeds to merge their commit, applying the proposals contained within it and progressing to a new epoch.

As already mentioned in the MLS specification [28, §14], clients can end up starved of the ability to submit a commit. This means that not all application features are suitable for implementing via OAM proposals, e.g., switching regular text messaging to have ordering would have significant negative impact on performance. However, we do not have the same requirement of consistency of text message ordering as we do for governance state updates. Since we expect governance state updates to be sent with less frequency than user text messages, our usage of OAMs achieves consistent state evolution with performance. The only case in which we would expect a high volume of ordered messages in a short period of time is with voting, and we describe an optimization in Section 2.5 that is able to handle this scenario effectively. We empirically evaluate the performance of our approach in Section 2.7.

The messaging layer API. To make precise the interface between our extended version of MLS and our governance layer, we define a messaging layer API. This API maps between higher-layer requests (in our context, the governance

Function	Description	MLS messages
send_uam	Send bytes to group, without ordering guarantee	Application msg
send_oam	Send bytes to group, with consistent ordering guarantee	OAM P+C
add_user	Add user to group	Add P+C, Welcome msg
remove_user	Remove user from group	Remove P+C
update_user	Update user's public keys	Update P+C
get_epoch	Retrieve current epoch number for a group	N/A

Figure 2.3: Functions exposed by our message layer API, including inputs and the MLS messages invoked to handle the API request in our implementation. Here P+C stands for the indicated proposal type followed immediately by a commit.

layer) to underlying MLS message types. We follow as closely as possible the OpenMLS API [7]. OpenMLS is a mature implementation of the MLS protocol. More pragmatically, we will build off OpenMLS in our implementations. A summary of our messaging layer API is shown in Figure 2.3. Here we focus on the subset of the API related to messages and group maintenance. Each API function call maps to one or more underlying MLS protocol messages types.

2.5 The Governance Layer

The messaging layer described in the last section provides cryptographic APIs upon which we can build a *governance layer*. This layer sits between applications and messaging, interposing on application layer requests to apply policies. A pseudocode specification of our core governance layer logic can be found in Appendix 2.13.

Recall from Section 2.3 our example of a governance flow: (1) a user U_2

Action	Description	Content	Governance	MLS Messages
Text message	append message to message history	✓		UAM
Invite	update cryptographic state and share with invitee		✓	OAM, Add, UAM
Kick	update cryptographic state and share with kickee		✓	OAM, Remove
Rename Group	modify group name in shared group state		✓	OAM
Define Role	define a new role as a set of allowed actions		✓	OAM
Assign Role	assign a role to a user		✓	OAM
Content takedown	remove specified content	✓		UAM
Report	send a report of received message	✓		UAM
Vote	send a vote on a proposed action		✓	OAM or UAM
Accept	acknowledge acceptance of invitation to group	✓		UAM

Figure 2.4: A subset of our supported actions, whether they affect governance state or content state, and the MLS messages they produce. UAM refers to an unordered application message, OAM refers to an ordered application message, and Add/Remove refers to a commit with the Add/Remove proposal.

proposes a vote to change community guidelines; (2) U_3 harasses U_2 after the guidelines change; (3) U_2 sends an abuse report to community moderator U_1; and finally (4) U_1 escalates by reporting the abuse to the platform moderator. In this section we detail the abstractions that our governance layer provides to realize such governance actions, and how they are implemented on top of our messaging layer.

Community structure. Our governance layer organizes users into groups, which are also tagged with community identifiers. A group consists of two or more users involved in a shared messaging channel. We do not make a distinction between DMs and GMs, rather these are both implemented as groups (of

size two or more, respectively). We assume distinct namespaces for usernames, group identifiers, and community identifiers. These are assumed public, but users can choose them to be semantically meaningless (like random numbers) — we support having internal groups names that remain private to the current members and can be chosen and modified by members. A user can initiate a group by invoking a group creation API call at the messaging layer. Then they can then add users via the messaging layer. The initiator of a group and group membership is platform-visible.

We associate to all users a public key and secret key for use by the governance layer, we refer to these as governance keys. Clients pick their governance keys, and the public key is registered with the AS like other public keys used in the messaging layer. As we discuss below, adding a separate pair of keys allows for clean separation of the layers, while incurring little performance impact, as we show.

Our governance layer supports a rich role-based access control (RBAC) [57] mechanism. The RBAC gates who can perform what kinds of governance actions (a notion we will define soon), what we refer to as a permission. A set of permissions defines a role. This will support, in our running example, having a community moderator U_1 that has the permissions to remove U_3, while other users do not have this ability. Our RBAC mechanism is more expressive than recent suggestions for cryptographic group administration [27].

Governance and content state. To support governance mechanisms, we need some consistent state replicated across clients. We refer to this state as the *governance state*. The governance state is a key-value store that includes information such as user roles and privileges within the group, current policies like prohib-

ited word lists, and the group name.

We delineate between governance state and all other group-related state, such as sent plaintext text or image content that's been sent to the group or via DMs. We refer to this non-governance state as *content state*. A key enabling architectural decision is that we can separate between aspects of group state for which governance only works should clients agree on that state, versus other state for which it is allowable for clients' views to diverge. This is important for performance and deployability: we show that useful governance policies can be implemented even when many aspects of shared group state are potentially inconsistent. In our running example, the group guidelines are part of the governance state because the group must agree on the current policies, but individual messages and reports end up as part of the content state. This means the group does not "agree" on the fact that a report occurred, but the moderator U_1 can verify that U_3's harassment occurred while the group agreed on a no-swearwords policy.

When a new user joins a group, the inviter includes in the welcome message a serialization of the current governance state. We assume some efficient canonical way of encoding the governance state. We use JSON in particular.

Actions. To guide and reason about how community state changes over time, our governance layer defines a set of possible actions. An *action* is a message broadcast within a group that can produce changes to the shared group state or content state. Examples of actions include sending a text message and changing the group name. We classify actions, depending on whether they affect the governance state, into two categories: *governance actions* which can change the governance state and the content state, and *content actions* which can only

change the content state. For example, sending a text message to a group only changes the content state, so it is a content action, while changing the group name changes the governance state, so it is a governance action. We provide more examples of actions and which types of state they may modify in Figure 2.4. All application-layer requests result in one or more actions.

To communicate an action, the governance layer prepares an action message. A header is constructed that includes the sender username, an action ID (a unique identifier for the action), the group ID, and the community ID. An unambiguous encoding of the action-related data follows, such as the plaintext data for sending content or the new group name. Finally, all this is serialized and signed using the sender's governance digital signing key. This signature is checked once a message is received, before it is evaluated by the RBAC or policy engine. While in theory we could utilize the messaging layer's digital signatures to provide accountability for action messages, using governance keys allows for clean separation between abstraction layers — otherwise we would have to modify the messaging API to expose low-level details of how MLS messages are framed. It also means we can swap out our MLS messaging layer with any API-compatible variant.

As an example of an action, consider when U_2 wants to report to U_1 the abusive DMs received from U_3. To do so, U_2 builds a report action whose payload consists of one or more action messages. In this case it would be U_3's DMs, which were, themselves, action messages that are signed. The report therefore includes a full serialization of the reported action messages, including their signatures, making it possible for U_1 to verify them.

Updating state via OAMs. Content actions result in a call to send_uam in the

messaging layer, while governance actions use ordered messages via send_oam. Governance actions need to use ordered messages to ensure that governance state is modified consistently across all clients. Since there are no ordering guarantees on UAMs, using them to encode governance actions can potentially fork governance state, leading to security and correctness issues. An ordered application message, when processed by a client, potentially changes the shared group state. As all clients begin from a common state and process OAMs in the same order, they retain a consistent group state.

Initializing governance state. While ordered messages allow current group members to perform consistent governance state updates, we require a separate mechanism to provide newly invited members with their initial governance state. In our design, we have a designated action for group state announcement, which contains an encoding of the group state at the epoch during which the new member joins. The group state announcement is sent by the inviter as an UAM. The new member assigns their current group state to the one in the encoding and includes a hash of this state in their `Accept` message, which is broadcast to the entire group. If a member detects this hash to be inconsistent, they can alert the new member and group (and/or platform) moderators. We analyze the security of this approach in Section 2.6.

There are alternate possible approaches in which a group state announcement could be sent via an ordered message that immediately follows the `Add` message. However, doing so results in a more complex state machine, is worse in terms of performance, and can lead to DOS attacks. In contrast, our approach has existing group members check the `Accept` message of the new member for

consistency with their current view of the governance state.

Policies. We support developer-defined policies. A policy is an arbitrary process defined using code which determines whether an action should be executed, in the context of a particular group. Here we adopt the viewpoint of, and some of the concepts underlying, PolicyKit [127], which built powerful governance mechanisms for settings where all traffic can be seen by a service. To support privacy we position policy enforcement at the clients, by way of a policy engine that is applied to broadcast actions. Policies must be written such that only actions broadcast in OAMs update shared state. This guarantees governance state consistency.

The policy engine defines an execution model for determining if actions should proceed. Following PolicyKit, we hardcode that our RBAC engine takes precedence, so that when processing an action the engine first checks if the user has a role whose permissions allow the action. If so the action is invoked. For example, a user who has the moderator role will typically have the permission to immediate remove a user from the group. If an action cannot immediately pass according to the RBAC, it is fed into the policy engine, which will determines when, if at all, the action will pass.

As with PolicyKit, our policies are defined by a template consisting of several functions that get called from within the policy engine: (1) `filter` defines the scope of the policy. It takes as input an action message and returns a boolean to indicate whether it is relevant to the policy. This allows policies to choose which kinds of actions they impact. (2) `init` defines how to initialize state for the policy's execution of a specific action, and can potentially modify the client state (3) `check` evaluates an action. It takes an action and returns one

of `passed`, `failed`, or `proposed`. The latter allows for policies that can't yet determine whether they pass or fail, such as when handling a vote action that requires waiting some period of time for votes to arrive. (4) `pass` determines the effect if the action passed. It takes as input the action and temporary state, and it modifies the governance or content state. (5) `fail` determines the response if the action failed. Normally this routine does nothing, but in some cases policies may want to display a message to users or clean up state All functions have access to the governance and content state held by the client.

The policy engine executes policies as follows. The engine is called anytime an action is broadcast to the group and does not pass the RBAC. The engine loops over all policies in a predetermined, developer-defined order. For each policy, the engine calls in turn `filter`, `init`, and then `check`. The first policy for which `filter` returns `true` will be the policy that governs this action. If `check` returns `proposed`, then the engine keeps track of the action as pending. If it returns `passed` then the engine calls `pass`, which executes the action and otherwise calls `fail`. Once an action passes or fails, these routines are expected to clean up state allocated by `init`. The policy engine will periodically attempt to pass proposed actions by re-running `check`, whose output can change based on the policy's current state.

Voting. Although our policy framework is quite general, we are interested in how it can enable collective action among users to effect change within the community. A classic example of this is voting, in which community members indicate their preference for a change, and carry it out if there is sufficient support. Our policy framework allows the expression voting procedures for arbitrary actions, including changing the group name and promoting users to

moderator status. Individual votes can be cast within ordered messages, however, our design admits an optimization for high-volume voting behavior. If many votes were to be cast simultaneously through individual ordered messages, there would be a high degree of contention and many retransmissions of votes. However, votes for a single poll can be aggregated in any order if we consider simple majority or threshold votes. Therefore, we allow clients to send votes in unordered messages. Multiple votes can then be batched into a single commit (for instance, when enough messages are collected to bring a vote to completion), at which point they are fed into the policy engine. As a result, a group can process many votes within a short period of time, as we show in Section 2.7.

Hierarchical governance. All the above facilities support community governance in a way that is, by design, opaque to the platform. But we also want to allow community members to escalate problems to moderators run by the platform, such as in our example when U_1 reports U_3 to the platform for their behavior. We refer to this as hierarchical governance.

Our governance layer therefore includes the ability to interact with a *moderation service (MS)* operated by the platform. This service can have the ability to receive reports, process reports, and limit users at the platform level, e.g., by instructing the AS to revoke a user's keys or temporarily blocking them from sending messages.

While there are multiple ways to build an MS, we do so by setting aside a distinguished username, e.g., `@moderation`, which is authorized for taking platform moderation action and receiving reports on behalf of the platform. This design choice enables us to reuse existing infrastructure and allow for conversa-

tions surrounding reported content. We define a structured report as a plaintext byte string consisting of a serialization of a username, sequence of one or more action messages, and an (optional) reason for the report (an arbitrary byte string in our current implementation). The structured report is sent to `moderation` via send_uam in a group that consists of the user and `moderation` (the group has some distinguished group identifier).

As implied by using a UAM, we do not require consistency: delayed or dropped reports can be resent by the user just like regular messages. The moderation service can verify digital signatures in the included action messages, providing reporter accountability. Unlike in traditional platforms, the technical capabilities of the MS are limited to user-level limiting (since communities are implemented client-side). In particular, the moderation service can block a user at the platform-level (either indefinitely or for a limited time), but cannot block specific messages, groups, or communities based on their content.

2.6 Security Evaluation

In this section, we analyze MlsGov in terms of its ability to achieve our security goals: governance integrity, accountability, and confidentiality.

Prior work has analyzed the MLS protocol [25, 26, 120], establishing that it achieves strong message confidentiality and authenticity. MLS additionally provides post-compromise and forward secrecy, though we will not need this for our subsequent analyses. Note that the modifications we made to the MLS messaging layer (ordered application messages) reuse the existing underlying proposal plus commitment mechanisms, and therefore inherits their security.

To analyze governance extensions with respect to our security definitions (Section 2.3), we enumerate possible attacks. For each attack, we analyzed the extent to which our system prevents, mitigates, or allows the eventual detection of the attack. This is in line with the methodology used in the Tor paper [49] and in the MLS Architecture Specification [32] to establish confidence in the security of complex protocols. Additional discussion of out-of-scope attacks and security goals appears in Appendix 2.10.

Attacks against integrity. We analyzed attacks that seek to undermine a group's governance state. At a high level, the authenticity of MLS and its hash transcript for epochs ensures that honest clients will reject maliciously generated state updates. Even with a colluding, malicious DS, the best an adversary can do in most cases is partition the group forever, a form of denial of service since it means the partitioned honest users can never be allowed to talk to each other again (lest they detect the attack). In more detail, we considered attacks including policy violation, impersonation, governance state partitioning, and invalid initial state, and vote suppression:

Policy violation: Suppose malicious clients collude to attempt to violate the policy of the group by trying to perform an unauthorized action, such as one malicious client performing an action outside that client's RBAC-defined role. Since all honest clients have the same governance state and run the same code to interpret the governance state, they will not accept this unauthorized action. This is true even if the malicious clients collude with a malicious DS.

Impersonation: A malicious client or DS could attempt to impersonate a member of the group and send messages as that other member. But the authenticity of MLS messages plus our assumption that the AS is trustworthy rules out

such impersonating messages being accepted by honest clients.

Governance state partitioning: Suppose a malicious client colludes with a malicious DS in an attempt to produce inconsistent governance state within the group. This means that the goal is to have two distinct subsets A, B of the group have different governance states; A and B must have at least one honest client each. We refer to this as a partitioning attack on the governance state. Since the governance state can only be updated by commit messages that are included in the MLS hash transcript, such a partitioning attack can proceed only as long as no honest client in A receives a message from B (or vice versa), as the first such message will not verify by the recipient. Thus, the adversary can at best split the group and never allow future communication.

Invalid initial state: Any malicious client, regardless of their RBAC permissions, can send an Invite message with an incorrect initial governance state. For example, consider our running example group, we could have that the abusive user U_3 instead behaves maliciously and invites a new member U_4 to the group, but providing an initial governance state that does not include the policy against swearing and has U_3 with the RBAC role to add users. Honest clients will reject this invite message, but the newly invited client does not know that this is invalid. However, when the new member sends an `Accept` message, this message will contain a hash of the received governance state. By collision resistance, that hash will not match the one expected by honest clients. Thus, while U_4 may send additional messages or interact with U_3, no honest user will accept U_4's messages and, moreover, as soon as they come online, they will detect that an attack occurred. Thus, even with collusion by the DS, the malicious client can at best partition the group.

Vote suppression: A malicious DS may attempt to prevent one or more client's votes from counting. Because votes are encrypted, it isn't directly revealed to the DS which are votes (and for what election). Thus, naively the DS would have to just drop all messages emanating from the target clients. But even if the DS can somehow precisely target UAMs and OAMs for dropping, a client can still detect if their votes are being suppressed: the transcript hash consistency mechanism enables clients to obtain a consistent ordering over commits, which contain all the registered client votes.

Attacks against confidentiality. Recall that for confidentiality we want to, by default, ensure the privacy of group content and governance actions from a malicious DS. This covers other confidentiality threats, like network adversaries, who see less than the DS. Here we rely primarily on the confidentiality of MLS messages, and do not consider traffic analysis attacks here (see discussion at the end of this section). We considered the following attacks:

Inferring the content of governance state: A malicious DS may attempt to infer the content of the governance state based on the transcript of messages it relays on behalf of clients. For example, the DS might want to identify moderators or admins. But all updates to governance state are sent through encrypted commit messages. Group state announcements that supply new members with the current governance state are sent via encrypted UAMs. By the confidentiality of the encryption scheme MLS uses, the DS cannot observe the content of the governance state.

Inferring content of unreported messages: All messages and reporting signatures are encrypted via MLS, and MLS' confidentiality guarantees ensure that even learning about one message does not leak any additional informa-

tion about another encrypted plaintext. As a result, even a malicious DS would not be able to infer anything about the contents of unreported messages beyond what is revealed by a reported message.

Inferring content of user votes: An adversarial DS may attempt to learn the value of votes that clients cast for policies that involve voting. Vote messages are encrypted through MLS, and therefore remain confidential as the DS observes only ciphertexts.

Inferring outcome of a vote: The DS could attempt to learn the outcome of a vote. However, votes are encrypted and aggregated on client devices. After aggregation, the change a vote produces, if successful, is applied to the local governance state. As a result, the DS cannot infer the result of a vote.

Attacks against accountability. A malicious user may attempt to circumvent accountability either by sending a message that is accepted by a recipient but unreportable to a moderator or framing an honest sender for having sent an incriminating message. We prevent both types of attacks via the authenticity properties of digital signatures used at the governance layer. Recall that deniability is not a goal of our system since MLS itself does not provide it.

Report evasion: A malicious user may attempt to arrange for their messages to not be reportable, violating what is often called sender binding. An attack here seeks to send a message that verifies for an honest recipient, but does not verify for a moderator. Since we use a standard digital signature, and we trust the AS to provide correct signature public keys, these verification procedures are equivalent, ruling out such sender binding attacks. We note that this also covers moderators and other privileged users within the group, and that all

UAMs and OAMs are reportable, including those associated to actions.

Fake reports: A malicious user can attempt to frame a user, violating what is often called receiver binding. Here the malicious client tries to trick a moderator into accepting a reported message that was not sent by the honest user specified in the report. Because we trust the AS, doing so would result in an existential forgery against the digital signature scheme.

2.7 Implementation and Evaluation

We now describe a concrete realization of our governance approach: a proof-of-concept messaging platform, called MlsGov, that supports an expressive set of governance policies. This implementation helped us explore the ease with which developers might build platforms with rich governance features. It also allowed us to assess performance overheads relative to governance-free encrypted messaging.

2.7.1 The Platform: MlsGov

We call our platform prototype MlsGov. We forked the Rust implementation OpenMLS [9] to add our new ordered application message proposal type and to modify the exposed API as needed. The changes to the library are minimal, reflecting our goal of modular design. We then implemented our governance layer logic in Rust. It totals 3,988 lines of code as counted by the `cloc` utility, not counting specific policies.

Action	Operation Latency & Traffic				
	Request Gen. (ms)	**Network Overhead** (ms)	**Post-Processing** (ms)	**Total Latency** (ms)	**Traffic** (KB)
Invite (for 63 invitees)	4.78	650.96	12.62	823.47	636.94
Add (for 63 invitees)	20.76	360.39	0.15	485.25	176.65
GovStateAnn.	0.58	258.08	0.12	259.02	9.62
Accept^s	1.28	254.2	0.02	255.72	3.65
RenameGroup	10.18	350.0	0.13	360.68	44.91
VotePropose (Rename)	10.12	313.45	0.16	324.07	33.95
Vote^s (Rename)	0.37	255.58	0.05	256.25	3.89
Send (10-char Text)	0.40	257.13	0.08	257.87	3.54
Send (100-char Text)	0.40	257.4	0.08	258.13	3.86

Action	Sync^s Latency & Traffic			
	Network Overhead (ms)	**Message Processing** (ms)	**Total Latency** (ms)	**Traffic** (KB)
Add (for 63 invitees)	482.88	2.03	486.18	124.92
Accept^s	625.19	28.77	655.29	223.79
RenameGroup	404.32	1.88	407.35	56.7
VotePropose (Rename)	392.95	1.87	395.99	51.22
Vote^s (Rename)	720.08	37.65	759.04	378.59
Send (10-char Text)	371.85	0.63	373.71	37.32
Send (100-char Text)	360.62	0.60	362.38	37.65

Figure 2.5: Latency breakdown for various messaging operations and traffic for a user in a group of 64 users in MlsGov (5-trial average). Clients are on US-East AWS instances while DS/AS are on US-West. For operations^s requested by multiple clients simultaneously, data from the 33rd starting client is used. Total latency represents the time between loading the pre-operation group state and saving the post-operation group state, including server processing and key package generation/update delays. The top table shows operation latency and traffic; the bottom table shows sync latency and traffic for applicable operations. Sync request generations take consistently < 0.01 ms. For results in a group of 1024, refer to Figure 2.7 in Appendix 2.12.

We implemented MlsGov by designing a set of policies, as we elaborate on below. We constructed a CLI client in Rust, totaling 1,431 lines of code. It is capable of managing histories and states for multiple groups in multiple communities. Additionally, we developed simple yet efficient DS and AS services, also in Rust.

Delivery service architecture. Our delivery service is responsible for ferrying ordered, unordered, and welcome messages between clients. Additionally, the DS distributes user-submitted cryptographic material (KeyPackages), which are used by other users to add them to MLS groups. In line with our asynchronous setting, users send and receive messages via issuing requests. There are separate requests for handling ordered and unordered messages. In our implementation, ordered messages are inserted into per-group synchronized queues to ensure total ordering. Unordered messages are placed in individual per-user queues. This means that we have that group membership is revealed to the DS (recall that we do not target membership privacy). When users send ordered messages, in the response, they receive all ordered messages that arrived before theirs in that group. This enables clients to break ties among ordered messages to ensure consistency, all while ensuring minimal lock usage which is important for achieving high throughput.

Included governance features. MlsGov includes RBAC that enables flexible permissions hierarchies among group members. For instance, our system can support having admins that can remove users from a group, add new members to the group, takedown content, change the (private) group name, assign a new moderator, and more. Importantly, admins have the ability to add more RBAC roles and permissions; a common pattern we expect is to have admins delegate

to moderators the ability to takedown content and remove regular users (except the admins). But this is just an example that we implemented, and different moderation hierarchies are configurable via governance actions.

We also built a policy to support voting. Any user can initiate a poll to decide on whether to perform a governance action, which votes to elect new moderators, change the name of the group, modify community guidelines, takedown some content, or perform other governance actions. Our initial implementation waits for all current members of the group to vote, and then executes the governance action, a rename action in our proof of concept, if a simple majority voted to do so. It would be easy to modify the policy to instead have the vote end after a set duration, and take a decision based on who participated (and even set thresholds on how many need to have participated). As described in Section 2.5, we optimized the process to minimize the usage of ordered messages, significantly reducing contention and hence total latency. We also support polls that perform no governance actions, as may often be the case when users want to vote on something that happens off the platform.

Our experience writing policy code indicates that it enables rapidly building rich governance features. We have also begun prototyping reputation systems (modifying user permissions dynamically based on reputation score), setting word filters to automatically block content (which would give a mechanism to enforce the profanity ban mentioned in our example from Section 2.3), and more.

2.7.2 Performance Evaluation

MlsGov is the first system we are aware of that builds rich extensible governance features in an E2EE setting. In this section we evaluate the performance overheads of our governance approach over the baseline of a basic messaging platform. For the latter we use as baseline a version MlsGov with all governance features turned off. We refer to this as MlsBase. This means that no authorization checks occur, reporting signatures are not included with messages, and the policy engine is not run. Our evaluation focuses on assessing the latency and bandwidth overheads incurred by our approach to governance, as well as its effect on scalability in terms of group size and server resources consumed.

We note that, as far as we are aware, ours is the first comprehensive end-to-end benchmark of a messaging system built on MLS. Hence, these absolute performance numbers may be of independent interest.

Experimental setup. We perform our experimental evaluations in a networked setting. We use AWS EC2 to run our AS and DS (on a m7g.medium in US-West-2), and our client machines (in US-East-2, 8 clients per t4g.small instance), in order to test in the WAN setting, with the only exception of server processing time analysis (where all instances are in US-East-2). See Appendix 2.12 for additional details.

Microbenchmarks. We present both client latency and client-to-server bandwidth metrics for various operations in Figures 2.6a and 2.6b across group sizes ranging from 8 to 1024. Latency is measured from the initiation of a client request to the end of processing of server responses, possibly containing new messages from other members. This latency breakdown includes request and

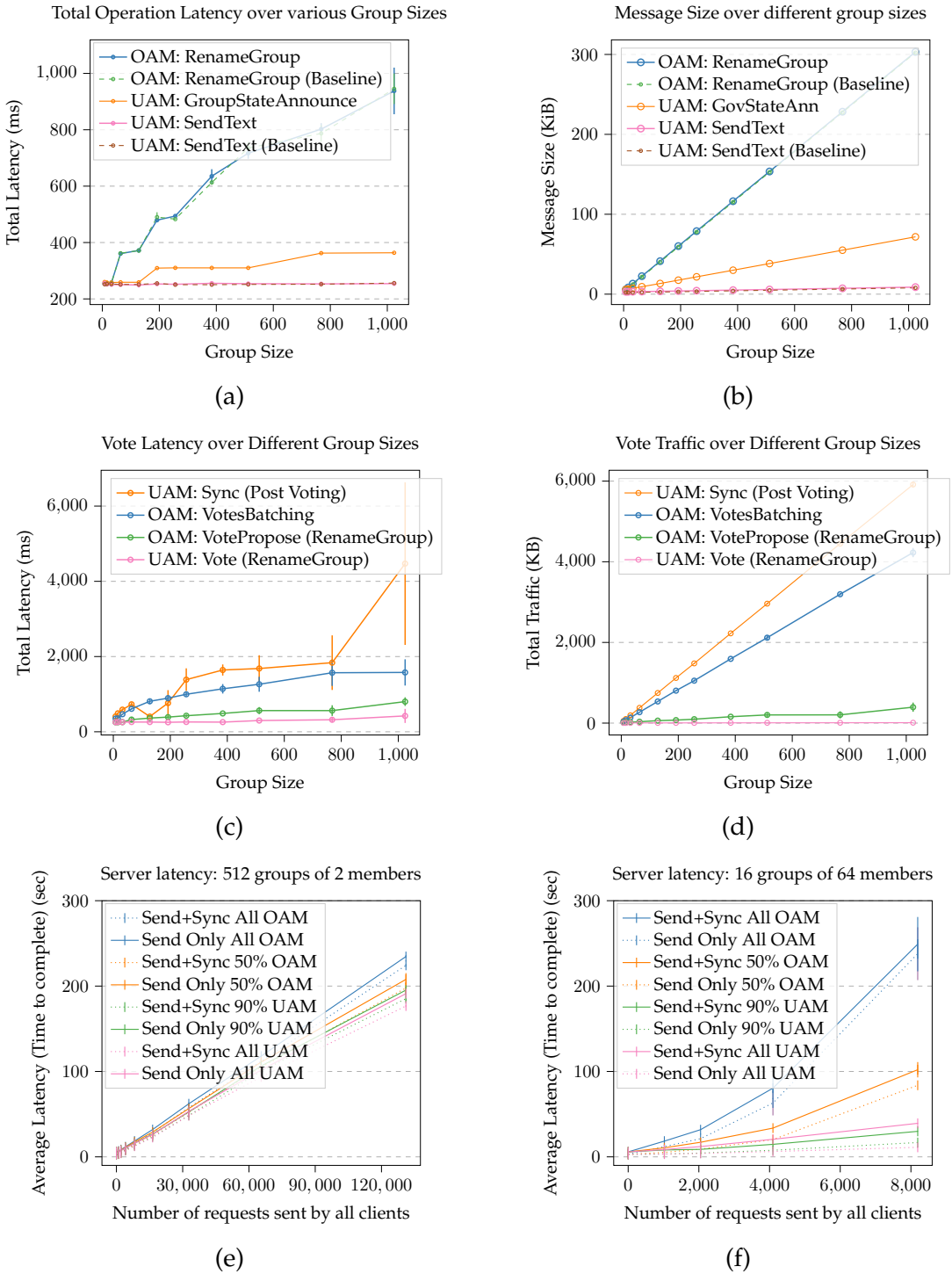

Figure 2.6: Experimental evaluation of MlsGov. Results, averaged over 5 trials, include standard deviation as error bars. In (a)(b), data is from the last starting client for multi-client operations. Only (e)(f) feature data from both servers and clients in US-East, instead of cross-regional – this was an optimization to allow for faster server benchmarks.

its corresponding sync request generation, network communication, and processing.

Most message types, when compared to MlsBase, bear extra bandwidth overheads due to digital signatures, each adding an average of 882.20 bytes on average across 5 trials of all group sizes. The impact on latency and bandwidth varies with group size, as illustrated in Figure 2.6a.

Governance state in group additions depends on group size, growing approximately linearly. This state is only relayed in a GroupStateAnnouncement message when adding users and is depicted in Figure 2.6b.

A significant portion of end-to-end latency is dominated by network channel establishment and data transmission, accounting for $99.6\% \sim 99.7\%$ for UAM and $68.8\% \sim 99.0\%$ for OAM, appeared as Network Overhead in Figure 2.5. MlsGov latency is mostly negligible, with exceptions being the addition of many members or syncing large message quantities in large groups. OAM bears a significant overhead than UAMs and incurs a longer generation and processing time. This is because OAMs are sent through commit messages, which by default carry out key rotations for forward secrecy and post compromise recovery. Our evaluations involve a worst-case configuration of an MLS group in which these updates are linear in size. Governance state announcements increase in size linearly in group size because the RBAC, which is part of the governance state, tracks information for each group member.

For a group of 64, a text message's transmission requires 258 ms and 3.54 KiB of bandwidth. Meanwhile, adding all 63 members takes 485 ms and incurs a bandwidth of 177 KiB. These metrics are tabulated in Figure 2.5. Notably, costs

for sending text messages remain constant, but costs scale linearly for group rename actions and governance state announcements, largely due to key rotations, request metadata, and RBAC-related data.

Voting macro-benchmark. To assess the performance of a complex governance procedure, we benchmarked a representative voting workload. For groups ranging from 16 to 1024 members, we measured latency and bandwidth (averaged over 5 trials, with standard deviations) for a vote to rename the group, reporting the results in Figures 2.6c and 2.6d. In our tests, one user starts the vote, and all members cast votes simultaneously via unordered messages. Once a client receives enough votes, it batches them in an ordered message.

We noted a linear rise in time and bandwidth (communication complexity of sent messages). For a group of 1024, it takes about $0.50\,(0.17)$ seconds (standard deviation in parentheses) for all to vote and $1.58\,(0.35)$ seconds for a member to batch and commit votes with an OAM. Our data shows operations use $9.53\,(0.02)$ KB network traffic per voter and $4227.70\,(113.53)$ KB for the batching member, making our voting approach viable for large groups.

Server evaluation. We evaluate how well our system implementation scales with different request loads by measuring how fast our delivery service can handle requests. The latency is defined as the timespan from the start of the first client's request to the end of the last client request.

We test 1024 users with various workloads: 100% unordered messages with 21-character strings, 100% ordered rename messages of length 11 ± 2 characters (we have the group names depend on the client name, which can vary in size), and randomly generated mixed workloads with either half or 90% unordered

requests, with the remainder as rename requests.

We vary the total number of requests for these four different workloads and measure total latency provided by our server and the achieved throughput. The offered workload ranges from 0 up to 2^{17} requests, capped at around 250 seconds. We performed the same benchmark in a setup with 512 and 16 groups of sizes 2 (direct messaging) and 64, both involving 1024 clients (see Appendix 2.12).

We report on our results in Figures 2.6e and 2.6f. In situations with ordering contention, when others' valid OAM arrives at the DS before a client's generated OAM does, the client needs multiple messages and communication rounds for its request completion. Larger group size means more read per request and also higher contention likelihood, which resulted in a slower completion.

Yet, even under challenging conditions where all OAMs are within large groups, our system can handle an average of 32.89 incoming requests (accompanied by over 2,072 message retrievals) every second. In a more typical scenario with 90% UAM and 10% OAM, the server processes a minimum of 127 requests (8192 message retrievals) per second for groups of 64, and 585.14 requests per second for groups of 2. Employing a more powerful server could decrease the message retrieval latency, but its efficacy may diminish with a high volume of ordered messages per group.

2.8 Conclusion

This paper introduces the novel goal of private hierarchical governance for encrypted group messaging. We show how community moderation systems widely used on plaintext platforms can be adapted to the E2EE setting while maintaining privacy, integrity, and accountability. Our solution is a radical departure from prior E2EE moderation approaches which focus on platform-driven moderation. As a result, private hierarchical governance opens up new possibilities for abuse mitigation that do not suffer from the transparency and accountability issues that arise with platform-driven solutions.

Our design pushes the execution of governance to client devices and makes use of the messaging layer to maintain shared encrypted state. Instead of focusing on hard-coding specific policies, our design enables a framework for expressing general policies, such as voting and content filter enforcement. Through enabling reporting to both platform and community moderators, our design provides channels to inform moderation at both levels. We conduct a security analysis of our design by reasoning through possible attack scenarios. We build and benchmark a prototype encrypted messaging platform that realizes private hierarchical governance in order to demonstrate its practicality.

2.9 MLS Background

In this section, we summarize details about the MLS protocol that are relevant to our work.

MLS architecture. MLS [32] relies on two services: an *authentication service (AS)* for managing user identities and a *delivery service (DS)* for transporting messages. The concrete design of these services are not specified, and implementors are free to design within the API [28].

The AS is a service that serves the role of a traditional PKI, mapping user identities (usernames) to certificates. Looking ahead, we use credentials that include for each user a long-lived digital signature public key. This long-lived public key can be used to verify the authenticity of further cryptographic keys, and ultimately allows cryptographically verifying, for example, sent messages as emanating from a particular sender username. As in any PKI, security relies on the AS being a trusted third-party that provides users with the most up-to-date views of these mappings. We suggest that deployments use a key transparency mechanism [41,94,95,112] to enable users or auditors to check for malicious behavior on the part of the authentication service.

The DS transfers (encrypted) messages between users in the network. In contrast to solutions built out of independent pairwise channels, in our implementation we opt for a server fan-out design. Clients send messages to the delivery service along with a list of intended recipients. The DS then forwards the messages along to those recipients. Over the course of its operation, the delivery service does not need to keep track of who belongs to which group, however, it

can infer membership based on which recipients a message specifies.

Protocol overview. The MLS protocol [28] provides a mechanism for group encrypted messaging. To do so, it uses the bhargavanTreekem18 protocol [33, 45] to allow a group to efficiently maintain a shared secret that evolves over time. KEM (key encapsulation mechanism) public keys are authenticated via long-lived signing keys that are, in turn, authenticated by the AS. bhargavanTreekem18 allows for efficient changes to group membership and provides strong forward secrecy and post-compromise security guarantees.

Protocol messages in MLS can be either public or private. Public MLS messages are signed by the sender. Private MLS messages are signed by the sender and then encrypted using a current group-held symmetric key with an appropriate authenticated encryption with associated data (AEAD) scheme (such as AES-GCM [93]). Group members can verify the sender of both private and public messages; public messages can additionally be verified by the DS should that be useful to applications.

MLS has two classes of messages, *application messages* and *handshake messages*. The former are private messages used to transmit plaintext data to the group. Delivery of application messages is best-effort, and MLS is explicitly designed to allow message reordering or even dropping of messages. Handshake messages are more complicated as they are used to maintain shared group state, such as the current bhargavanTreekem18. To add a new client to a group, the user adding the new client prepares a welcome message. These include a serialization of the current shared cryptographic state, and are sent as a private message.

The shared group state must evolve consistently over time, as users join and leave, with updates to user KEM public keys for forward secrecy, etc. MLS therefore requires a consensus mechanism to ensure that clients agree on this evolution.

2.10 Additional Security Analysis

Our design prevents the platform from directly learning about governance actions, messages, the group name/topic, block list, votes cast, election outcomes, and member profiles. However, the platform may be able to infer via traffic analysis who serves as the moderator of the group, the outcome of a vote, and the type of governance messages sent in the group. Whether such attacks will be effective in practice is unclear, given that all messages are encrypted and the possibility of deploying countermeasures that have been explored in other contexts such as TLS [52].

To elaborate, suppose a group has a policy in which only moderators can add new members. Although `Add` messages are encrypted, the DS may still infer which messages add new members through keeping track of the recipient list of messages. For instance, the DS may notice that after U_1 sent a message (whose contents are encrypted), the recipient list in U_1's next message contains one more user, U_2. As a result, the DS can infer that the first encrypted message probably added U_2 to the group and that U_1 is authorized to add new members to the group. These inferences are likely to work in some settings and not others (e.g., due to noise), and so future work will be needed to evaluate their practicality. What's more, our approach to governance is amenable to deployment of traffic

analysis mitigations such as padding, dummy messages, and metadata private messaging.

Only received action messages can be reported with cryptographic assurance to the moderator, others will only be trustworthy should client software be honest. To elaborate, in a conversation between Alice and Bob, Alice can report the messages she receives from Bob but cannot cryptographically prove that Bob received particular messages from her. This limitation also exists for other asymmetric cryptographic message franking solutions [113]. The reason is that nothing prevents a malicious reporter with modified client software from generating a new action message, signing it, and reporting it to a moderator—without actually sending the action message to any group.

Another related limitation is that ordering information of messages is not currently cryptographically verifiable, and only trustworthy should client software be honest. In fact there is no absolute ordering of content actions, since these are encoded as UAMs, and so this issue is in some sense fundamental. That said, one might add DS-signed time stamps to (encrypted) messages to enforce some partial ordering.

Our reporting mechanisms do not immediately enable the reporting of deviation from protocol behavior. For instance, a malicious client sending an honest client an incorrect initial state is not reportable. Even though the incorrect group state announcement is reportable, proving that it is incorrect would require reporting every commit sent in the group in order to compute what the correct group state should be. This is in general not practical or desirable. Future work could explore providing cryptographic assurance for commit sequences using zero-knowledge proofs. Regardless, honest clients can still issue claims to the

platform moderator that a client generated an incorrect group state announcement. If many such reports are received, say from a majority of a group, a platform moderator would have reason to take action against the reported inviter.

MLS provides strong forward-secrecy (FS) and post compromise security (PCS). FS entails the confidentiality of messages sent before a compromise occurs and PCS guarantees the confidentiality of messages sent after a healing procedure following a compromise. We now discuss how our addition of governance, in particular, our mechanism for shared encrypted state, interacts with the FS and PCS properties of MLS. Recall that state updates are sent via encrypted commit messages and that new users learn the current aggregate state via a message sent by a current group member. These update messages are protected by FS and PCS, but the reliance of our system on maintaining long-term aggregate state changes the implications of FS. In particular, when a group state update message (sent upon a new user joining a group) is compromised, the attacker can infer the contents of prior messages that led to the aggregate state they observed. This issue surrounding FS seems inherent to similar systems that must maintain long-term accumulated state, such as end-to-end encrypted backups. Semantic information from compromised messages can also leak information about prior messages sent, even if those messages are cryptographically protected by FS. Finally, given that users are often not required to turn on disappearing messages, device compromise trivially reveals to an attacker past messages in the clear, which are stored on-device.

In summary, our governance mechanism requires maintaining long-term encrypted state that may reveal information about messages sent before FS ratchets. The extent of such leakage is highly dependent on what specific policies

are deployed. Furthermore, we cannot deploy a "disappearing messages" type feature for governance state in a straightforward manner. If our aim were to hide information about past governance messages, we could do so by re-setting governance state (like the list of moderators) periodically, but this would likely be prohibitive from a usability standpoint. To be clear, the addition of governance does not impact forward secrecy of content-carrying messages. Rather, maintaining long-term state that gradually evolves over the lifetime of a group necessitates that prior governance messages be reflected within the current governance state.

2.11 Additional Implementation Details

Communications and Persistent States. We designed and implemented a custom protocol for communication between clients and the two servers. To do so we use standard approaches, and used JSON for data serialization and web sockets for transferring messages between clients and servers.

Signature. We use the ed25519-dalek [48] implementation of ed25519 for governance digital signatures. Clients sign all actions and include the signature in the request.

Unordered and Ordered Voting. We implemented both voting in all ordered and unordered (except for the start and ending batch-commit message(s) being ordered) format. When all clients start to vote at the same time, ordered voting can have very high contention that even if coupled with an exponential back-off mechanism (a common retry strategy), the completion time could still last

for minutes for bigger groups. Unordered voting on the other hand brings voting time down to seconds, but unordered messages are relayed best-effort and could be lost, with a remedy that the client can batch-commit their unordered vote themselves should their vote does not appear in other members' batch-commit.

2.12 Additional Experimental Results and Details

Additional results. We include additional experimental results for our system in Figure 2.7, which reports micro-benchmark results for a group consisting of 1024 members.

Client Instances. To mimic performance on a low-budget device, every 8 clients run on a t4g.small instance, which has 2 GiB of RAM and 2 vCPU, and network bandwidth up to 5 Gbps.

Server Instance. The AS and DS run on a single m7g.medium instance, which has 4 GiB of RAM and 1 vCPU, and network bandwidth up to 12.5 Gbps.

Text Field Length. Text message content consists of strings of 10 ± 1 (default unless specified otherwise) or 100 characters. Rename message content contains strings of 15 ± 1 characters. Vote message content contains the string "yes". Invites add a single new user with a name consisting of $1 \sim 4$ characters to the group.

Benchmarking Tooling. We use boto3 with SSH (Paramiko) to automate the process of creating and running instances for experiments. We use an t4g.xlarge

Action	Operation Latency & Traffic Request Gen. (ms)	Network Delay (ms)	Post-Processing (ms)	Total Latency (ms)	Traffic (KB)
Invite (1023 invitees)	$66.94^{(1.09)}$	$1170.27^{(90.75)}$	$200.25^{(0.17)}$	$1866.37^{(103.33)}$	$10254.38^{(2.85)}$
Add (1023 invitees)	$337.32^{(0.75)}$	$551.04^{(25.3)}$	$1.3^{(0.09)}$	$1283.11^{(39.56)}$	$2752.14^{(0.9)}$
GovStateAnnoun.	$1.95^{(0.07)}$	$298.87^{(22.03)}$	$0.53^{(0.04)}$	$363.62^{(1.94)}$	$72.12^{(0.11)}$
Accept	$3.84^{(0.24)}$	$254.45^{(3.67)}$	$0.07^{(0.07)}$	$259.12^{(3.36)}$	$9.26^{(0.02)}$
RenameGroup	$141.37^{(1.09)}$	$644.03^{(82.48)}$	$0.5^{(0.02)}$	$937.78^{(82.64)}$	$600.5^{(0.16)}$
VotePropose (Rename)	$94.4^{(29.66)}$	$574.26^{(74.46)}$	$0.85^{(0.05)}$	$801.44^{(114.64)}$	$394.29^{(113.98)}$
Vote (Rename)	$0.63^{(0.01)}$	$547.78^{(183.57)}$	$0.05^{(0.0)}$	$549.16^{(172.16)}$	$9.53^{(0.02)}$
Send (10-char Text)	$0.72^{(0.03)}$	$271.94^{(1.28)}$	$0.08^{(0.01)}$	$273.83^{(1.24)}$	$9.17^{(0.02)}$
Send (100-char Text)	$0.65^{(0.03)}$	$257.31^{(0.62)}$	$0.08^{(0.0)}$	$259.01^{(0.54)}$	$9.47^{(0.02)}$

Action	Sync Latency & Traffic Network Delay (ms)	Message Processing (ms)	Total Latency (ms)	Traffic (KB)
Invite/Add/Ann (1023 invitees)	$3563.57^{(1948.93)}$	$20.10^{(0.83)}$	$3585.59^{(1542.01)}$	$1795.32^{(0.47)}$
Accept	$2535.59^{(1149.73)}$	$614428.41^{(51687.51)}$	$616969.1^{(51930.98)}$	$3529.61^{(0.64)}$
RenameGroup	$1476.81^{(266.87)}$	$8.21^{(2.59)}$	$1486.52^{(248.27)}$	$748.49^{(0.15)}$
VotePropose (Rename)	$1242.37^{(358.08)}$	$6.27^{(2.45)}$	$1250.14^{(264.72)}$	$645.38^{(56.95)}$
Vote (Rename)	$3606.64^{(620.86)}$	$602.40^{(26.95)}$	$4214.05^{(447.57)}$	$5913.89^{(56.18)}$
Send (10-char Text)	$613.35^{(43.62)}$	$0.62^{(0.04)}$	$615.59^{(40.18)}$	$454.1^{(0.08)}$
Send (100-char Text)	$593.48^{(9.72)}$	$0.65^{(0.02)}$	$595.85^{(9.37)}$	$454.38^{(0.09)}$

Figure 2.7: Operation latency breakdown for a user in a group of 1024 users in MlsGov (5-trial average (std.)). Clients are on US-East AWS instances while DS/AS are on US-West. For operations requested by multiple clients simultaneously, data from the 513th starting client is used. Total latency represents the time between loading the pre-operation group state and saving the post-operation group state, including server processing and key package generation/update delays.

instance to issue the operation command to all instances, which can reach 1024 clients (128 instances) within 3 seconds unless the commands are too demanding and are draining instance resources.

Server Evaluation Detail. The workload is generated by each client looping through requests consisted of the target distribution, with all clients starts uniformly in the loop. We disable group state saving to reduce client-induced overheads. To mimic natural interactions, users send sync messages before their requests in these workloads.

2.13 Pseudocode Specification

We present a pseudocode specification of our governance protocol in Figure 2.8 and Figure 2.9. We indicate explicitly where our system calls out to MLS. The MLS.SendUnordMsg call corresponds to sending an MLS `ApplicationMessage`. The MLS.SendOrdMsg corresponds to usage of our new ordered application message proposal type. An overview of the MLS-level functionalities can be found in the MLS protocol specification [28] and OpenMLS documentation [7]. We use the variable st_{mls} to denote state associated with the MLS messaging layer. The variable st_{gov} represents shared governance state within a group. The local state a client maintains is represented by st_{loc}. In addition to storing the shared governance state, st_{loc} contains the content state st_{con}, which may differ between users of the same group, due to a lack of ordering guarantees for content-carrying messages. Groups have associated identifiers, denoted as gid. The signature key pair $(pk_{\text{R}}, sk_{\text{R}})$ enables reporting signatures on messages.

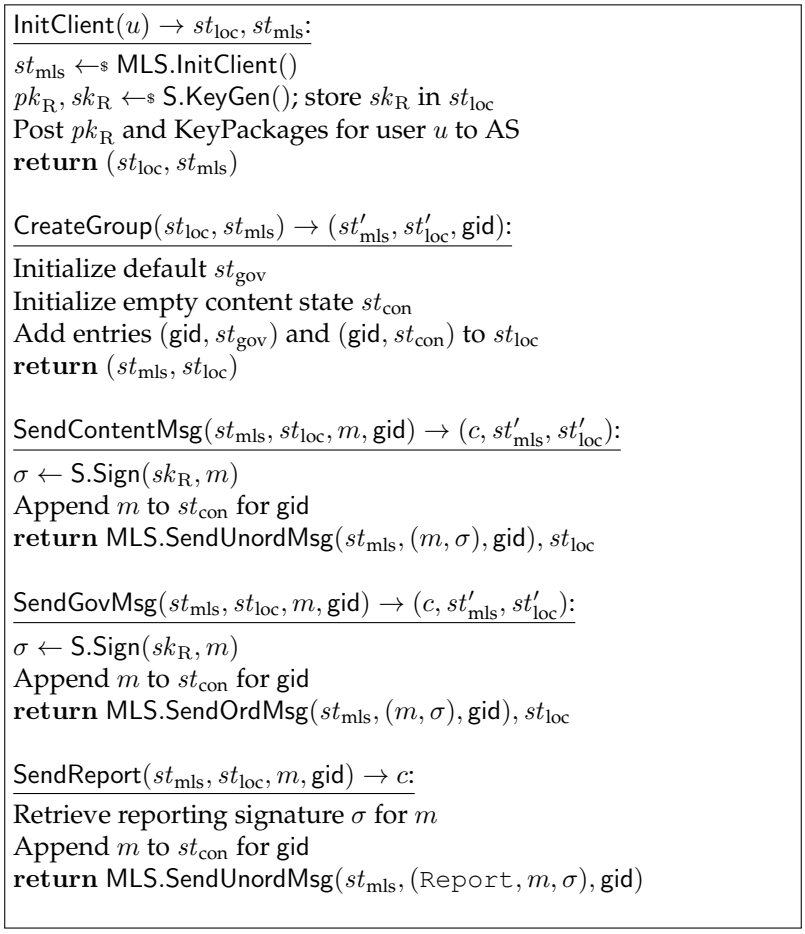

Figure 2.8: A pseudocode specification of our governance protocol.

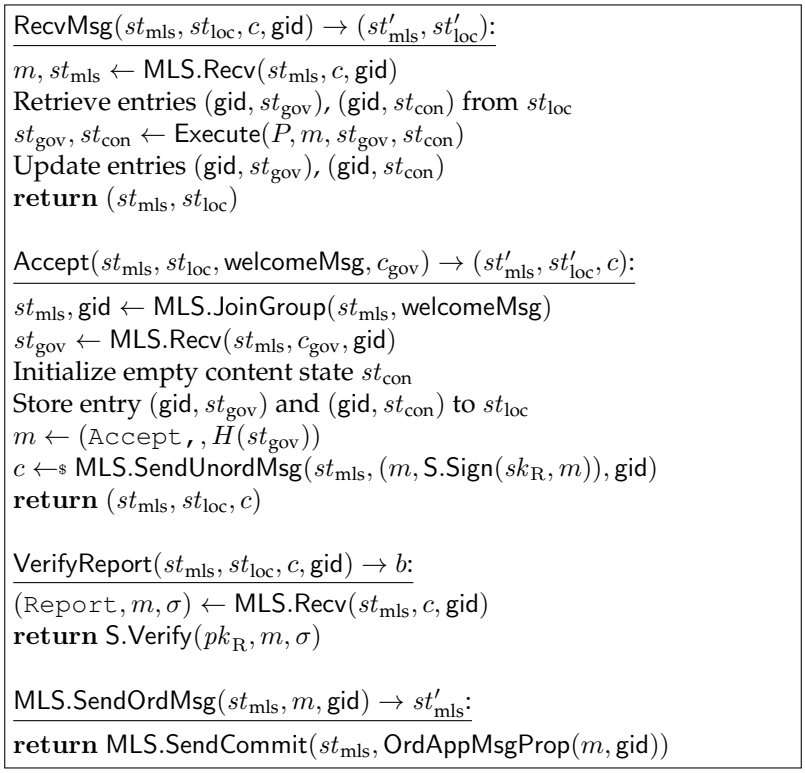

Figure 2.9: A pseudocode specification of our governance protocol (continued).

CHAPTER 3

TRANSCRIPT FRANKING FOR ENCRYPTED MESSAGING

3.1 Introduction

End-to-end encrypted (E2EE) messaging is used by billions of people through platforms like Whatsapp, Signal, and iMessage [124]. As a result, users enjoy strong security and privacy protections even in the face of messaging platform compromise by malicious insiders, remote attackers, or government overreach. Abuse, hate, and harassment, however, are not prevented or mitigated by encryption, and encrypted messaging platforms are used to spread misinformation, incitements of violence, and illegal content [87]. As a result, a rapidly growing body of work has sought to provide trust and safety features for encrypted messaging, without diminishing its privacy benefits [105].

One important line of work targets secure reporting of abusive messages (see [105]). When users receive harmful content, they can report it to the platform, which can in turn take appropriate action against the user that sent the problematic content. Such user-driven reporting features are widespread on plaintext platforms and play an instrumental role in content moderation across a wide range of abuse types [100]. In encrypted messaging, however, the platform cannot trivially verify that a report corresponds to a transmitted message.

Facebook's message franking feature [1] was the first to target cryptographically verifiable abuse reports. Message franking targets not compromising the confidentiality of unreported messages, and preventing attacks that undermine the trustworthiness of reporting: users should not be able to report messages

that were not sent, nor be able to send messages that cannot be reported. These security properties were first formalized in [63] as receiver binding and sender binding, respectively. While Facebook's first design had a sender binding vulnerability [50], we now have message franking protocols with strong assurance in their security [50, 63, 65]. Subsequent work extended to provide asymmetric message franking schemes (AMFs) [67, 113] for two-party sender-anonymous messaging, group AMFs [78], franking for two-party channels [65], and message franking that allows only revealing parts of messages [81].

All those treatments of message franking only support reporting individual messages. In practice, however, moderators typically need visibility into more of a conversation to make judgements, and indeed existing abuse reporting workflows do report surrounding messages when one is reported [11, 63]. Recent work [122] reports that users would find it useful to have more agency in specifying what portions of their private conversations are disclosed to moderators, which is not something current approaches offer.

Despite this, to date, there has been no attempt to show how to provide cryptographically verifiable reporting of multi-message conversations. Near-at-hand approaches, including those used in practice, do not provide a satisfying level of security. Consider reporting with message franking: each individual message can be verified along with a platform timestamp of when it was sent. But a malicious client can simply undetectably omit messages from a report. For example consider the conversation between Alice and Bob shown in Figure 3.1. If Alice reports the conversation with omission of m_1 it blocks moderators from interpreting Bob's message m_3 as replying to m_1 rather than mocking Alice's loss. A more subtle issue is that even high-precision timestamps do not establish

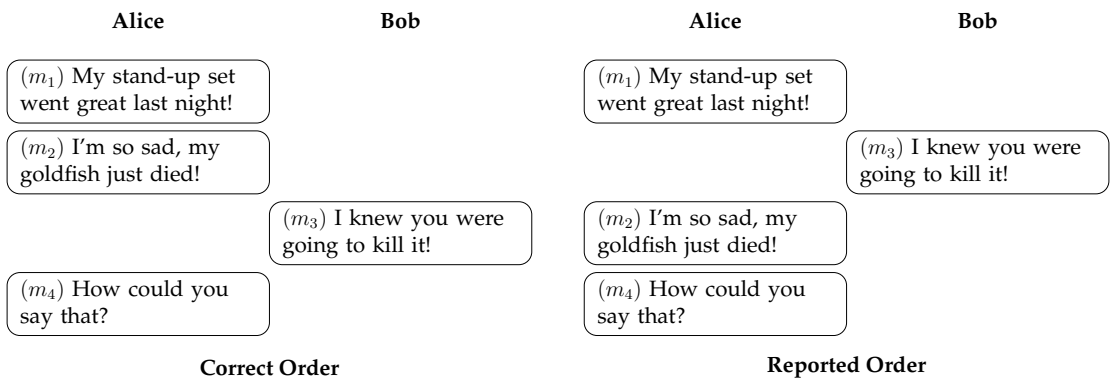

Figure 3.1: An example conversation in which message ordering impacts interpretation. A report of this conversation should confirm for moderators the causal ordering.

strict causal ordering [79]. Let $m < m'$ represent that m was received before m' was sent. Returning to the figure, it could be that $m_2 < m_3$, or it could be that m_3 was sent before Bob received m_2. The result in the latter case would be Bob's view of the conversation being different from Alice's.

The problem of conversation ordering and moderation has been known to practitioners since at least 2014 [89], but only recently has there been a first effort to address it by Chen and Fischlin [44]. They propose a message franking protocol, $\mathsf{MFCh}_{\mathrm{cFB}}$, in which clients report observed message ordering via franking metadata alongside content. While their approach provides a novel augmentation of message franking with causality, they stop short of providing a fleshed out solution for multi-message franking. Their security modeling considers reporting individual received messages. Meanwhile, allowing reporters to disclose sent messages is crucial to multi-message reporting with full context. One could consider a natural extension of $\mathsf{MFCh}_{\mathrm{cFB}}$ in which the report function is invoked for each message and both parties are involved in disclosing each other's messages. This, however, is susceptible to denial of service as the abusive party can simply refuse to cooperate and go offline, withholding crucial context.

Moreover, $\mathsf{MFCh}_{\mathrm{cFB}}$ is susceptible to integrity attacks. Since causality data is client-generated, malicious clients can provide incorrect information about message ordering. Consider the example conversation in Figure 3.1. We demonstrate an attack in which Bob makes Alice observe the view on the left while making it so that Alice can only report the view on the right. Bob simply sends m_3 after having received m_2 but attaches causality metadata to m_3 indicating he has seen only m_1. Hence, Bob can force Alice to observe an upsetting message ordering while making seem as if it were due to the network reordering messages.

Our contributions. We suggest a new approach that we call *transcript franking*. This cryptographic protocol goal allows users to report some or all of two-party or group conversations with stronger security guarantees about message ordering and omissions. We define the syntax and semantics of a transcript franking scheme and provide formal security definitions. We go on to detail transcript franking schemes for both the two-party and group messaging settings; our schemes are practical to deploy and avoid issues like those above, clarifying in a single report all relevant information about the messages reported, including causal ordering. To do so, our schemes deviate from Chen-Fischlin's approach of using client-provided causality metadata, instead taking advantage of the fact that the platform can establish causal ordering over ciphertexts and check that reports are consistent with it. We prove that our new schemes meet our new security goals under standard assumptions.

We treat both two-party (direct) and group messaging settings; we explain further each, starting with the former.

Two-party transcript franking. We start with the two-party case, where only

two clients are involved in a conversation (sometimes called direct messaging). Our goal is to enable either participant in the encrypted conversation to report all or part of the conversation to the platform. Our starting point is symmetric message franking (SMF) [63, 65], in which the clients encrypt messages using committing authenticated encryption with associated data (AEAD) and the platform MACs (portions of) the resulting ciphertexts to produce a reporting tag. To report a message, the ciphertext, secret encryption key, and the reporting tag are all sent to the provider.

In this setting where we rely on fast symmetric primitives, the veracity of a ciphertext can only be checked by those who have access to the secret keys—before reporting, just the two clients. Prior work on SMFs took this to mean that one cannot support reporting a sent message, since the reporter could be unreliable. But if we want to allow reporting more of a conversation, we must support this. Intuitively, our approach will be to leverage acknowledgements of received ciphertexts to register their validity for their use in reporting.

To do so, we first enrich the architectural model to surface how the provider manages sending and receiving events separately. This more accurately captures the asynchronous nature of messaging. Formally, the platform is represented by a pair of algorithms TagSend and TagRecv. We allow the platform to maintain per-conversation state; we also give a way to outsource this state securely to clients. A client sends a message by running an algorithm Snd and submitting the resulting ciphertext to the platform by calling TagSend. To receive a message, a recipient client runs an algorithm Rcv on it, and if it accepts the message, indicates so to the server, which then calls TagRecv. The latter readies a cryptographic reception acknowledgement for both the sender and the receiver.

A set of messages can be reported by submitting their ciphertexts, their corresponding secret keys, and platform-provided reporting tags. The moderator can process them via an algorithm Judge that verifies them, and produces a causal graph providing the moderator with: (1) a partial ordering over all messages that implies a total ordering over messages sent in either direction; and (2) indication of gaps—unreported messages sent between reported messages.

We require that transcript franking schemes provide confidentiality and integrity of messages that aren't reported in the face of a malicious provider. More complex is the security of reporting, in which the platform is trusted, but clients are not. Here we formalize two security goals for transcript franking schemes, which can be viewed as strengthening SMF sender and receiver binding. Coming up with definitions that capture transcripts, rather than individual messages, proved challenging. For example, any given conversation can now give rise to all sorts of possible reports: the entire conversation or any subset of that conversation, including possibly just a single message.

Our first definition is *transcript reportability*. Here the security game tasks an adversary controlling one client with interacting with the provider and some other honest client. The adversary attempts to generate a message transcript such that the honest client cannot successfully report some adversarially-chosen subset of messages. In the case where only a single message is reported, this coincides with sender binding, but it goes much beyond since it requires that any subset of the conversation be reportable, including messages sent by the reporter.

The second main security notion is *transcript integrity*. Intuitively we want to not only ensure that no maliciously generated report can frame someone as hav-

ing sent something when they have not, but, moreover, ensure that the reported transcript does not lie about ordering or omissions. Perhaps counterintuitively, we increase the adversarial power over prior treatments of receiver binding by allowing the adversary to control *both* clients in a conversation to arbitrarily deviate from the protocol. The adversary has oracles to send ciphertexts to the platform and register having received them. The game keeps track of a ground truth graph of message transmission. This matches the view of the platform in terms of sending and receiving event orderings for each transmitted ciphertext. Then, the adversary's goal is to come up with either a report that generates a causality graph inconsistent with the ground truth graph, or two reports that are inconsistent with each other. We define consistency relative to our causality graph abstraction.

The quad-counter construction. We now turn to constructions. We want to ensure practicality, using fast cryptographic mechanisms and avoiding unrealistic storage constraints. The main underlying idea is to have the platform carefully attest to the observed causal order of ciphertexts. Since we allow stateful platforms, we could of course just store a log of all sending and reception events that occurred, with their corresponding ciphertexts. But this would be prohibitively expensive in terms of storage. Instead, we can use MAC'd counters of events. Namely, for each party we have a pair of counters, a sending counter incremented each time that party sends a ciphertext, and a reception counter incremented each time a ciphertext is registered as received. The platform generates a cryptographic acknowledgement for each send and receive event: a tuple including the party identifiers, the type of event (send or receive), a binding commitment to the ciphertext, and the current counters, together with a MAC over the tuple using a platform-held secret key. Cryptographic acknowledgements

are shared with both parties. The platform need only retain the four counters, hence the name of this protocol as the quad-counter construction (QCC).

A reporter can choose any subset of messages, and submits the identity of the sender, the message itself, the cryptographic commitment to the message, and both the sending and reception cryptographic acknowledgements. The platform can then verify each tuple, and construct a causality graph that indicates the precise causality order of the reported messages plus how many send and receive events were omitted from the report.

We show that, under standard assumptions on the MAC and commitment scheme, we achieve transcript integrity.

Extensions. The quad-counter construction readily extends to work in group messaging settings, by having a sending and reception counter for each party (for a total of $2n$ counters where n is the number of participants). The cryptographic acknowledgements are otherwise similar to the two-party case. We formalize this setting, showing how our definitions and results handle it securely.

One potential downside of the quad-counter construction and its generalization, as compared to in-use (but insecure) approaches like timestamps, is that the platform must now keep per-conversation state. While this state is tiny, it would be better to avoid, due to it complicating large-scale deployments where one would prefer to have platform servers stateless and able to load balance any message event across any server without having to synchronize state. We describe an extension to our approach which outsources the state to the clients, at the cost of slightly extra data being sent to the platform, and relying on honest users to report replays. See Section 4.6 for details.

3.2 Preliminaries

3.2.1 General Notation and Primitives

Let $\mathbb{Z}^*$ denote the non-negative integers $\{0, 1, \ldots\}$. To indicate the range of elements $\{0, 1, \ldots, N-1\}$, we use the shorthand $[N]$. The symbol $\lambda \in \mathbb{Z}^*$ denotes the security parameter. For an adversary $\mathcal{A}$ in a game $\mathbf{G}$, we use $\Pr[\mathbf{G}(\mathcal{A}) \Rightarrow x]$ to denote the probability that the $\mathbf{G}$ outputs x when run with adversary $\mathcal{A}$. With a tuple, we refer to its elements via 0-indexed positions or names. For instance, $c.x$ or $c[0]$ refer to the first element of the tuple $c = (x, y, z)$. Multiple indexing is also allowed: $c.(x, z)$ extracts the first and third elements of the tuple, while $c[1:2]$ extracts the last two elements (for ranges, both the start and end indices are inclusive). We indicate structs as sets of variables, e.g., $s \leftarrow \{x, y, z\}$. When s is in scope, we allow accessing x directly in order to simplify notation. The notation $d \leftarrow D(x)$ indicates obtaining the output d from a deterministic algorithm D, when fed input x. For a randomized algorithm R, we write $r \leftarrow\!\!\$\ R(x)$ to denote obtaining the output r from the input x. We utilize the terms *algorithm*, *routine*, and *procedure* interchangeably.

Bidirectional channels. To model two-party communication, we make use of a bidirectional channel abstraction, borrowing syntax from [44]. A bidirectional channel Ch consists of three procedures $\mathsf{Ch} = (\mathsf{Init}, \mathsf{Snd}, \mathsf{Rcv})$, defined over a key space $\mathcal{K}$, a message space $\mathcal{M}$, a party space $\{0, 1\}$, a ciphertext space $\mathcal{C}$, and a state space $\mathcal{S}$. The variable P and the labels $\{0, 1\}$ are used to indicate the two parties. Let the notation $\bar{P}$ be shorthand for $1 - P$. We elaborate on these procedures below.

- $st \leftarrow \mathsf{Init}(P, k)$ generates initial channel state $st \in \mathcal{S}$ for party $P \in \{0, 1\}$ with key $k \in \mathcal{K}$.
- $st', c \leftarrow\!\!\$\ \mathsf{Snd}(P, st, m)$ produces a ciphertext $c \in \mathcal{C}$ from party $P \in \{0, 1\}$ for plaintext $m \in \mathcal{M}$ and client state st, yielding updated state st'.
- $st', m, i \leftarrow \mathsf{Rcv}(P, st, c)$ decrypts a ciphertext $c \in \mathcal{C}$, received by the party $P \in \{0, 1\}$, to plaintext $m \in \mathcal{M}$ along with a sending index $i \in \mathbb{Z}^*$.

Correctness of a channel requires that all honestly sent messages can be successfully received with the correct sending index. For security, we expect standard confidentiality and integrity properties. See [44] for more details.

Message authentication code. A message authentication code (MAC) consists of the algorithms $\mathsf{MAC} = (\mathsf{KGen}, \mathsf{Tag}, \mathsf{Ver})$ defined over a key space $\mathcal{K}$, a message space $\mathcal{M}$, and a tag space $\mathcal{T}$. The key generation procedure KGen outputs a random key $k \in \mathcal{K}$. The procedure $\mathsf{Tag}(k, m)$ outputs a tag $t \in \mathcal{T}$ for a message $m \in \mathcal{M}$. To verify a tag t on a message m, one runs the procedure $\mathsf{Ver}(k, m, t)$, which outputs $b \in \{0, 1\}$. An output of 1 indicates a valid tag on the message while an output of 0 indicates an invalid tag. For a MAC to be considered secure, it has to satisfy existential unforgeability under a chosen message attack (EUF-CMA). This means that, when given oracle access to $\mathsf{Tag}(k, \cdot)$ and $\mathsf{Ver}(k, \cdot, \cdot)$, for $k \leftarrow\!\!\$\ \mathsf{KGen}()$, an adversary $\mathcal{A}$ has a negligible probability of producing (m, t), where m is not previously queried to Tag, that passes the verification check. We denote this probability as the advantage $\mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{A})$.

Commitment scheme. A commitment scheme CS consists of a pair of algorithms $(\mathsf{Com}, \mathsf{VerC})$ defined over a message space $\mathcal{M}$, a key space $\mathcal{K}$, and a commitment space $\mathcal{Q}$. The randomized algorithm $\mathsf{Com}(m)$ outputs a pair

$(k, c) \in \mathcal{K} \times \mathcal{Q}$, where c is the commitment and k is a key that allows opening the commitment to the original message $m \in \mathcal{M}$. In terms of security, a commitment scheme is often required to be *hiding* and *binding*. To satisfy the hiding property, the commitment c must reveal nothing about the message m. We formalize this via a real-or-random notion that requires that no efficient adversary can distinguish a commitment from a random bit-string of the same length. The binding property requires that an adversary $\mathcal{A}$ has a negligible probability in producing a tuple (m, k, m', k', c), where $m \neq m'$, $\mathsf{VerC}(m, k, c) = 1$, and $\mathsf{VerC}(m', k', c) = 1$.

Message franking. User-driven reporting for end-to-end encrypted messaging is captured by message franking [1, 50, 63]. For now, we focus our attention on two-party conversations between users. For the sake of notational simplicity, we elide associated metadata that clients or the server may associate with the message (e.g., timestamps). In accordance with our usage of messaging channels, we draw on message franking channels [65]. We opt for the syntax used in Chen and Fischlin's work [44]. The procedure $\mathsf{Rcv}(P, st, c)$ outputs st', m, k_f, i, where k_f is a key opening the commitment $c.c_f$, which is stored as part of the ciphertext c. There is also a server tagging procedure $\mathsf{SrvTag}(st_S, P, c_f)$, which outputs a tag t on a franking commitment c_f, which corresponds to a ciphertext sent by party P. A reporting procedure $\mathsf{Rprt}(st_S, P, m, k_f, c_f, t)$ verifies for the server that t is a server tag on c_f, which opens to a message m.

3.2.2 Causality Graphs

In order to model message transmission in a way that accounts for asynchronous networks, we use causality graphs. We define our causality graph object here, which draws heavily on the graph formalism used in [44]. A causality graph G modeling two-party messaging is a directed bipartite graph consisting of two disjoint sets of vertices V_0 and V_1 and an edge set E. An edge is a pair of vertices (v_1, v_2) where $v_1 \in V_P$ and $v_2 \in V_{\bar{P}}$ for some $P \in \{0, 1\}$. We write $V = V_0 \cup V_1$ to refer to the full set of vertices within the graph. Each vertex $v \in V$ contains four pieces of data: an action type $t \in \{S, R\}$, a sending counter cs, a reception counter cr, and a message $m \in \mathcal{M}$. Indeed, these four pieces of data are sufficient to uniquely identify a vertex within a single conversation. We can therefore define a vertex space $\mathcal{V}$ as the direct product $\{S, R\} \times \mathbb{Z}^* \times \mathbb{Z}^* \times \mathcal{M}$, and the edge space $\mathcal{E}$ as $\mathcal{V} \times \mathcal{V}$. The action type t indicates whether the vertex corresponds to a sending (S) event or a reception (R) event. For any message sent from P to $\bar{P}$, there is an edge from a sending vertex in P to a receiving vertex in $\bar{P}$. We define the sub-graph relation as follows: for two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we write $G_1 \subseteq G_2$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. We will also allow causality graphs that do not contain messages, which will become useful for our security definitions. Let G be a causality graph; we denote the message-excluded version of G as $\widetilde{G} = (\{v[0:2] \mid v \in G.V\}, \{(v_1[0:2], v_2[0:2]) \mid (v_1, v_2) \in G.E\})$.

The counters cs and cr are monotonically increasing counters over sending and reception event respectively for each party $P \in \{0, 1\}$. The addition of a sending event to V_P increments the sending counter for V_P while the addition of a reception event increments the reception counter. Consider a sending vertex

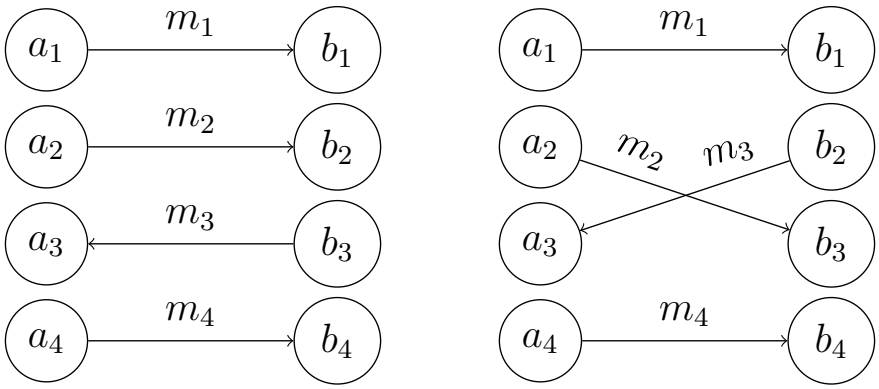

Figure 3.2: Examples of causality graphs. Let the a nodes correspond to Alice and the b nodes correspond to Bob. The left graph corresponds to a situation in which both Alice and Bob view the left ordering from Figure 3.1. The right graph leads to Alice viewing the left ordering and Bob viewing the right ordering from the same figure.

$v \in V_P$. We have that $v.t = S$, there are $v.cs - 1$ sending events that precede v and $v.cr$ reception events that precede v. The nodes in V_P are totally ordered by the lexicographic ordering over v given by $(v.cs, v.cr)$. For a message m' sent from P to $\bar{P}$, there are vertices $v_s \in V_P$ and $v_r \in V_{\bar{P}}$, where $v_s.t = S$, $v_r.t = R$, $v_s.m = v_r.m = m'$, and $(v_s, v_r) \in E$.

Now that we have introduced the semantics of the data contained within the graph, we define graph operations associated with sending and receiving messages, and how they modify the data contained within the graph.

Graph initialization. An empty graph G consists of empty vertex sets V_0 and V_1 along with an empty edge set E. Each vertex set V_P has an associated send counter cs and reception counter cr. We use the notation $V_P.cs$ and $V_P.cr$ to refer to these counters for party P. Initially, $V_P.cs = V_P.cr = 0$ for $P \in \{0, 1\}$. The counters will be treated as auxiliary state as opposed to part of the graph.

We use the symbol ε to denote the empty graph.

Addition of send operation. We use the notation $G \leftarrow G + (\mathtt{S}, P, m)$ to denote the addition of a sending operation for message m' from party P. First, we increment $V_P.cs$ ($V_P.cs \leftarrow V_P.cs + 1$), then create a new vertex v with $v.t = S$, $v.cs = V_P.cs$, $v.cr = V_P.cr$, and $v.m = m'$. Finally, we add v to V_P. When updating a non-message-inclusive graph, we simply write $G \leftarrow G + (\mathtt{S}, P)$.

Addition of reception operation. The addition of a reception for the message with sending index i from party $\bar{P}$ is denoted by $G \leftarrow G + (\mathtt{R}, P, i)$. Note that this operation enforces that such a message with sending index i exists in $G.V_{\bar{P}}$. First, we increment $V_P.cr$, then create a new vertex v with $v.t = R$, $v.cs = V_P.cs$, $v.cr = V_P.cr$, $v.m = v_s.m$, where $v_s \in V_{\bar{P}}$, $v_s.t = S$, $v_s.cs = i$ (by construction, there is only one such vertex). We add v to V_P and add (v_s, v) to E.

Graph validity and consistency. A causality graph G is valid if there exists a sequence of send and reception operations that lead to its construction from an empty graph. A sub-graph G' is valid if there exists a valid graph G such that $G' \subseteq G$. Two graphs G and G' are consistent if there exists a valid graph G^* such that $G \subseteq G^*$ and $G' \subseteq G^*$. We write $G \approx G'$ to indicate that G and G' are consistent, and we use $G \not\approx G'$ to indicate that they are inconsistent. Notions of validity and consistency will be important in our security definitions.

Partial ordering over events. As we mentioned before, there is a total ordering over the events within V_0 and V_1. For $v_1, v_2 \in V_P$, we have that $v_1 < v_2$ if $v_1.cs \leq v_2.cs$, $v_1.cr \leq v_2.cr$, and $(v_1.cs, v_1.cr) \neq (v_2.cs, v_2.cr)$. The causality partial ordering $v_1 \leq v_2$ for $v_1, v_2 \in V$ is defined as the transitive closure of these individual total orders along with the edge relation (i.e., $v_1' \leq_E v_2'$ if $(v_1', v_2') \in E$).

Observe that this is an ordering over the sending and reception of messages as opposed to an ordering of the messages themselves.

Contiguity and gaps. When a moderator views a sub-graph of a full conversation, it will be useful to understand which events are contiguous (in the sense that no other events can be ordered between them) and which events have gaps. This makes clear where there might be missing context, which can later be provided by either party if requested. The inclusion of sending and reception counters within the graph provides rich information that allows interpretation of the contiguity of events. Consider two vertices $v, v' \in V_P$ where $v < v'$. If we have that $\max(v'.cs - v.cs, v'.cr - v.cr) = 1$, then by construction there can be no $v'' \in V_P$ such that $v < v'' < v'$, so v and v' are contiguous with one another, among the vertices in V_P. Taking into consideration the edges in E, we can ascertain whether any nodes in $V_{\bar{P}}$ can be ordered between them. If we have that $(v'.cs - v.cs, v'.cr - v.cr) = (a, b)$ for $v, v' \in V_P$, we know that there are a send events and b reception events that occurred after v, including v'. In this way, these counters provide rich interpretability of gaps within sub-graphs.

3.3 Two-Party Transcript Franking

Prior work on message franking has focused on reporting single messages that were received by the reporting user. Often, single messages do not contain sufficient context for understanding online harassment. Hence, we aim to extend message franking to enable reporting sequences of messages. We propose a new cryptographic primitive we call transcript franking that captures this goal. In addition to providing integrity over the contents of messages contained within

a sequence, a transcript franking scheme must have cryptographic assurance over the ordering and contiguity of messages reported within a sequence. Due to the inherent asynchrony of messaging, honest users may observe differing but valid views of the message order. We aim for users and platform moderators to be able to see the view of the transcript from the perspective of each party as well the causal dependencies between messages.

Limitations of current approaches. The original Facebook white-paper that introduced message franking suggests including server timestamps within the data tagged by the server [1]. As prior work points out, this is insufficient for capturing causal dependencies between messages since it does not directly reflect client-side views and capture message concurrency [44]. Furthermore, timestamps do not assure integrity over the contiguity of reported messages within a conversation. Recent work suggests incorporating causality metadata into message franking channels, however since this metadata is client-generated, it can deviate from the actual ordering of sending and reception events that clients experience [44]. As we explain in the introduction, this leads to attacks in which a malicious party can force the other party to view an upsetting message sequence while only being able to report a plausibly benign one.

A further limitation is that all prior treatments of message franking allow reporters to disclose only messages they have received from the other party. From a motivational standpoint, this seems reasonable since the goal of reporting is to demonstrate that an abusive party, presumably the non-reporting party, sent harmful content. However, when reporting multiple messages within a conversation, a reporter may have to include their own sent messages to provide sufficient context. In Figure 3.1, for instance, Alice should be able to show her

message that precedes the message Bob sends in order for the moderator to evaluate whether it is abusive. The obvious solution of a reporter disclosing their own sent messages is insufficient since the corresponding ciphertexts may be malformed for the other party, and existing treatments of message franking provide no way for recipients to indicate this to the service provider. An alternative would be to involve both parties in the reporting process. Yet this can lead to a denial of service attack in which the accused party refuses to participate. Even with an honest accused party, requiring both parties to be online is an unfavorable limitation. Hence, we must devise a reporting protocol that is not contingent upon the participation of the non-reporting party. Note that such a protocol may still allow the non-reporting party to continue to disclose more context if they decide to do so.

Our suggested platform model. In line with the model used by the messaging layer security (MLS) standard [28], we conceptualize the platform as providing a delivery service (DS) for message transmission and an authentication service (AS) for managing user identities. Additionally, we consider a platform-managed moderation service (MS) for accepting user reports and taking action against abusive user accounts. Clients issue a `Send` request with a payload containing the message ciphertext to the DS. To receive messages, a client issues a `Recv` request, to which the server responds with outstanding message ciphertexts. In a real platform, clients may be identified via usernames or phone numbers. Our transcript franking abstraction, on the other hand, simply identifies parties within a conversation via the numeric labels $\{0, 1\}$ for direct messaging.

Our model assumes a client receives a notification that indicates when the DS has successfully received their message and another notification when the

recipient client device has successfully received the message. These two events correspond to the first and second checkmarks shown in widely used messaging platforms such as Whatsapp and Signal [10, 12]. Messages may still be dropped or reordered, but we will concern ourselves with reporting only messages that were successfully received. We remark that this differs slightly from traditional platform models for message franking, in which the platform simply tags a message once it is sent and shares this tag with just the recipient. It turns out this model will be crucial to achieving our transcript franking security goals.

Transcript franking syntax and semantics. We draw heavily on [44] as inspiration for our syntax and will later provide a comprehensive comparison between their approach to incorporating causality into message franking and our notion of transcript franking. A transcript franking scheme is a tuple of algorithms $\mathsf{TF} = (\mathsf{SrvInit}, \mathsf{Init}, \mathsf{Snd}, \mathsf{Rcv}, \mathsf{TagSend}, \mathsf{TagRecv}, \mathsf{Judge})$, defined over a server state space $\mathcal{S}_S$, a client state space $\mathcal{S}_C$, a key space $\mathcal{K}$, a message space $\mathcal{M}$, a commitment space $\mathcal{Q}$, a franking key space $\mathcal{K}_f$, a message-sent tag space $\mathcal{T}_S$, and a reception tag space $\mathcal{T}_R$. We detail each algorithm below. Input and output variables names are unique, and we indicate the "type" (the set of possible values) and description of a variable only the first time it is introduced in order to reduce verbosity. In general, st will refer to client or server state the before the invocation of a routine, while st' refers to the updated state after the invocation.

- $st_S \leftarrow\!\!\$\ \mathsf{SrvInit}()$ outputs an initial server state $st_S \in \mathcal{S}_S$.
- $st \leftarrow\!\!\$\ \mathsf{Init}(P, k)$ outputs initial client state $st \in \mathcal{S}_C$ for party $P \in \{0, 1\}$ and a secret shared channel key $k \in \mathcal{K}$.
- $st', c \leftarrow\!\!\$\ \mathsf{Snd}(P, st, m)$ is a client procedure that produces a ciphertext $c \in \mathcal{C}$ and updated client state $st' \in \mathcal{S}_C$ for party P, with original client state

$st \in \mathcal{S}_C$, corresponding to an input message $m \in \mathcal{M}$. The ciphertext c contains a franking tag $c_f \in \mathcal{Q}$, which is a commitment to m, and a sending index $i \in \mathbb{Z}^*$.

- $st'_S, t_s \leftarrow \mathsf{TagSend}(st_S, P, c_f)$ is a server procedure that produces a tag $t_s \in \mathcal{T}_S$ for a message sending event, where P is the sending party, and c_f is the franking tag for the message.
- $st_S, t_r \leftarrow \mathsf{TagRecv}(st_S, P, c_f)$ is a server procedure that produces a tag $t_r \in \mathcal{T}_R$ for a message reception event by receiving party P. This procedure is invoked only when the receiving client indicates that the message was successfully received and valid.
- $st', m, k_f, i \leftarrow \mathsf{Rcv}(P, st, c)$ is a client procedure that processes a received ciphertext $c \in \mathcal{C}$ and decrypts it to a message $m \in \mathcal{M} \cup \{\perp\}$ with sending index $i \in \mathbb{Z}^*$ and a franking key $k_f \in \mathcal{K}_f$. The message m is $\perp$ if and only if decryption fails.
- $G \leftarrow \mathsf{Judge}(st_S, \rho)$ takes as input the server state st_S as well as a client-provided report $\rho \in \{(\{0,1\} \times \mathcal{M} \times \mathcal{K}_f \times \mathcal{Q} \times \mathcal{T}_S \times \mathcal{T}_R)\}$, which is a set of tuples of the form $(P, m, k_f, c_f, t_s, t_r)$. This procedure verifies the report and, if the report is valid, produces a causality graph $G \in (\mathcal{V} \times \mathcal{E}) \cup \{\perp\}$ for the messages contained within the report. If the reporting information is invalid, the procedure outputs $\perp$. Note, we enforce here that only messages that have been sent and received can be reported.

To illustrate the semantics of a transcript franking scheme, we provide a brief example of how these procedures are invoked. At initialization time, the server runs SrvInit to produce an initial state. When two clients initiate a conversation, both clients run $\mathsf{Init}(P, k)$ over a shared key k to obtain initial client states. When

client P sends a message m, it obtains c from the output of $\mathsf{Snd}(P, st, m)$. The ciphertext c is sent to a platform server, which then invokes $\mathsf{TagSend}(st_S, P, c_f)$ to produce a tag t_s, which is returned to P. Eventually, $\bar{P}$ contacts the server to retrieve outstanding messages. Upon doing so, the server transmits the ciphertext c, along with t_s, to $\bar{P}$, which decrypts it by invoking Rcv. Upon indicating valid reception to the platform, the server runs $\mathsf{TagRecv}$ to produce t_r, which is sent to both P and $\bar{P}$. When party P wishes to report a set of messages, they compile $(P, m, k_f, c_f, t_s, t_r)$ for each message in ρ. The moderator then runs Judge to produce a causality graph G.

Correctness. Informally, correctness requires that all honestly generated and tagged messages can be successfully received and that any sub-graph of the full causality graph can be reported. We make this precise with the game shown in Figure 3.3, and define the correctness advantage of an adversary $\mathcal{A}$ as

$$\mathrm{Adv}^{\mathrm{corr}}_{\mathsf{TF}}(\mathcal{A}) = \Pr[\mathbf{G}^{\mathrm{corr}}_{\mathsf{TF}}(\mathcal{A}) = 1] \ .$$

Formally, a transcript franking scheme TF achieves correctness if $\mathrm{Adv}^{\mathrm{corr}}_{\mathsf{TF}}(\mathcal{A}) = 0$ for all adversaries $\mathcal{A}$.

Tagging reception events. We discuss in detail what it means for the platform to tag a successful reception event. Recall that our goal is to enable reporters to include their own sent messages within a report without interaction from the other party. The reception tag serves as a way to achieve this goal, acting as an acknowledgement from the other party that the reported message was received. However, in order for this acknowledgement to be meaningful, we must carefully consider at which point the platform generates the reception tag. One option is to generate the tag once the recipient sends a message to the service provider indicating that their verification check passed, meaning the franking

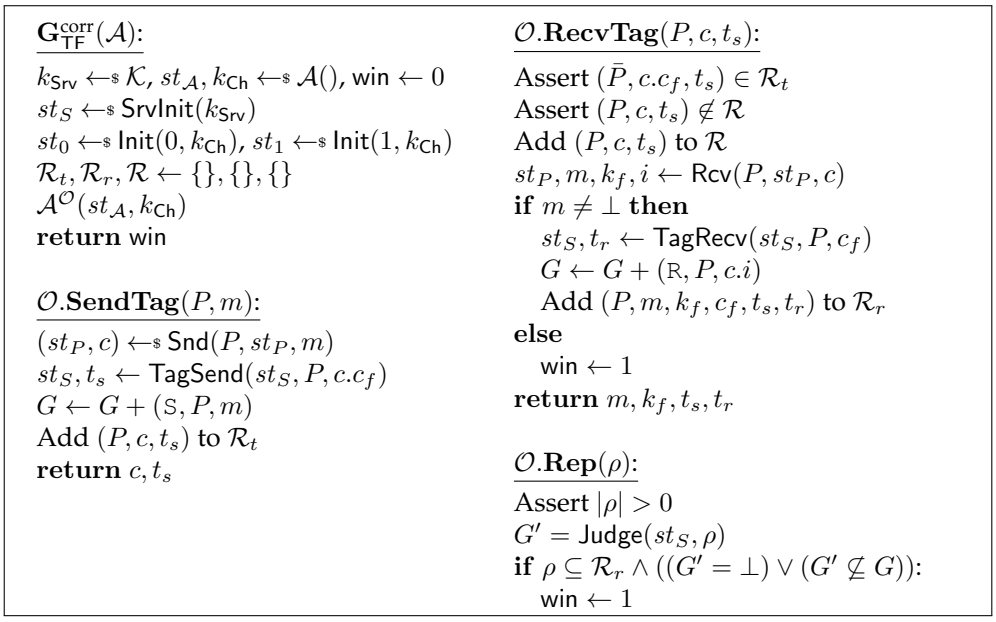

Figure 3.3: The security game for transcript franking correctness.

tag corresponds to the received plaintext. This would require two round-trips, the first for to retrieve the message, and the second to explicitly tell the server that it was well-formed.

Another option is to do this in one round trip: immediately tag the reception event and send the reception tag along with the ciphertext. If the ciphertext is malformed, the recipient can issue a complaint to the service provider, nullifying the reception tag in question. Therefore, two round trips are made only if the ciphertext is malformed. An implementation may also enforce a reasonable time window within which to make such a complaint. We discuss receiver acknowledgement further in Section 4.6.

Comparison to Chen-Fischlin. Observe that our formalism, unlike that of [44] includes two server-side tagging procedures as opposed to one. This makes possible acknowledgement of message receipt by the platform and message delivery to the recipient client device. As a result, the server, as opposed to client

devices, becomes the authority on message ordering, leading to additional security benefits as we discuss next. Instead of having a single Init procedure shared by the client and server, we specify SrvInit, and Init. The syntax and semantics of the message franking channel in Chen-Fischlin does not enable parties to report their own sent messages while our formalism does. While Chen and Fischlin focus on two-party channels, we show how to enable transcript franking for group channels in Section 3.6. We provide a more in-depth comparison in Appendix 3.10.

3.4 Security Definitions for Two-Party Transcript Franking

In this section, we introduce security notions for transcript franking. Our setting requires that the platform is the same entity that handles moderation reports. Recall that we defined the transcript franking syntax and semantics in Section 3.3. Our security definitions formalize notions of confidentiality and accountability for the reporting process. Accountability consists of two properties, reportability and transcript integrity, which we further explain in the remainder of the section.

Threat model, informally. As the platform is trusted for handling user reports, we trust it to serve as source of ground truth for the ordering of messages. This does not mean that we trust the platform with the contents of messages or that we assume a malicious platform will not attempt to maul ciphertexts. We do not explicitly model the public key infrastructure used to authenticate users, though we note that solutions such as key transparency [43, 80, 88, 95, 112] enable PKIs

without placing full trust in the service provider.

Transcript reportability. When a client accepts a message as valid, it should be the case that this message can be successfully reported to the moderator as well. Transcript reportability, which is formally specified by a security game in Figure 3.4, is a security property we define that captures this goal. The adversary $\mathcal{A}$ attempts to craft a report, containing messages accepted by an honest recipient, that does not verify for the moderator. We define the reportability advantage of an adversary $\mathcal{A}$ for a transcript franking scheme TF as follows:

$$\mathrm{Adv}^{\text{tr-rep}}_{\mathsf{TF}}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{tr-rep}}_{\mathsf{TF}}(\mathcal{A}) = 1] \ .$$

Transcript integrity. To model malicious reporters that attempt to trick moderators into accepting incorrect causality graphs, we define a security notion called transcript integrity, which is captured by the game in Figure 3.5. The adversary $\mathcal{A}$ controls both parties and has access to three oracles: $\mathbf{SendTag}$, $\mathbf{RecvTag}$ and $\mathbf{Rep}$. A ground truth causality graph G is maintained by the game and updated by $\mathbf{SendTag}$ and $\mathbf{RecvTag}$. The adversary wins if it can produce two valid reports, where at least one is not a sub-graph of the ground truth causality graph, or where the generated sub-graphs are inconsistent.

To elaborate, we recall some definitions regarding causality graphs that were given in Section 3.2. First, recall that $\tilde{G}$ refers to the graph with message labels removed, allowing us to compare with the ground-truth causality graph G maintained by the game. Second, two causal sub-graphs are consistent if there exists a valid causality graph G' of which they are both sub-graphs. This final consistency condition means that there is a unique ground truth causality graph from which sub-graphs can be reported. We view this as a natural lifting of the

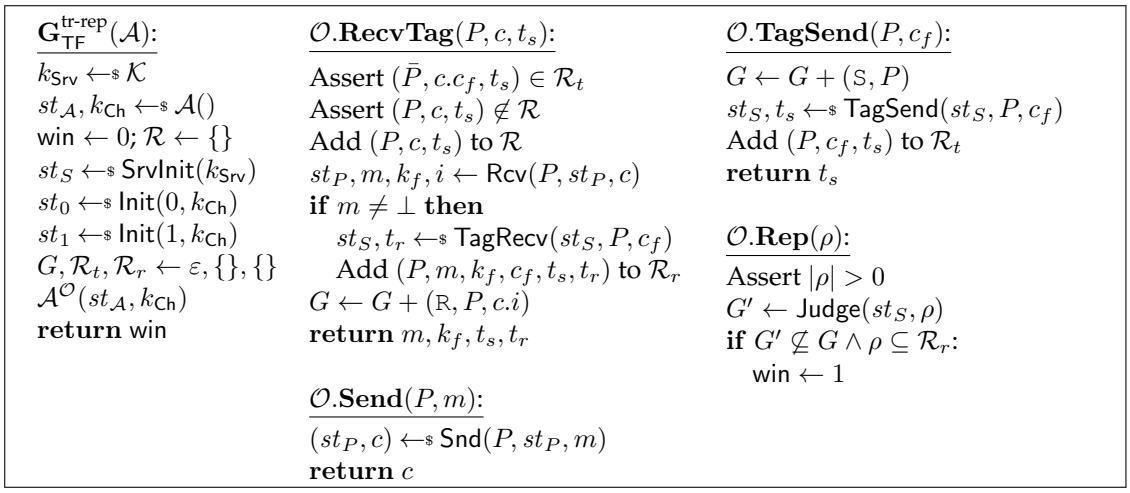

Figure 3.4: The security game for transcript reportability.

receiver binding notion proposed in [63] to the multi-message setting. The advantage of an adversary $\mathcal{A}$ in the transcript integrity game is defined as follows:

$$\mathrm{Adv}^{\text{tr-int}}_{\mathsf{TF}}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{tr-int}}_{\mathsf{TF}}(\mathcal{A}) = 1] \ .$$

Confidentiality. In order for a transcript franking scheme to achieve confidentiality, the reporting process must not reveal any information about unreported messages. Of course, the underlying messaging channel itself must provide confidentiality as well. We formalize this property in a security game in Figure 3.6, inspired by the real-or-random multi-opening confidentiality notion proposed in [63]. Our definition uses the function $\mathsf{clen} : \mathcal{M} \rightarrow \mathbb{Z}^*$, which outputs the length of a ciphertext for plaintext m. The ROR-advantage against the confidentiality of a transcript franking scheme TF for an adversary $\mathcal{A}$ is:

$$\mathrm{Adv}^{\text{conf}}_{\mathsf{TF}}(\mathcal{A}) = |\Pr[\mathbf{G}^{\text{conf}}_{\mathsf{TF},0}(\mathcal{A})] - \Pr[\mathbf{G}^{\text{conf}}_{\mathsf{TF},1}(\mathcal{A})]| \ .$$

$\underline{\mathbf{G}^{\text{tr-int}}_{\mathsf{TF}}(\mathcal{A})}$:
$k_{\mathsf{Srv}} \leftarrow\!\!\$\ \mathcal{K}$; win $\leftarrow 0$
$st_{\mathcal{A}}, k_{\mathsf{Ch}} \leftarrow\!\!\$\ \mathcal{A}()$
$st_S \leftarrow\!\!\$\ \mathsf{SrvInit}(k_{\mathsf{Srv}})$
$st_0 \leftarrow\!\!\$\ \mathsf{Init}(0, k_{\mathsf{Ch}})$
$st_1 \leftarrow\!\!\$\ \mathsf{Init}(1, k_{\mathsf{Ch}})$
$G, \mathcal{R}_t, \mathcal{R} \leftarrow \varepsilon, \{\}, \{\}$
$\mathcal{A}^{\mathcal{O}}(st_{\mathcal{A}}, k_{\mathsf{Ch}})$
return win

$\underline{\mathcal{O}.\mathbf{SendTag}(P, c)}$:
$G \leftarrow G + (\mathtt{S}, P)$
$st_S, t_s \leftarrow \mathsf{TagSend}(st_S, P, c.c_f)$
Add (P, c, t_s) to $\mathcal{R}_t$
return t_s

$\underline{\mathcal{O}.\mathbf{RecvTag}(P, c, t_s)}$:
Assert $(\bar{P}, c, t_s) \in \mathcal{R}_t$
Assert $(P, c, t_s) \notin \mathcal{R}$
Add (P, c, t_s) to $\mathcal{R}$
$st_S, t_r \leftarrow \mathsf{TagRecv}(st_S, P, c.c_f)$
$G \leftarrow G + (\mathtt{R}, P, c.i)$
return t_r

$\underline{\mathcal{O}.\mathbf{Rep}(\rho_1, \rho_2)}$:
Assert $|\rho_1| > 0$ and $|\rho_2| > 0$
$G_1 \leftarrow \mathsf{Judge}(st_S, \rho_1)$
$G_2 \leftarrow \mathsf{Judge}(st_S, \rho_2)$
if $G_1 \neq \perp \wedge G_2 \neq \perp \wedge$
$((G_1 \not\subseteq G) \vee (G_2 \not\subseteq G)$
$\vee (G_1 \not\approx G_2))$:
win $\leftarrow 1$

Figure 3.5: The security game for transcript integrity.

$\underline{\mathbf{G}^{\text{conf}}_{\mathsf{TF},b}(\mathcal{A})}$:
$k_{\mathsf{Srv}} \leftarrow\!\!\$\ \mathcal{K}_S$; $k_{\mathsf{Ch}} \leftarrow\!\!\$\ \mathcal{K}_C$
$st_{\mathcal{A}} \leftarrow\!\!\$\ \mathcal{A}()$, $\mathcal{Y} \leftarrow \{\}$
$st_S \leftarrow\!\!\$\ \mathsf{SrvInit}(k_{\mathsf{Srv}})$
$st_0 \leftarrow\!\!\$\ \mathsf{Init}(0, k_{\mathsf{Ch}})$, $st_1 \leftarrow\!\!\$\ \mathsf{Init}(1, k_{\mathsf{Ch}})$
$\hat{b} \leftarrow \mathcal{A}^{\mathcal{O}}(st_{\mathcal{A}})$
return $\hat{b}$

$\underline{\mathcal{O}.\mathbf{Send}(P, m)}$:
$(st_P, c) \leftarrow\!\!\$\ \mathsf{Snd}(P, st_P, m)$
$\mathcal{Y} \leftarrow \mathcal{Y} \cup \{c\}$
return c

$\underline{\mathcal{O}.\mathbf{Recv}(P, c, t_s)}$:
Assert $c \in \mathcal{Y}$
$st_P, m, k_f, i \leftarrow \mathsf{Rcv}(P, st_P, c)$
return m, k_f

$\underline{\mathcal{O}.\mathbf{ChalSend}(P, m)}$:
$(st_P, c_0) \leftarrow\!\!\$\ \mathsf{Snd}(P, st_P, m)$
$c_1 \leftarrow\!\!\$\ \{0,1\}^{\mathsf{clen}(m)}$
return c_b

Figure 3.6: The security game for transcript franking confidentiality.

3.5 Our Construction

The key idea of our construction is to report platform-tagged acknowledgements of message sending and reception. These acknowledgements contain counters, maintained as part of the server state, that allow the moderator to reliably reconstruct a portion of the causality graph corresponding to the platform's view of message transmission. Since our construction uses four counters per conversation, we call it the quad-counter construction (QCC). We present

the pseudocode specification of our construction in Figure 3.7, which specifies how the service provider handles state for a single conversation between two parties. Parallelizing this for multiple conversations can be done in a straightforward manner, as we further discuss in Section 4.6.

Client logic. The client procedures Init, Snd, and Rcv comprise a secure messaging channel with reportable franking tags c_f, committing to plaintext content m, with the opening k_f. Indeed, these three procedures form a message franking channel [44, 65].

Server logic. Upon server initialization, sending and reception counters for each party are initialized to 0 and the server samples a MAC key. When a party P sends a message, the server increments the send counter for P and tags the send event. Similarly, it increments the reception counter for P when P successfully receives a message, and then produces a tag for this event. In the pseudocode, the symbols S and R are labels that denote sending and reception events respectively.

Reporting. To report a set of messages, a client compiles the message contents m, the franking key k_f, the franking tag c_f, the send tag TagSend, and the reception tag t_r for each message. The client then forwards this information to the platform within a single report object ρ. The platform verifies the commitments for each message along with its own MAC tags. Then, it uses the indexes within the tags to construct and order the vertices for the sub-graph, and it adds edges between vertices that correspond to the same message. A moderator can interpret the contiguity of vertices as explained in Section 3.2.

Security proofs. We now demonstrate the security of our transcript frank-

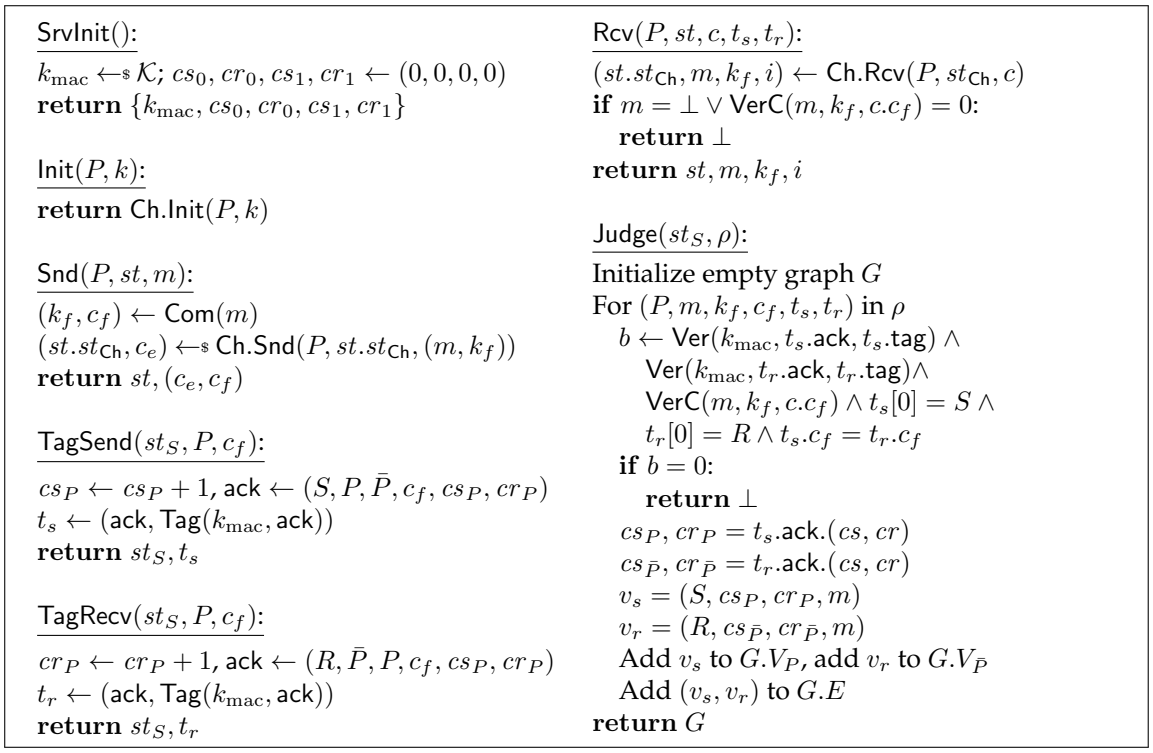

Figure 3.7: Pseudocode for our two-party transcript franking construction.

ing construction, TF. We begin by proving transcript integrity. The following lemma will be helpful for our proof.

Lemma 1. *Let G_1 and G_2 be two valid two-party causality sub-graphs. Suppose $\widetilde{G_1} \approx \widetilde{G_2}$ but $G_1 \not\approx G_2$. Then there must be $v_1 \in G_1.V$ and $v_2 \in G_2.V$ such that $v_1[0:2] = v_2[0:2]$ but $v_1.m \neq v_1.m$.*

Proof. Assume for the sake of contradiction that for all $v_1 \in G_1.V$ and $v_2 \in G_2.V$, that if $v_1[0:2] = v_2[0:2]$, then $v_1.m = v_2.m$. Since $\widetilde{G_1} \approx \widetilde{G_2}$, there exists some valid causality graph G such that $\widetilde{G_1}, \widetilde{G_2} \subseteq G$. This means that there exists a sequence of send and receive operations that constructs G. We can use the same sequence of operations to generate a valid message-inclusive graph G^*, such that $G_1, G_2 \subseteq G^*$, contradicting the assumption that $G_1 \not\approx G_2$. In each send operation, we simply include the message corresponding to the vertex $v \in$

$G_1.V \cup G_2.V$, if the counters for that send operation correspond to a vertex in the union of the two vertex sets. By our initial assumption a unique such vertex exists. For all other send operations, we can include an arbitrary message, the empty string for instance. This completes the proof.

□

Theorem 1. *Let* TF *be the transcript franking scheme given in Figure 3.7. Let* $\mathcal{A}$ *be a transcript integrity adversary against* TF*. Then we give an EUF-CMA adversary* $\mathcal{B}$ *and V-Bind adversary* $\mathcal{C}$ *such that*

$$\mathrm{Adv}^{\text{tr-int}}_{\mathsf{TF}}(\mathcal{A}) \leq \mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{B}) + \mathrm{Adv}^{\text{v-bind}}_{\mathsf{CS}}(\mathcal{C}) \ .$$

Adversaries $\mathcal{B}$ *and* $\mathcal{C}$ *run in time that of* $\mathcal{A}$ *plus a small overhead.*

Proof. We proceed via a sequence of game hops with Game $\mathbf{G}_0$ equivalent to $\mathbf{G}^{\text{tr-int}}_{\mathsf{TF}}$, defined in Figure 3.5. To aid with future game definitions, we provide some additional bookkeeping, initializing an empty set $\mathcal{R}'$ at the start of the game. Game $\mathbf{G}_0$ adds $(S, P, \bar{P}, c.c_f, G.(cs_P, cr_P))$ to $\mathcal{R}'$ before the **return** statement of **SendTag**. Similarly, it adds $(R, \bar{P}, P, c.c_f, G.(cs_P, cr_P))$ to $\mathcal{R}'$ before the **return** statement of **RecvTag**.

The adversary $\mathcal{A}$ can only win if it is able to produce ρ_1, ρ_2 such that $\widetilde{G_1} \not\subseteq G$ or $\widetilde{G_2} \not\subseteq G$ or $G_1 \not\approx G_2$, where $G_1 \leftarrow \mathsf{Judge}(st_S, \rho_1)$, $G_2 \leftarrow \mathsf{Judge}(st_S, \rho_2)$, and G is the ground truth graph maintained by the game. From (ρ_1, ρ_2), we will show that we can break either the unforgeability of the MAC or the binding property of the commitment.

We consider two cases: (1) the adversary wins with $\widetilde{G_1}$ or $\widetilde{G_2}$ not a subgraph of G or (2) the adversary wins with $G_1 \not\approx G_2$, but $\widetilde{G_1}, \widetilde{G_2} \subseteq G$. The

first case will allow us to reduce to the EUF-CMA security of the MAC while the second allows us to reduce to the binding security of the commitment. Each case will correspond to a distinct failure event. Let $\mathbf{G}_1$ be the same as $\mathbf{G}_0$, except we abort and output 0, right before setting the win flag, if $\mathcal{A}$ produces a valid (ρ_1, ρ_2) in case (1). We denote the event that this abort occurs as F_1. Let $\mathbf{G}_2$ be the same as $\mathbf{G}_1$ except we abort and output 0 at the same location if $\mathcal{A}$ produces a valid (ρ_1, ρ_2) in case (2). We denote the event that this abort occurs as F_2. Note that $|\Pr[\mathbf{G}_0(\mathcal{A}) \Rightarrow 1] - \Pr[\mathbf{G}_1(\mathcal{A}) \Rightarrow 1]| \leq \Pr[F_1]$ and $|\Pr[\mathbf{G}_1(\mathcal{A}) \Rightarrow 1] - \Pr[\mathbf{G}_2(\mathcal{A}) \Rightarrow 1]| \leq \Pr[F_2]$. Furthermore, $\Pr[\mathbf{G}_2(\mathcal{A}) \Rightarrow 1] = 0$, so we have $\mathrm{Adv}^{\text{tr-int}}(\mathcal{A}) = \Pr[\mathbf{G}_0(\mathcal{A}) \Rightarrow 1] \leq \Pr[F_1] + \Pr[F_2]$.

We now demonstrate an adversary $\mathcal{B}$ where $\mathrm{Adv}^{\text{euf-cma}}(\mathcal{B}) = \Pr[F_1]$. The adversary $\mathcal{B}$ perfectly simulates $\mathbf{G}_0$ to $\mathcal{A}$, while routing Tag and Verify calls to its challenger oracles. If F_1 occurs, then we have that $\widetilde{G_i} \not\subseteq G$ for some $i \in \{1, 2\}$. For r an element of ρ, define $f(r) = \{r.t_s.\mathsf{ack}, r.t_r.\mathsf{ack}\}$. Observe that $\widetilde{G_i} \not\subseteq G$ implies there is some $r^* = (P^*, m^*, k_f^*, c_f^*, t_s^*, t_r^*) \in \rho_i$, such that $f(r^*) \not\subseteq \mathcal{R}'$.

To see why this is, observe that the construction mirrors the updates of the causality graph perfectly. Put formally, if $\bigcup_{r' \in \rho} f(r') \subseteq \mathcal{R}'$ and $G' \leftarrow \mathsf{Judge}(\rho)$, then $\widetilde{G'} \subseteq G$. We have that $G.(cs_P, cr_P, cs_{\bar{P}}, cr_{\bar{P}}) = (0, 0, 0, 0)$ and the server counters $(cs_P, cr_P, cs_{\bar{P}}, cr_{\bar{P}}) = (0, 0, 0, 0)$ at the start of the game. When **SendTag** is called, we increment $G.cs_P$ and $st_S.cs_P$. Similarly, when **RecvTag** is invoked, we increment $G.cr_P$ and $st_S.cr_P$. A simple proof by induction on the number of oracle calls shows that $G.(cs_P, cr_P) = st_S.(cs_P, cr_P)$ for $P \in \{0, 1\}$ by the end of each call to **SendTag** and **RecvTag**. This means that $v \in \widetilde{G'}.V$ implies $v \in G.V$ and $e \in \widetilde{G'}.E$ implies $e \in G.E$.

If F_1 occurs, then we retrieve the $r^* = (P^*, m^*, k_f^*, c_f^*, t_s^*, t_r^*)$ in question, and

observe that $\mathsf{Verify}(k_{\mathrm{mac}}, t_s^*.\mathsf{ack}, t_s^*.\mathsf{tag}) = 1$ and $\mathsf{Verify}(k_{\mathrm{mac}}, t_r^*.\mathsf{ack}, t_r^*.\mathsf{tag}) = 1$, because the output of Judge is not $\perp$. We must have that at least one of $t_r^*.\mathsf{ack}$ or $t_s^*.\mathsf{ack}$ was not queried to the MAC challenger oracle, otherwise both would be in $\mathcal{R}'$. Let t^* denote this tag. We output $(t^*.\mathsf{ack}, t^*.\mathsf{tag})$ as a forgery.

Now, we construct adversary $\mathcal{C}$ where $\mathrm{Adv}_{\mathsf{CS}}^{\text{v-bind}}(\mathcal{C}) = \Pr[F_2]$. If F_2 occurs, we have that the adversary $\mathcal{A}$ was able to trigger $G_1 \not\approx G_2$ while $\widetilde{G_1}, \widetilde{G_2} \subseteq G$. By Lemma 1 there exists $v_1 \in G_1$ and $v_2 \in G_2$ such that $v_1[0:2] = v_2[0:2]$ but $v_1.m \neq v_2.m$. Note that there must also be a single c_f and $k_f^{(1)}, k_f^{(2)}$ such that $\mathsf{VerC}(v_1.m, k_f^{(1)}, c_f) = 1$ and $\mathsf{VerC}(v_2.m, k_f^{(2)}, c_f) = 1$. This breaks the binding property of the commitment, and so $\mathcal{C}$ outputs $(v_1.m, k_f^{(1)}, v_2.m, k_f^{(2)}, c_f)$ to win with probability $\Pr[F_2]$. This completes the proof. □

We now show that our scheme also achieves perfect reportability.

Theorem 2. *For all adversaries $\mathcal{A}$, we have* $\mathrm{Adv}_{\mathsf{TF}}^{\text{tr-rep}}(\mathcal{A}) = 0$.

Proof. Observe that the check that an honest recipient performs in Rcv, namely, that $\mathsf{VerC}(m, k_f, c.c_f) = 0$, is replicated in Judge. The only way Judge can return $\perp$ is if this check fails, if any of the MAC checks fail, or if the input ρ is malformed. Since we are dealing with an honest reporter, this cannot be the case, so Judge must always return a non-$\perp$ value.

□

Our scheme achieves correctness via the correctness of the underlying channel Ch, the correctness of the MAC scheme $(\mathsf{Tag}, \mathsf{Verify})$, the correctness of the commitment scheme $(\mathsf{Com}, \mathsf{VerC})$, and the fact that the counters in our construction perfectly mirror those of the ground truth graph (see the proof of Theo-

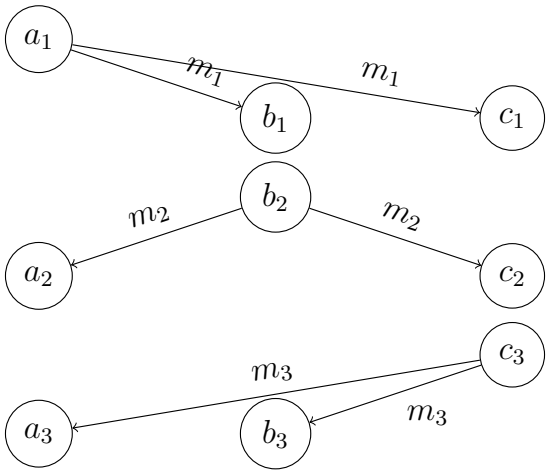

Figure 3.8: Example of group causality graph

rem 1). Observe that our construction boils down to a commit-then-encrypt scheme, which was proven secure for the multi-opening real-or-random confidentiality notion in [63], hence we omit the proof of confidentiality here.

3.6 Multi-party Transcript Franking

Up to this point, our constructions have considered transcript franking in the two-party direct messaging context. We now discuss how our approach generalizes to an arbitrary number of parties. A group consists of a set of N parties $\{0, \ldots, N-1\}$. The goal of a group transcript franking construction is to be able to reconstruct a causality graph like that shown in Figure 3.8. Note that, unlike the two-party setting, one send event can correspond to multiple reception events, as there are now multiple recipients. Each edge corresponds to a single copy of the broadcast message. We build on the multi-party channel communication graph formalism proposed in [91], adapting it to our causality graph abstraction.

Causality graphs for group messaging. For N-party communication, a causality graph is an N-partite graph $G = (V, E)$, where $V = \bigcup_{i \in [N]} V_i$. All vertex sets V_i for $i \in [N]$ are disjoint. The edge set E consists of pairs (v, v') where $v \in V_i$ and $v' \in V_j$, where $i, j \in [N]$ and $i \neq j$. The vertex space, as before, is $\{S, R\} \times \mathbb{Z}^* \times \mathbb{Z}^* \times \mathcal{M}$. The notation and updates for adding a send event is the same as the two party version. For adding a reception event to party P_R from party P_S, we write $G \leftarrow G + (\mathtt{R}, P_S, P_R, c.i)$, incrementing the reception counter for P_R before adding the vertex. We then add an edge between the sending vertex and the new reception vertex, as before. The partial ordering over events is given by the transitive closure over the total orders for each vertex set and the edge relation. The total order within each vertex set is given by the lexicographic ordering over $v.(cs, cr)$ for $v \in V_P$. Event contiguity and gaps can be interpreted similarly as the two-party case, as described in Section 3.2.

Group messaging channels. A group messaging channel Ch is defined as the tuple $\mathsf{Ch} = (\mathsf{Init}, \mathsf{Snd}, \mathsf{Rcv})$. The main difference here is that instead of $P \in \{0, 1\}$, we have that $P \in [N]$, and our reception procedure accepts the identity of the sending party P_S associated with the ciphertext c.

- $st \leftarrow\!\!\$\ \mathsf{Init}(P, k)$ outputs initial client state $st \in \mathcal{S}_C$ for a new channel for Party $P \in [N]$ and a key $k \in \mathcal{K}$.
- $st', c \leftarrow\!\!\$\ \mathsf{Snd}(P, st, m)$ is a client procedure that produces a ciphertext $c \in \mathcal{C}$ corresponding to an input message $m \in \mathcal{M}$, and an updated client state $st' \in \mathcal{S}_C$.
- $st', m, i \leftarrow \mathsf{Rcv}(P_R, st, P_S, c)$ is a client procedure that processes a received

ciphertext c from party $P_S \in [N]$ to party $P_S \in [N]$ (where $P_S \neq P_R$) and decrypts it to a message $m \in \mathcal{M} \cup \{\perp\}$ with sending index $i \in \mathbb{Z}^*$. The message m can be $\perp$ if decryption fails.

Group transcript franking syntax and semantics. The Init, Snd, and Rcv procedures inherit the syntactic changes discussed above, and our TagRecv procedure accepts an additional argument for the sending party $P_S \in [N]$. This additional argument is required in the group case since a client can receive a message from more than one party. We inherit notational conventions from Section 3.4.

- $st_S \leftarrow\!\!\$\ \mathsf{SrvInit}(N)$ outputs initial server state $st_S \in \mathcal{S}_S$ for an N-party group.
- $st \leftarrow\!\!\$\ \mathsf{Init}(P, k)$ outputs initial client state $st \in \mathcal{S}_C$ for a new channel for Party $P \in [N]$ and a key $k \in \mathcal{K}$.
- $st', c \leftarrow\!\!\$\ \mathsf{Snd}(P, st, m)$ is a client procedure that produces a ciphertext $c \in \mathcal{C}$ corresponding to an input message $m \in \mathcal{M}$, and an updated client state $st' \in \mathcal{S}_C$.
- $st'_S, t_s \leftarrow \mathsf{TagSend}(st_S, P_S, c_f)$ is a server procedure that produces a tag $t_s \in \mathcal{T}_S$ for a message sending event, where $P_S \in [N]$ is the sending party, $c_f \in \mathcal{Q}$ is the franking tag for the message, and $st'_S \in \mathcal{S}_S$ is the updated server state.
- $st_S, t_r \leftarrow \mathsf{TagRecv}(st_S, P_S, P_R, c_f)$ is a server procedure that produces a tag $t_r \in \mathcal{T}_R$ for a message reception event by receiving party $P_R \in [N]$ for a message sent by party $P_S \in [N]$. This procedure is invoked only when the receiving client indicates that the message was successfully received and valid.

- $st', m, k_f, i \leftarrow \mathsf{Rcv}(P_R, st, P_S, c)$ is a client procedure that processes a received ciphertext $c \in \mathcal{C}$ and decrypts it to a message $m \in \mathcal{M} \cup \{\perp\}$ and a franking key $k_f \in \mathcal{K}_f$. The message m can be $\perp$ if decryption fails.
- $G \leftarrow \mathsf{Judge}(st_S, \rho)$ takes as input the server state $st_S \in \mathcal{S}_S$ as well as a client-provided report ρ, which is a set of tuples of the form $(P_S, P_R, m, k_f, c_f, t_s, t_r)$. This procedure verifies the report and, if the report is valid, produces a causality graph $G \in (\mathcal{V} \times \mathcal{E}) \cup \{\perp\}$ for the messages contained within the report. If the reporting information is invalid, the procedure outputs $\perp$.

Security definitions. Our security notions in the group setting are a natural extension of those for the two-party setting. The main difference is that N parties are initialized and any of these parties can send and receive messages within the same channel. Correctness and confidentiality definitions generalize in a straightforward manner, hence we omit full descriptions of them for brevity. We present our group transcript reportability definition in Figure 3.9 and our group transcript integrity definition in Figure 3.10. To denote the advantage of an adversary $\mathcal{A}$ in the N-party reportability game, we write $\mathrm{Adv}^{\text{tr-rep}}_{\mathsf{TF},N}(\mathcal{A})$. Similarly, $\mathrm{Adv}^{\text{tr-int}}_{\mathsf{TF},N}(\mathcal{A})$ is the advantage of $\mathcal{A}$ in the N-party transcript integrity game.

Our construction. The group messaging transcript franking construction generalizes naturally from the two-party version. We provide a pseudocode specification of our group transcript franking protocol in Figure 3.11. Counter updates happen nearly identically in TagSend and TagRecv, except now N pairs of counters are maintained, one for each party.

Security analysis. The security analysis of our group construction closely mir-

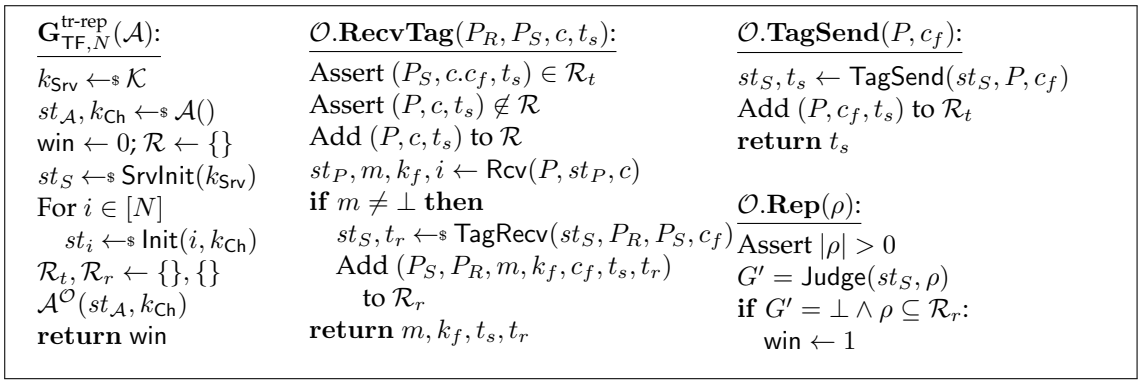

Figure 3.9: The security game for transcript reportability for N-party messaging.

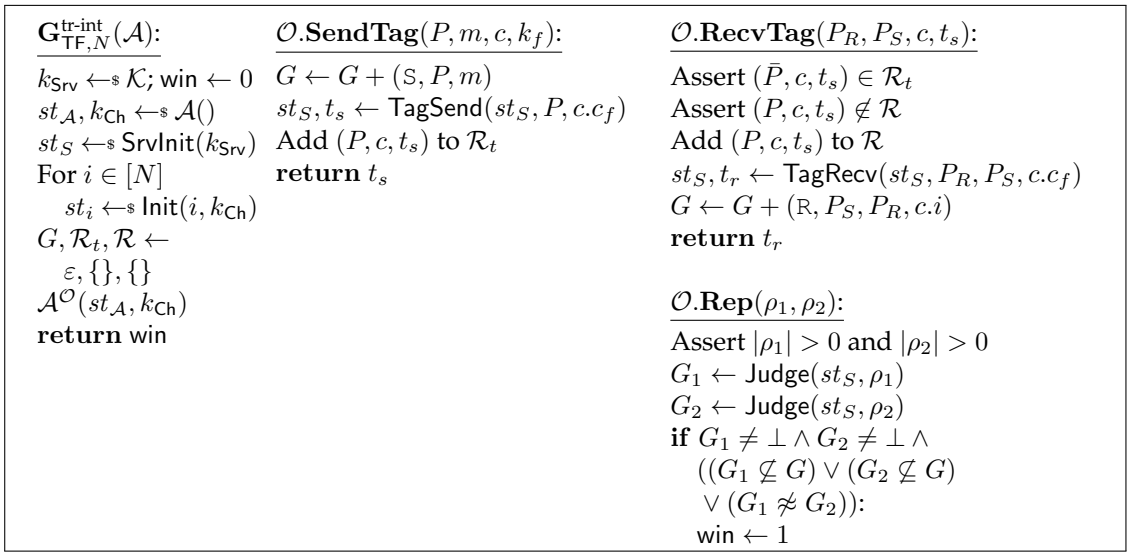

Figure 3.10: The security game for group transcript integrity for N-party messaging.

rors that of the two-party construction. As with the two-party construction, our group construction achieves perfect reportability because the Rcv procedure performs the same commitment checks as the Judge procedure. Below, we prove the transcript integrity of our scheme.

Theorem 3. *Let* TF *be the group transcript franking scheme given in Figure 3.11. Let* $\mathcal{A}$ *be an* N*-party group transcript integrity adversary against* TF*. Then we give an*

$\underline{\mathsf{SrvInit}(N)}$:
$k_{\mathrm{mac}} \leftarrow\!\!\$\ \mathcal{K}$
For $i \in [N]$ **do** $cs_i, cr_i \leftarrow 0, 0$
return $\{k_{\mathrm{mac}}\} \cup \{cs_i, cr_i\}_{i\in[N]}$

$\underline{\mathsf{Init}(P, k)}$:
return $\mathsf{Ch.Init}(P, k)$

$\underline{\mathsf{Snd}(P, st, m)}$:
$(k_f, c_f) \leftarrow \mathsf{Com}(m)$
$(st.st_{\mathsf{Ch}}, c_e) \leftarrow\!\!\$\ \mathsf{Ch.Snd}(P, st.st_{\mathsf{Ch}}, (m, k_f))$
return $st, (c_e, c_f)$

$\underline{\mathsf{TagSend}(st_S, P, c_f)}$:
$cs_P \leftarrow cs_P + 1$, $\mathsf{ack} \leftarrow (S, P, c_f, cs_P, cr_P)$
$t_s \leftarrow (\mathsf{ack}, \mathsf{Tag}(k_{\mathrm{mac}}, \mathsf{ack}))$
return st_S, t_s

$\underline{\mathsf{TagRecv}(st_S, P_R, P_S, c_f)}$:
$cr_{P_R} \leftarrow cr_{P_R} + 1$
$\mathsf{ack} \leftarrow (R, P_S, P_R, c_f, cs_{P_R}, cr_{P_R})$
$t_r \leftarrow (\mathsf{ack}, \mathsf{Tag}(k_{\mathrm{mac}}, \mathsf{ack}))$
return st_S, t_r

$\underline{\mathsf{Rcv}(P_R, st, P_S, c)}$:
$(st.st_{\mathsf{Ch}}, m, k_f, i) \leftarrow \mathsf{Ch.Rcv}(P_R, st_{\mathsf{Ch}}, P_S, c)$
if $m = \perp \vee \mathsf{VerC}(m, k_f, c.c_f) = 0$:
  return $\perp$
return st, m, k_f, i

$\underline{\mathsf{Judge}(st_S, \rho)}$:
Initialize empty graph G
For $(P_S, P_R, m, k_f, c_f, t_s, t_r)$ in ρ:
  $b \leftarrow \mathsf{Ver}(k_{\mathrm{mac}}, t_s.\mathsf{ack}, t_s.\mathsf{tag}) \wedge$
    $\mathsf{Ver}(k_{\mathrm{mac}}, t_r.\mathsf{ack}, t_r.\mathsf{tag}) \wedge$
    $\mathsf{VerC}(m, k_f, c.c_f) \wedge$
    $t_s[0] = S \wedge t_r[0] = R \wedge$
    $t_s.c_f = t_r.c_f$
  if $b = 0$:
    return $\perp$
  $cs_P, cr_P = t_s.\mathsf{ack}.(cs, cr)$
  $cs_{P_R}, cr_{P_R} = t_r.\mathsf{ack}.(cs, cr)$
  $v_s = (S, cs_P, cr_P, m)$
  $v_r = (R, cs_{P_R}, cr_{P_R}, m)$
  if $v_s \notin G.V_P$
    Add v_s to $G.V_P$
  Add v_r to $G.V_{P_R}$, add (v_s, v_r) to $G.E$
return G

Figure 3.11: Pseudocode for our N-party transcript franking construction.

EUF-CMA adversary $\mathcal{B}$ and V-Bind adversary $\mathcal{C}$ such that

$$\mathrm{Adv}^{\text{tr-int}}_{\mathsf{TF},N}(\mathcal{A}) \leq \mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{B}) + \mathrm{Adv}^{\text{v-bind}}_{\mathsf{CS}}(\mathcal{C}) \ .$$

Adversaries $\mathcal{B}$ and $\mathcal{C}$ run in time that of $\mathcal{A}$ plus a small overhead.

Proof. We proceed as with the proof of the two-party construction. The games $\mathbf{G}_0$, $\mathbf{G}_1$, and $\mathbf{G}_2$ are defined as before, with a similar security argument. Here, we highlight notable differences in the group case. The additional bookkeeping $\mathcal{R}'$ is initialized to $\{\}$, as before, at the start of the game. Game $\mathbf{G}_0$ adds $(S, P_S, P_R, c.c_f, G.(cs_P, cr_P))$ to $\mathcal{R}'$ before the **return** statement of **SendTag**. Similarly, it adds $(R, P_S, c.c_f, G.(cs_P, cr_P))$ to $\mathcal{R}'$ before the **return** statement of **RecvTag**. The failure events F_1 and F_2 are defined as in the proof of Theorem 1.

We now demonstrate an adversary $\mathcal{B}$ where $\mathrm{Adv}^{\text{euf-cma}}(\mathcal{B}) = \Pr[F_1]$. The adversary $\mathcal{B}$ perfectly simulates $\mathbf{G}_0$ to $\mathcal{A}$, while routing Tag and Verify calls to its challenger oracles. If F_1 occurs, then we have that $\widetilde{G_i} \not\subseteq G$ for some $i \in \{1, 2\}$. For r an element of ρ, define $f(r) = \{r.t_s.\mathsf{ack}, r.t_r.\mathsf{ack}\}$. Observe that $\widetilde{G_i} \not\subseteq G$ implies there is some $r^* = (P_S^*, P_R^*, m^*, k_f^*, c_f^*, t_s^*, t_r^*) \in \rho_i$, such that $f(r^*) \not\subseteq \mathcal{R}'$.

This results from the fact that $\bigcup_{r' \in \rho} f(r') \subseteq \mathcal{R}'$ and $G' \leftarrow \mathsf{Judge}(\rho)$ implies $\widetilde{G'} \subseteq G$. We have that $G.(cs_P, cr_P) = (0, 0)$ for $P \in [N]$ and the server counters $st_S.(cs_P, cr_P) = (0, 0)$ for $P \in [N]$ at the start of the game. When **SendTag** is called, we increment $G.cs_P$ and $st_S.cs_P$. Similarly, when **RecvTag** is invoked, we increment $G.cr_P$ and $st_S.cr_P$. This means that $v \in \widetilde{G'}.V$ implies $v \in G.V$ and $e \in \widetilde{G'}.E$ implies $e \in G.E$. This allows us to produce a forgery as shown in the proof for the two-party case.

Now, we construct adversary $\mathcal{C}$ where $\mathrm{Adv}^{\text{v-bind}}_{\mathsf{CS}}(\mathcal{C}) = \Pr[F_2]$. If F_2 occurs, we have that the adversary $\mathcal{A}$ was able to trigger $G_1 \not\approx G_2$ while $\widetilde{G_1}, \widetilde{G_2} \subseteq G$. It is straightforward to see that the group version of Lemma 1 holds, so there exists $v_1 \in G_1$ and $v_2 \in G_2$ such that $v_1[0:2] = v_2[0:2]$ but $v_1.m \neq v_2.m$. We proceed as in the proof of Theorem 1 to produce a binding violation. □

3.7 Discussion and Extensions

Composing causality preservation with transcript franking. Our work is primarily concerned with how malicious parties can interfere with the reporting process. Causality preservation as proposed in [44] aims to model how parties in a messaging channel can obtain consistent causality graphs in the presence of a malicious service provider. Indeed, transcript franking can be instantiated

with a channel that achieves strong causality preservation to reap these benefits.

Reports with redacted messages. We remark that our construction allows clients to report transmission patterns of messages, via causality sub-graphs, without having to disclose every message within the causality sub-graph. Doing so simply requires omitting the opening key k_f for the messages that a client wishes to redact within a report. In Chen and Fischlin's construction of a message franking channel with causality, this would not be possible as the causality metadata is committed to alongside the plaintext message [44].

Malicious clients refusing acknowledgement. In our protocol, clients indicate reception of a well-formed ciphertext to the server before a reception event tag is created. Malicious clients may refuse to make this acknowledgement to the server, preventing the sender of that message from being able to report it. We can mitigate this via notifying senders of message delivery and recipient validation separately, thereby flagging malicious behavior in-band. Upon detecting this behavior, a client may refuse to further communicate with the misbehaving client, but we do not yet support cryptographically reporting this misbehavior.

To explain in more detail, when P sends a message to $\bar{P}$, there are three key events in the course of message transmission: (1) the reception of the message sent from P by the platform server, (2) the reception of the message by $\bar{P}$, and (3) platform reception of a valid message acknowledgement from $\bar{P}$.

If $\bar{P}$ is malicious, it could refuse to indicate the validity of the message, omitting step (3) as described above. When (an honest) P notices that only events (1) and (2) occurred for a particular message, while $\bar{P}$ continues to send and acknowledge subsequent messages, it knows that $\bar{P}$ is acting in an aberrant man-

ner and will halt interactions with $\bar{P}$ (tear down the conversation and alert the user). So detection of deviation is built into our protocol. But our protocol does not enable P to cryptographically prove to a moderator that $\bar{P}$ misbehaved in the way described above. A messaging system might trust client software to report such misbehavior, but cryptographically secure reporting of this class of misbehavior is an open question.

The above issue is analogous to an honest client receiving a message with a malformed franking tag in the standard single-message franking setting. Here the recipient should drop the message, but cannot cryptographically prove to the moderator that the sender sent a malformed ciphertext.

One might wonder if the key-committing aspect of the encryption scheme can come to the rescue here: if a ciphertext can be decrypted only under one key, then the sender needs to simply reveal the key in order to show the ciphertext is well-formed. In messaging protocols, these keys are often intended for one-time use and revealing them does not compromise forward-secrecy or post compromise security. The issue is that there is no straightforward way to prove that the recipient should have been able to derive a particular key from the ratcheting protocol. Hence, proving that a message is decryptable is not sufficient to show that the recipient should have been able to decrypt it. Designing protocols that allow for proving such statements is an interesting future direction.

Deployment considerations. Our definitions consider single conversations, however our constructions can be parallelized in a straightforward manner for multiple conversations. Indeed, the same server MAC key can be used, and as [1] suggests, periodically rotated. Tags will additionally have to include conversation specific identifiers in order to ensure that messages cannot be falsely

reported across conversations. This would amount to appending the identifier cid to ack before tagging it in TagSend and TagRecv. Instead of using numerical indices to identify parties, one might use unique user identifiers. This is especially important for groups as membership can change over time, hence so can the mapping between party indexes and user identities within a group. Presenting causality information in a user-friendly way to messaging parties and content moderators is an open question, which was also raised in [44]. Concretely, both the MAC and the commitment scheme can be instantiated with HMAC-SHA-256 [30]. A drawback of our proposed construction is that the server must maintain counters for each party in each channel. At scale, keeping track of this state can be prohibitive. In the remainder of this section, we describe a modification of our scheme that mitigates this issue.

Outsourced-storage transcript franking. We now propose mechanisms for allowing clients to store the counters while the server verifies how it is updated. In our no-server-storage solution, clients send these counters in the associated data of their messages. We present a solution for two-party transcript franking with outsourced storage in the remainder of this section. In Appendix 3.11, we provide a security analysis of this solution and outline its generalization to group transcript franking. We present pseudocode for our outsourced construction in Figure 3.12, highlighting the procedures that differ from the server-storage version.

Formally, an outsourced two-party transcript franking scheme is a tuple of algorithms $\mathsf{OTF} = (\mathsf{SrvInit}, \mathsf{Init}, \mathsf{Snd}, \mathsf{Rcv}, \mathsf{TagSend}, \mathsf{TagRecv}, \mathsf{Judge}, \mathsf{JudgeReplay})$, defined over a server state space $\mathcal{S}_S$, a client state space $\mathcal{S}_C$, a key space $\mathcal{K}$, a message space $\mathcal{M}$, a commitment space $\mathcal{Q}$, a franking key space $\mathcal{K}_f$, an initial-

ization tag space $\mathcal{T}_I$, a message-sent tag space $\mathcal{T}_S$, and a reception tag space $\mathcal{T}_R$. We detail these algorithms below:

- $st_S, t_0^{(0)}, t_0^{(1)} \leftarrow\!\!\$\ \mathsf{SrvInit}()$ outputs an initial server state $st_S \in \mathcal{S}_S$, along with initialization tags $t^{(0)}, t^{(1)} \in \mathcal{T}_I$ for each party.
- $st \leftarrow\!\!\$\ \mathsf{Init}(P, k)$ is defined as in Section 3.4.
- $st', c \leftarrow\!\!\$\ \mathsf{Snd}(P, st, m)$ is defined as in Section 3.4.
- $st'_S, t_s \leftarrow \mathsf{TagSend}(st_S, P, c_f, t)$ is a server procedure that produces a tag $t_s \in \mathcal{T}_S$ for a message sending event, where P is the sending party, c_f is the franking tag, and $t \in \mathcal{T}_S \cup \mathcal{T}_R \cup \mathcal{T}_I$ is the last tag issued for P.
- $st_S, t_r \leftarrow \mathsf{TagRecv}(st_S, P, c_f, t)$ is a server procedure that produces a tag $t_r \in \mathcal{T}_R$ for a message reception event by receiving party P. As with $\mathsf{TagSend}$, $t \in \mathcal{T}_S \cup \mathcal{T}_R \cup \mathcal{T}_I$ is the last tag issued for P.
- $st', m, k_f, i \leftarrow \mathsf{Rcv}(P, st, c)$ is defined as in Section 3.4.
- $G \leftarrow \mathsf{Judge}(st_S, \rho)$ is defined as in Section 3.4.
- $P \leftarrow \mathsf{JudgeReplay}(st_S, t, t')$ is a server procedure that takes as input two tags $t, t' \in \mathcal{T}_S \cup \mathcal{T}_R \cup \mathcal{T}_I$. It outputs a party $P \in \{0, 1\}$ if it determines that the tags constitute a replay attempt by P, or $\perp$ if no replay attempt is detected.

Preventing fast-forwards. When sending a message, a client increments its send counter and appends to the causality data a server tag for the previous counter. This prevents the client from incrementing the counter too far into the future, resulting in what we term a *fast-forward* attack. Doing so would require the client to forge a MAC, since it would have to produce a valid server

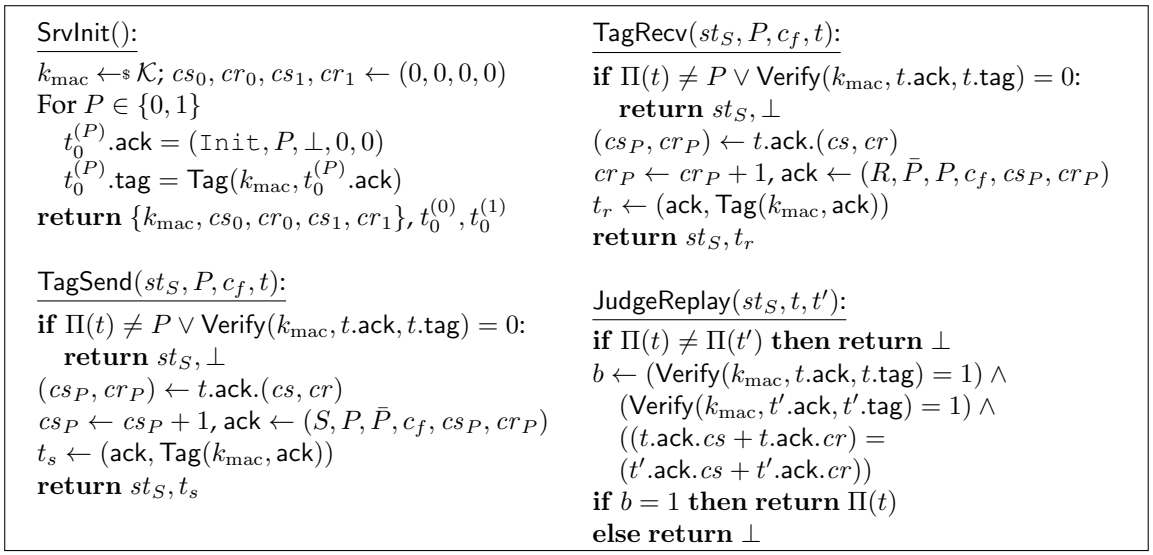

Figure 3.12: Pseudocode for our two-party transcript franking construction with outsourced storage. Let $\Pi(t)$ be the sending party if t is a sending tag and the receiving party if t is a reception tag. The routines Init, Snd, Rcv, and Judge remain unchanged relative to the pseudocode given in Figure 3.7.

tag on a message the server had not tagged before. Syntactically, this means we modify TagRecv and TagSend to take in an additional input t, the latest tag issued by the server to party P. At the initialization of a conversation, the server provides each party P with a special starting tag $t_0^{(P)}$, which are additional outputs from SrvInit, where $t_0^{(P)}.\mathsf{ack} = (\mathtt{Init}, P, \perp, 0, 0)$. We write $\mathsf{TagSend}(st_S, P, c_f, t)$ and $\mathsf{TagRecv}(st_S, P, c_f, t)$. In our modified construction, the server first checks that t is a valid tag and obtains the initial values of the counters as $(cs_P, cr_P) \leftarrow t.\mathsf{ack}.(cs, cr)$ instead of retrieving them from its own storage, for both TagSend and TagRecv. If the check on t fails, TagRecv and TagSend output $\perp$.

Preventing replays. If a client attempts to send a message with a repeated past counter, an honest recipient can report the repeated counters to the server as proof of sender misbehavior. Such a report provides resistance against rollback attacks for counters. For message reception tags, we follow the same exact ap-

proach as it applies to receive counters. We add a new procedure to the construction: $P \leftarrow \mathsf{JudgeReplay}(st_S, t, t')$, where P is the party that attempted the replay. If the provided tags do not constitute proof of a replay, then the output is $\perp$. In our modified construction, the server returns P if $(\mathsf{Verify}(k_{\mathrm{mac}}, t.\mathsf{ack}, t.\mathsf{tag}) = 1)$, $(\mathsf{Verify}(k_{\mathrm{mac}}, t'.\mathsf{ack}, t'.\mathsf{tag}) = 1)$, $((t.\mathsf{ack}.cs + t.\mathsf{ack}.cr) = (t'.\mathsf{ack}.cs + t'.\mathsf{ack}.cr))$, and $(t.\mathsf{ack} \neq t'.\mathsf{ack})$, where P is the sending party within a send acknowledgement or the receiving party of a reception acknowledgement – if these are not the same between t and t', then we output $\perp$. To submit a false replay report framing an honest party would require a client to forge a MAC. We term the submission of a false replay a *replay framing attack*, for which we provide a game-based definition in Appendix 3.11.

Replay reportability. In addition to ensuring that honest clients cannot be falsely framed for attempting a replay, we must guarantee that actual replays by malicious clients are reportable. For a party P that has been issued a server tag $t \in \mathcal{T}_S \cup \mathcal{T}_R \cup \mathcal{T}_I$, that then generates t' and t'' by invoking TagRecv and/or TagSend with the same t given as the previous tag, $\mathsf{JudgeReplay}(st_S, t', t'')$ must output P. This holds for the construction in Figure 3.12 by the checks performed in JudgeReplay.

3.8 Related Work

Message franking. Message franking has been studied in various settings. Symmetric message franking provides a reporting solution when the platform houses the moderation endpoint for receiving reports, and when sender and recipient identities are known [1, 50, 63, 65]. Our own work is situated within this

setting. Asymmetric message franking (AMF) generalizes to metadata private platforms and allows for third-party moderation [67,113]. Recent work has also generalized AMFs to group messaging [78]. There are also proposed reporting mechanisms built from secret sharing [54]. All of these works consider message franking at the single-message level.

Causality in cryptographic protocols. Prior work has investigated incorporating causality in cryptographic channels [55, 91]. Notably, recent work by Chen and Fischlin has introduced stronger causality notions and shown how to combine them with message franking protocols [44]. However, as we have discussed, their message franking formalism does not meet our goals for transcript franking, due to its reliance on client-reported causality information and the inability for reporters to disclose their own sent messages. See Appendix 3.10 for more details. The distributed systems literature has long considered the problem of ordering events over communication networks via devising notions of logical time [79] and distributed snapshots [40].

Cryptographic Abuse Mitigation. In addition to enabling user-driven reporting, other cryptographic solutions have been proposed for targeting abuse in encrypted messaging. For messaging platforms that allow forwarding content, message trace-back is a cryptographic primitive that allows a platform to determine the origin of harmful content [67,98,115]. Message franking concerns user-driven content reporting. Recent work has also considered automated reporting for messages that match a list of known harmful content [34]. Such proposals have been met with strong criticism from privacy advocates. Follow-on work has attempted to navigate the privacy vs. moderation trade-off through added transparency and placing limitations on what content can be traced [29, 104].

Another line of work explores how cryptography can be used to aid with user-blocking [103, 114].

3.9 Conclusion

Existing treatments of message franking only consider how reporters can disclose individual messages that they receive. This is insufficient for including necessary context within reports. Our work provides definitions and constructions for transcript franking, an extension of message franking protocols that allows reporting sequences of messages with strong guarantees over message ordering and contiguity. We generalize our results to multi-party messaging and show how to securely outsource state to clients, allowing for more practical deployment. How our techniques can be generalized to asymmetric message franking, in order to be applicable to metadata-private and third-party moderation settings, remains an interesting open problem.

3.10 Comparison with Causality Preservation

Recent work by Chen and Fischlin considers the problem of assuring ordering integrity for messages within cryptographic channels, as well as extending this integrity to message franking [44]. They introduce a security notion called *causality preservation*, which captures the ability for two parties to recover a consistent causal dependency graph over the messages they exchange, even in the presence of a malicious network provider. To achieve causality preservation, clients self-report the order in which messages were sent and received via addi-

tional metadata. As a result, clients obtain a shared view of the partial ordering in which messages were sent and received. This partial ordering is captured by a causality graph, which we describe in Section 3.2. Our work additionally considers the problem of multi-message franking in the group setting while Chen and Fischlin focus on two-party messaging.

Overview of MFC with causality preservation. The causality metadata is incorporated into a message franking scheme, enabling reporting of this order in addition to the contents of the messages. Such context is valuable as the ordering and contiguity of messages sent within a conversation can heavily influence a moderator's interpretation of the reported messages. To illustrate what this metadata looks like, we briefly recall the causal message franking channel $\mathsf{MFCh}_{\mathrm{cFB}}$ presented in the Chen-Fischlin paper.

A sender attaches causal metadata consisting of a queue Q and an index i_R to each sent message. The queue Q contains the actions performed by the sender that have not yet been acknowledged by the recipient. Messages can be uniquely identified by their sending index and party. An index with a bar $\bar{i}$ indicates a received message with sending index i. Actions recorded in Q simply consist of these indices. Once the recipient indicates the latest message it has received, the sender removes all elements from Q up to and including the one associated with that latest message. To communicate this, the index i_R, sent alongside Q, indicates the largest message index received by the sender from the other party, and the value i_S keeps track of the largest $\bar{i}_R$ received from the other party. Intuitively, the message with sending index $\bar{i}_R$, now confirmed to have been received by the other party, contains all elements of Q up to and including the sending action $\bar{i}_R$, allowing those actions to be safely removed from Q. De-

spite the optimization that i_S and i_R enable, Q can grow arbitrarily large, and the overall bandwidth can increase in a quadratic manner if messages are not acknowledged. In single-message franking, a user reports only messages they have received. In causality-preserving message franking, the same is true, except these messages contain metadata about the order in which other messages have been sent and received.

Reporting self-sent message contents. Recall that the message franking channel formalism allows you to only report the messages and content received from the other party. This poses an issue for transcript franking since we may have to report messages sent by the reporter themselves. Currently, The Chen-Fischlin construction does not have a solution to this problem. Their reporting formalism only allows reporting messages received from the other party. One workaround would be to require an interactive reporting process in which each party reports the messages of the other party, filling out the causality graph. Of course, this is less than desirable, especially in the case where an abusive party refuses to participate. In our solution, we propose a way in which clients can explicitly acknowledge reception of well-formed messages, thereby allowing the senders of those messages to independently report them.

Misbehavior by malicious clients. Via Q, clients self-report orderings of message sending and reception events. An issue here is that clients can self-report arbitrary such orderings, even ones that do not align with how messages were transmitted through the service provider. Figure 3.1 illustrates an example for which arbitrary self-reported messages orderings can enable malicious behavior.

Beyond the ability to deviate from the actual interleaving of messages, a

malicious sender or reporter can lie about messages having been dropped or delivered out of order. For these reasons, relying on client-reported orderings is insufficient for reporting sequences of messages.

Attack on prior construction. The augmented Facebook message franking channel construction presented in [44], $\mathsf{MFCh}_{\mathrm{cFB}}$, allows clients to report message orders that do not align with the ground truth. We illustrate this via a simple attack that mirrors the idea presented in Figure 3.1. In order to provide a fair comparison, we first describe a natural lifting from the Chen-Fischlin notion for message franking channels to our setting. To instantiate a Chen-Fischlin-style message franking channel in our setting, we define a TagRecv function that simply returns $\perp$ for the tag. The Judge function is defined in the natural way, by running Extr repeatedly for all messages involved in the report in order to construct the full graph.

The adversary $\mathcal{A}$ chooses Party 0 as the malicious party (this is opposite of the attack presented in Section 3.1, but it applies in the same manner) and issues the following oracle calls. We specify the exact causal metadata that Party 0 embeds within the messages it sends to achieve this goal within each send call.

Sequence 1:

1. $c_1 \leftarrow\!\!\$\ \mathbf{SendTag}(0, m_1; Q = (), i_R = -1)$
2. $\mathbf{RecvTag}(1, c_1)$
3. $c_2 \leftarrow\!\!\$\ \mathbf{SendTag}(1, m_2; Q = (\bar{1}), i_R = 1)$
4. $\mathbf{RecvTag}(0, c_2)$
5. $c_3 \leftarrow\!\!\$\ \mathbf{SendTag}(0, m_3; Q = (1), i_R = -1)$
6. $\mathbf{RecvTag}(1, c_3)$

7. $c_4 \leftarrow\!\!\$\ \mathbf{SendTag}(0, m_4; Q = (1, 2, \bar{1}), i_R = 1)$
8. $\mathbf{RecvTag}(1, c_4)$

The adversary $\mathcal{A}$ embeds causality metadata that suggests Party 0 having observed the ordering (m_1, m_3, m_2, m_4). Meanwhile, Party 1 observes the ordering (m_1, m_2, m_3, m_4), which also aligns with what Party 0 should have observed had it not deviated from the protocol. The ordering of **SendTag** and **RecvTag** calls differs from the ordering specified in the Q sent along with m_4. Below, we show the causality metadata Party 0 would have attached had it followed the protocol honestly:

Sequence 2:

1. $c_1 \leftarrow\!\!\$\ \mathbf{SendTag}(0, m_1; Q = (), i_R = -1)$
2. $\mathbf{RecvTag}(1, c_1)$
3. $c_2 \leftarrow\!\!\$\ \mathbf{SendTag}(1, m_2; Q = (\bar{1}), i_R = 1)$
4. $\mathbf{RecvTag}(0, c_2)$
5. $c_3 \leftarrow\!\!\$\ \mathbf{SendTag}(0, m_3; Q = (\bar{1}), i_R = 1)$
6. $\mathbf{RecvTag}(1, c_3)$
7. $c_4 \leftarrow\!\!\$\ \mathbf{SendTag}(0, m_4; Q = (\bar{1}, 2), i_R = 1)$
8. $\mathbf{RecvTag}(1, c_4)$

As a result, the graph recovered from Extr differs from the graph maintained by the security game. The adversary $\mathcal{A}$ can issue a report, from either party, indicating the wrong causal ordering, winning with probability 1.

Impossibility result. The attack we just presented works because the server has

no way to tag reception events. Since the message franking channel presented in [44] does not enable such tagging, it is impossible for any scheme within their model to satisfy our transcript integrity security notion. Intuitively, any scheme that doesn't enable the server to certify when messages are received requires the recipient party to self-report when reception events occur. In particular, the attack we just presented generalizes to any scheme that does not enable the server to tag reception events. Hence, we extend the message franking model to allow the server to tag reception events, via that TagRecv procedure.

Theorem 4. *Any message franking channel* MFCh *that does not tag message reception events does not satisfy transcript integrity.*

Proof. We consider Sequence 1 and Sequence 2 as defined above and note that they provide a scenario in which the sending events occur in the same order, but the reception events occur in a different order. Given that client input cannot be trusted to report the ordering of reception events, the Judge routine needs to somehow recover the causality graph given just information about the order in which messages were sent, which it can record upon invocation of TagSend. Since Sequences 1 and 2 have the same ordering of send operations ordering, but correspond to different causality graphs due to the difference in reception ordering, we see that it is impossible for Judge to distinguish between these two scenarios in general. In the transcript integrity game, the adversary can randomly choose to execute Sequence 1 or Sequence 2 with probability $1/2$. For any fixed Judge routine, the probability of outputting the correct causality graph is at most $1/2$. Hence, no scheme that fails to consider reception events can achieve our notion of transcript integrity. □

3.11 Security for Outsourced-storage Transcript Franking

In this section, we elaborate on the security analysis of our outsourced transcript franking scheme, which we introduced in Section 4.6. Correctness for outsourced transcript franking is captured by the game in Figure 3.13. We adapt the security notions for server-side-storage transcript franking to the outsourced setting and prove that our outsourced scheme achieves security.

Transcript integrity. In Figure 3.14, we present our transcript integrity definition for outsourced transcript franking. The goal of the adversary is the same as in the non-outsourced transcript integrity game, except now the adversary must submit valid server tags to generate new ones. Moreover, the adversary cannot replay tags in a way that is undetected by JudgeReplay. The advantage of an adversary $\mathcal{A}$ in the outsourced storage transcript integrity game is

$$\mathrm{Adv}^{\text{o-tr-int}}_{\mathsf{TF}}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{o-tr-int}}_{\mathsf{TF}}(\mathcal{A}) = 1] \ .$$

Replay framing. In Figure 3.15, we present a security game for replay framing. Given two parties that honestly interact in the protocol, the adversary attempts to generate a replay that is detected by JudgeReplay, in effect framing an honest party for attempting to replay a server tag. The advantage of an adversary $\mathcal{A}$ in the outsourced storage replay framing game is

$$\mathrm{Adv}^{\text{o-fr}}_{\mathsf{TF}}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{o-fr}}_{\mathsf{TF}}(\mathcal{A}) = 1] \ .$$

Security proofs. We now provide proofs for the transcript integrity and reply framing security of our outsourced storage transcript franking scheme. The following theorems establish the security of our outsourced construction.

$\underline{\mathbf{G}^{\text{corr}}_{\mathsf{TF}}(\mathcal{A})}$:
$k_{\mathsf{Srv}} \leftarrow\!\!\$\ \mathcal{K}$, $st_{\mathcal{A}}, k_{\mathsf{Ch}} \leftarrow\!\!\$\ \mathcal{A}()$, $\mathsf{win} \leftarrow 0$
$st_S, t_0^{(0)}, t_0^{(1)} \leftarrow\!\!\$\ \mathsf{SrvInit}(k_{\mathsf{Srv}})$
$st_0 \leftarrow\!\!\$\ \mathsf{Init}(0, k_{\mathsf{Ch}})$, $st_1 \leftarrow\!\!\$\ \mathsf{Init}(1, k_{\mathsf{Ch}})$
$t_0, t_1 \leftarrow t_0^{(0)}, t_0^{(1)}$
$\mathcal{R}_t, \mathcal{R}_r \leftarrow \{\}, \{\}$
$\mathcal{A}^{\mathcal{O}}(st_{\mathcal{A}}, k_{\mathsf{Ch}})$
return win

$\underline{\mathcal{O}.\mathbf{SendTag}(P, m)}$:
$(st_P, c) \leftarrow\!\!\$\ \mathsf{Snd}(P, st_P, m)$
$st_S, t_P \leftarrow\!\!\$\ \mathsf{TagSend}(st_S, P, c.c_f, t_P)$
$G \leftarrow G + (\mathtt{S}, P, m)$
Add (P, c, t_P) to $\mathcal{R}_t$
return c, t_P

$\underline{\mathcal{O}.\mathbf{RecvTag}(P, c, t_s)}$:
Assert $(\bar{P}, c.c, t_s) \in \mathcal{R}_t$
$st_P, m, k_f, i \leftarrow \mathsf{Rcv}(P, st_P, c)$
if $m \neq \perp$ **then**
  $st_S, t_P \leftarrow\!\!\$\ \mathsf{TagRecv}(st_S, P, c_f, t_P)$
  $G \leftarrow G + (\mathtt{R}, P, c.i)$
  Add $(P, m, k_f, c_f, t_s, t_r)$ to $\mathcal{R}_r$
else
  $\mathsf{win} \leftarrow 1$
return m, k_f, t_s, t_P

$\underline{\mathcal{O}.\mathbf{Rep}(\rho)}$:
Assert $|\rho| > 0$ $G' = \mathsf{Judge}(st_S, \rho)$
if $\rho \subseteq \mathcal{R}_r \wedge ((G' = \perp) \vee (G' \not\subseteq G))$:
  $\mathsf{win} \leftarrow 1$

Figure 3.13: The security game for outsourced-storage transcript franking correctness.

$\underline{\mathbf{G}^{\text{o-tr-int}}_{\mathsf{TF}}(\mathcal{A})}$:
$k_{\mathsf{Srv}} \leftarrow\!\!\$\ \mathcal{K}$; $\mathsf{win} \leftarrow 0$
$st_{\mathcal{A}}, k_{\mathsf{Ch}} \leftarrow\!\!\$\ \mathcal{A}()$
$st_S, t_0^{(0)}, t_0^{(1)} \leftarrow\!\!\$$
  $\mathsf{SrvInit}(k_{\mathsf{Srv}})$
$st_0 \leftarrow\!\!\$\ \mathsf{Init}(0, k_{\mathsf{Ch}})$
$st_1 \leftarrow\!\!\$\ \mathsf{Init}(1, k_{\mathsf{Ch}})$
$G, \mathcal{R}_r, \mathcal{R} \leftarrow \varepsilon, \{\}, \{\}$
$\mathcal{A}^{\mathcal{O}}(st_{\mathcal{A}}, k_{\mathsf{Ch}}, t_0^{(0)}, t_0^{(1)})$
return win

$\underline{\mathcal{O}.\mathbf{SendTag}(P, c, t)}$:
Assert for all $t_1, t_2 \in \mathcal{R}$,
  $\mathsf{JudgeReplay}(st_S, t_1, t_2) = 0$
$st_S, t_s \leftarrow \mathsf{TagSend}(st_S, P, c.c_f, t)$
if $t_s = \perp$ **then return** $\perp$
$G \leftarrow G + (\mathtt{S}, P)$
Add (P, c, t_s) to $\mathcal{R}_t$, add t_s to $\mathcal{R}$
return t_s

$\underline{\mathcal{O}.\mathbf{RecvTag}(P, c, t_s, t)}$:
Assert $(\bar{P}, c, t_s) \in \mathcal{R}_t$
Assert for all $t_1, t_2 \in \mathcal{R}$,
  $\mathsf{JudgeReplay}(st_S, t_1, t_2) = 0$
$st_S, t_r \leftarrow \mathsf{TagRecv}(st_S, P, c.c_f, t)$
if $t_r = \perp$ **then return** $\perp$
$G \leftarrow G + (\mathtt{R}, P, c.i)$
Add t_r to $\mathcal{R}$
return t_r

$\underline{\mathcal{O}.\mathbf{Rep}(\rho_1, \rho_2)}$:
Assert $|\rho_1| > 0$ and $|\rho_2| > 0$
$G_1 \leftarrow \mathsf{Judge}(st_S, \rho_1)$
$G_2 \leftarrow \mathsf{Judge}(st_S, \rho_2)$
if $G_1 \neq \perp \wedge G_2 \neq \perp \wedge$
  $((\widetilde{G_1} \not\subseteq G) \vee (\widetilde{G_2} \not\subseteq G)$
  $\vee (G_1 \not\approx G_2))$:
  $\mathsf{win} \leftarrow 1$

Figure 3.14: The security game for outsourced-storage transcript integrity.

Theorem 5. *Let* TF *be our outsourced-storage transcript franking scheme in Figure 3.12. Let $\mathcal{A}$ be a transcript integrity adversary against* TF*. Then we give EUF-CMA adversaries $\mathcal{B}$ and $\mathcal{C}$, and a V-Bind adversary $\mathcal{D}$, such that*

$$\mathrm{Adv}^{\text{o-tr-int}}_{\mathsf{TF}}(\mathcal{A}) \leq \mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{B}) + \mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{C}) + \mathrm{Adv}^{\text{v-bind}}_{\mathsf{CS}}(\mathcal{D}) \ .$$

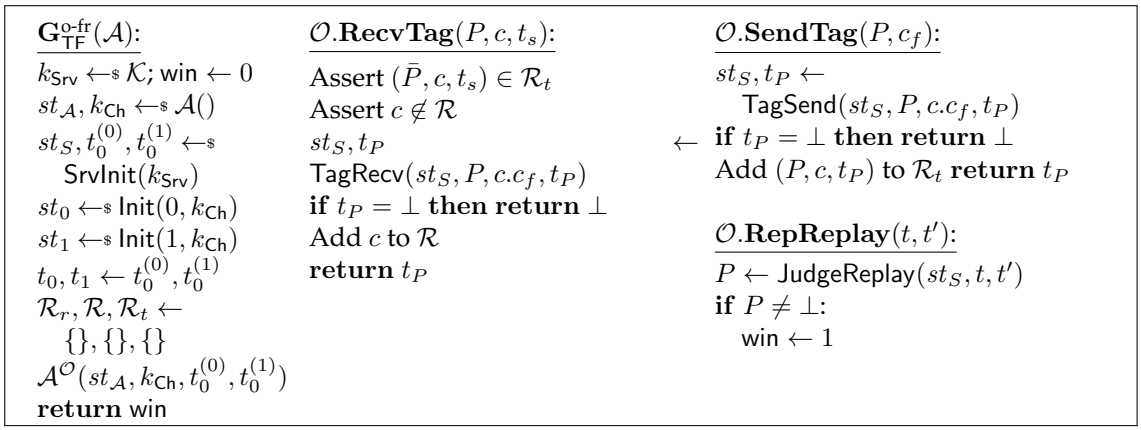

Figure 3.15: The security game for outsourced storage replay framing.

Adversaries $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ run in time that of $\mathcal{A}$ plus a small overhead.

Proof. We proceed via a sequence of game hops. Define $\mathbf{G}_0$ to be the same as $\mathbf{G}^{\text{o-tr-int}}_{\mathsf{TF}}$ with the additional bookkeeping as defined in the proof of Theorem 1. Let $\mathbf{G}_1$ be the same, except we abort if at any point $G.(cs_P, cr_P) \neq st_S.(cs_P, cr_P)$. Let F_1 denote this event. We have $|\Pr[\mathbf{G}_0(\mathcal{A}) \Rightarrow 1] - \Pr[\mathbf{G}_1(\mathcal{A}) \Rightarrow 1]| \leq \Pr[F_1]$, and we will construct $\mathcal{B}$ such that $\mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{B}) = \Pr[F_1]$. We have that $\mathcal{B}$ simulates $\mathbf{G}_0$ to $\mathcal{A}$ while routing Tag and Verify calls to its challenger oracles. Observe that the only way for F_1 to occur is for $\mathcal{A}$ to produce a $t^* \notin \mathcal{R}$, which means that t^* was never queried to the Tag oracle, hence $\mathcal{B}$ outputs it as a forgery.

The rest of the proof proceeds similarly to the proof of Theorem 1. □

Theorem 6. *Let* TF *be our outsourced-storage transcript franking scheme in Figure 3.12. Let $\mathcal{A}$ be a replay framing adversary against* TF*. Then we give an EUF-CMA adversary $\mathcal{B}$ such that*

$$\mathrm{Adv}^{\text{o-fr}}_{\mathsf{TF}}(\mathcal{A}) \leq \mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{B}) \ .$$

Adversary $\mathcal{B}$ runs in time that of $\mathcal{A}$ plus a small overhead.

Proof. Our adversary $\mathcal{B}$ perfectly simulates $\mathbf{G}^{\text{o-fr}}_{\mathsf{TF}}$ to $\mathcal{A}$ while routing calls to Tag and Verify to its own challenger oracles. For any two tags t_1, t_2 output by the Tag oracle, we must have that $\mathsf{JudgeReplay}(st_S, t_1, t_2) = \perp$, since these tags are the output of the honest TagRecv and TagSend procedures. Therefore, if $\mathcal{A}$ wins, it must have produced a pair (t, t') where at least one of these tags was not output by the Tag challenger oracle. Let t^* be that tag. The adversary $\mathcal{B}$ outputs t^* and wins with the same probability that adversary $\mathcal{A}$ wins. □

Group outsourced transcript franking. The construction and analysis we provide for outsourced two-party transcript franking generalizes naturally to the N-party group messaging setting. We sketch the necessary modifications here. As with the two-party outsourced setting, we have that SrvInit, in addition to st_S, outputs a list of initial tags $t_0^{(0)}, \ldots, t_0^{(N-1)}$. The server-side tagging procedures TagSend and TagRecv accept an additional argument t for the previous tag issued to a party. We include an additional procedure $\mathsf{JudgeReplay}(st_S, t, t')$, which outputs a party P if (t, t') constitute a replay attack by P, or $\perp$ otherwise. We outline our outsourced group transcript franking construction in Figure 3.16.

$\underline{\mathsf{SrvInit}(N)\text{:}}$
$k_{\mathrm{mac}} \leftarrow\!\!\$\ \mathcal{K}$
For $P \in [N]$
  $t_0^{(P)}.\mathsf{ack} = (\mathtt{Init}, P, \perp, 0, 0)$
  $t_0^{(P)}.\mathsf{tag} = \mathsf{Tag}(k_{\mathrm{mac}}, t_0^{(P)}.\mathsf{ack})$
  $cs_i, cr_i \leftarrow 0, 0$
return $\{k_{\mathrm{mac}}\} \cup \{cs_i, cr_i\}_{i\in[N]}, \{t_0^{(i)}\}_{i\in[N]}$

$\underline{\mathsf{TagSend}(st_S, P, c_f, t)\text{:}}$
if $\Pi(t) \neq P \vee \mathsf{Verify}(k_{\mathrm{mac}}, t.\mathsf{ack}, t.\mathsf{tag}) = 0$:
  return $st_S, \perp$
$(cs_P, cr_P) \leftarrow t.\mathsf{ack}.(cs, cr)$
$cs_P \leftarrow cs_P + 1, \mathsf{ack} \leftarrow (S, P, c_f, cs_P, cr_P)$
$t_s \leftarrow (\mathsf{ack}, \mathsf{Tag}(k_{\mathrm{mac}}, \mathsf{ack}))$
return st_S, t_s

$\underline{\mathsf{TagRecv}(st_S, P_R, P_S, c_f, t)\text{:}}$
if $\Pi(t) \neq P_R \vee \mathsf{Verify}(k_{\mathrm{mac}}, t.\mathsf{ack}, t.\mathsf{tag}) = 0$:
  return $st_S, \perp$
$(cs_{P_R}, cr_{P_R}) \leftarrow t.\mathsf{ack}.(cs, cr)$
$cr_{P_R} \leftarrow cr_{P_R} + 1$
$\mathsf{ack} \leftarrow (R, P_S, P_R, c_f, cs_{P_R}, cr_{P_R})$
$t_r \leftarrow (\mathsf{ack}, \mathsf{Tag}(k_{\mathrm{mac}}, \mathsf{ack}))$
return st_S, t_r

$\underline{\mathsf{JudgeReplay}(st_S, t, t')\text{:}}$
if $\Pi(t) \neq \Pi(t')$ **then return** $\perp$
$b \leftarrow (\mathsf{Verify}(k_{\mathrm{mac}}, t.\mathsf{ack}, t.\mathsf{tag}) = 1) \wedge$
  $(\mathsf{Verify}(k_{\mathrm{mac}}, t'.\mathsf{ack}, t'.\mathsf{tag}) = 1) \wedge$
  $((t.\mathsf{ack}.cs + t.\mathsf{ack}.cr) =$
  $(t'.\mathsf{ack}.cs + t'.\mathsf{ack}.cr))$
if $b = 1$ **then return** $\Pi(t)$
else return $\perp$

Figure 3.16: Pseudocode for our N-party transcript franking construction with outsourced storage. Let $\Pi(t)$ be the sending party if t is a sending tag and the receiving party if t is a reception tag.

CHAPTER 4

GROUP STATE MANAGEMENT AND REPORTING FOR ENCRYPTED PLATFORMS

4.1 Introduction

Billions of users of end-to-end encrypted (E2EE) messaging applications enjoy numerous security and privacy benefits [124]. Text messages, images, videos, and other media are encrypted under keys accessible only to the endpoints of the communication [76]. Therefore, platform compromise or law enforcement overreach cannot threaten user privacy; the platform has no way to see the plaintext content exchanged by users. While beneficial for privacy, this poses a challenge for content moderation, the process by which a platform sets and enforces rules for the type of content it hosts and transmits [61,75]. Meanwhile, online harms, such as harassment and misinformation, have grown increasingly prevalent in recent years [110]. These harms manifest on E2EE platforms as well, such as widespread misinformation on WhatsApp [87]. The tension between mitigating online abuse and E2EE has led to a new Crypto Wars [24,72], with governments and law enforcement calling for client-side scanning [34] for abusive materials while privacy advocates label these efforts as harmful back-doors that threaten security and civil liberty [23].

To better navigate these tensions, recent work proposed *private hierarchical governance*, which provides community moderation for end-to-end encrypted messaging groups, while still allowing reporting to the platform [96]. With community moderation, users are given tools to set and enforce moderation policies within their own groups [68]. Moderation policies may specify access control

for various group members, such as allowing some members to take down posts and remove abusive users. These policies can also automatically filter content based on words/URLs. Examples of tools that provide these functionalities include the Reddit Automoderator [68] and Discord moderation bots [74]. Some tools even allow users to specify community policies via code [127].

Importing these tools to the E2EE setting must be done with care so as not to undermine privacy benefits. Such tools keep track of the privileges associated with each group member, a list of group rules, the content filter for the group, etc. This information may be sensitive and therefore should not be revealed to the platform by default. Thus, E2EE community moderation necessitates shared encrypted state (SES) among group members. The MlsGov system is an example of one such approach that achieves this [96].

In addition to empowering communities to perform their own moderation, platforms must have mechanisms for enforcing their own policies, especially in the presence of communities that are largely malicious. Within encrypted platforms, user-driven reporting mechanisms largely fulfill this role. In encrypted messaging, reporting allows users to disclose messages sent within groups to platform moderators. If the platform determines that the messages demonstrate a violation of platform policies, it can take action by restricting users and/or groups via bans and content takedown. Securely achieving this functionality is the goal of message franking, a cryptographic mechanism that allows message disclosure without compromising the privacy of unreported messages, achieving reportability of valid messages, and disallowing the reporting of content that was never sent through the platform [50, 63]. All current work on message franking considers only the messages as reportable content. However,

when groups maintain shared encrypted state, disclosing this state and reporting messages in a way that ties them to the shared state at the time they were sent becomes crucial for reporting. Although the MlsGov system from prior work provides management for SES, it does not support reportability of this state [96].

To see the utility of reporting shared state, consider the following example. A messaging platform includes within its moderation guidelines a rule against promoting hate speech. Despite this, a group includes hateful language within its name or community guidelines, which are encrypted and not visible to the platform. After joining the group, a non-abusive member is shocked to discover this and reports it to the platform. Upon receiving the report, the platform asks the user to disclose the moderator(s) of the group for further intervention. It is impossible to securely support this type of interaction using existing franking techniques that apply to standard content-bearing messages. Beyond reporting abusive messages, it would also be beneficial to flag malicious clients to the platform. Such clients may produce invalid or even non-decryptable SES. The ability to rapidly identify such clients is crucial to ensuring smooth group operation. Furthermore, it would be beneficial to situate group messages within the context of the shared state at a particular point in time.

For example, consider a group policy, expressed in the shared state, that mandates that group moderators place members on a mute-list (also part of the shared state) upon violation of group norms. Reporting messages contextualized within the shared state would allow a group member to raise moderator negligence to the platform.

Goal. We aim to specify a mechanism for maintaining shared encrypted state

among group members in a way that facilitates reporting of this state. We refer to such a mechanism as a shared encrypted state (SES) franking scheme. A group begins with an empty initial state, denoted by $\perp$, that members modify via update messages that follow a total ordering. We identify versions of the state via an epoch counter i, which is initially 0. The evolution of group state is defined via a deterministic group state update function $gs' \leftarrow \mathsf{GSUp}(P, gs, u)$, where P is the sender of the message, gs is the shared state before the update at epoch i, u is the update message, and gs' is the shared state at epoch $i + 1$. Our system must be able to recover from invalid or malformed updates that malicious clients may attempt to submit. Messages within a group can also indicate the epoch within which they were sent. Clients should be able to report that the shared state at some epoch i^* was gs^*.

Technical challenges. Maintaining SES in MlsGov involves ordered encrypted updates that clients apply to their local state [96]. Newly joined clients receive the aggregate group state as an encrypted blob. All clients achieve a consistent state because they process the updates in the same order. If a malicious client issues an invalid update, honest clients will simply ignore it and still achieve a valid and consistent shared state.

A natural straw solution to SES franking is simply reporting these update messages and having the moderator reconstruct the shared state by applying the updates one after the other. This suffers from several issues. First, existing message franking schemes are tailored to reporting individual messages. A moderator would not be able to tell if certain updates were omitted or reordered. To fix this, one could append an epoch counter to the updates. Yet, for long-lived groups, this would require disclosing prohibitively many messages.

In addition to being undesirable from a bandwidth standpoint, this is bad for privacy as it reveals the full history and evolution of the group state as opposed to the group state at a particular point in time.

Iterating on this, one may consider applying message franking on top of broadcasts of the consolidated group state gs. The issue is that the platform cannot verify that the state is valid or accepted by all honest group members. Furthermore, this solution does not adequately handle malformed ciphertexts, which do not decrypt correctly for honest group members. A malicious client may post a ciphertext for which only they know the decryption key and attempt to issue a report of the group state.

There are powerful cryptographic tools that allow clients to proactively assert the validity of updates. For a relation $R(x, w)$ over public statements x and witnesses w, a zero-knowledge proof system is a protocol that allows a prover to convince a verifier that there is some w such that $R(x, w) = 1$ without having to reveal w [66]. Recently, zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) have gained traction in the realm of verifiable computation [62]. In principle, we could use zk-SNARKs to have clients reveal the group state at a particular epoch without revealing prior updates. We could also use them to have the client prove to the server that the ciphertext they submit for the SES correctly applies an update to the prior SES. However, zk-SNARKs can be prohibitively expensive for complex computations, leading to concerns for scalability State of the art SNARKs that do not require trusted setup incur large costs in terms of latency and proof size just to prove knowledge of the pre-image of a SHA256 hash [84]. Proving statements about shared encrypted states must incur even larger costs. Given that we target deployment on mobile

devices, we must adopt a lightweight approach.

Our SES construction. We now provide a brief overview of our construction for SES franking. To update the state, a client supplies an update token u and a commitment c_f to new group state gs' after the update, and an opening k_f to the commitment. Both u and k_f are sent via the encrypted group channel and are tagged by the server, which also tags c_f. The server acts as the authority on ordering, determining which update is applied if two arrive simultaneously, attempting to build off the same previous update. The server also maintains a counter for the updates in order to be able to tie content messages to particular versions of the shared state. Platform moderators can thereby know the group state at the time a particular message was sent, which may add useful context that informs the interpretation of a particular reported message. For instance, a reporter could show that another group member is violating group norms included within the shared state by disclosing messages demonstrating this and tying them to the shared state at the time they were sent.

We ensure group state validity and reportability via a natural acknowledgement and flagging mechanism, assuming an honest majority of clients at any given point in time. Upon receiving an update message u and commitment c_f, a client applies u to their local view of the state to obtain gs'. The client then checks to see if c_f verifies with gs', u and k_f. If this check verifies, the client sends an acknowledgement of validity to the server, which the server tags. Otherwise, if decryption fails or if the commitment does not verify, the client flags this to the server. For a group of N clients, assume $\geq t$ are honest. Then a group state can be reported once $\geq t$ acknowledgements for it are tagged by the server. Conversely, an update can be reported as malformed once $\geq t$ flags are received

for it. To enable efficient reporting, we make use of aggregate MACs [73].

Our key insight is that we can use simpler and more efficient cryptographic primitives if we target post hoc reportability of deviation by malicious clients as opposed to preventing deviation proactively. Proactive prevention while maintaining privacy would require use of costly generic zero-knowledge proofs, which would be burdensome on lightweight client devices, and on servers dealing with billions of clients. An added benefit of our approach is that honest clients can continue to proceed normally even in the presence of invalid updates. The invalid updates are simply ignored. Reporting to the server, however, can only happen once a threshold ($\geq t$) number of acknowledgements or flags are produced. The value we choose for t depends on the application setting and report purpose, which we further discuss in Section 4.3. Our usage of aggregate MACs allows for scalability in threshold reporting by lowering the bandwidth associated with sending acknowledgements. In Section 4.6, we discuss optimizations that this enables for both storage on the server and on client devices.

Beyond governance state. Although our instantiation of shared encrypted state greatly benefits governance for encrypted messaging groups, we remark that this primitive is useful in other contexts. File sharing services and other E2EE applications that require shared state can benefit from our paradigm as well. A user may wish to report abuse in a file within a shared cloud folder for instance. With new developments in E2EE social media with platforms such as Germ [17], efforts to bring E2EE to the ActivityPub protocol [18], and E2EE collaborative document editing [2], reportability of shared encrypted state will find

other applications.

Contributions. In this work, we formalize a primitive for shared encrypted state in encrypted messaging groups and extend message franking to accommodate verifiable disclosure of this shared state to platform moderators.

- We define our threat model and specify syntax, semantics, and security definitions for SES franking in Section 4.3.
- We give a construction for SES and provide a formal security analysis in Section 4.4.
- To show our construction is practical, we provide benchmarks of an implementation of it in Section 4.5.
- We provide deployment considerations and discuss various extensions in Section 4.6.

4.2 Preliminaries

Notation. The symbol $\mathbb{Z}^+$ shall refer to the non-negative integers $\{0, 1, 2, \ldots\}$. The notation $\{0,1\}^\ell$ denotes the set of bit-strings of length ℓ. To indicate the range $\{0, 1, \ldots, N-1\}$ for some $N \in \mathbb{Z}^+$, we use the shorthand $[N]$. Assigning a variable y to the value v is shown as $y \leftarrow v$. Assigning y to the output of a deterministic algorithm (equivalently, procedure or routine) Alg on the input x is written as $y \leftarrow \mathsf{Alg}(x)$. If Alg is randomized, we write $y \leftarrow\!\!\$\ \mathsf{Alg}(x)$. To denote sampling a value r uniformly at random from a set S, we write $r \leftarrow\!\!\$\ S$. We express security via games involving an adversary, which we often denote via

$\mathcal{A}$, drawing on the code-based games framework of Bellare and Rogaway [31].

Group channels. We adopt an encrypted group messaging channel formalism inspired by [91] and [44]. We define a group channel Ch as a tuple of algorithms $(\mathsf{Init}, \mathsf{Snd}, \mathsf{Rcv})$ defined over a key space $\mathcal{K}$, message space $\mathcal{M}$, party space $\mathcal{P}$, and ciphertext space $\mathcal{C}$:

- $st \leftarrow\!\!\$\ \mathsf{Init}(k)$ initializes the channel with a randomly sampled shared key $k \in \mathcal{K}$, producing client state st.
- $c \leftarrow\!\!\$\ \mathsf{Snd}(P, st, m)$ takes as input a party $P \in \mathcal{P}$, client state st, and message $m \in \mathcal{M}$. It outputs a ciphertext $c \in \mathcal{C}$.
- $m, i \leftarrow \mathsf{Rcv}(P_r, st, P_s, c)$ takes as input the receiving party $P_r \in \mathcal{P}$, client state st, sending party $P_s \in \mathcal{P}$, and ciphertext c. It outputs a decrypted message $m \in \mathcal{M}$ and message sequence counter $i \in \mathbb{Z}^+$.

A secure channel provides confidentiality and integrity properties – we adopt the notions discussed in [44]. Concretely, this channel would be implemented via a protocol such as MLS [25], and can also provide forward secrecy (FS) and post compromise security (PCS). The advantage of our channel abstraction is that it hides low-level details of the underlying key-agreement mechanism, which will be advantageous when we specify the message franking component of our protocol. Note that this abstraction does not explicitly handle dynamic groups. We discuss an extension of our approach to dynamic groups in Section 4.6.

Commitment schemes. A commitment scheme consists of two algorithms $\mathsf{CS} = (\mathsf{Com}, \mathsf{VerC})$. The randomized algorithm Com takes as input a message

m and outputs the tuple (k, c) where c is the commitment and k is the opening key. The algorithm $\mathsf{VerC}(m, k, c)$ outputs 1 if k is a valid opening of c and 0 otherwise. A commitment scheme is hiding if c reveals nothing about m and binding if it is impossible to find $m \neq m'$ and k, k' such that both $\mathsf{VerC}(m, k, c)$ and $\mathsf{VerC}(m, k', c')$ output 1. HMAC-SHA-256 with a random key instantiates such a commitment scheme [63].

Message franking. A message franking scheme allows reporting of messages sent through an E2EE platform while maintaining confidentiality for unreported messages. The first proposal for message franking was specified in the Facebook secret conversations whitepaper [1] and later analyzed in [63], which formalizes message franking in terms of committing AEAD. Each ciphertext has associated with it a hiding and binding commitment c_f to the plaintext. The platform tags c_f for sent ciphertexts via a message authentication code (MAC). Later, when a user wishes to report a message, it provides the opening (m, k_f) to c_f along with the platform-generated MAC tag τ. The platform verifies the MAC and the commitment c_f to determine whether the reported message is valid.

Aggregate MACs. A message authentication code (MAC) consists of a tuple of algorithms $(\mathsf{KGen}, \mathsf{Tag}, \mathsf{Ver})$. The randomized key generation algorithm $\mathsf{KGen}()$ outputs a symmetric key k. The tagging algorithm $\mathsf{Tag}(k, m)$ takes as input this key k and a message m and outputs a tag τ. The verification algorithm $\mathsf{Ver}(k, m, \tau)$ verifies that τ is a valid tag on the message m. Verification typically amounts to recomputing the tag on m using k and verifying that this matches τ. Aggregate MACs [73], similar to aggregate signatures [35], allow one to combine a set of MAC tags over multiple pairs of keys and messages into a single

tag that can later be verified over that set of key-message pairs. The basic idea is that a set of MAC tags can be aggregated via an XOR operation. Verification amounts to recomputing and XORing the tags, then checking equality. The EUF-CMA game for aggregate MACs is essentially the same as the standard game, except that the adversary submits a set of messages and a single aggregate tag to the challenger. Winning the aggregate MAC game requires the adversary to produce a tag that verifies against this set of messages, and for there to be at least one message within this set that was never queried to the challenger's MAC oracle.

Shared state. We model the evolution of shared state within a group via an intuitive abstraction that we refer to as a *shared state mechanism (SSM)*. An SSM is defined over a party space $\mathcal{P}$, a shared state space $\mathcal{G}$, and an update space $\mathcal{U}$. We use the variable $gs \in \mathcal{G}$ to denote the shared group state at any given point in time, which is measured in *epochs*. An epoch $i \in \mathbb{Z}^+$ identifies a version of gs. At group creation time, $i = 0$ and $gs = \perp$. Shared state evolves as group members issue updates. The state evolution is defined via a function $\mathsf{GSUp} : \mathcal{P} \times \mathcal{G} \times \mathcal{U} \to \mathcal{G}$. For a party $P \in \mathcal{P}$, group state $gs \in \mathcal{G}$, and update $u \in \mathcal{U}$, the procedure $\mathsf{GSUp}(P, gs, u)$ outputs the new group state gs' after applying the update u issued by P to gs. This operation advances the group state gs at epoch i to the new group state gs' at epoch $i + 1$. The epoch counter i can be maintained alongside the group state. Figure 4.1 provides a visual representation of this process. We illustrate our abstraction with a simple example. Consider an SSM that maintains a key-value store. Update messages are of the form $(\texttt{put}, k, v)$ and the group state consists of a set of key-value pairs. An example execution can look like the following

1. $(P_1, \perp, (\texttt{put}, \texttt{a}, 1)) \rightarrow \{(a, 1)\}$
2. $(P_3, \{(a, 1)\}, (\texttt{put}, \texttt{b}, 2)) \rightarrow \{(a, 1), (b, 2)\}$
3. $(P_2, \{(a, 1), (b, 2)\}, (\texttt{put}, \texttt{a}, 2)) \rightarrow \{(a, 2), (b, 2)\}$

On the left-hand side of each arrow, we show the initial state and on the right-hand side we show the state after applying the specified update. Observe that clients achieve consistent shared state by applying the same updates in the same order. Looking forward, the platform will bear the responsibility of ensuring a consistent ordering of these updates. Prior work incorporates client-side verification mechanisms to ensure consistency over the sequence of updates being applied [96].

This mechanism for maintaining shared state is quite generic and can support a wide range of computations beyond maintaining a shared key-value store. MlsGov uses this style of shared updates to support policies such as maintaining group roles, voting for moderators, and implementing updatable content filters [96]. Any mechanism for which the shared state is a deterministic function of the update messages can be supported by this approach.

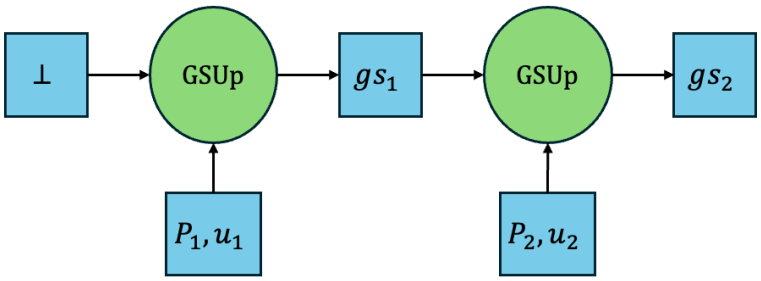

Figure 4.1: A depiction of shared state evolution for two epochs. The initial shared state gs_0 is $\perp$. Then party P_1 applies update u_1 to obtain the shared state gs_1 for epoch 1. This updated state gs_1 is the output of the shared update function GSUp on inputs (P_1, gs_0, u_1). Party P_2 then applies u_2 to gs_1 to obtain the new shared state gs_2 for epoch 2.

4.3 Shared Encrypted State

In this section, we provide an overview of how we model our system, our threat model, introduce SES franking syntax and semantics, and provide our security definitions.

Platform model. In E2EE messaging, the platform is often conceptualized as providing an authentication service (AS) for user identity management and a delivery service (DS), which transmits message ciphertexts – these are terms that we borrow from the MLS specification [28]. For the purposes of this work, we do not explicitly model user identity management and hence we focus on the DS-oriented aspect of the platform. In line with standard symmetric message franking, we assume that the DS knows sender and recipient identities, and is responsible for tagging and verifying messages. Trusting the platform in this way is reasonable as it is the moderation entity to which users report messages.

Clients can issue updates to the SES for groups to which they belong. These updates can be sent via an independent encrypted channel, separate from the one used for standard group messages. We rely on the platform to act as the authority on ordering state updates. Consequently, it acts as a tie-breaker in case of a simultaneous update to the SES. The platform manages an SES epoch counter for each group.

Settings for SES. The primary motivation for SES is maintaining shared state for encrypted group messaging. Often, this state pertains to group governance, containing information such as the list of moderators of the group, community guidelines, and group member reputations. With the release of the WhatsApp communities feature, more advanced group management functionality has en-

tered mainstream E2EE messaging. Meanwhile, WhatsApp acknowledges the role it has as a platform to moderate abusive communities [125]. The WhatsApp security whitepaper does not make any mention of encrypting the shared state associated with groups [13]. However, doing so is crucial to upholding the goals of end-to-end encryption, and SES franking enables platform moderation for such a setting.

Beyond the group messaging context, settings such as encrypted collaborative document editing and E2EE social media can benefit from SES reporting. CryptPad is an example of an E2EE platform that supports real-time document editing [2]. Recent work has also explored bringing end-to-end encryption to services such as Git [83], providing yet another application of shared encrypted state. Facilitating abuse reporting in such settings will also be of interest. SES enables users to report files containing harmful content.

Newer social media platforms provide features that require shared encrypted state. Peergos, which provides E2EE file sharing and social media features [19], serves as one such example. Germ, a messaging platform, supports encrypted user profiles that can be updated over time [17]. Supporting full-fledged abuse reporting for these platforms requires reportability of SES.

Threat model. We aim to achieve confidentiality of unreported messages, even to a platform that may tamper with messages. One may wonder about the implications of platform compromise for management of shared group state and reporting, especially considering that the platform acts as the ordering authority on messages. Honest clients can maintain ordering information locally and compare it with the ordering reported by the platform. Any deviation here would flag malicious platform behavior to honest clients. Furthermore, a mali-

cious platform could simply choose to not produce valid reporting tags or fail to properly verify tags. This would also become apparent to an honest client during the reporting process.

SES franking syntax and semantics. An N-party SES scheme provides interfaces for updating and obtaining the latest version of the shared state. In addition, it allows reportability of the shared state to a platform moderator. Formally, an SES scheme is defined as a tuple of algorithms $\mathsf{SES} = (\mathsf{SrvInit}, \mathsf{Init}, \mathsf{Put}, \mathsf{Get}, \mathsf{TagPut}, \mathsf{TagGet}, \mathsf{GenReport}, \mathsf{Judge})$, defined in relation to a key space $\mathcal{K}$, a server state space $\mathcal{S}_S$, a client state space $\mathcal{S}_C$, a ciphertext space $\mathcal{C}$, a shared state message space $\mathcal{M}$, an update space $\mathcal{U}$, report space $\mathcal{E}$, a franking tag space $\mathcal{F}$, and a server tag space $\mathcal{T}$. An instantiation of an SES franking scheme is parameterized by a group state update function GSUp. We detail the algorithms below for a group of size $N \in \mathbb{Z}^+$.

- $st_S \leftarrow\!\!\$\ \mathsf{SrvInit}()$ is a randomized procedure that outputs an initial server state st_S.
- $st \leftarrow\!\!\$\ \mathsf{Init}(P, k)$ outputs an initial client state $st \in \mathcal{S}_C$ for party $P \in [N]$ with shared initial key $k \in \mathcal{K}$.
- $st', c, gs, cc \leftarrow\!\!\$\ \mathsf{Put}(P, st, u)$ is a client procedure that takes a party $P \in [N]$, client state $st \in \mathcal{S}_C$, update $u \in \mathcal{U}$. Upon executing a Put operation, the client updates their local state to st' and produces a ciphertext $c \in \mathcal{C}$ that encrypts the update, along with other information that enables verification and reporting. The updated group state gs is also returned.
- $st', m, k_f \leftarrow \mathsf{Get}(P_r, st, P_s, c)$ is a client procedure run by receiving party $P_r \in [N]$. It decrypts the ciphertext $c \in \mathcal{C}$ from sending party P_s, where

$P_s \in [N]$, with receiving party client state $st \in \mathcal{S}_C$. In the event of successful decryption, we have $m = (u, gs, cc)$, where $u \in \mathcal{U}$ is the update, $gs \in \mathcal{M}$ is the new group state for the new epoch $cc \in \mathbb{Z}^+$. If decryption fails, $m = \perp$. The epoch counter is incremented once a new valid ciphertext is processed, updating the local version of the shared state. An invalid ciphertext will not increment cc. The value k_f is the franking key, which is used in the event of a report.

- $st'_S, t_p \leftarrow \mathsf{TagPut}(st_S, P, c_f)$ is a server procedure that outputs a reporting tag $t_p \in \mathcal{T}$ on a group state update franking tag $c_f \in \mathcal{F}$ from party $P \in [N]$, using the server state $st_S \in \mathcal{S}_S$.
- $st'_S, t_g \leftarrow \mathsf{TagGet}(st_S, P_s, P_r, c_f)$ is a server procedure for tagging a ciphertext with franking tag $c_f \in \mathcal{F}$ upon client acknowledgement by $P_r \in [N]$ of a message sent by $P_s \in [N]$, producing $t_g \in \mathcal{T}$, using server state $st_S \in \mathcal{S}_S$.
- $\rho \leftarrow \mathsf{GenReport}(P_s, t_p, T_g, u, gs, cc, cs, c_f, k_f)$ is a client procedure that generates report information $\rho \in \mathcal{E}$ for a group state modification $u \in \mathcal{U}$ to yield $gs \in \mathcal{M}$ initiated by sending party $P_s \in [N]$ for a set of get tags $T_g \in \{\mathcal{T}\}$ and a put tag $t_p \in \mathcal{T}$. The procedure also accepts the client epoch counter $cc \in \mathbb{Z}^+$, the server epoch counter $cs \in \mathbb{Z}^+$, as well as the franking tag $c_f \in \mathcal{F}$, and the opening k_f.
- $st'_S, b \leftarrow \mathsf{Judge}(st_S, P_s, R, u, gs, cc, cs, \rho, c_f)$ is a server procedure that takes as input the sending party $P_s \in [N]$, the set of recipient parties $R \in \{\mathbb{Z}^+\}$, server state $st_S \in \mathcal{S}_S$, state update $u \in \mathcal{U}$, group state $gs \in \mathcal{M}$, client/server epoch numbers $cc, cs \in \mathbb{Z}^+$, and report information $\rho \in \mathcal{E}$, and franking tag c_f. It outputs a bit $b \in \{0, 1\}$: 1 if the gs is determined to have been the group state at epoch cc resulting from update u and 0 otherwise.

This abstraction hides the details of the underlying CGKA mechanism for updating the shared secret. Our syntax also elides metadata, such as timestamps and group identifiers, which message franking systems often include within server-produced tags [1]. Observe that our treatment does not explicitly incorporate the threshold for the number of recipients, but rather has the Judge routine accept a set of recipient identifiers for clients that acknowledged successful reception of the update.

Security definitions. For security, we propose two accountability notions specifically tailored to our setting. The accountability definitions, in line with prior work on message franking, include sender and receiver binding notions that we refer to as reportability and integrity. We take inspiration from the channel-oriented definitions used by [44]. We restrict our attention to adversaries that provide well-formed inputs to the game oracles. If an input to an oracle is malformed, we assume the oracle outputs $\perp$. Our games and pseudocode specifications make use of the $\mathbf{assert}\ P$ macro for a boolean predicate P, which will cause the oracle or algorithm in which they appear to immediately return $\perp$ if P is false, similar to the behavior of the **require** macro in [27].

Correctness. We opt for a game-based definition of correctness in order to capture the stateful nature of our primitive. Our game captures the set of all valid interactions with the SES mechanism. The goal of the adversary is to attempt to induce an invalid state. We formalize correctness via a security game in Figure 4.2. The correctness advantage of an adversary $\mathcal{A}$ is defined as follows:

$$\mathrm{Adv}^{\text{ses-corr}}_{\mathsf{SES},N}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{ses-corr}}_{\mathsf{SES},N}(\mathcal{A}) = 1]$$

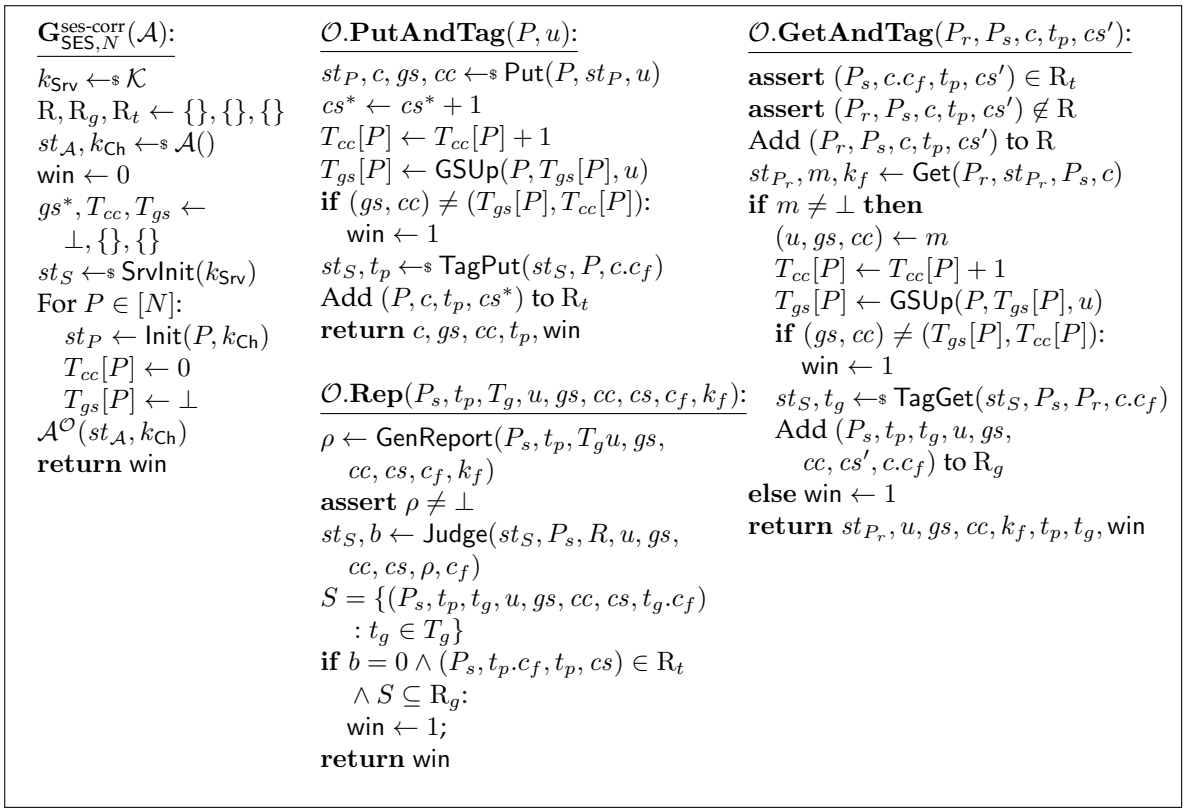

Figure 4.2: The security game for SES Correctness.

A correct SES scheme is one for which for all efficient adversaries $\mathcal{A}$, we have that $\mathrm{Adv}^{\text{ses-corr}}_{\mathsf{SES},N}(\mathcal{A}) = 0$ for all values $N \in \mathbb{Z}^+$. The game initializes a server key and allows the adversary to pick a channel key.

The adversary is provided with a **PutAndTag** oracle, which executes a Put operation with the specified update u, and then a TagPut operation over this update. Similarly, there is an oracle **GetAndTag** that executes a Get operation followed by a TagGet operation. Finally, the oracle **Rep** generates and judges a report for the specified inputs. The remaining security games follow a similar structure with slight variations reflecting the particular security properties they model.

Note that the adversary can win if it forces clients to obtain states that are

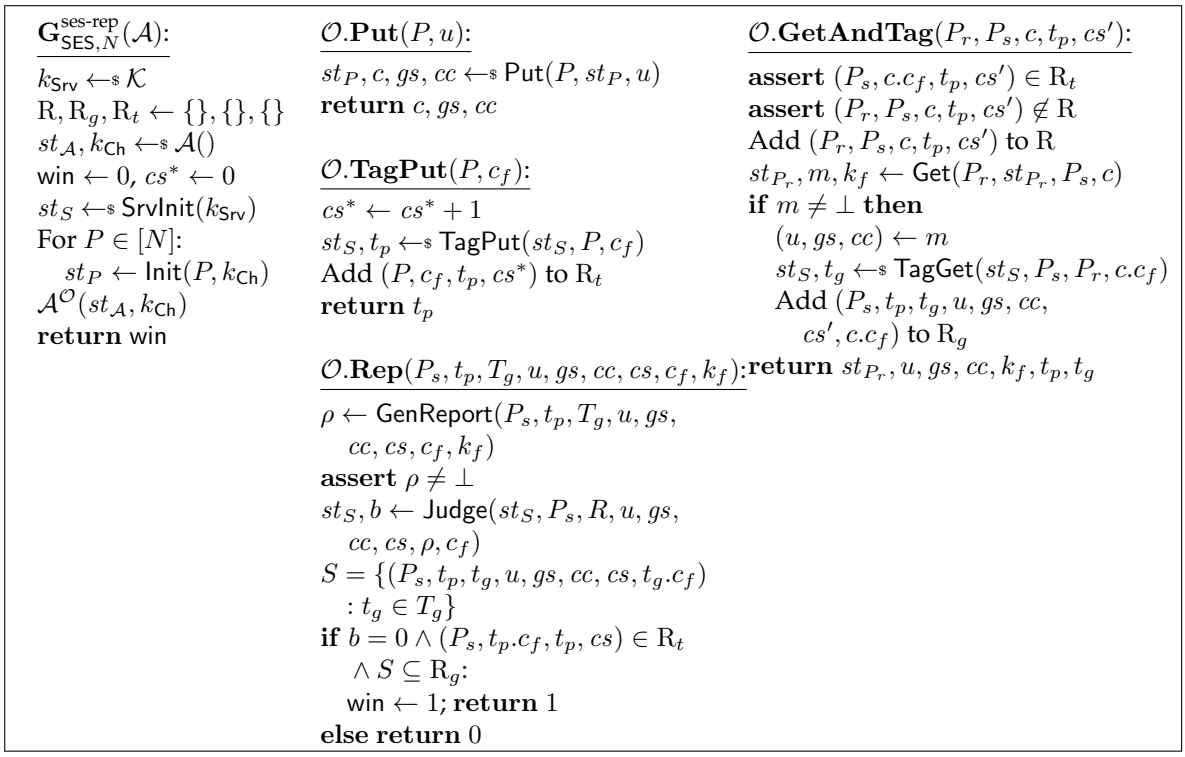

Figure 4.3: The security game for SES Reportability. This game is similar to the correctness game, however the main differences are that we model a malicious sender that does not have to execute valid Put operations.

inconsistent with the accounting the game performs using the tables T_{cc} and T_{gs}. Observe that all ciphertexts provided to the **GetAndTag** must have been the output of an honest Put operation from the **PutAndTag**, hence the adversary wins if a ciphertext does not properly decrypt.

Reportability. Our reportability definition is akin to what is often known as sender binding in the message franking literature. At a high-level, we require that all messages that successfully decrypt and verify for an honest client must be reportable to the judge, even for a malicious sender, in our case, a party performing a Put operation. We formalize this in the security game in Figure 4.3. The reportability advantage of an adversary is defined as follows:

$$\mathrm{Adv}^{\text{ses-rep}}_{\mathsf{SES},N}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{ses-rep}}_{\mathsf{SES},N}(\mathcal{A}) = 1]$$

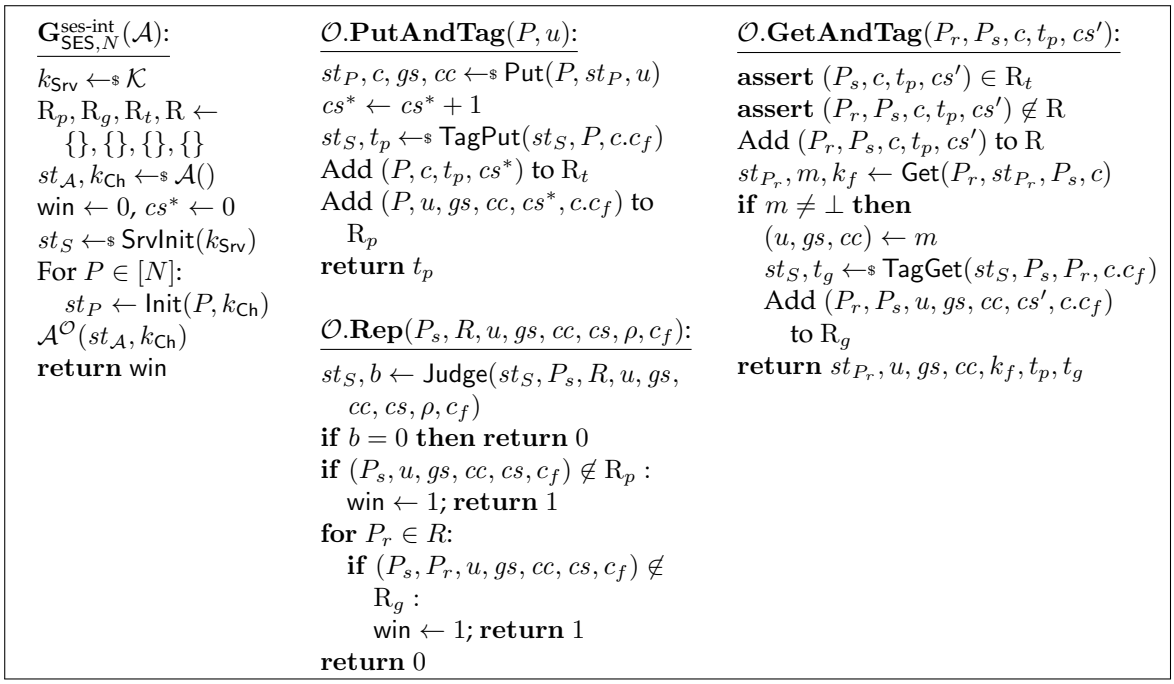

Figure 4.4: The security game for SES Integrity.

The adversary $\mathcal{A}$ wins the game if it is able to generate a report that should verify, according to the bookkeeping the game performs, but ends up failing to verify with the Judge procedure. The check happens in the **Rep** oracle. The set R_g accumulates all messages for which Get successfully executes. The adversary wins if it can produce report information ρ that does not validate for the Judge procedure, even though ρ is generated from valid tags.

Integrity. Integrity for SES franking is analogous to the receiver binding property in symmetric message franking. To provide integrity, it must be computationally infeasible for a malicious reporter to produce a never-tagged SES report that verifies for the judge. The corresponding SES integrity advantage is:

$$\mathrm{Adv}^{\text{ses-int}}_{\mathsf{SES},N}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{ses-int}}_{\mathsf{SES},N}(\mathcal{A}) = 1]$$

Formally, the adversary $\mathcal{A}$ has to produce report information ρ, such that

it verifies for the judge while deviating from the ground truth that the game maintains in its own bookkeeping. The set R_p keeps track of the state and update information associated with Put operations from the **PutAndTag** oracle. The set R_g keeps track of the state and update information successfully received by honest clients and is updated in the **GetAndTag** oracle. The adversary wins if it can successfully report state information that contradicts the accounting that R_p and R_g perform.

Confidentiality. Confidentiality requires that the franking tag does not reveal anything to the server about the contents of the underlying plaintext to get. This property is inherited in a straightforward manner from the hiding properties of the commitment scheme that will be used in our construction presented in Section 4.4. We omit a detailed security game for brevity.

Choosing a threshold. Our SES formalism allows flexibility for the number of Get tags required to establish the validity of a report. The choice of how many such tags are required depends on the context of a report. Consider the following example: Suppose the platform receives multiple reports about policy violations within a particular group. In order to intervene, the platform wishes to reach out to the current moderator(s) of the group. The identity of the group moderators is part of the SES, and so verifiably disclosing it would require a report. Preventing malicious users from impersonating as a moderator to the platform requires a sufficiently large (e.g., a majority) threshold. On the other hand, if someone wishes to report a moderator for changing the group guidelines to contain abusive language, it would suffice to include just one Get tag. The worst a malicious user could do is implicate themselves for having done something objectionable. Similarly, with encrypted profiles, typically just one

user (the owner of the profile) updates the shared state. Hence, a single Get tag suffices to report a user for issuing an update to their profile that goes against platform policies. Indeed, in order for an abusive client to be able to have their harmful content viewed by an honest client, the ciphertext and franking tag must be well-formed, otherwise the harmful content will not be displayed.

4.4 Our Construction

In this section, we outline our construction for SES franking. First, we discuss some intuitive straw proposals and point out their weaknesses in order to motivate our approach. We then specify our construction and then analyze its security using the definitions we established in Section 4.3.

Challenges with straw solutions. In Section 4.1, we outlined two straw solutions for maintaining reportable shared encrypted state. The first involved reporting all updates sent within the group to the moderator, allowing them to independently reconstruct the consolidated group state. The second entailed sending and franking the full group state every time it is updated. We now discuss in more detail why standard message franking approaches employed in these solutions fail to achieve our goals for SES franking.

The first solution relies on maintaining ordering integrity over the updates. Although there are approaches to message franking that enable reporting messages with such information [44], such approaches do not directly allow reporting which total ordering is agreed upon by all honest clients. This suggests that there must be some sort of acknowledgement mechanism so that clients can indicate which updates are accepted in the event of conflicting updates. Fur-

thermore, the number of updates can grow arbitrarily large, making reporting expensive over time, linear in the number of updates. In certain deployments with disappearing messages, group members may not even have access to all updates that led to a particular group state. This is also a burden on client-side storage.

Broadcasting the full group state appears somewhat better in terms of mitigating these issues, yet this causes the bandwidth to significantly increase for every update. Identifying valid group state broadcasts remains an issue with this solution, again suggesting the importance of acknowledgements of validity. We therefore consider a hybrid of these two solutions: broadcast individual group updates, but commit to the consolidated group state after applying this update. On top of this, we include a mechanism for clients to indicate that the update and commitment are valid.

Overview of our construction. We present our construction in Figure 4.5. The key insight is that the franking tag commits to the update u, the full group state after the update gs, and the client state counter cc. This way, update messages can be succinct, as opposed to having to convey the full group state, yet still commit to the full group state, enabling reporting. Meanwhile, the server tagging procedure incorporates a server state counter cs, which maintains a logical time that the server can use to tie standard content-bearing messages to particular versions of the group state. Furthermore, our use of aggregate MACs decreases the bandwidth required for reporting and reduces client-side tag storage requirements. See Section 4.6 for a discussion of the optimizations that tag aggregation affords.

Client procedures. The client initialization procedure initializes the channel via

a shared key. The shared state gs is initialized to $\perp$ while the client state epoch counter cc is initialized to 0. A Put operation takes as input an update u. The client commits to the update, the new group state, and a counter for the version associated with the group state.

A Get operation takes as input this ciphertext. Upon decryption, a client applies the update and verifies the commitment against the update, new group state, and new state counter. If this verification fails, the client reverts the group state to what it was before.

Server procedures. The server initialization procedure samples a key for computing MACs and initializes a counter cs to keep track of versions of the shared encrypted state. The server tags Put operations via TagPut, MAC-ing over the party that performed the Put along with the franking commitment c_f, and a server-side update counter cs. This server counter is necessary since the client counter may be out of sync due to malformed updates. Including a server counter additionally allows the server to link non-shared-state messages within a group to particular versions of the group state. The server counter also enforces a total order over updates while preventing replays. Once a client indicates successful reception and verification of a ciphertext, it prompts the server to initiate a TagGet, which additionally MACs over the receiving party. The output of a TagGet serves as a cryptographic acknowledgement of successful message reception.

Reporting. A client generates report information using the GenReport procedure, which XORs all the Get tags together along with the Put tag. In fact, this creates an aggregate MAC tag [73]. The report information includes the identifiers associated with the recipients that validate the committed state. Upon

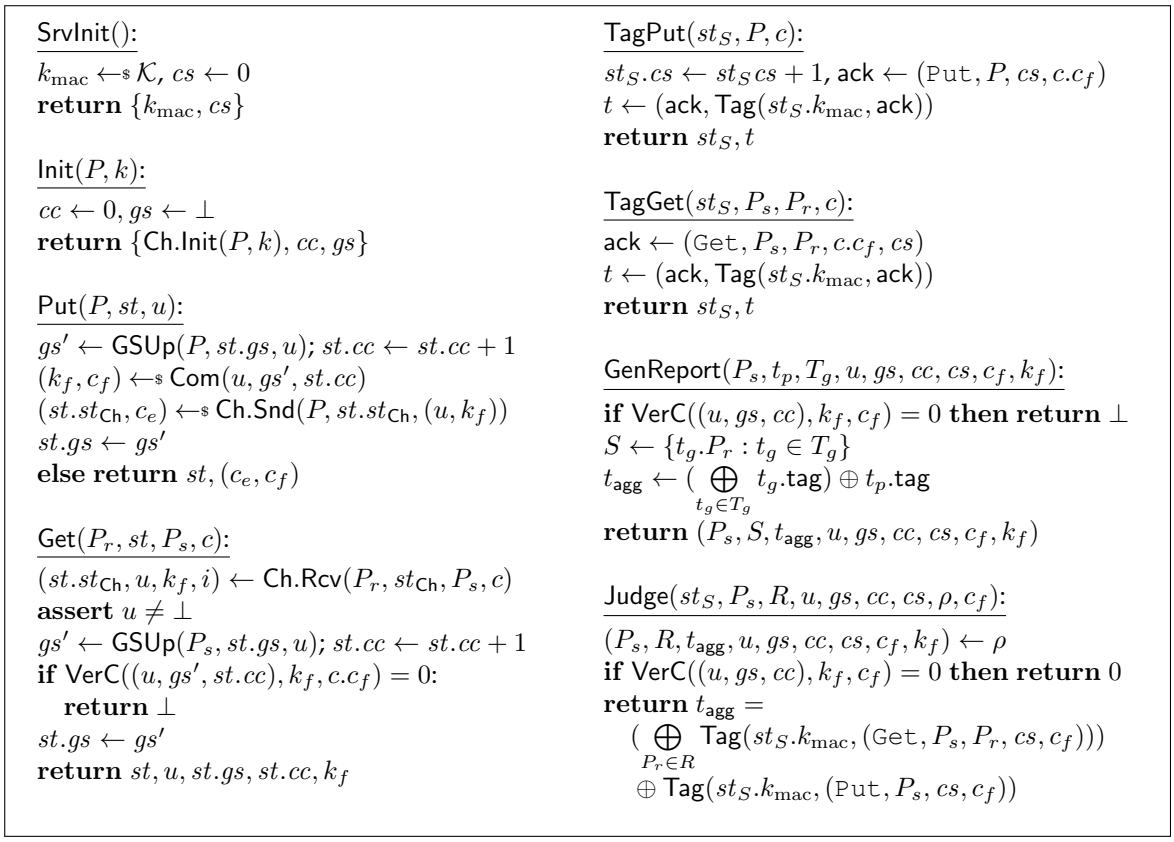

Figure 4.5: Pseudocode for our SES construction.

report verification, the server re-generates these tags for the identified recipients and XORs them together with a regenerated t_p, comparing the result to the provided aggregate MAC in the report information. The server also checks the franking tag commitment to ensure the validity of the reported triple of update, shared group state, and state counter.

Security analysis. We now show that our construction achieves reportability and integrity as defined in Section 4.3. Correctness of the scheme follows from the correctness of the underlying encrypted messaging channel, the aggregate MAC, and the commitment scheme, hence we omit a full security proof here. In the remainder of this section, we provide proofs of reportability and integrity for our SES construction SES^*.

Theorem 7. *Let* SES^* *be the construction in Figure 4.5 and let* $\mathcal{A}$ *be a reportability adversary. For all* $N \in \mathbb{Z}^+$*, we have that* $\mathrm{Adv}^{\text{ses-rep}}_{\mathsf{SES}^*,N}(\mathcal{A}) = 0$.

Proof. The checks performed in the Judge procedure mirror those performed in the Get procedure. Hence, if the commitment verifies for the client, it must also verify for the judge. □

Theorem 8. *Let* SES^* *be the construction in Figure 4.5 and let* $\mathcal{A}$ *be an integrity adversary. For all* $N \in \mathbb{Z}^+$ *we give an aggregate MAC EUFCMA adversary* $\mathcal{B}$*, and a commitment V-bind adversary* $\mathcal{C}$*, such that* $\mathrm{Adv}^{\text{ses-int}}_{\mathsf{SES}^*,N}(\mathcal{A}) \leq \mathrm{Adv}^{\text{agg-euf-cma}}_{\mathsf{AggMAC}}(\mathcal{B}) + \mathrm{Adv}^{\text{v-bind}}_{\mathsf{CS}}(\mathcal{C})$*, where* $\mathcal{B}$ *and* $\mathcal{C}$ *run in the time of* $\mathcal{A}$ *plus some small constant.*

Proof. We proceed via a sequence of game hops. The game G_0 is the SES Integrity game. Our goal is to bound $\Pr[G_0(\mathcal{A}) = 1]$. Our analysis considers two cases: the adversary wins by forging the aggregate tag t_{agg} or the adversary wins by opening the commitment c_f to a value other than what was originally committed. We define failure events F_1 and F_2 respectively corresponding to these scenarios.

More precisely, let F_1 denote the event that $\mathcal{A}$ wins and $(P_s, \cdot, \cdot, \cdot, cs, c_f) \notin \mathrm{R}_p$ or there is some $P_r \in R$ such that $(P_s, P_r, \cdot, \cdot, \cdot, cs, c_f) \notin \mathrm{R}_g$, where $(P_s, R, t_{\mathsf{agg}}, u, gs, cc, cs, c_f, k_f) = \rho$. Let F_2 denote the event that $\mathcal{A}$ wins without either of these events occurring.

We define G_1 to be the same as G_0, except an abort occurs if F_1 occurs. The game G_2 is the same as G_1, except an abort occurs if F_2 occurs. Observe that $\Pr[G_2(\mathcal{A}) = 1] = 0$ and therefore, by standard arguments that $\Pr[G_0] \leq \Pr[F_1] + \Pr[F_2]$.

We give an aggregate MAC EUF-CMA adversary $\mathcal{B}$ such that $\mathrm{Adv}^{\text{euf-cma}}(\mathcal{B}) \geq \Pr[F_1]$. The adversary $\mathcal{B}$ simulates the SES Integrity game to $\mathcal{A}$, routing calls to the MAC to its own challenger in the aggregate MAC EUF-CMA game. If F_1 occurs, then $\mathcal{A}$ produced a value that verifies, but was never queried to the MAC challenger oracle. Thus, $\mathcal{B}$ outputs

$$(\{(P_s, cs, c_f)\} \cup \{(P_s, P_r, cs, c_f) : P_r \in R\}, t_{\mathsf{agg}})$$

as a forgery, winning the game with probability $\Pr[F_1]$.

We give an adversary $\mathcal{C}$ such that $\mathrm{Adv}^{\text{v-bind}}(\mathcal{C}) \geq \Pr[F_2]$. Now it remains to show that F_2 must produce binding violation. The adversary $\mathcal{A}$ can only win by having Judge output 1 while $(P_s, u, gs, cc, cs, c_f) \notin \mathrm{R}_p$ or $(P_s, P_r, u, gs, cc, cs, c_f) \notin \mathrm{R}_g$ for some $P_r \in R$. Simultaneously, the aggregate MAC must verify for the set of messages $\{(P_s, P_r, cc, c_f) : P_r \in R\} \cup \{(P_s, cc, c_f)\}$ and c_f must open for the message (u, gs, cc). If the adversary wins without F_1 occurring (i.e. the event F_2), then by definition of F_1, we have that $(P_s, \cdot, \cdot, \cdot, cs, c_f) \in \mathrm{R}_p$ and for all $P_r \in R$ we have $(P_s, P_r, \cdot, \cdot, \cdot, cs, c_f) \in \mathrm{R}_g$. Let (u', gs', cc') be the tuple associated with c_f stored in R_p and R_g – by construction and by the way these sets are updated in the game, only one such tuple can exist. The adversary $\mathcal{C}$ can find this tuple by simulating the integrity game and noting the oracle arguments supplied by $\mathcal{A}$. Moreover, we must have that $(u', gs', cc') \neq (u, gs, cc)$ for the adversary to win. If F_2 occurs, then we must have that $\mathsf{VerC}((u, gs, cc), k_f, c_f) = 1$. This gives a collision $((u', gs', cc'), k'_f, (u, gs, cc), k_f, c_f)$. Thus, $\mathcal{C}$ wins with probability $\Pr[F_2]$.

□

4.5 Implementation and Evaluation

In order to show the practicality of our SES construction, we implemented a proof of concept in Rust and performed various microbenchmarks.

Implementation details. We chose Rust [21] for our implementation due to its high performance guarantees. For cryptographic operations we use AWS Libcrypto [15]. For serialization, we use the Serde crate [22]. The core implementation consists of 443 lines of code, as counted by the `cloc` utility [16].

Experimental setup. We performed microbenchmarks using the popular Criterion tool [64]. In particular, we measure the latency associated with various SES franking operations, and we report the bandwidth (communication complexity) of various objects of interest in our construction. All of our experiments ran on an AWS `t2.medium` EC2 instance, with 4 GiB of RAM and 2 vCPUs [14]. The shared group state we opted for in our benchmarks is a simple key value store. Updates to the group state consist of a key-value assignment. We test with workloads consisting of key value pairs where both keys and values are base 64 encodings of 32 random bytes. The main factors influencing the performance of the SES operations are the size of the state and the number of clients. We find that this setup allows us to exercise the core functionality of our SES franking construction and understand its scalability as a function of the size of the shared state and number of parties involved.

Put and get operations. The performance of Put and Get operations depends primarily on the number of operations within the group state. This is because the group state is part of the commitment which is generated in the Put oper-

ation and checked in the Get operation. With our setup, it is the only input to the commitment that is variable in size. Initially, the group state is empty for all clients. A single group member emits updates as other group members process them. For the purposes of our benchmarking, it suffices to measure the timing of the Get operation for a single client over repeated iterations. We present our measurements in Figure 4.6. Over 1024 Put operations, we measure a mean Put ciphertext size of 1255 bytes with a standard deviation of 15.3 bytes. This minor variability is due to the nature of our serialization approach with Serde [22].

Server tagging operations. Our tagging operations run in time independent of the number of group members, the size of the update, and the size of the group state. As such, we report on their latency directly. For the TagPut operation, we record a mean latency of 2117.5 ns with a standard deviation of 15.4 ns. The TagGet operation measures at a mean 1749.7 ns with a standard deviation of 15.3 ns. We compute these values from 100 Criterion samples. Note that an individual Criterion sample consists of multiple iterations of the benchmarked function to ensure accuracy.

Report creation and judging. The latency measurements for report creation and verification are shown in Figure 4.7. We observe that judging takes significantly longer due to the validation of the aggregate MAC requiring multiple MAC evaluations – one for each included Get tag. The generation of the aggregate tag simply requires XOR-ing the individual tags together, which is much faster.

Memory overhead of reporting. Our usage of aggregate MACs [73], greatly improves the bandwidth required for sending reports. A naive approach would require sending t MAC tags whereas our approach requires sending only one.

We show the bandwidth overhead associated with reporting in Figure 4.8 for a report of a group state after 1024 update operations. The size of the group state itself is roughly 92 KiB. We define the overhead as the difference between the size of the report object, which contains the group state, and the group state itself. This allows us to quantify how much bandwidth the cryptographic information incurs. In our measurements, we make use of Concise Binary Object Representation (CBOR) serialization [20]. We observe that the overhead grows linearly with the number of client get tags included within the report. This is due to the fact that the report includes a list of recipient identifiers. Although we can aggregate the MAC tag itself, we have to include this list to reconstruct the messages against which we are checking. Nonetheless, this overhead amounts to 3.1 KiB for a group of 1024 members. Furthermore, reporting is a somewhat infrequent operation, hence this overhead is not prohibitive.

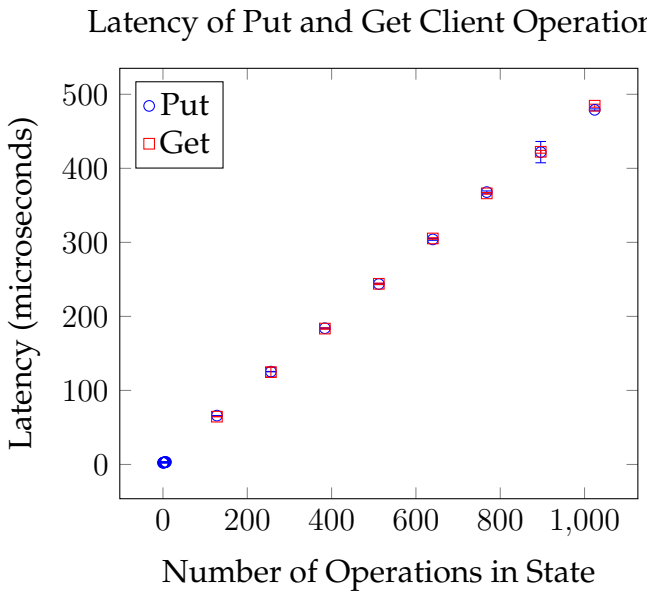

Figure 4.6: The latency of Put and Get operations. We report the mean values taken over 100 Criterion samples along with error bars showing one standard deviation.

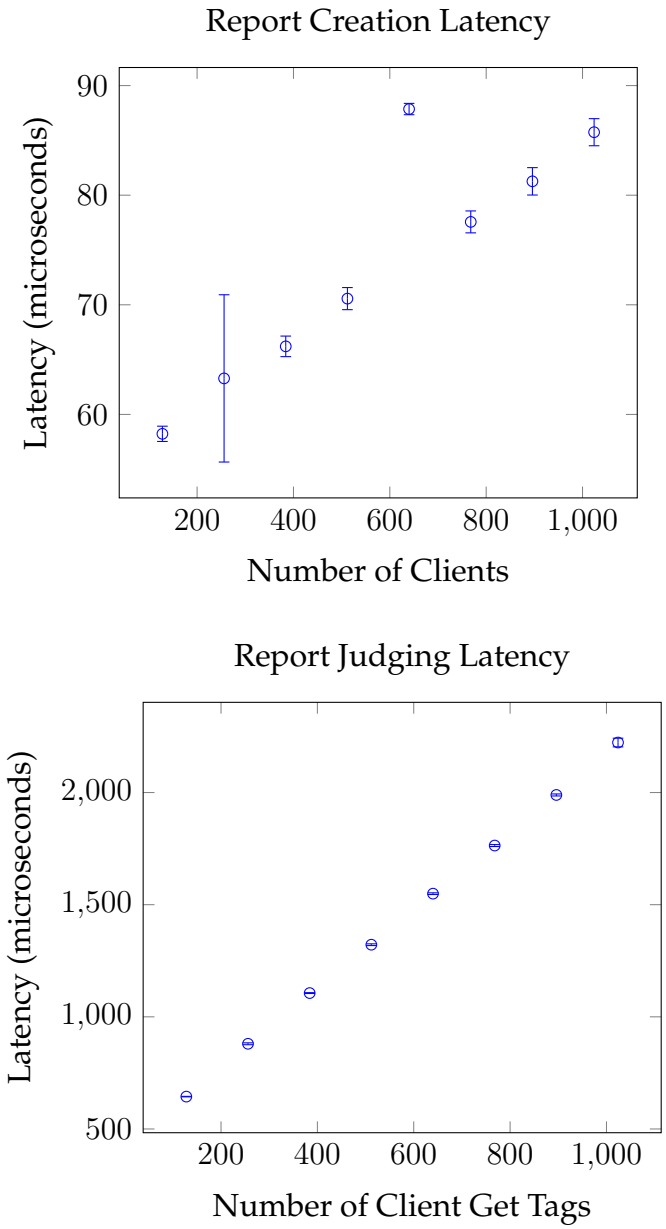

Figure 4.7: Latency associated with report creation and report judging. The plotted points show the mean measurements taken over 100 Criterion samples. The error bars show one standard deviation.

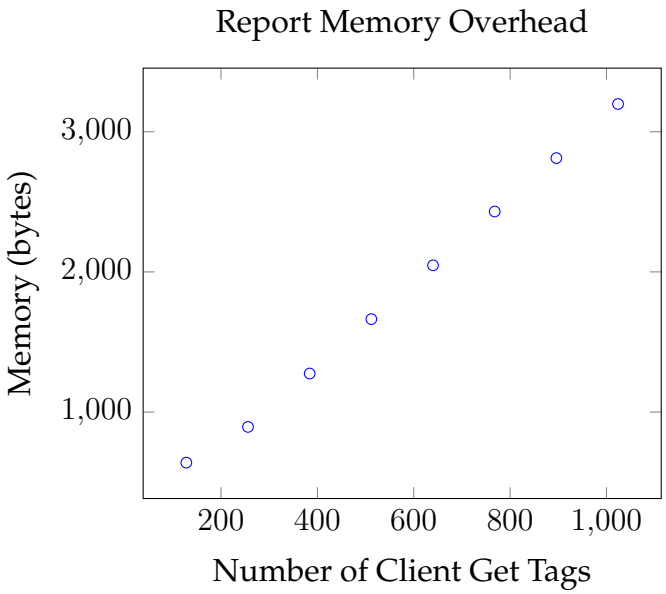

Figure 4.8: Memory overhead of cryptographically verifiable group state reporting, measured as the difference in the length of the encoding of the state and the length of the full report object with cryptographic material attached.

4.6 Discussion

In this section, we discuss deployment considerations for SES, extensions to our approach, and comment on limitations.

Integration with messaging. Our presentation provides an analysis of SES as an isolated component of an encrypted platform. We now discuss how it can integrate with a messaging platform, such as MlsGov [96], that supports shared state for groups. Recall that SES is reported with a client-side counter that indicates the version of the state and a server-side counter that keeps track of the number of updates. These counters may diverge in the event that clients submit invalid updates. The server counter can be included as metadata associated with non-shared-state messages in order to indicate the governance state at the time a message was sent. This way, abuse reports can include governance state

as associated context.

Selective opening. There exist message franking approaches that allow reporters to disclose portions of a message as opposed to the message in its entirety. For example, recent work has proposed constructions that enable revealing portions of a message at the block level that contain offensive content [81]. This approach, as well as other approaches based on vector commitments, such as Merkle trees, can be composed with our construction to allow the client to disclose parts of the state as opposed to the entirety of it. This has benefits for improving privacy and reducing the report size.

Optimizations. Our use of aggregate MACs enables server-side and client-side optimizations, beyond the benefits we have already discussed in saving report bandwidth. As clients receive Get tags, they can accumulate a running XOR for each message, noting which recipients have contributed the tags. This saves space over storing individual tags separately, and this approach is particularly beneficial for large groups. An interval tree can be used to allow clients to efficiently keep track of which group members have issued Get tags for which messages. One may also consider pruning reporting data for older messages. Indeed, this aligns with implementations of disappearing messages.

The server can similarly accumulate Get tags for messages as it prepares to deliver them to group members. When a group member receives a message, the server can send a running XOR of all Get tags at the time the message is being delivered. This saves on server storage as well as bandwidth.

Dynamic groups. To provide a streamlined presentation and analysis, we model a group messaging abstraction that handles static groups. That is, we

assume that the membership of a group is defined and fixed at its creation, and we do not explicitly model how members can be added to or removed from the group. We point out that extension to dynamic groups boils down to bootstrapping new members with the existing state of the group. A straightforward way to do this is to send newly invited members the current state of the group. Then, the inviter can issue a no-op update, one that does not change the state of the group, committing to the current shared state via our SES mechanism using c_f. Although the full state only needs to be sent to the new members, c_f is broadcast to all members via the platform. Therefore, producing a discrepancy between the state new members receive and the shared state maintained by existing members requires producing a binding violation for c_f.

Limitations. A limitation of our approach is that it requires a threshold number of Get tags to enable reporting of shared state, depending on the particular use case, as we discussed in Section 4.3. This leads to a potential lag in report verification for the platform. A requisite number of honest clients must come online to obtain these acknowledgements of state validity. In most scenarios, this is not a significant drawback, considering reporting is a highly asynchronous process. Moderators may not address a report long after it is submitted, at which point the appropriate Get tags have been accumulated. Moderators can also speculatively review and act on reports and roll back a decision if a state update is later flagged as malicious.

4.7 Related Work

E2EE reporting. Early work on E2EE reporting focused on *symmetric* message franking, a setting in which the moderation entity is the platform itself [1, 50, 63]. This is the setting that our work targets. Later, asymmetric message franking (AMF) considered metadata-private and third-party moderation settings [67, 113]. Follow-on work has also explored extending AMFs to the group messaging setting [78]. Beyond reporting messages, there has been a line of work on tracing the source of viral misinformation that is forwarded via E2EE messaging platforms [29, 67, 98, 115]

Private scanning. While reporting provides a reactive moderation mechanism, a line of work has explored cryptographic protocols that enable the platform to proactively scan for harmful content. Most prominently, Apple proposed a system, based on private set intersection, for privately scanning iCloud photos for child sexual abuse material (CSAM) [34]. Due to backlash from academics and civil society, the protocol was never deployed. Nonetheless, some academic proposals provide alternate system designs with better accountability and transparency [104, 111]. Although SES allows for proactive moderation at the group level, it does not tackle platform-driven proactive moderation.

Cryptographic group management. The Signal private groups paper details a mechanism for group-level access control in the sealed-sender setting [42]. Their approach makes use of anonymous credentials that allow privileged users to demonstrate they are authorized to perform membership management operations without revealing their identity. Administrative CGKA proposes an augmentation on CGKA protocols that supports access control for group ad-

ministrators [27]. Private hierarchical governance contributes a mechanism for broader group moderation policies for encrypted messaging groups [96]. SES builds off of this approach to provide a richer reporting functionality that supports disclosure of shared state and platform-verifiable detection of malicious clients.

Verifiable computation and zero-knowledge. As discussed in Section 4.1, one could use techniques from the verifiable computation literature [117] in order to enforce that only valid updates to the group state are performed. Recent work has proposed stateful anonymous credentials backed by zero-knowledge proofs [108]. Such techniques are computationally heavy, and hence would not work well in our setting to generalize to sophisticated shared state mechanisms, such as the policies supported by MlsGov [96], or for large-scale E2EE platforms. Our goal in this paper is to provide a lightweight solution based on simple and efficient cryptographic primitives to encourage streamlined adoption.

4.8 Conclusion

Our work initiates the study of shared encrypted state (SES) franking, motivated by applications in encrypted messaging and E2EE platforms broadly. We present security definitions and constructions for SES franking, in addition to an implementation and experimental evaluation, demonstrating the practical nature of our constructions. With the increasing prevalence of E2EE platforms that support novel rich features comes a growing reliance on shared encrypted state. This work provides a foundation for handling this state in a way that is compatible with content moderation and user safety. Future work may examine

how to realize SES franking in metadata-private scenarios and situations with relaxed state consistency requirements.

CHAPTER 5

SOCIETAL IMPLICATIONS

In this chapter, we discuss the benefits and challenges raised by the contributions we have detailed in the preceding chapters. In particular, we provide recommendations for how these contributions can be deployed in a way that best balances user safety, agency, and privacy for end-to-end encrypted platforms. While the results presented in this thesis focus on the technical infrastructure that enables enhanced safety features for E2EE platforms, we touch upon the human-related factors with which practitioners must grapple when implementing these solutions in the real world.

5.1 Private Hierarchical Governance

The focus of Chapter 2 is to provide tools to realize a broad range of governance policies, as opposed to prescribing specific instantiations of well-designed policies. These tools facilitate community moderation. Notably, however, these tools, and private hierarchical governance broadly, do not address groups in which all participants are bad actors. In a fully abusive group, no user would report harmful content to the platform. Private scanning approaches, such as the one proposed by Apple [34], are tailored for this use case, however they raise concerns regarding surveillance and censorship. Follow-on work has examined how transparency can be paired with these approaches to enable accountability on the part of the platform [104, 111], yet these efforts are orthogonal to the goal of providing community moderation tools to users, which is the goal of our work. Providing governance tools to users helps automate the work of community moderation, allowing local interventions without requiring action from the

platform, which is more supportive of user agency and privacy.

It may be unclear to users what policies are "good" in terms of promoting positive outcomes and protecting against autocratic takeovers of the community. Certain policies could, for instance, allow some group members to accumulate power and cement control of the group. Fields such as social choice theory and game theory may offer insights for policy design. There is also an opportunity for HCI work investigating how users choose and navigate different governance policies. In an actual deployment scenario, the platform should present users with a set of safe and well-designed default policies, along with an interface for customization. Works such as Pika [121] can serve as the foundation for usable policy design interfaces. Pika similarly provides a set of good defaults and identifies the exploration of "good" policies as future work.

5.2 Transcript Franking

Transcript franking enables more flexible reporting processes by allowing users to disclose sequences of messages within a conversation with strong integrity guarantees on the causal ordering over those messages. Although our constructions give users control over the exact context they disclose, there remain usability questions surrounding interfaces for disclosing messages and how to display and interpret message causal ordering. Further work will have to be done to determine appropriate designs for reporting procedures that allow the revelation of sufficient context so that moderators can make informed decisions, while minimizing the impact to user privacy. Our constructions allow for incremental disclosure, which allows users to add context in additional rounds,

which is useful for moderators who request additional information to interpret the report.

5.3 Shared Encrypted State Franking

Shared encrypted state franking is motivated by the need to provide more flexible reporting in the private hierarchical governance setting, in addition to accommodating broader forms of E2EE platforms and social media. This raises questions about what E2EE social media should look like and how user-driven reporting should fit within it. Most current E2EE platforms center messaging. As platforms expand in functionality, accommodating larger groups and facilitating user and content discovery, employing safety mechanisms becomes increasingly important. Harmful content can disseminate more rapidly and reach users with greater ease, relative to the messaging setting. Accordingly, E2EE platforms must set appropriate safeguards when adding new features in order to limit the spread of harmful content. Such safeguards may include limiting group sizes, mandating structured community moderation, and throttling content forwarding and dissemination. Prior work has discussed where and how content moderation should fit within the technology stack [51]. There is a general understanding that platforms are more responsible for content than cloud services, due to their role in distributing and amplifying content. As encrypted platforms grow in functionality, especially in their capacity to disseminate content, so too does their responsibility in engaging in moderation.

When using SES franking to report governance-related information in a group, there remain open questions regarding how platforms should take ac-

tion. For instance, if a report of the governance state reveals that a group is involved in activity that violates platform guidelines, it is unclear how the platform ought to intervene. Platform moderators can reach out to community moderators, whose identities are also contained within the governance state, and urge them to govern in accordance with platform policies. If group moderators fail to comply, the platform may sanction them, demote their status, or even remove them from the group. However, it is not obvious who would be appointed in their place and how governance in the group should proceed thereafter. Future work should investigate users preferences surrounding platform intervention in such cases.

CHAPTER 6

CONCLUSION

In addition to providing strong privacy protections for users, end-to-end encryption provides defense in depth. Even if a platform itself is compromised, user data remains protected. Still, users must be protected from online harms, which continue to grow, accelerated now by the advent of powerful AI tools, which can facilitate the creation and dissemination of harmful content. The expansion of such tools also raises concerns regarding mass surveillance and data collection. End-to-end encryption enables powerful privacy protections for users. It is therefore imperative, especially now, that digital platforms begin moving toward end-to-end encryption. Yet while doing so, they must provide users with robust protections. At a time when online harms and threats to user privacy are rising, developing trust and safety mechanisms for E2EE platforms is critical.

For end-to-end encrypted platforms, governance and enhanced reporting capabilities serve as a strong foundation for user safety. This thesis presents novel approaches toward furthering these goals. End-to-end encryption is rapidly spreading beyond the realm of messaging. I hope this work serves as a foundation for securing and building safety into new and emerging E2EE platforms.

BIBLIOGRAPHY

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

[2] Cryptpad: End-to-end encrypted collaboration suite, 2022. https://blog.cryptpad.org/images/whitepaper.pdf.

[3] Matrix specification. matrix.org, 2022. https://spec.matrix.org/v1.2/.

[4] mjolnir. github.com, 2022. https://github.com/matrix-org/mjolnir.

[5] Moderating on Discord. discord.com, 2022. https://discord.com/moderation.

[6] Moderation in matrix. matrix.org, 2022. https://matrix.org/docs/guides/moderation.

[7] OpenMLS. github.com, 2022. https://github.com/openmls/openmls.

[8] Reddit mods. reddithelp.com, 2022. https://mods.reddithelp.com/hc/en-us.

[9] An open-source implementation of the messaging layer security protocol. Web page, 2023. https://openmls.tech/.

[10] How do i know if my message was delivered or read?, 2024. https://support.signal.org/hc/en-us/articles/360007320751-How-do-I-know-if-my-message-was-delivered-or-read.

[11] How to block and report someone, 2024. https://faq.whatsapp.com/1142481766359885/?helpref=hc_fnav&cms_platform=web.

[12] How to check read receipts, 2024. https://faq.whatsapp.com/665923838265756/?helpref=uf_share.

[13] Whatsapp encryption overview, 2024. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[14] Amazon ec2 instance types, 2025. https://aws.amazon.com/ec2/instance-types/.

[15] Aws libcrypto for rust, 2025. https://github.com/aws/aws-lc-rs.

[16] cloc, 2025. https://github.com/AlDanial/cloc.

[17] Germ network, 2025. https://www.germnetwork.com/.

[18] Messaging layer security over activitypub, 2025. https://swicg.github.io/activitypub-e2ee/mls.

[19] Peergos, 2025. https://peergos.org/.

[20] Rfc 8949 concise binary object representation (cbor), 2025. https://www.rfc-editor.org/rfc/rfc8949.html.

[21] Rust, 2025. https://www.rust-lang.org/.

[22] Serde, 2025. https://serde.rs/.

[23] Harold Abelson, Ross Anderson, Steven M Bellovin, Josh Benaloh, Matt Blaze, Jon Callas, Whitfield Diffie, Susan Landau, Peter G Neumann, Ronald L Rivest, et al. Bugs in our pockets: the risks of client-side scanning. *Journal of Cybersecurity*, 10(1):tyad020, 2024.

[24] Harold Abelson, Ross Anderson, Steven M Bellovin, Josh Benaloh, Matt Blaze, Whitfield" Whit" Diffie, John Gilmore, Matthew Green, Susan Landau, Peter G Neumann, et al. Keys under doormats. *Communications of the ACM*, 58(10):24–26, 2015.

[25] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of mls. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1463–1483, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of mls. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 34–68, Cham, 2022. Springer Nature Switzerland.

[27] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1253–1270, Anaheim, CA, August 2023. USENIX Association.

[28] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The messaging layer security (mls) protocol, 2024. https://datatracker.ietf.org/doc/rfc9420/.

[29] James Bartusek, Sanjam Garg, Abhishek Jain, and Guru-Vamsi Policharla. End-to-end secure messaging with traceability only for illegal content. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 35–66, Cham, 2023. Springer Nature Switzerland.

[30] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 1–15, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[31] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 409–426, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[32] Benjamin Beurdouche, Eric Rescorla, Emad Omara, Srinivas Inguva, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture-10, Internet Engineering Task Force, December 2022. Work in Progress.

[33] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris, May 2018.

[34] Abhishek Bhowmick, Dan Boneh, Steve Myers, Kunal Talwar, and Karl Tarbe. The Apple PSI system, 2021. https://www.apple.com/child-safety/pdf/Apple_PSI_System_Security_Protocol_and_Analysis.pdf.

[35] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, ed-

itor, *Advances in Cryptology — EUROCRYPT 2003*, pages 416–432, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[36] Nikita Borisov, Ian Goldberg, and Eric A. Brewer. Off-the-record communication, or, why not to use PGP. In *WPES*, pages 77–84. ACM, 2004.

[37] Robyn Caplan. Content or context moderation? Data & Society Research Institute, 2018. `https://datasociety.net/library/content-or-context-moderation/`.

[38] Eshwar Chandrasekharan, Shagun Jhaver, Amy Bruckman, and Eric Gilbert. Quarantined! examining the effects of a community-wide moderation intervention on reddit. *ACM Trans. Comput.-Hum. Interact.*, 29(4), mar 2022.

[39] Eshwar Chandrasekharan, Umashanthi Pavalanathan, Anirudh Srinivasan, Adam Glynn, Jacob Eisenstein, and Eric Gilbert. You can't stay here: The efficacy of reddit's 2015 ban examined through hate speech. *Proceedings of the ACM on Human-Computer Interaction*, 1(CSCW):1–22, 2017.

[40] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[41] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less</> trust. In *CCS*, pages 1639–1656. ACM, 2019.

[42] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1445–1459, New York, NY, USA, 2020. Association for Computing Machinery.

[43] Brian Chen, Yevgeniy Dodis, Esha Ghosh, Eli Goldin, Balachandar Kesavan, Antonio Marcedone, and Merry Ember Mou. Rotatable zero knowledge sets. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022*, pages 547–580, Cham, 2022. Springer Nature Switzerland.

[44] Shan Chen and Marc Fischlin. Integrating causality in messaging channels. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology –*

EUROCRYPT 2024, pages 251–282, Cham, 2024. Springer Nature Switzerland.

[45] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *CCS*, pages 1802–1819. ACM, 2018.

[46] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE Symposium on Security and Privacy*, pages 321–338. IEEE Computer Society, 2015.

[47] Kate Crawford and Tarleton Gillespie. What is a flag for? social media reporting tools and the vocabulary of complaint. *New Media & Society*, 18(3):410–428, 2016.

[48] dalek cryptography. ed25519-dalek. GitHub, 2023. `https://github.com/dalek-cryptography/ed25519-dalek`.

[49] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, pages 303–320. USENIX, 2004.

[50] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 155–186, Cham, 2018. Springer International Publishing.

[51] Joan Donovan. Navigating the tech stack: When, where and how should we moderate content? *Centre for International Governance Innovation*, 2019.

[52] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE Symposium on Security and Privacy*, pages 332–346, 2012.

[53] Peter Elkind, Jack Gillum, and Craig Silverman. How facebook undermines privacy protections for its 2 billion whatsapp users. ProPublica, 2021. `https://www.propublica.org/article/how-facebook-undermines-privacy-protections-for-its-2-billion-whatsapp-users`.

[54] Saba Eskandarian. Abuse reporting for Metadata-Hiding communication

based on secret sharing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3205–3221, Philadelphia, PA, August 2024. USENIX Association.

[55] Patrick Eugster, Giorgia Azzurra Marson, and Bertram Poettering. A cryptographic look at multi-party channels. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 31–45, 2018.

[56] Hany Farid. An overview of perceptual hashing. *Journal of Online Trust and Safety*, 1(1), Oct. 2021.

[57] David Ferraiolo and Richard Kuhn. Role-based access controls. In *Proceedings of the 15th National Computer Security Conference*, 1992.

[58] R Stuart Geiger. Bot-based collective blocklists in twitter: the counterpublic moderation of harassment in a networked public space. *Information, Communication & Society*, 19(6):787–803, 2016.

[59] Tarleton Gillespie. *Custodians of the Internet: Platforms, content moderation, and the hidden decisions that shape social media*. Yale University Press, 2018.

[60] Tarleton Gillespie. Content moderation, AI, and the question of scale. *Big Data & Society*, 7(2):2053951720943234, 2020.

[61] James Grimmelmann. The virtues of moderation. *Yale JL & Tech.*, 17:42, 2015.

[62] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[63] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 66–97, Cham, 2017. Springer International Publishing.

[64] Brook Heisler. Criterion.rs documentation, 2025. `https://bheisler.github.io/criterion.rs/book/`.

[65] Loïs Huguenin-Dumittan and Iraklis Leontiadis. A message franking

channel. In Yu Yu and Moti Yung, editors, *Information Security and Cryptology*, pages 111–128, Cham, 2021. Springer International Publishing.

[66] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, page 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

[67] Rawane Issa, Nicolas Alhaddad, and Mayank Varia. Hecate: Abuse reporting in secure messengers with sealed sender. Cryptology ePrint Archive, Report 2021/1686, 2021. `https://ia.cr/2021/1686`.

[68] Shagun Jhaver, Iris Birman, Eric Gilbert, and Amy Bruckman. Human-machine collaboration for content regulation: The case of reddit automoderator. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 26(5):1–35, 2019.

[69] Shagun Jhaver, Seth Frey, and Amy X Zhang. Decentralizing platform power: A design space of multi-level governance in online social platforms. *Social Media+ Society*, 9(4):20563051231207857, 2023.

[70] Jialun Aaron Jiang, Charles Kiene, Skyler Middler, Jed R Brubaker, and Casey Fiesler. Moderation challenges in voice-based online communities on discord. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–23, 2019.

[71] Ari Juels, Dario Catalano, and Markus Jakobsson. *Coercion-Resistant Electronic Elections*, pages 37–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[72] Seny Kamara, Mallory Knodel, Emma Llansó, Greg Nojeim, Lucy Qin, Dhanaraj Thakur, and Caitlin Vogus. Outside looking in: Approaches to content moderation in end-to-end encrypted systems, 02 2022.

[73] Jonathan Katz and Andrew Y. Lindell. Aggregate message authentication codes. In Tal Malkin, editor, *Topics in Cryptology – CT-RSA 2008*, pages 155–169, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[74] Charles Kiene and Benjamin Mako Hill. Who uses bots? a statistical analysis of bot usage in moderation teams. In *Extended abstracts of the 2020 CHI conference on human factors in computing systems*, pages 1–8, 2020.

[75] Kate Klonick. The new governors: The people, rules, and processes governing online speech. *Harv. L. Rev.*, 131:1598, 2017.

[76] Mallory Knodel, Sofía Celi, Olaf Kolkman, and Gurshabad Grover. Definition of end-to-end encryption. Cryptology ePrint Archive, Paper 2024/2085, 2024. https://eprint.iacr.org/2024/2085.

[77] Anunay Kulshrestha and Jonathan Mayer. Identifying harmful media in End-to-End encrypted communication: Efficient private membership computation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 893–910. USENIX Association, August 2021.

[78] Junzuo Lai, Gongxian Zeng, Zhengan Huang, Siu Ming Yiu, Xin Mu, and Jian Weng. Asymmetric group message franking: Definitions and constructions. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 67–97, Cham, 2023. Springer Nature Switzerland.

[79] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[80] Julia Len, Melissa Chase, Esha Ghosh, Kim Laine, and Radames Cruz Moreno. OPTIKS: An optimized key transparency system. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4355–4372, Philadelphia, PA, August 2024. USENIX Association.

[81] Iraklis Leontiadis and Serge Vaudenay. Private message franking with after opening privacy. In *International Conference on Information and Communications Security*, pages 197–214. Springer, 2023.

[82] Ian Levy and Crispin Robinson. Thoughts on child safety on commodity platforms. *arXiv preprint arXiv:2207.09506*, 2022.

[83] Ya-Nan Li, Yaqing Song, Qiang Tang, and Moti Yung. End-to-end encrypted git services. Cryptology ePrint Archive, Paper 2025/1208, 2025.

[84] Junkai Liang, Daqi Hu, Pengfei Wu, Yunbo Yang, Qingni Shen, and Zhonghai Wu. SoK: Understanding zk-SNARKs: The gap between research and practice. Cryptology ePrint Archive, Paper 2025/172, 2025.

[85] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew K. Miller. Honeybadgermpc and asynchromix:

Practical asynchronous MPC and its application to anonymous communication. In *CCS*, pages 887–903. ACM, 2019.

[86] Joshua Lund. Technology preview: sealed sender for Signal, 2018.

[87] Caio Machado, Beatriz Kira, Vidya Narayanan, Bence Kollanyi, and Philip Howard. A study of misinformation in whatsapp groups with a focus on the brazilian presidential elections. In *Companion Proceedings of The 2019 World Wide Web Conference*, WWW '19, page 1013–1019, New York, NY, USA, 2019. Association for Computing Machinery.

[88] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean Lawlor. Parakeet: Practical key transparency for end-to-end encrypted messaging. Cryptology ePrint Archive, Paper 2023/081, 2023. `https://eprint.iacr.org/2023/081`.

[89] Moxie Marlinspike. Private group messaging, 2014. `https://signal.org/blog/private-groups/`.

[90] Moxie Marlinspike and Trevor Perrin. The X3DH key agreement protocol, 2016.

[91] Giorgia Azzurra Marson. *Real-World Aspects of Secure Channels: Fragmentation, Causality, and Forward Security*. PhD thesis, Technische Universität Darmstadt, Darmstadt, 2017.

[92] J Nathan Matias. The civic labor of volunteer moderators online. *Social Media+ Society*, 5(2):2056305119836778, 2019.

[93] David A. McGrew and John Viega. The security and performance of the galois/counter mode (GCM) of operation. In *INDOCRYPT*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2004.

[94] Sarah Meiklejohn, Pavel Kalinnikov, Cindy S. Lin, Martin Hutchinson, Gary Belvin, Mariana Raykova, and Al Cutter. Think global, act local: Gossip and client audits in verifiable data structures. *CoRR*, abs/2011.04551, 2020.

[95] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: bringing key transparency to end

users. In *USENIX Security Symposium*, pages 383–398. USENIX Association, 2015.

[96] Armin Namavari, Barry Wang, Sanketh Menda, Ben Nassi, Nirvan Tyagi, James Grimmelmann, Amy Zhang, and Thomas Ristenpart. Private Hierarchical Governance for Encrypted Messaging . In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2610–2629, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[97] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[98] Charlotte Peale, Saba Eskandarian, and Dan Boneh. Secure complaint-enabled source-tracking for encrypted messaging. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1484–1506, New York, NY, USA, 2021. Association for Computing Machinery.

[99] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[100] Riana Pfefferkorn. Content-oblivious trust and safety techniques: Results from a survey of online service providers. *Journal of Online Trust and Safety*, 1(2), Feb. 2022.

[101] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. Deniable authentication and key exchange. In *CCS*, pages 400–409. ACM, 2006.

[102] Sarah T Roberts. *Behind the screen*. Yale University Press, 2019.

[103] Michael Rosenberg, Mary Maller, and Ian Miers. Snarkblock: Federated anonymous blocklisting from hidden common input aggregate proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 948–965, 2022.

[104] S. Scheffler, A. Kulshrestha, and J. Mayer. Public verification for private hash matching. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 253–273, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[105] Sarah Scheffler and Jonathan Mayer. SoK: Content Moderation for End-to-End Encryption. *Proceedings on Privacy Enhancing Technologies*, 2023(2):403–429, 2023.

[106] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[107] Joseph Seering, Tony Wang, Jina Yoon, and Geoff Kaufman. Moderator engagement and community development in the age of algorithms. *New Media & Society*, 21(7):1417–1443, 2019.

[108] Maurice Shih, Michael Rosenberg, Hari Kailad, and Ian Miers. zk-promises: Anonymous moderation, reputation, and blocking from anonymous credentials with callbacks. Cryptology ePrint Archive, Paper 2024/1260, 2024.

[109] Nicolas P Suzor. *Lawless: The secret rules that govern our digital lives*. Cambridge University Press, 2019.

[110] Kurt Thomas, Devdatta Akhawe, Michael Bailey, Dan Boneh, Elie Bursztein, Sunny Consolvo, Nicola Dell, Zakir Durumeric, Patrick Gage Kelley, Deepak Kumar, Damon McCoy, Sarah Meiklejohn, Thomas Ristenpart, and Gianluca Stringhini, editors. *SoK: Hate, Harassment, and the Changing Landscape of Online Abuse*, 2021.

[111] Kurt Thomas, Sarah Meiklejohn, Michael A. Specter, Xiang Wang, Xavier Llorà, Stephan Somogyi, and David Kleidermacher. Robust, privacy-preserving, transparent, and auditable on-device blocklisting, 2023. `https://arxiv.org/abs/2304.02810`.

[112] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. Versa: Verifiable registries with efficient client audits from RSA authenticated dictionaries. In *CCS*, pages 2793–2807. ACM, 2022.

[113] Nirvan Tyagi, Paul Grubbs, Julia Len, Ian Miers, and Thomas Ristenpart. Asymmetric message franking: Content moderation for metadata-private end-to-end encryption. In *Advances in Cryptology - CRYPTO 2019*, volume 11694 of *Lecture Notes in Computer Science*, pages 222–250. Springer, 2019.

[114] Nirvan Tyagi, Julia Len, Ian Miers, and Thomas Ristenpart. Orca: Blocklisting in Sender-Anonymous messaging. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2299–2316, Boston, MA, August 2022. USENIX Association.

[115] Nirvan Tyagi, Ian Miers, and Thomas Ristenpart. Traceback for end-to-end encrypted messaging. In *Proceedings of the 2019 ACM SIGSAC Con-*

ference on Computer and Communications Security, CCS '19, page 413–430, New York, NY, USA, 2019. Association for Computing Machinery.

[116] Nik Unger and Ian Goldberg. Improved strongly deniable authenticated key exchanges for secure messaging. *Proc. Priv. Enhancing Technol.*, 2018(1):21–66, 2018.

[117] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *Theory of Cryptography*, pages 1–18, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[118] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP*, pages 137–152. ACM, 2015.

[119] Emily A. Vogels. Online harassment occurs most often on social media, but strikes in other places, too, 2021. https://www.pewresearch.org/short-reads/2021/02/16/online-harassment-occurs-most-often-on-social-media-but-strikes-in-other-places-too/.

[120] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. Treesync: Authenticated group management for messaging layer security. Cryptology ePrint Archive, Paper 2022/1732, 2022. https://eprint.iacr.org/2022/1732.

[121] Leijie Wang, Nicolas Vincent, Julija Rukanskaitė, and Amy X. Zhang. Pika: Empowering non-programmers to author executable governance policies in online communities, 2024.

[122] Leijie Wang, Ruotong Wang, Sterling Williams-Ceci, Sanketh Menda, and Amy X. Zhang. "is reporting worth the sacrifice of revealing what i've sent?": Privacy considerations when reporting on End-to-End encrypted platforms. In *Nineteenth Symposium on Usable Privacy and Security (SOUPS 2023)*, pages 491–508, Anaheim, CA, August 2023. USENIX Association.

[123] Sarah West. Raging against the machine: Network gatekeeping and collective action on social media platforms. *Media and Communication*, 5:28, 09 2017.

[124] WhatsApp. Two billion users – connecting the world privately. WhatsApp Blog, 2020. https://blog.whatsapp.com/two-billion-users-connecting-the-world-privately.

[125] WhatsApp. Sharing our vision for communities on whatsapp. Whatsapp Blog, 2022. https://blog.whatsapp.com/sharing-our-vision-for-communities-on-whatsapp.

[126] WhatsApp. About blocking and reporting contacts, 2023. https://faq.whatsapp.com/414631957536067.

[127] Amy X. Zhang, Grant Hugh, and Michael S. Bernstein. Policykit: Building governance in online communities. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 365–378, New York, NY, USA, 2020. Association for Computing Machinery.