Private Hierarchical Governance for Encrypted Messaging

Armin Namavari[1], Barry Wang[2], Sanketh Menda[1], Ben Nassi[1], Nirvan Tyagi[3,4]
James Grimmelmann[2], Amy Zhang[3], Thomas Ristenpart[1]
[1] Cornell Tech [2] Cornell University [3] University of Washington [4] Stanford University

***Abstract*—The increasing harms caused by hate, harassment, and other forms of abuse online have motivated major platforms to explore hierarchical governance. The idea is to allow communities to have designated members take on moderation and leadership duties; meanwhile, members can still escalate issues to the platform. But these promising approaches have only been explored in plaintext settings where community content is public to the platform. It is unclear how one can realize hierarchical governance in the huge and increasing number of online communities that utilize end-to-end encrypted (E2EE) messaging for privacy.**

We propose private hierarchical governance systems. These should enable similar levels of community governance as in plaintext settings, while maintaining cryptographic privacy of content and governance actions not reported to the platform. We design the first such system, taking a layered approach that adds governance logic on top of an encrypted messaging protocol; we show how an extension to the message layer security (MLS) protocol suffices for achieving a rich set of governance policies. Our approach allows developers to rapidly prototype new governance features, taking inspiration from a plaintext system called PolicyKit. We build a prototype E2EE messaging system called MlsGov that supports content-based community and platform moderation, elections of community moderators, votes to remove abusive users, and more.

1. Introduction

Today, end-to-end encryption (E2EE) helps protect the private communications of billions of people [74] from data breaches, overzealous advertisers, and snooping governments. At the same time, surveys [64, 70] 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 [60]. Content-oblivious approaches [57] do not handle most types of abuse, and those that allow reporting content [1, 28, 37, 38, 66, 76] rely on a centralized platform-operated moderation service that, in turn, relies on a combination of automated tools and large groups of human moderators [25, 35, 36, 44]. Such centralized moderation infrastructure represents a dangerous accumulation of power [63, 73] and struggles with nuanced, contextualized, or community-specific abuse [30].

Some plaintext platforms, like Reddit [7] and Discord [4, 41] instead employ a hierarchical governance structure. Certain community members, known as *community moderators*, define and enforce community policies, while moderators 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 [39]. 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 [40], and much work has explored the design of tools to support or enable community governance [33, 39, 43, 77] 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 [54, 56] or state-machine replication [61] 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) [13, 14] 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 [12]. We expect our extension to be of broad utility; here we show how we can build a governance layer distributed across clients in a group that agree on state via OAMs.

Our governance layer provides a full role-based access control (RBAC) [32] mechanism and a policy engine, inspired by prior work [77], 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 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.

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

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 [18, 34]. 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 [44, 59], 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 [50, 62]. However, the platform still retains the power to step in and enforce its rules, including banning accounts and removing or quarantining entire communities [19, 20], forming a two-level hierarchical governance structure [40].

Some community-reliant platforms provide a powerful set of tools to communities to automatically carry out policies [62]. For instance, subreddit moderators can use the Automoderator tool to automatically filter and act on posts based on their content [39]. Discord has a strong ecosystem of third-party tools that community moderators can use to customize their community [43]. More recently, academic work has explored richer governance approaches, where communities can define and execute arbitrarily complex governance procedures written in code [77], 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 [28, 37, 38, 66]. 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 [38, 55, 67].

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 [46]. Perceptual hashing [31] 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 multi-party computation can be used to navigate these challenges while respecting user privacy [16, 45]. Such proposals have been met with criticism owing to their potential to undermine the privacy goals of E2EE [9] 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 [2], a protocol for federated E2EE messaging, provides some basic community moderation features via role-based access control (RBAC) [5]. However, information about user roles is directly visible to the homeserver (platform server). Mjolnir [3] 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 [75]. 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 [22]. 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 [77].

The recently proposed A-CGKA construction [12] 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 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 1. We define governance privacy and integrity in detail Section 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.

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

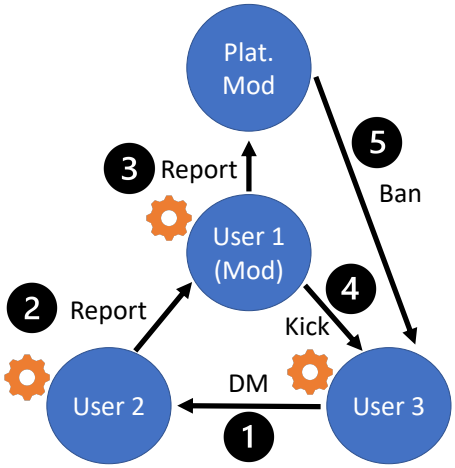

Figure 2: Diagram of an example governance scenario in which a user U_2 reports another user U_3 to a community moderator U_1, who kicks that user from the community and escalates by reporting the abuse to the platform's moderation systems. This results in a temporary ban of U_3 from the platform. Execution 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.

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.

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 actions 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 [40], 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 [14] 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 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 [77] 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 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 [21, 52, 53, 65]. In accordance with the MLS Architecture Document [14] §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., [14]), 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 [66], 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 [16] and others [45]. While automated detection might be necessary to detect abusive groups, privacy advocates raised serious concerns about their deployment (c.f., [9]). 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, 37, 66], 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 [72]. 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 [24, 27, 47, 48, 69]. 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 techniques 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 [17] and, relatedly, coercion resistance [42]. MLS does not provide deniability. But just like metadata privacy our system should be adaptable to a deniable encryption layer (e.g., [17, 49, 58, 68]), by replacing our use of non-deniable digital signatures with deniable message franking tools [66]. 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.

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 [13, 14] 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 [13] 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 A. 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 TreeKEM, 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 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 [13, §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 5 that is able to handle this scenario effectively. We empirically evaluate the performance of our approach in Section 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 layer) to underlying MLS message types. We follow as closely as possible the OpenMLS API [6]. 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 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.

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 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.

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 E.

Recall from Section 3 our example of a governance flow: (1) a user U_2 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.

Action	Description	Content State	Governance State	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 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.

Our governance layer supports a rich role-based access control (RBAC) [32] 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 [12].

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 prohibited 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 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 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 [77], 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 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 conversations 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.

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 [10, 11, 71], 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 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 [27] and in the MLS Architecture Specification [14] to establish confidence in the security of complex protocols. Additional discussion of out-of-scope attacks and security goals appears in Appendix B.

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 information 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.

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.

7.1. The Platform: MlsGov

We call our platform prototype MlsGov. We forked the Rust implementation OpenMLS [8] 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.

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

Action	Operation Latency & Traffic					Sync^s Latency & Traffic			
	Request Gen. (ms)	**Network Overhead** (ms)	**Post-Processing** (ms)	**Total Latency** (ms)	**Traffic** (KB)	**Network Overhead** (ms)	**Message 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	482.88	2.03	486.18	124.92
GovStateAnn.	0.58	258.08	0.12	259.02	9.62				
Accept^s	1.28	254.2	0.02	255.72	3.65	625.19	28.77	655.29	223.79
RenameGroup	10.18	350.0	0.13	360.68	44.91	404.32	1.88	407.35	56.7
VotePropose (Rename)	10.12	313.45	0.16	324.07	33.95	392.95	1.87	395.99	51.22
Vote^s (Rename)	0.37	255.58	0.05	256.25	3.89	720.08	37.65	759.04	378.59
Send (10-char Text)	0.40	257.13	0.08	257.87	3.54	371.85	0.63	373.71	37.32
Send (100-char Text)	0.40	257.4	0.08	258.13	3.86	360.62	0.60	362.38	37.65

Figure 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. Sync request generations take consistently < 0.01 ms. For results in a group of 1024, refer to Figure 7 in Appendix D.

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 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 3), and more.

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 D for additional details.

Microbenchmarks. We present both client latency and client-to-server bandwidth metrics for various operations in Figures 6a and 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 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 6a.

Governance state in group additions depends on group size, growing approximately linearly. This state is only

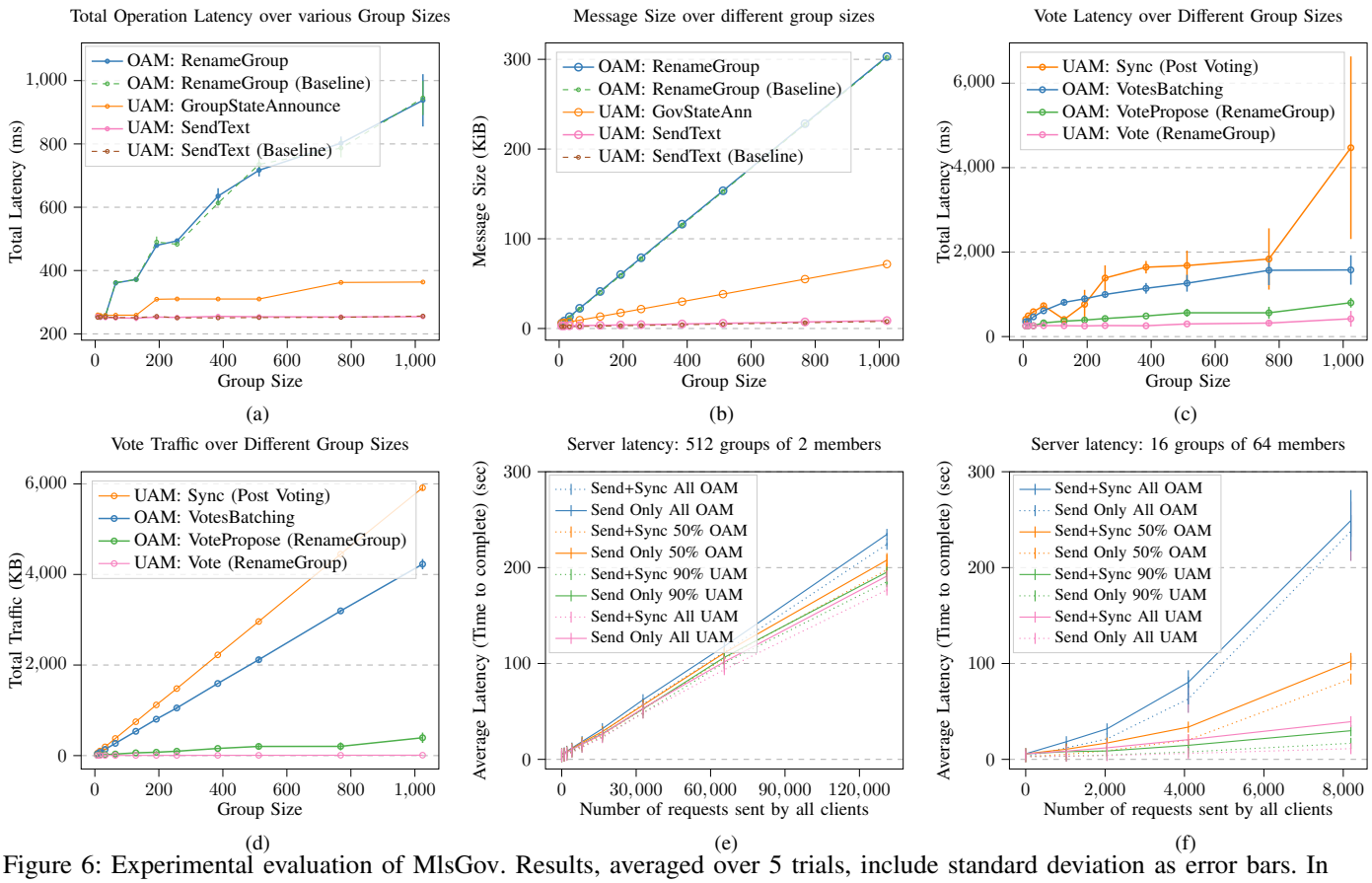

Figure 6: Experimental evaluation of MlsGov. Results, averaged over 5 trials, include standard deviation as error bars. In (a)(b), data is from the last starting client for multi-client operations. Only (e)(f) feature data from both servers and clients in US-East, instead of cross-regional – this was an optimization to allow for faster server benchmarks.

relayed in a GroupStateAnnouncement message when adding users and is depicted in Figure 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 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 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 6c and 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 D).

We report on our results in Figures 6e and 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.

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.

Acknowledgements

This work was funded in part by NSF grant CNS-2120651. We thank the reviewers and our shepherd for their helpful feedback.

References

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

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

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

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

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

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

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

[9] Hal Abelson, Ross Anderson, Steven M. Bellovin, Josh Benaloh, Matt Blaze, Jon Callas, Whitfield Diffie, Susan Landau, Peter G. Neumann, Ronald L. Rivest, Jeffrey I. Schiller, Bruce Schneier, Vanessa Teague, and Carmela Troncoso. Bugs in our pockets: The risks of client-side scanning, 2021.

[10] 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.

[11] 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.

[12] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. Cryptology ePrint Archive, Paper 2022/1411, 2022. https://eprint.iacr.org/2022/1411.

[13] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-17, Internet Engineering Task Force, December 2022. Work in Progress.

[14] 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.

[15] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: asynchronous decentralized key management for large dynamic groups a protocol proposal for Messaging Layer Security (MLS), 2018.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] Melissa Chase, Trevor Perrin, and Greg Zaverucha. *The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption*, page 1445–1459. Association for Computing Machinery, New York, NY, USA, 2020.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

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

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

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

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

[45] 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.

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

[47] 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.

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

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

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

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

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

[60] Sarah Scheffler and Jonathan R. Mayer. Sok: Content moderation for end-to-end encryption. *Proc. Priv. Enhancing Technol.*, 2023(2):403–429, 2023.

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

[62] 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.

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

[64] 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.

[65] 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.

[66] 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.

[67] Nirvan Tyagi, Ian Miers, and Thomas Ristenpart. Traceback for end-to-end encrypted messaging. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 413–430, New York, NY, USA, 2019. Association for Computing Machinery.

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

[69] 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.

[70] 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/.

[71] 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.

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

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

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

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

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

[77] 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.

Appendix A. MLS Background

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

MLS architecture. MLS [14] 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 [13].

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 [21, 52, 53, 65] 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 [13] provides a mechanism for group encrypted messaging. To do so, it uses the TreeKEM protocol [15, 23] 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. TreeKEM 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 [51]). 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 TreeKEM. 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.

Appendix B. 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 [29].

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 [66]. 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.

Appendix C.
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 [26] 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.

Appendix D.
Additional Experimental Results and Details

Additional results. We include additional experimental results for our system in Figure 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 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.

Appendix E.
Pseudocode Specification

We present a pseudocode specification of our governance protocol in Figure 8. In particular, we demonstrate how clients create and process messages within the system, including user reports. 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 [13] and OpenMLS documentation [6]. 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.

Action	Operation Latency & Traffic: Request Gen. (ms)	Network Delay (ms)	Post-Processing (ms)	Total Latency (ms)	Traffic (KB)	Sync Latency & Traffic: Network Delay (ms)	Message Processing (ms)	Total Latency (ms)	Traffic (KB)
Invite (1023 invitees)	$66.94^{(1.09)}$	$1170.27^{(90.75)}$	$200.25^{(0.17)}$	$1866.37^{(103.33)}$	$10254.38^{(2.85)}$	$3563.57^{(1948.93)}$	$20.10^{(0.83)}$	$3585.59^{(1542.01)}$	$1795.32^{(0.47)}$
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	$3.84^{(0.24)}$	$254.45^{(3.67)}$	$0.07^{(0.07)}$	$259.12^{(3.36)}$	$9.26^{(0.02)}$	$2535.59^{(1149.73)}$	$614428.41^{(51687.51)}$	$616969.1^{(51930.98)}$	$3529.61^{(0.64)}$
RenameGroup	$141.37^{(1.09)}$	$644.03^{(82.48)}$	$0.5^{(0.02)}$	$937.78^{(82.64)}$	$600.5^{(0.16)}$	$1476.81^{(266.87)}$	$8.21^{(2.59)}$	$1486.52^{(248.27)}$	$748.49^{(0.15)}$
VotePropose (Rename)	$94.4^{(29.66)}$	$574.26^{(74.46)}$	$0.85^{(0.05)}$	$801.44^{(114.64)}$	$394.29^{(113.98)}$	$1242.37^{(358.08)}$	$6.27^{(2.45)}$	$1250.14^{(264.72)}$	$645.38^{(56.95)}$
Vote (Rename)	$0.63^{(0.01)}$	$547.78^{(183.57)}$	$0.05^{(0.0)}$	$549.16^{(172.16)}$	$9.53^{(0.02)}$	$3606.64^{(620.86)}$	$602.40^{(26.95)}$	$4214.05^{(447.57)}$	$5913.89^{(56.18)}$
Send (10-char Text)	$0.72^{(0.03)}$	$271.94^{(1.28)}$	$0.08^{(0.01)}$	$273.83^{(1.24)}$	$9.17^{(0.02)}$	$613.35^{(43.62)}$	$0.62^{(0.04)}$	$615.59^{(40.18)}$	$454.1^{(0.08)}$
Send (100-char Text)	$0.65^{(0.03)}$	$257.31^{(0.62)}$	$0.08^{(0.0)}$	$259.01^{(0.54)}$	$9.47^{(0.02)}$	$593.48^{(9.72)}$	$0.65^{(0.02)}$	$595.85^{(9.37)}$	$454.38^{(0.09)}$

Figure 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.

$\underline{\mathsf{InitClient}(u) \to st_{\text{loc}}, st_{\text{mls}}:}$
$st_{\text{mls}} \leftarrow\!\!\$\ \mathsf{MLS.InitClient}()$
$pk_{\text{R}}, sk_{\text{R}} \leftarrow\!\!\$\ \mathsf{S.KeyGen}()$; store sk_{R} in st_{loc}
Post pk_{R} and KeyPackages for user u to AS
return $(st_{\text{loc}}, st_{\text{mls}})$

$\underline{\mathsf{CreateGroup}(st_{\text{loc}}, st_{\text{mls}}) \to (st'_{\text{mls}}, st'_{\text{loc}}, \mathsf{gid}):}$
Initialize default st_{gov}
Initialize empty content state st_{con}
Add entries $(\mathsf{gid}, st_{\text{gov}})$ and $(\mathsf{gid}, st_{\text{con}})$ to st_{loc}
return $(st_{\text{mls}}, st_{\text{loc}})$

$\underline{\mathsf{SendContentMsg}(st_{\text{mls}}, st_{\text{loc}}, m, \mathsf{gid}) \to (c, st'_{\text{mls}}, st'_{\text{loc}}):}$
$\sigma \leftarrow \mathsf{S.Sign}(sk_{\text{R}}, m)$
Append m to st_{con} for gid
return $\mathsf{MLS.SendUnordMsg}(st_{\text{mls}}, (m, \sigma), \mathsf{gid}), st_{\text{loc}}$

$\underline{\mathsf{SendGovMsg}(st_{\text{mls}}, st_{\text{loc}}, m, \mathsf{gid}) \to (c, st'_{\text{mls}}, st'_{\text{loc}}):}$
$\sigma \leftarrow \mathsf{S.Sign}(sk_{\text{R}}, m)$
Append m to st_{con} for gid
return $\mathsf{MLS.SendOrdMsg}(st_{\text{mls}}, (m, \sigma), \mathsf{gid}), st_{\text{loc}}$

$\underline{\mathsf{SendReport}(st_{\text{mls}}, st_{\text{loc}}, m, \mathsf{gid}) \to c:}$
Retrieve reporting signature σ for m
Append m to st_{con} for gid
return $\mathsf{MLS.SendUnordMsg}(st_{\text{mls}}, (\texttt{Report}, m, \sigma), \mathsf{gid})$

$\underline{\mathsf{RecvMsg}(st_{\text{mls}}, st_{\text{loc}}, c, \mathsf{gid}) \to (st'_{\text{mls}}, st'_{\text{loc}}):}$
$m, st_{\text{mls}} \leftarrow \mathsf{MLS.Recv}(st_{\text{mls}}, c, \mathsf{gid})$
Retrieve entries $(\mathsf{gid}, st_{\text{gov}})$, $(\mathsf{gid}, st_{\text{con}})$ from st_{loc}
$st_{\text{gov}}, st_{\text{con}} \leftarrow \mathsf{Execute}(P, m, st_{\text{gov}}, st_{\text{con}})$
Update entries $(\mathsf{gid}, st_{\text{gov}})$, $(\mathsf{gid}, st_{\text{con}})$
return $(st_{\text{mls}}, st_{\text{loc}})$

$\underline{\mathsf{Accept}(st_{\text{mls}}, st_{\text{loc}}, \mathsf{welcomeMsg}, c_{\text{gov}}) \to (st'_{\text{mls}}, st'_{\text{loc}}, c):}$
$st_{\text{mls}}, \mathsf{gid} \leftarrow \mathsf{MLS.JoinGroup}(st_{\text{mls}}, \mathsf{welcomeMsg})$
$st_{\text{gov}} \leftarrow \mathsf{MLS.Recv}(st_{\text{mls}}, c_{\text{gov}}, \mathsf{gid})$
Initialize empty content state st_{con}
Store entry $(\mathsf{gid}, st_{\text{gov}})$ and $(\mathsf{gid}, st_{\text{con}})$ to st_{loc}
$m \leftarrow (\texttt{Accept}, , H(st_{\text{gov}}))$
$c \leftarrow\!\!\$\ \mathsf{MLS.SendUnordMsg}(st_{\text{mls}}, (m, \mathsf{S.Sign}(sk_{\text{R}}, m)), \mathsf{gid})$
return $(st_{\text{mls}}, st_{\text{loc}}, c)$

$\underline{\mathsf{VerifyReport}(st_{\text{mls}}, st_{\text{loc}}, c, \mathsf{gid}) \to b:}$
$(\texttt{Report}, m, \sigma) \leftarrow \mathsf{MLS.Recv}(st_{\text{mls}}, c, \mathsf{gid})$
return $\mathsf{S.Verify}(pk_{\text{R}}, m, \sigma)$

$\underline{\mathsf{MLS.SendOrdMsg}(st_{\text{mls}}, m, \mathsf{gid}) \to st'_{\text{mls}}:}$
return $\mathsf{MLS.SendCommit}(st_{\text{mls}}, \mathsf{OrdAppMsgProp}(m, \mathsf{gid}))$

Figure 8: A pseudocode specification of our governance protocol.

Appendix F.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

F.1. Summary

This paper brings the ability for a group or community on an end-to-end encrypted platform to have moderators and rules that are internally enforced while maintaining the ability for the platform itself to have ultimate authority over moderation decisions. The proposed solution builds on top of the consensus mechanisms included in MLS. The authors use the existing MLS channels plus independent governance public keys associated to each group member to propose actions, and RBAC to determine if the group member is allowed to take such an action. Additionally, group members maintain governance state to ensure an attacker cannot fork governance actions by group members. The authors implement their proposal and report on the performance of their implementation. Their experiments indicate that their proposal scales reasonably well.

F.2. Scientific Contributions

- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field
- Establishes a New Research Direction

F.3. Reasons for Acceptance

1) The paper shows how to overcome a key impediment to large-scale deployments of encrypted social media. Governance for end-to-end encrypted messaging is a long known issue and this paper advances the discussion.
2) The implementation and evaluation demonstrate feasibility of the solution. This paper creates an open source implementation built on top of MLS that will be useful in understanding the capabilities that MLS can provide for governance actions

F.4. Noteworthy Concerns

While the paper considers various attack scenarios in its security evaluation, it does not formally analyze its contributions.