GLYPH Security Overview

Knight et al.

Sentriqs, Inc.

Savannah, Georgia, USA

Abstract

This paper is intended for a technical audience and provides sufficient discussion to assure outside parties that the security measures taken by our (Sentriqs') GLYPH platform infrastructure and software appropriately conceal and protect its users' data. It outlines the three major user-facing components of the GLYPH platform and the encryption protocols employed in each to achieve that end. It also discusses the techniques used for securely archiving communications and distributing data around the globe to meet potential data sovereignty regulations.

1 Introduction

GLYPH is a suite of productivity tools that facilitate secure collaboration for teams in corporate and governmental organizations. It was designed with a security-first mindset. The three fundamental components of the platform—messaging, file management, and video conferencing—are each secured using state-of-the-art authentication techniques, like passkeys and biometric authentication, along with novel and next-generation encryption techniques, like Messaging Layer Security (MLS). In the sections that follow, we outline the use of these techniques in each of these components, respectively.

GLYPH's security posture assumes the following to be true. The system is designed to remain secure even when these conditions are true:

- our server infrastructure is generally secure but will be compromised to some degree at some point in the future, consistent with industry-standard threat modeling assumptions;
- communication occurs over an untrusted network where:
  - any and all network communication can be intercepted and stored indefinitely and
  - standard TLS connections are proxied by malicious adversaries; and
- a sophisticated adversary may have computing resources capable of a sustained brute-force attempt against a decryption key over an extended period.

In section 2, we first outline the entities and actors defined in the GLYPH platform that represent various circles of trusted access and those that perform actions within them, respectively. In section 3, we outline the mechanisms by which we authenticate users, establish trust, and authorize requests made to the GLYPH server infrastructure. Because the server cannot be fully trusted, we implement a multi-party ends-to-ends encrypted communication protocol, called Messaging Layer Security (MLS)[3], outlined in section 4. Files shared through GLYPH are treated differently than text messaging, and section 5 discusses the protocol for securely storing them in the cloud while ensuring that only authorized end users are able to access them. Our implementation of secure video conferencing is discussed in section 6. Additionally, GLYPH seeks to support and facilitate the compliance requirements of our customers, many of whom must be able to furnish all corporate communications upon request or must ensure corporate data resides within specific jurisdictions. Section 7 outlines our internal tool for archiving client communication while maintaining privacy, client data ownership, and security guarantees. And section 8 touches upon the distributed architecture that enables data sovereignty compliance.

2 System Model

In this section, we define the actors and entities that interact within our system and that are germane to the discussion in later sections.

An *Account* in GLYPH is a collection of groups, some amount of storage, a series of video conferencing sessions, and a set of memberships. Each membership is a grant extended to a specific user, giving that user varying levels of access to the Account and its components. This can allow the account's members to add groups to or remove groups from the account; add, remove, or access files in the account's storage; begin, join, or end a video conferencing session in the account; and search for other members in the account.

A *Group* is a time-ordered collection of messages, a set of files, a set of ratchets, and a set of group memberships. It provides and maintains the cryptographic context that supports the secure transfer of information between users and their devices, which we will outline in greater detail in the sections 4 and 5.

Each *group membership* is (1) a grant extended to a specific user, providing a level of access to perform certain actions on a Group, and (2) a list of the user's devices currently participating in the Group. This access can allow a user on a participating device to post messages to the group, share files with or read files shared by other group members, add members to or remove members from the group, add and remove their own devices to and from the group, add and remove the devices of other members to and from the group, or start a video conference with the other group members.

Permission is assigned at the user level, but each device independently maintains a copy of the Group's state—most importantly its cryptographic state—in order to maintain an ends-to-ends encrypted channel among the participating devices of all users, and, thus, each device must be explicitly enrolled in the group to securely set up this cryptographic state.

Membership permissions, both at the account member and group member level, are maintained by, enforced by, and govern access to the GLYPH server infrastructure. Thus, in order to participate in the conversation of a particular group, a user must be properly permissioned in the account that owns the group and the group itself. The server infrastructure authorizes each request from a given user's device against these permissions and denies those that fail this check. This includes requests to forward along a given message to a particular Group.

3 Authentication and Authorization

Numerous multi-factor authentication options are available in GLYPH that challenge a person to prove they are who they're claiming to be. We briefly outline these options here since we generally conform to the standards and best practices of each. For a more in-depth discussion, see the references provided.

We require every user to choose a password conforming to typical formatting rules, and passwords are properly salted, peppered[18], and hashed with the Argon2 hashing function[5, 4]. Additional options are as follows:

- Passkeys[17], the user-facing name given to the WebAuthn protocol[11], are a promising and potential replacement for MFA altogether and provide the means of authenticating a user with a private key stored in a browser, phone, or specially-designed device[1];
- Time-based one-time passwords (OTPs)[13] are derived from a shared secret stored both on our server and in a device owned by the user; and
- YubiKey[12] provides users with a portable hardware device that can store both private keys for WebAuthn and shared secrets for OTPs.

Authenticated devices are issued JWT tokens for the logged-in user containing sufficient information to authorize subsequent requests made to the server. To conduct the transfer of these tokens from server to device, they are individually encrypted at the application level to avoid exposing them to any intermediate proxies between the server and device. For each issuance, a new key pair is chosen to encrypt the tokens to avoid replay attacks.

[1]The WebAuthn protocol is often implemented with modern biometric authentication, like Face ID, Touch ID, and Windows Hello, as a step in its flow on devices that offer those options.

3.1 Onboarding

The GLYPH system extends trust in a federated manner. Our engineers extend trust to our account management and customer service teams. Our team members extend trust to our clients' administrators. And those administrators extend trust to their organizations' members.

This is facilitated through an invitation mechanism involving *magic links*, which must be necessarily shared with new individuals through means other than GLYPH's secure communication channels. Magic links come in two flavors: (1) a one time consumable where the link is deactivated once it's used to create a new user account in the system, and (2) a multi-use link that allows for the creation of multiple user accounts with pre-scoped account access. Both flavors are time gated to expire after a configurable amount of time.

In some cases, we rely on the user's sole access to their email inbox or SMS SIM card as a means of authentication. In typical fashion, a magic link is sent to these locations with further instructions, and we trust that only the intended user would have access to these locations, such that only they would then be able to continue with an operation.

Alternatively, we also leverage novel, near-field communication hardware provided by modern smartphones to allow privileged existing users to transmit a magic link invite in-person, directly phone-to-phone, which greatly reduces the risk of the link being intercepted.

4 Messaging

In this section, we outline our steps to ensure private, text-based communication. Text communication in GLYPH is organized around a Group, which is owned by an Account and has a set of members, who are authorized to participate in the group's communication and who elect to participate in the group on one or more devices. The state of this membership, i.e., inclusion in a particular group, is stored both on our server infrastructure and with each participating device. The server infrastructure references this state when deciding to permit group operations and admit messages for transmission. And devices reference this state when deciding to trust or decode a given group message.

Each participating device—and only participating devices—also stores the necessary cryptographic information to conduct our proprietary implementation of an ends-to-ends encryption scheme known as Messaging Layer Security (MLS)[3], which is an iteration on Asynchronous Ratcheting Trees[6], improved upon in [1], and was standardized in RFC 9420[2] (well after we released our implementation). In utilizing this scheme, group members mitigate the risks of an untrusted network and a server infrastructure that may one day be compromised.

4.1 Background

Here, we outline some key components of modern secure group communication that are germane to a comparison of MLS with its alternatives and motivate its use in GLYPH.

4.1.1 Ratchets. A *ratchet* is a cryptographic mechanism that produces a sequence of deterministic secrets over time using some *state* and a *key derivation function (KDF)*, where the state is a cryptographically difficult to guess secret that is valid input for the KDF. At a given step in the sequence, the KDF is applied to the state at least twice: once to produce a new state value and once to produce a secret to be used when encrypting some package of data. This is colloquially referred to as a "turn" of the ratchet.

The KDF imparts an important property to the sequence of secrets in that we can reasonably assume no function exists that is capable of computing the input state from either of the outputs. That is to say, if one secret in sequence becomes compromised, no prior secret in the sequence can be compromised. But subsequent secrets in sequence can be computed if the KDF is known. This property then extends to data encrypted by each secret in the sequence, and it's typically referred to as Forward Secrecy (FS).

Without employing ratchets, two entities intending to securely communicate would first employ a public key encryption algorithm to effectively share a secret amongst themselves that no other entity could intercept. This secret would then be used repeatedly to encrypt each message that the two sent to one another. For example, if one entity is a web browser and another is a web server, the two would engage this two step process to establish a short-lived but secure session[2], and the messages traded between the two would be requests for and contents of web assets needed to render a given web page. And the secret they share is discarded once the session is terminated.

If the entities are two people conducting a chat session, the secret they share may be reused for days, weeks, or years.

Assuming an attacker has access to every message sent between the two entities, there are two means of reading the content contained in those messages: (1) breaking the public key encryption that was used to share the secret, which reveals the secret to them, or (2) guessing the secret itself. And, once the work is done to discover the secret, every message in the session is compromised.

Ratchets improve upon this by using the secret shared through the public key encryption technique as the initial state for a ratchet, and both entities maintain a copy of that ratchet and its state. Then, for each message to be sent, the ratchet is turned, a new secret is produced, and that secret is used to encrypt the message. The recipient can then turn its copy of the ratchet to produce the same secret and decrypt the message.

[2]Formally defined by the Transport Layer Security (TLS) protocol

Because a single ratchet shared amongst the two entities can lead to issues when both entities try to send at the same time using the same secret in the sequence, a *double ratchet protocol* assigns a separate ratchet to each entity, where both entities store copies of both ratchets and where one entity turns its assigned ratchet to encrypt its outgoing messages and turns the other ratchet only when receiving and decrypting an incoming message.

With this mechanism in place, another important property emerges, called *Post-Compromise Security* (PCS)[7], which allows the two entities to reset the states of their ratchets over time through another round of applying the public key encryption process to share a new pair of secrets between the two, which, when applied through the KDF, creates an entirely different sequence of future secrets. Assuming this reinitialization can be done securely, this essentially heals a compromised session from a breach by an outside attacker. Thus, at the moment of a breach, where an attacker discovers one secret in the sequence, Forward Secrecy ensures the privacy of the session for all messages prior to the breach, and Post-Compromise Security ensures the privacy of the session after the reinitialization is performed.

4.1.2 Multi-party Communication. The double ratchet protocol in the discussion above can naturally be extended to more than two parties, i.e., the formation of a group. Each member of the group is assigned a ratchet and must also store a copy of the ratchets assigned to every other member of the group. That member will use its ratchet when encrypting an outgoing message and the copies of other ratchets when decrypting an incoming message. This basic setup applies to MLS or its more problematic alternatives.

To illustrate the problems of those alternatives, we must first discuss how public key cryptography works in more detail for those unfamiliar with the process. Typically, before any private communication can be achieved, an entity, Alice, must create a public and private key pair using a well-known algorithm, e.g., an Elliptic Curve algorithm, like secp256k1, or a Module Lattice algorithm, like ML-KEM-1024. And the public key must be published or shared in a location to which other entities have access. Some other entity, Bob, seeking to communicate with Alice, will acquire her public key, perform a computation in the well-known algorithm using that public key along with a private key of Bob's choosing, and produce at least two outputs: (1) an encrypted message that can be openly sent to Alice and (2) the secret the two will share. Upon receiving it, Alice feeds that message and the private key corresponding to the public key she previously published through the algorithm to produce the same secret that Bob computed for himself.

The key feature here is that, being in possession of her private key, only Alice can perform the computation to arrive at the secret that Bob also arrived at. Where any discussion about maintaining one or more secrets across multiple people or devices inevitably grapples with the problem of securely introducing that secret shared state to a new member of the

group, it almost universally boils down to this initial starting condition of Alice creating a key pair and keeping the private key to herself to ensure her privacy and security as well as that of the system. And, thus, initial communication with Alice must be encrypted exclusively and individually for her. Technology does not currently exist that allows us to mix multiple public keys within a single encryption step such that any of the corresponding set of private keys are solely capable of decrypting the result[3].

Public key cryptography is invoked in our group communication scenario with ratchets when the group changes. When a new member joins, a new ratchet is created for the member with its initial state, and that state must be distributed to every existing member of the group so that their copy of the ratchet can be created for the new member as well. Similarly, removing a member—such that the group ensures the departing member no longer possesses the secrets to decrypt future messages—is akin to an event that compromises the group's secrets, in which case every copy of every ratchet must be discarded and a new one created. In these cases, one or more members of the group do not have a secure channel over which to transmit the secrets needed to initialize or sync the copies of their ratchets and thus must fall back upon another mechanism, which is usually an exchange using public key cryptography as described above.

4.1.3 Sender Keys. Where there are several ways a group might set up ratchets amongst its members, the most efficient, other than MLS, is called *Sender Keys*. As described above, each member of the group is assigned a ratchet to be used when encrypting an outgoing message and maintains copies of other members' ratchets to be used when decrypting inbound messages from them, respectively. Additionally, each member is required to publish a public key as well.

When a new member joins the group, they obtain the public key for every existing group member, create their ratchet, and encrypt its state multiple times, using each public key in turn to build and send a separate message for each group member, which only that group member can read.

When reinitializing the entire group due to a departing member or a compromise, each existing or remaining group member must perform this action, too, as if they are joining the group anew. Thus every member obtains every other member's public keys, recreates their ratchet state, and then builds and sends a message for each other member of the group.

This performs well enough for small group sizes. But, when group sizes become large, these become costly and time consuming operations to perform.

[3]in unrelated schemes, a single message is often encrypted with one secret key, and that secret key is encrypted multiple times, once using the public key of each intended recipient of a message

4.2 Messaging Layer Security

MLS improves on Sender Keys by arranging the collection of ratchets into a tree structure, specifically a left-balanced binary tree where the bottom-most leaf nodes of the tree represent the members of the group and store the group state associated with that member, which includes a public key for that member and their ratchet state, among other bookkeeping values as depicted in figure 1.[4]

The interior nodes of the tree (non-leaf nodes) store additional public and private keys to be used when an event occurs requiring the group to change, e.g., a member being added or removed.

Each member of the group maintains a copy of the tree along with some other group secrets, most notably a public/private key pair that new members may use to join the group, called the *external join key*. However, the information contained in these copies differs slightly from member to member. The private keys associated with the interior nodes are not known to every member of the group. Instead, a given member only stores the private keys in its copy of the ratchet tree for the nodes that sit along the path from its corresponding leaf up to the root of the tree.

We refer to this sequence of nodes, running from a member's leaf up to the root, as that member's *direct path*, and to the siblings of those nodes as its *copath*. The operations that manipulate the tree are what furnish MLS with its decisive advantage over Sender Keys. The essential observation is that the only secrets a member must replace in order to rotate the group key are those lying along its own direct path. The remainder of the tree is left untouched.

When a member initiates a change—whether to add a device, remove one, or simply heal the group following a suspected compromise—it performs an *update*. It creates a fresh secret at random, termed a *path secret*, for the lowest node on its direct path, the node associated with that member. From this secret, it derives a public/private key pair to associate with this node of the tree. For the immediate parent of this node, the process repeats, but the path secret for this parent is derived from the earlier child path secret using a KDF. This new path secret for the parent is then used to derive a public/private key pair to assign to the parent node. And this repeats again, ascending up to the root of the tree. In this way, the key pair assigned to each node in the direct path is fixed deterministically by the one beneath it and the topmost yields a new root secret held by the entire group. Because this derivation proceeds in one direction only, possession of a node's path secret confers every parent's secret while revealing nothing of its descendants.

By conveying this collection of path secrets to them, the other group members are able to derive the new key pairs and assign them to the corresponding nodes in their copies of the ratchet tree as well as derive the root secret at the top of

[4]Some implementations may store separate trees for the public keys and for the ratchet secrets

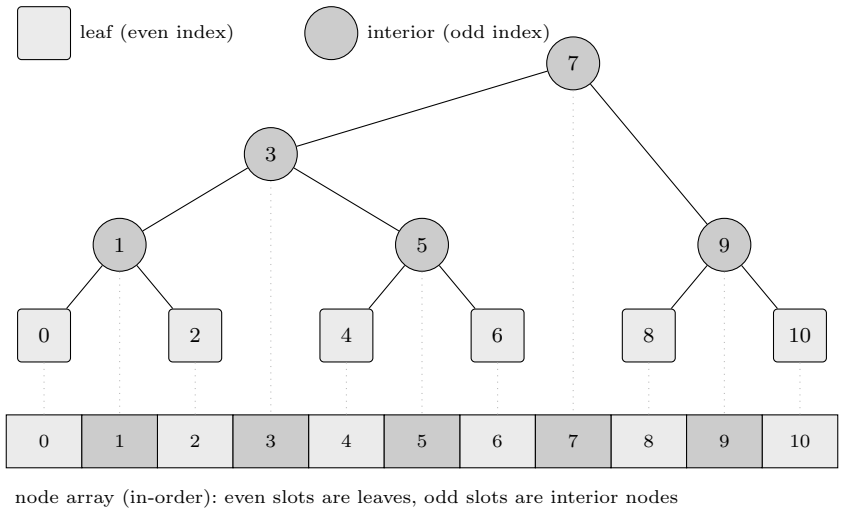

Figure 1: A left-balanced ratchet tree for a six-member group, each node labeled by its position in the underlying array. The tree is not perfectly balanced: its left subtree is filled and the remaining members hang lower on the right, giving the tree its characteristic left-heavy shape. Numbering the nodes from left to right in the order they are encountered places every leaf at an even index and every interior node at an odd one, which lets the whole structure, together with each node's parent and children, be stored and navigated as the flat array shown beneath.

the tree. To accomplish this, a key observation is that each of the nodes in the direct path has a sibling node in the tree where the public key assigned to that node does not change as the update proceeds. Thus, the acting member can utilize this set of public keys to encrypt the path secrets before broadcasting the update, as depicted in figure 2.

Because the corresponding private keys to the public keys used in this update are only known to the group members whose assigned nodes reside below the node holding a given public key, only those members can decrypt the corresponding path secret. And all members can decrypt at least one path secret. Then, with those path secrets, the members can derive the same public/private key pairs to assign to the respective interior nodes to which the path secrets correspond, bringing their tree state up to date with the member that initiated the update and maintaining the invariant that they retain the private keys for the nodes in their own direct paths.

At this point, then, all members of the group are able to derive the root secret at the top of the tree. This root secret is the input needed to derive what is called the *key schedule* for the group, which is a set of other keys and secrets the group needs and which includes the public/private key pair that external entities can use to join the group unilaterally.

One of these derived secrets is used to initiate the process of reinitializing the ratchets for every member of the group. Here, the KDF is employed, starting at the root of the tree, the initial secret is passed through the KDF twice to derive a secret for the left child and the right child of the tree. Then each of those secrets is passed through the KDF to derive secrets for the left and right grandchildren. This cascades repeatedly down the tree until disparate secrets are derived for each of the children in the tree, and these secrets are the new starting state values for the ratchets held by the group.

Previously, to reinitialize every ratchet held by the group, sender keys required every member to create its own secret, individually encrypt it for every other member of the group, and individually transmit it to each member of the group. MLS reduces the number of encrypted secrets to those in the update path, and it requires only a single message be broadcast to the group, i.e., multiple copies of the same message fanned out to group members. This amounts to an enormous savings and enables very large group sizes suitable for enterprise environments.

4.2.1 Joining a Group. Outside entities wishing to join a group require that groups publish a public copy of the ratchet tree containing the public keys associated with the nodes of the tree along with the public key specifically reserved for external joiners to use.

Joining a group mirrors an existing member adding a new member to a group. In both cases, new nodes are added to the tree by appending one interior node followed by the new leaf node to the tree where the public key associated with the leaf node is set to the public key previously published by the new group member. For list or array implementations of the tree, this amounts to appending to the end of the list or array. The tree structure is updated accordingly. And the acting member performs an update as described above.

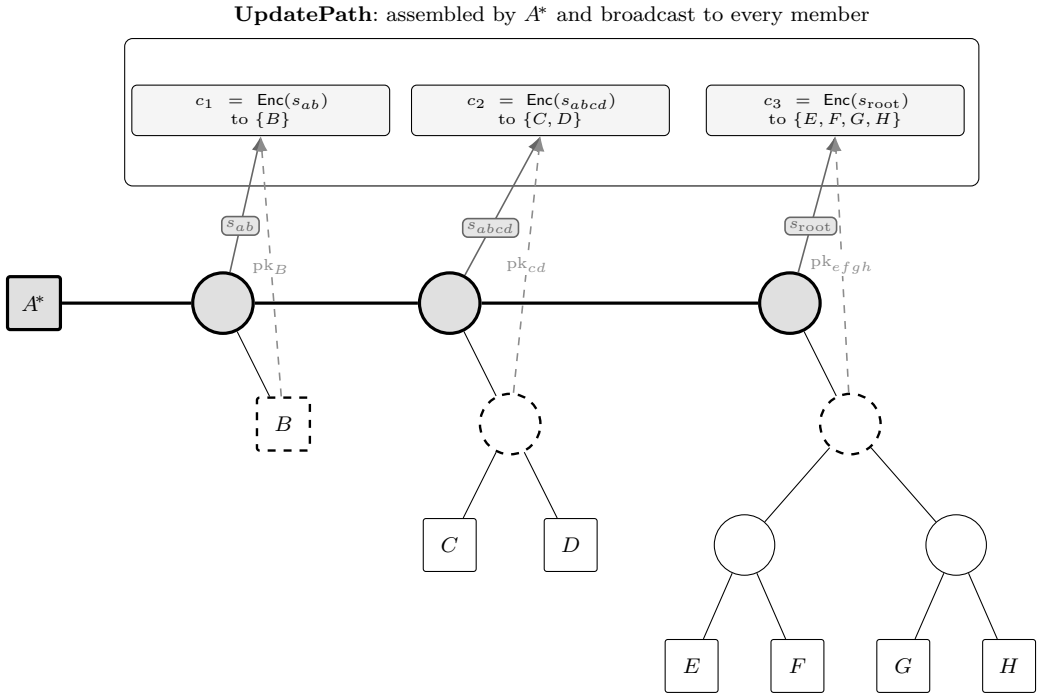

Figure 2: An update by member A^* in an eight-member ratchet tree, with A^*'s direct path to the root drawn as a horizontal spine and each copath subtree dangling beneath it. The acting member walks every node along this path (shaded), deriving a fresh *path secret* s for each followed by a new public/private key pair. Above the spine, the figure traces how the broadcast UpdatePath is built: each path secret is encrypted using the public key stored at the corresponding copath node (dashed arrows) to form one ciphertext c_i, addressed to exactly the members beneath that copath node. The message therefore carries just three ciphertexts ($\log_2 8$) rather than the per-member redistribution a sender-keys scheme would require.

Assuming the requisite public keys are available, an outsider can append its own node to the tree, and compute the update using the public keys published for the tree.

The two cases differ in a couple of ways. First, the update message, like all MLS messages, requires that additional metadata and a digital signature be transmitted with the message that (1) identifies and verifies the sending entity, (2) carries information to prevent replay attacks, and (3) is encrypted such that only group members can read the metadata. For intra-group communication, the key schedule and ratchet tree provide a sending secret and signing key used for these purposes. An outsider uses the published external join public key mentioned above along with its own private key to secure the message metadata.

Given this, it is incumbent upon the existing group members to accept or deny an update sent by a given external entity, which may need to rely upon other mechanisms in the system for authenticating such a request, e.g., our system model described above in section 2.

Given an untrustworthy server, an adversary who either compromises the server infrastructure or compromises an existing user's credentials would be able to install a fraudulent membership and device definition. They could then perform a join operation on groups of interest, adding the ratchet for the fraudulent device among those for the legitimate devices in the group. The other devices would rely on the server's information to validate the join operation and proceed to update their set of ratchets. At this point, no prior communication would be exposed to the fraudulent device, but the server would work to forward the future communication to this device, making it unlikely for the device to miss future message exchanges. For this reason, among others, each device reports to its user when new devices have joined or been added to the group independently and in response to its local action of adding ratchets to the group's state. Thus, the adversary cannot inconspicuously gain access to group communication. And the other members of the group can take action to correct the situation.

4.3 Post-Quantum Encryption

To further motivate GLYPH's proprietary implementation of MLS, we briefly discuss the threat imposed by quantum computing to traditional cryptography.

It is known that a large enough quantum computer—one with a sufficient number of qubits—will be able to efficiently find the prime factors of the large numbers used in RSA encryption and to efficiently compute discrete logarithms for breaking elliptic curve encryption[20, 19]. Given the regularly advancing state of the art in quantum computing hardware, it is therefore only a matter of time before a quantum computer exists that can compromise the security of systems utilizing these, the most common, public key cryptographic algorithms. Estimates for the advent of this computer range anywhere from twelve months to ten years, which is considerably shorter than the amount of time required to perform a brute force attack against a given public key in these algorithms.

Given this time horizon, adversaries may adopt a strategy of intercepting encrypted content now in order to decrypt it later once a suitable quantum computer becomes available to do so. If private data transmitted today is still valuable beyond this horizon, it then becomes imperative to adopt encryption algorithms capable of withstanding an attack from quantum-capable adversaries as soon as possible.

At the time of this writing, it is still believed that AES-256-GCM is safe in light of advancing quantum computing; no algorithm is yet known for deriving a secret from some ciphertext.

However, as discussed above, while individual messages transmitted in GLYPH are therefore believed to be secure, MLS still utilizes public key cryptography for adding devices to, removing devices from, and joining a given group. Most notably, this occurs when transmitting initialization information to a newly added device in a given group. Each device in GLYPH publishes one or more public keys to the GLYPH servers, and one of these is used to seal a secure envelope containing the entire group state that will be transmitted to the new device. If a quantum-capable adversary intercepts and decrypts this envelope, he/she/ze will obtain a complete set of ratchets for the group, which can be used to decrypt all subsequently intercepted group messages.

Fortunately, in July 2022, the National Institute of Standards and Technology (NIST) selected several public key algorithms believed to be safe from quantum computing, and in August 2024 it published the first of these as finalized standards. Among them are the CRYSTALS-Kyber[9] and CRYSTALS-Dilithium[8] algorithms—standardized as ML-KEM in FIPS 203[15] and ML-DSA in FIPS 204[14]—for key encapsulation and digital signatures, respectively. More recently, in March 2025, NIST selected a fifth algorithm, HQC[16], to serve as a backup key-encapsulation mechanism. Because HQC is built on error-correcting codes rather than the structured lattices underlying ML-KEM, it hedges against the possibility that a future advance weakens the lattice assumption. NIST expects to publish a draft HQC standard in 2026 and a final standard in 2027.

4.4 GLYPH and MLS

GLYPH deploys a proprietary implementation of MLS that predates and differs from the reference implementation in several ways:

- the wire format differs significantly, that is the byte-for-byte arrangement of data in each message when transmitted over an IP-switched network;
- GLYPH extends the key schedule to include additional keys needed for broadcast and submission group types;
- where resumption keys (not previously discussed) and the file encryption keys must survive a group update and newly externally joining members do not possess these secrets when they perform the update to join the group, GLYPH extends the protocol with a mechanism by which existing members are later able to share these long-lived secrets with newly joined members (see section 5 below); and
- GLYPH's implementation is engineered to allow alternative encryption algorithms to be employed within the MLS framework more quickly than other implementations, which allows us to pursue post-quantum cryptography more aggressively.

Expanding on the final item above, each of the operations described above in section 4.2 is assembled from a handful of cryptographic primitives. They are:

(1) A source of cryptographically secure *randomness*;

(2) A collision-resistant *hash function*;

(3) A *key derivation function (KDF)*, as previously described;

(4) A means of deterministically deriving a key pair from a secret, which is the most exotic primitive listed here;

(5) A *key encapsulation mechanism (KEM)*, which supports the process by which two distributed entities arrive at a shared secret (see section 4.1.2);

(6) An authenticated block or stream encryption scheme, currently AES-256-GCM; and

(7) A *digital signature* scheme.

Among these, items 4, 5, and 7 employ public key cryptography and are, thus, vulnerable to the threats imposed by quantum computing. As mentioned in section 4.3, NIST has approved (or is in the process of approving) and standardized new algorithms that serve these primitives. Being specific and well-defined problems with essentially equivalent interfaces, swapping in these alternatives does not disrupt the MLS protocol or infrastructure, allowing us to reasonably assert a post-quantum secure implementation of MLS.

5 File Storage

In GLYPH, files are treated in a different manner than text messages, but the Group and its requisite state serve as the

basis for limiting access to files to those devices participating in it.

We utilize cloud storage to make files widely available to participating devices, which are often offline and, thus, unable to service any kind of peer-to-peer requests for a given file. When uploading a file, a device chooses or creates a secret encryption key with which to encrypt the file (currently using AES-256). A cipher stream is set up such that the encrypted contents of the file are streaming to our server infrastructure without ever revealing the contents to the network or our server infrastructure[5].

Once uploaded, a reference to the uploaded location must be included in a message transmitted to the group. If needed, the secret encryption key may also be included in the message. Thus, subject to the security mechanisms of MLS described above, only participating devices in a group will be able to receive, decrypt, and utilize this encryption key. And, though the file exists in our cloud infrastructure where it potentially could be downloaded by anyone, only those devices will be able to understand its contents.

Additionally, for the sake of ease and usability, group file access is not subject to forward secrecy or post-compromise security insomuch as a newly added device (1) will have access to the files shared with the group prior to the moment the device gains access to the group communication and (2), by virtue of having access to the decryption key in plain text while participating in the group, could potentially store the key in order to access the file after leaving the group in the future. The set of decryption keys used for file encryption are shared with newly added or joining devices by another device in the group. And it would be untenable to require another device to choose new keys, download, re-encrypt, and upload every file every time a device is removed from the group. As such, we rely on server permissions to deny access to group files for unauthorized members.

Moreover, downloaded and decrypted files are stored locally on the device by the user and may persist in that form beyond the term of the device's membership in the group. Note, however, that, where possible[6], we store files in their encrypted form and only decrypt them as requested by the user.

Metadata about each file—such as the file name, content type, and file size—are stored in plain text on the server. This facilitates file browsing in the UI, administration, and storage metering.

6 Video Conferencing

A video conference in GLYPH is currently referred to as a Huddle. Access to a given huddle is governed by group membership at the user level. That is, if a user is a member of a group, they may participate in a huddle from any authenticated device regardless of whether that device participates in the text messaging of the group.

A given huddle can operate in one of two ways: a performance-optimized mode or an ultra-secure mode. Both utilize standard WebRTC sessions that are encrypted with AES-128 or AES-256 and negotiated with a traditional Diffie-Hellman key exchange, but participation remains governed by the aforementioned device and membership restrictions at the group level. The performance-optimized mode configures the participating devices to stream audio and video channels to a trusted network of forwarding media servers, which handle decryption, fan-out, re-encryption, and delivery to each participant. This mode ensures all participants use AES-256 encryption and is well-suited for the vast majority of enterprise use cases. Organizations with the most stringent confidentiality requirements may prefer the ultra-secure peer-to-peer mode described below.

The ultra-secure mode configures the participating devices to stream their audio/video channels in direct peer-to-peer fashion. That is, each device transmits each of its audio/video frames multiple times, once to every other device participating in the huddle, which can impact performance on resource-constrained devices with a relatively large number of participants. Also, many web browsers prevent web applications from participating in the WebRTC session initiation and unilaterally prioritize AES-128 over AES-256 when negotiating an encryption scheme, which prevents us from guaranteeing that AES-256 is used when configuring a huddle in this manner. However, this mode ensures end-to-end encryption of all streams.

7 Archiving and Auditing Compliance

The organizations of some industries are required to maintain an auditable record of all communication conducted within the organization, which presents a challenge for end-to-end encrypted communications platforms, like GLYPH, that are designed for maximal privacy and security. To service these industries, GLYPH offers an archiving tool that can be enabled at the account level or at an individual group level.

Once enabled, our server infrastructure requires that a special device be a participating member of every affected group and relies upon any of the participating user devices to perform the necessary MLS operation to include it. When a message is received at the server, it is first passed to and processed by this special device, which must possess a copy of the group's encryption state to do so. If the special device fails to process the message, the server will discard it, forego forwarding it to other group members, and inform the sender of the failure. As such, the group in question will be effectively muted until the archiving device is included in group communication and working properly.

[5]We break this condition in our browser-based web app and utilize a streaming proxy to encrypt and store a given file while we wait for broader adoption of the File System Access API[10] and implementations of the streaming webcrypto API[21].

[6]Here, the web browser is again the case where we cannot store downloaded files in their encrypted form.

This special device may reside within GLYPH's cloud infrastructure, in some other authorized cloud, or on premise for a given organization.

To maintain privacy, GLYPH configures an Account with a public/private key pair, where the private key is presented to and stored by the account holder in a manner of their choosing and the public key is stored with the Account in the GLYPH infrastructure.

As messages are decrypted, they are appended to an unencrypted log for the given group, stored locally on an encrypted disk, and formatted to facilitate query engines that will later need to operate over the collection of messages and their metadata. At regular intervals, an archiving encryption key is randomly chosen for the account, encrypted with the Account's public key, saved in the archive, and used to encrypt the most recent section of log. The encrypted log section is then moved to long term storage, and the unencrypted version is discarded along with the key that created it.

As such, after each interval, the GLYPH infrastructure has no access to the contents of the archive up to that point in the history of the log.

When the account holder must service an audit or compliance request, they can download a portion of the archive sufficient to cover the request along with the accompanying keys, use their private key to decrypt the keys, and then decrypt the log with each key in turn.

As noted above, enabling archiving requires the archiving device to hold a copy of the group's encryption secrets for the duration of its participation. This is an inherent tradeoff of any compliance-grade archiving solution operating on end-to-end encrypted communications. However, provided that any infrastructure breaches are corrected and combined with our archiving scheme above, the system still maintains a version of forward secrecy and post-compromise security while complying with archiving and auditing requirements. That is, at the time an adversary gains access to the archiving machine, they will have no access to message history before that point in time. And, upon correcting that breach and performing an update on the group, the adversary will have no access to future messages sent within the group.

8 Data Sovereignty

Some governments require or may soon require that an organization's data reside within the government's jurisdiction. To accommodate this, we adopt a multi-server architecture whereby we can deploy our system to data centers residing within the jurisdiction in question. A given Account, whose owner is subject to the laws of this jurisdiction, is then assigned to this data center. Each member of the account directs their device(s) to connect to, authenticate with, and communicate with our servers in this data center when interacting with this account. This ensures that the limited amount of data we store in our infrastructure with respect to this account properly resides within this jurisdiction. And it allows a single device to interact with multiple accounts where each account may reside in a different jurisdiction.

Note that the user's location, which could be temporarily or permanently outside the jurisdiction, is irrelevant. The account defines the jurisdiction, and any data the user generates while interacting with the groups of that account is thereby subject to the account's constraints and stored in proper jurisdiction.

References

[1] Joël Alwen et al. "Security Analysis and Improvements for the IETF MLS Standard for Group Messaging". In: *Advances in Cryptology – CRYPTO 2020*. Ed. by Daniele Micciancio and Thomas Ristenpart. Cham: Springer International Publishing, 2020, pp. 248–277. ISBN: 978-3-030-56784-2.

[2] R. Barnes et al. *The Messaging Layer Security (MLS) Protocol*. RFC RFC 9420. Internet Engineering Task Force (IETF), July 2023. DOI: 10.17487/RFC9420. URL: https://www.rfc-editor.org/rfc/rfc9420.html.

[3] B. Beurdouche et al. *The Messaging Layer Security (MLS) Architecture*. RFC RFC 9750. Internet Engineering Task Force (IETF), Apr. 2025. DOI: 10.17487/RFC9750. URL: https://www.rfc-editor.org/rfc/rfc9750.html.

[4] A. Biryukov et al. *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*. RFC RFC 9106. Internet Engineering Task Force (IETF), Sept. 2021. DOI: 10.17487/RFC9106. URL: https://www.rfc-editor.org/rfc/rfc9106.html.

[5] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. *Argon2: the memory-hard function for password hashing and other applications*. visited on 06/14/2022. 2017. URL: https://github.com/P-H-C/phc-winner-argon2/blob/master/argon2-specs.pdf.

[6] Katriel Cohn-Gordon et al. "On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1802–1819. ISBN: 9781450356930. DOI: 10.1145/3243734.3243747. URL: https://doi.org/10.1145/3243734.3243747.

[7] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. "On Post-compromise Security". In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 2016, pp. 164–178. DOI: 10.1109/CSF.2016.19.

[8] *CRYSTALS – Dilithium*. visited on 01/27/2023. URL: https://pq-crystals.org/dilithium/index.shtml.

[9] *CRYSTALS – Kyber*. visited on 01/27/2023. URL: https://pq-crystals.org/kyber/index.shtml.

[10] *File System Access API*. visited on 06/19/2026. URL: https://developer.mozilla.org/en-US/docs/Web/API/File_System_API.

[11] J. Hodges et al. *Web Authentication: An API for Accessing Public Key Credentials Level 2*. W3C Recommendation. visited on 06/19/2026. World Wide Web Consortium (W3C), Apr. 2021. URL: https://www.w3.org/TR/webauthn-2/.

[12] *How the YubiKey Works*. visited on 06/19/2026. URL: https://www.yubico.com/products/how-the-yubikey-works/.

[13] D. M'Raihi et al. *TOTP: Time-Based One-Time Password Algorithm*. visited on 06/14/2022. 2011. URL: https://datatracker.ietf.org/doc/html/rfc6238.

[14] National Institute of Standards and Technology. *Module-Lattice-Based Digital Signature Standard*. Federal Information Processing Standards Publication FIPS 204. U.S. Department of Commerce, Aug. 2024. DOI: 10.6028/NIST.FIPS.204. URL: https://doi.org/10.6028/NIST.FIPS.204.

[15] National Institute of Standards and Technology. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. Federal Information Processing Standards Publication FIPS 203. U.S. Department of Commerce, Aug. 2024. DOI: 10.6028/NIST.FIPS.203. URL: https://doi.org/10.6028/NIST.FIPS.203.

[16] National Institute of Standards and Technology. *NIST Selects HQC as Fifth Algorithm for Post-Quantum Encryption*. visited on 06/19/2026. Mar. 2025. URL: https://www.nist.gov/news-events/news/2025/03/nist-selects-hqc-fifth-algorithm-post-quantum-encryption.

[17] *Passkeys*. visited on 06/14/2022. URL: https://developer.apple.com/passkeys/.

[18] *Pepper (cryptography)*. visited on 06/14/2022. URL: https://en.wikipedia.org/wiki/Pepper_(cryptography).

[19] John Proos and Christof Zalka. "Shor's Discrete Logarithm Quantum Algorithm for Elliptic Curves". In: *Quantum Info. Comput.* 3.4 (July 2003), pp. 317–344. ISSN: 1533-7146. eprint: quant-ph/0301141. URL: https://arxiv.org/abs/quant-ph/0301141.

[20] P. W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. SFCS '94. USA: IEEE Computer Society, 1994, pp. 124–134. ISBN: 0818665807. DOI: 10.1109/SFCS.1994.365700. URL: https://doi.org/10.1109/SFCS.1994.365700.

[21] *Streams for the Web Cryptography API*. visited on 06/19/2026. 2022. URL: https://github.com/WinterTC55/proposal-webcrypto-streams/blob/main/explainer.md.