GLACIER

# Cryptography Whitepaper

This whitepaper explains the design decisions and cryptography within Glacier.

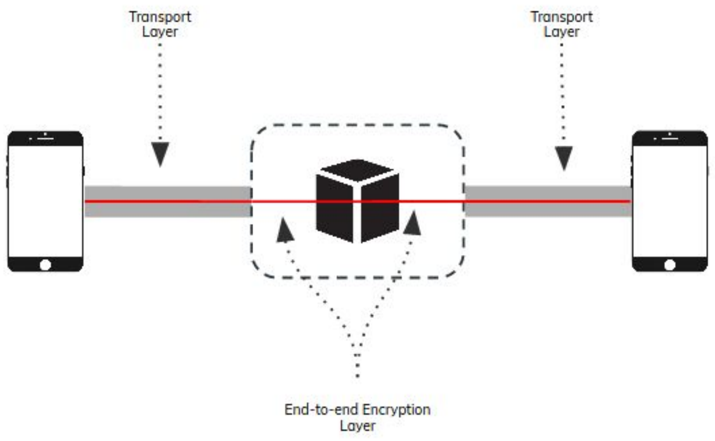
## Overview

Glacier uses multiple layers of encryption to protect messages, files, and devices within an organization.

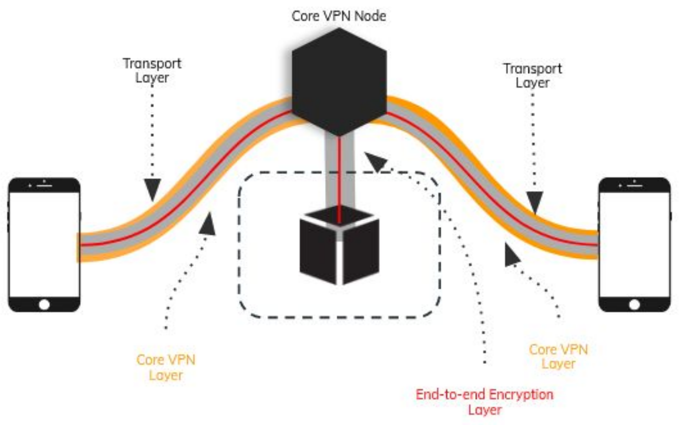
**End-to-end encryption layer:** Secures the message content from the sender to the receiver.
**Transport layer:** Secures the end-to-end encryption layer and header information in transit to the server.
**Core Obfuscation VPN layer:** Optional layer that secures both the transport layer and end-to-end encryption layer with an encrypted tunnel. This tunnel provides additional security, anonymity, and obfuscation.

These layers are described in further detail throughout this whitepaper.

*End-to-end encryption and transport layer encryption:*

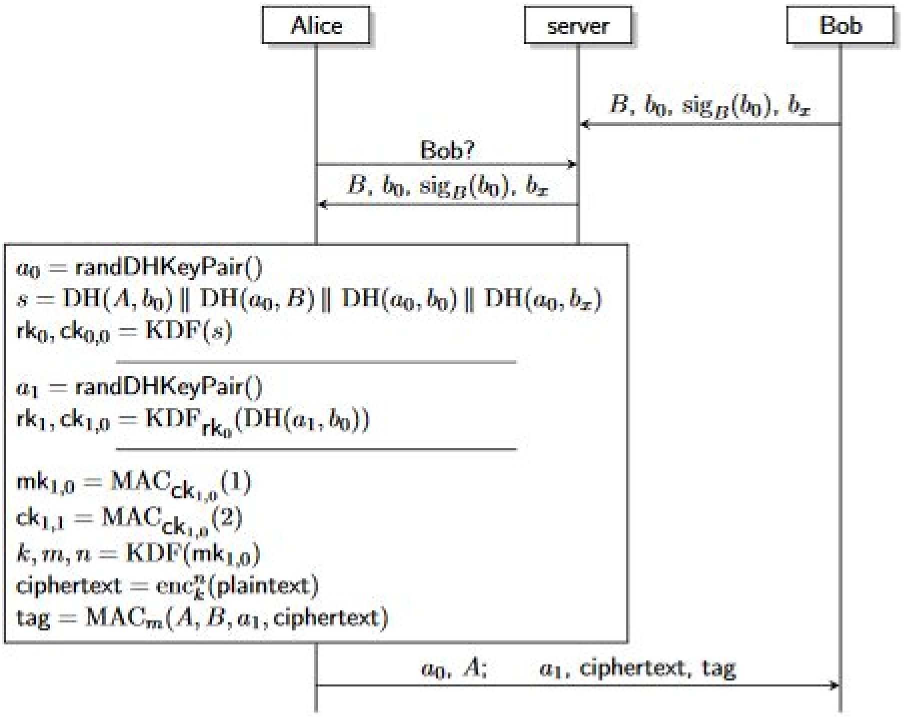


*End-to-end encryption, transport layer encryption, and Core VPN layer encryption:*



## End-to-End Encryption

In Glacier, all messages (text, media, audio, video) are end-to-end encrypted with the OMEMO (OMEMO Multi-End Message and Object Encryption) protocol, an adaptation of the Signal Protocol created by Open WhisperSystems. OMEMO is designed to work in an environment where users can have multiple devices associated with the same user account.


Alice
server
Bob
$B,\ b_0,\ \mathrm{sig}_B(b_0),\ b_x$
Bob?
$B,\ b_0,\ \mathrm{sig}_B(b_0),\ b_x$
$a_0 = \mathsf{randDHKeyPair}()$
$s = \mathrm{DH}(A, b_0) \parallel \mathrm{DH}(a_0, B) \parallel \mathrm{DH}(a_0, b_0) \parallel \mathrm{DH}(a_0, b_x)$
$\mathsf{rk}_0, \mathsf{ck}_{0,0} = \mathrm{KDF}(s)$
$a_1 = \mathsf{randDHKeyPair}()$
$\mathsf{rk}_1, \mathsf{ck}_{1,0} = \mathrm{KDF}_{\mathsf{rk}_0}(\mathrm{DH}(a_1, b_0))$
$\mathsf{mk}_{1,0} = \mathrm{MAC}_{\mathsf{ck}_{1,0}}(1)$
$\mathsf{ck}_{1,1} = \mathrm{MAC}_{\mathsf{ck}_{1,0}}(2)$
$k, m, n = \mathrm{KDF}(\mathsf{mk}_{1,0})$
$\mathsf{ciphertext} = \mathrm{enc}_k^n(\mathsf{plaintext})$
$\mathsf{tag} = \mathrm{MAC}_m(A, B, a_1, \mathsf{ciphertext})$
$a_0,\ A;\quad a_1,\ \mathsf{ciphertext},\ \mathsf{tag}$

The figure shows the start of a conversation between Alice and Bob. In this abstracted example, the participants are identified by their name. In reality, the users are identified by their Glacier ID (@alice and @bob) which is an adaptation of an XMPP Jabber ID (JID).

## Notation

The following notation is used: $KDF_s(i)$ derives a key using salt *s*, info data *i* and a constant label that is unique for each KDFcomputation in the figure. When no salt is specified, the constant value 0 is used. $MAC_k(m)$ computes an authentication tag on message m, using key *k*. $enc^n_k$ (m) computes the symmetric encryption of message *m*, using key *k* and  nonce/initialization vector *n*. To  keep the above diagram simple, the precise meaning for asymmetric keys notation depends on the context, but it is straightforward. For example: $a_0$ refers to the entire key pair when generated, to the private key when used in the DH computation and to the public key when sent in the message. Only public keys are sent in messages.

## Prekeys

First Bob uploads his client-side generated key material to the server so that he can be contacted by Alice. He sends his long-term identity key *B*, his signed pre key $b_0$ with corresponding signature $sig_B(b_0)$ and a One-time-use prekey $b_x$. Bob can go offline at this point, the server will now act as an online cache for others that want to initiate a conversation with Bob.

## TripleDH

When Alice wants to talk with Bob, she requests the cached data from the server. The server complies and Alice can initiate the TripleDH handshake. She first generates her own one-time key pair $a_0$. She combines the keys by concatenating the results of the DH computations and computes *s*, a shared secret that initializes the
Double Ratchet. Using the KDF function, Alice computes the initial root key $rk_0$.

## DH ratchet (every reply)

Alice updates the root key with the DH ratchet. She first generates a fresh random key pair $a_1$ and does a DH computation with the latest DH key she received from Bob (initially $b_0$). Using the previous root key $rk_0$ as a seed for the KDF, she computes a new root key $rk_1$ and a new sending chain key $ck_{1,0}$. At this point, Alice should delete the old root key $rk_0$ and her previous key pair $a_0$ to ensure forward secrecy.

## Chain ratchet (every message)

Alice derives a message key ($mk_{1,0}$) and a new chain key $ck_{1,1}$ from the old chain key $ck_{1,0}$ and she deletes the old chain key for forward secrecy. Alice derives three keys from the mk with th eKDF: an encryption key *k*, an authentication key *m* and a nonce/initialization vector *n*. She encrypts the plaintext message and computes an authentication tag over the (public) identity keys and the ciphertext. She then sends the Glacier message to Bob, consisting of her one-time key $a_1$, the ciphertext and the authentication tag. Only with the *PreKeySignalMessage* (the first message) will she also include her first one-time key $a_0$ and her identity key *A*. Bob can use the key material from the *PreKeySignalMessage* to initiate the root ratchet and receiving chain ratchet, from which the key material can be derived to validate and decrypt the message. This diagram implicitly also shows how the conversation continues. Every time the user replies to a message, the steps below the first horizontal line are taken: the root key is updated with a fresh random DH computation and a new sending chain ratchet is initialized. For every additional message, the sending chain key is updated and a fresh message key is used to encrypt user messages. Note that both users have one root ratchet and two chain ratchets: one for sending and one for receiving.

## PublicKey verification

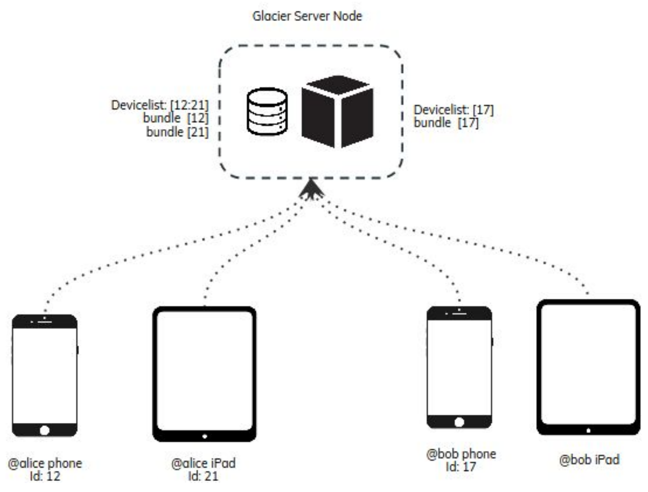
In order to ensure that no man-in-the-middle attack has taken place, Alice needs to verify that the identity key she has connected with indeed belongs to Bob. Users must manually verify the identity key “Glacier Device ID” or “fingerprint” (the full public key) of the other party.

## Message counters

Messages might arrive out of order and can even arrive after the DH ratchet has been forwarded. Therefore, the sender of the message also includes two counters: one for how many messages were sent under the current ratchet and one for the total under the previous ratchet. With these counters, the receiver can see exactly which messages did not (yet) arrive and store only the corresponding message key *mk*.

## Multiple prekeys

In a real-world situation, Bob would want more than one person to be able to communicate with him, so he uploads multiple prekeys to the server. In the case of the Glacier application, Alice only gets a single one-time prekey from the server. When the server runs out of prekeys, Alice can complete the handshake without Bob’s one-time prekey. This message has reduced forward secrecy, because Bob cannot delete the signed prekey $b_0$ immediately after use. When Bob receives a *PreKeySignalMessage*, he sends a fresh signed prekey to the server, so that the key that is cached on the server gets updated. Bob needs to know which signed prekey and which one-time prekey Alice used in her computation, so each prekey has its own identifying number. Alice includes that number in the *PreKeySignalMessage* and sends Bob, unauthenticated and unencrypted.



**Used cryptographic primitives**

The following primitives are in use:

- enc: AES in CBC mode using PKCS5
- MAC: HmacSHA256
- KDF: HKDF using HmacSHA256
- DH: X25519
- sig: Ed25519

## OMEMO Usage

Glacier supports multiple devices associated to the same account. A Glacier session is set up between each device. Messages are encrypted and authenticated with a random key and the encryption of that key is sent as message content of a Glacier message.

**Example**: Alice has authenticated to Glacier from two devices, while Bob has only authenticated his phone and wants to authenticate his iPad as well. In order to authenticate his iPad, Bob generates a random 31-bit device id and registers it by adding it to his device list on the Glacier server. He then generates a random identity key *B*, a signed prekey $b_0$ with corresponding signature sig($b_0$) and 100 one-time prekeys $b_x$. He then uploads this in an OMEMO bundle. This bundle contains the same information that Bob caches on the server.

Assume Alice wants to send an Glacier encrypted message from her phone. She encrypts and authenticates her message using a randomly generated key. For every device that Alice wants to send the encrypted message to, she fetches the entire bundle from the server. If she wants to add more of her own devices in the conversation, she gets their bundles as well from the server. Alice creates a PreKeySignalMessage for every device by picking a random one-time prekey from each bundle and encrypting the randomly generated key to each device. She combines all information in a single MessageElement: the encrypted payload (`<payload/>`), the plaintext iv (`<iv/>`), the sender id (`sid`) and the encrypted random key (`<key/>`) tagged with the corresponding receiver id (`rid`).

Bob's device can decrypt the message by selecting the correct `<key/>` element based on `rid` attribute and use it to initialize the session on his side. At this point, Alice's phone has set up a Glacier session with each of the devices. If Bob wants to reply, he still needs to initialize a session with Alice's other device, so he also needs to download all bundles and initialize Glacier sessions by sending a PreKeySignalMessage where necessary. If all devices (but one) have sent a message, each device will have a pairwise Glacier session setup.

## Group messages

Group messages (multi-user chats) are no different from a multi-device chat: to send a message to all users, the sending device sets up a Glacier session with each of the participating devices, regardless of the user to which the device belongs.

## File Transfer

A device wishing to share an end-to-end encrypted file first generates a 32 byte random key and a 12 byte random IV. After successfully requesting a slot for HTTP upload the file is encrypted with AES-256 in Galois/Counter Mode (GCM) on the fly while uploading it via HTTP. The authentication tag is appended to the end of the file.

To share the file the entity converts the HTTPS URL, the key and the IV to an `aesgcm://` URL. Both IV and key are converted to their hex representation of 24 characters and 64 characters respectively and concatenated for a total of 88 characters (44 bytes). The IV comes first followed by the key. The resulting string is put in the anchor part of the aesgcm URL.

***File transfer example:***

```
https://node1.somedomain.tld/4a771ac1-f0b2-4a4a-9700-f2a26fa2bb67/tr%C3%A8s%20cool.jpg

IV: 8c3d050e9386ec173861778f
Key: 68e9af38a97aaf82faa4063b4d0878a61261534410c8a84331eaac851759f587ed40ca58
```

***Resulting URL:***

```
aesgcm://node1.somedomain.tld/4a771ac1-f0b2-4a4a-9700-f2a26fa2bb67/tr%C3%A8s%20cool.jp
g#8c3d050e9386ec173861778f68e9af38a97aaf82faa4063b4d0878a61261534410c8a84331eaac851759
f587ed40ca58
```

## Purging Keys

The Glacier application allows users to purge the key of other devices, which irreversibly marks the key as compromised.

## Confidentiality and Forward Secrecy

OMEMO is an end-to-end encryption protocol based on the Double Ratchet. It provides the following guarantees:

| | |
|---|---|
| **Confidentiality** | Nobody else except sender and receiver is able to read the content of a message. |
| **Forward Secrecy** | Compromised key material does not compromise previous message exchanges. It has been demonstrated that OMEMO provides only weak forward secrecy (it protects the session key only once both parties complete the key exchange). |
| **Break-in Recovery** | A session which has been compromised due to leakage of key material recovers from the compromise after a few communication rounds. |
| **Authentication** | Every peer is able to authenticate the sender or receiver of a message, even if the details of the authentication process is out-of-scope for this specification. |
| **Integrity** | Every peer can ensure that a message was not changed by any intermediate node. |
| **Deniability** | X3DH is weakly offline deniable and provides no online deniability, as far as the research shows. |
| **Asynchronicity** | The usability of the protocol does not depend on the online status of any participant. |

## Client-Server Protocol Description

Glacier communicates with three different types of servers. To transport chat messages, access the directory and to download/upload encrypted media files, HTTPS/TLS is used.

**Chat protocol:** Transports the end-to-end encrypted incoming and outgoing messages between the client and the Glacier servers over TLS 1.2.

**User Authentication:** The clients initially authenticate to the Glacier systems through an out-of-band authentication server. This data is encrypted in transit with TLS 1.2 and ECDHE. Data within the user authentication servers are encrypted at rest in accordance with industry standards (see Data at Rest).

**File Upload:** The file upload servers are used for temporary storage of large media data (e.g. images, videos, audio recordings). Such media is not sent directly via the chat protocol.

## Push Notifications

Glacier uses the standard push notification service provided by the operating system. This is necessary to ensure reliable delivery of push notifications, as mobile operating systems place high importance on conserving battery power, and thus severely restrict background operation of apps (such as would be needed to maintain a separate, app-specific persistent connection for push notifications).

### What Glacier sends to the app server

**FCM Token:** The FCM token is essentially the address of the device that will be used to address push notifications. The address is generated by the Google Play Service on the user’s phone.

**Secure Device ID**: Since the FCM token can change over time the app server needs a second unique ID per device so when it receives a new FCM token it knows whether this one replaces an existing device or is for an entirely new device. Glacier uses the Android ID which is a unique string that is generated by the Android operating system and persists over the life time of the app.

The app server then generates a random string (Node ID in the language of the XEP) and stores a combination of the FCM Token, the node ID, the domain of the account and a hashed string of the Glacier ID + the secure device ID.

### What the Glacier app sends to the server

After registering with the server, The Glacier app sends the node ID and the Glacier ID of the Push server to the Glacier messaging servers.

## What the Glacier server sends via Google to the user's device

Upon receiving a push request from the server, the app server looks up the hashed Glacier ID + secure device ID combination and the FCM token. It then sends the hash via Google to the user's device, using the token.

The hash can then be used by Glacier to determine which of a number of accounts should be woken up. The hash is not reversible. Google only sees that hash and nothing else.

For additional information see https://github.com/iNPUTmice/p2

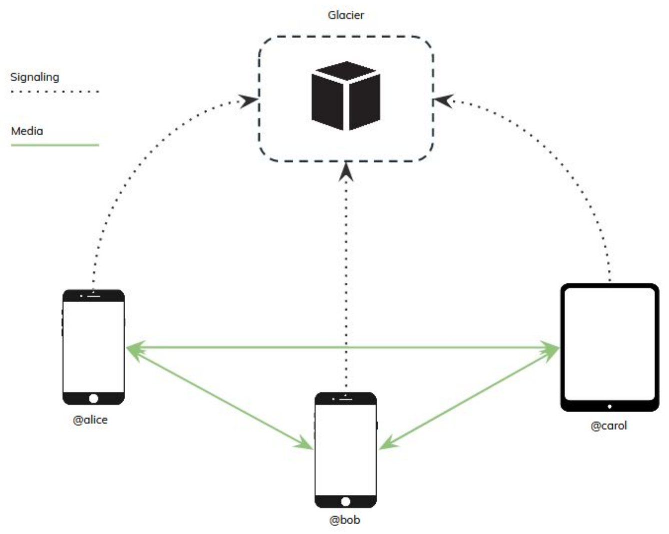
## Address Book Synchronization

The Glacier app does not synchronize contacts with any external server.

## Glacier Voice and Video Calls

Media shared in Peer-to-Peer Rooms is encrypted end-to-end and can never be accessed by Glacier. Each Participant in a Peer-to-Peer Room negotiates a separate DTLS/SRTP connection to every other participant. All media published to or subscribed from the call is sent over these secure connections, and is encrypted only at the sender and decrypted only at the receiver.

Glacier does not mediate in the media exchange, which takes place through direct communication among the Glacier users. The only exception is when media exchange requires TURN. In that case, a TURN server will blindly relay the encrypted media bits to guarantee connectivity. The TURN server cannot decrypt or manipulate the media.



The above picture illustrates the architecture of a Glacier voice/video call.

### Audio Encoding

Audio data is encoded by OPUS.

### Video Encoding

Video data is encoded by VP9, VP8 or H.264.

### Call Data Records (CDRs)

Glacier temporarily stores the username, randomly generated authentication tokens, keys and push tokens necessary for setting up a call or transmitting a message. These logs are stored for 7 days for debugging purposes.

## Glacier FileSafe

FileSafe uses Amazon Web Service (AWS) authentication mechanisms and AWS's TransferUtility library to securely transfer files from the Glacier client to the cloud storage endpoint. All data transmitted from the Glacier client to the FileSafe storage endpoint is encrypted through a HTTPS/TLS connection. Data at rest is secured with server-side encryption using AWS KMS (SSE-KMS). Organizations can also provide a customer managed CMK. Currently only available for Android.

## Glacier Group File Pinning

### iOS

Uploaded files that are "pinned" to a room, don't expire, and can be listed/retrieved by the group occupants. The files are encrypted by the server during upload. While browsing the attachments, the client receives the key material necessary to decrypt them.

To retrieve the list of attachments the client queries the list of files attached to a given room by sending an IQ-get to the group. The server responds with an IQ-result that contains the attachments, each with `url`, `cipher`, `key`, `iv`, and `tag` attributes. The url attribute points to the HTTPS URL of the encrypted file, the cipher hard-coded to AES-256-GCM, and the remaining three attributes contain the key material required to decrypt the downloaded file. During the upload process the file is encrypted using AES-256 GCM before storing it.

### Android

File Pinning is not currently available for Android.

## Core VPN

Glacier's cloud based immutable VPN infrastructure is launched unique for each organization. Glacier controls change across multiple system dimensions in order to increase uncertainty for attackers, reduce their window of opportunity, increase the costs of their attack efforts, and keep end user devices anonymous.

Glacier routes all device (or whitelisted apps) data through an encrypted tunnel to a temporary server provisioned for the users organization.

### Android

On Android, Glacier leverages the OpenVPN protocol to establish an encrypted tunnel from the user device, through the obfuscation node(s) to the Glacier Core.

**Procedures to establish a secure Core VPN connection:**

1. Alice enables the Glacier Core VPN within the Glacier app
2. Alice's device makes an OpenVPN connection to the Glacier Obfuscation Node on some non standard UDP or TCP port
3. Alice's source IP address is removed from the packet and routed through the Glacier Obfuscation network
4. Alice's device establishes the tunnel and starts routing app traffic through the tunnel

**Used cryptographic primitives**
The following primitives are in use for Android:

- Encryption cipher: AES 256bit
- DH Param Bits: 3072
- Hash Algorithm: AES-256

### iOS

On iOS, Glacier Core only supports IKEv2 with strong crypto (AES-GCM, SHA2, and P-256).

Optionally, ads, malware, and other malicious content are blocked using a local DNS resolver on the IPSEC endpoint. Glacier has integrated publicly available threat intelligence feeds to identify threat actors and provide a broad array of capabilities to block exploits, malware, ransomware, spyware, and other potentially harmful sites. These domains or hosts are updated in our lists from our Threat Intelligence (TI) partners, who supply us with data on feeds that are rapidly updated and distributed as new domain-based risks emerge.

**Procedures to establish a secure Core VPN connection:**

1. When Bob enables Glacier Core VPN for the first time, Glacier installs a `mobileconfig` (Apple profile) on your device (this allows users to leverage iOS's connect on demand feature)
2. Bob's device then makes an IPSEC IKEv2 tunnel to the temporary Glacier IPSEC endpoint

3. Bob's device routes all device traffic through the tunnel.

## Glacier Core Servers

Each organization is assigned a unique virtual environment within our cloud based infrastructure for messaging, voice, VPN, and file transfer services.

### Data at Rest

Glacier encrypts both the boot and data volumes of each Core server. The following types of data are encrypted:

- Data at rest inside the volume
- All data moving between the volume and the instance
- All snapshots and backups created from the volume
- All volumes created from those snapshots and backups

Glacier encrypts each volume with a data key using the industry-standard AES-256 algorithm. The data key is stored on-disk with the encrypted data. Data keys never appear on disk in plaintext. For more advanced configurations, organizations can also provide a customer managed CMK.

Glacier servers leverage a Key Management Service (KMS) to encrypt and decrypt server volumes as follows:

1. The Glacier server volume sends a `GenerateDataKeyWithoutPlaintext` request to the KMS, specifying the Customer Master Key (CMK).
2. KMS generates a new data key, encrypts it under the CMK and sends the encrypted data key to the Glacier instance volume to be stored with the volume metadata.
3. The Glacier server volume sends a `CreateGrant` request to KMS, so that it can decrypt the data key.
4. KMS decrypts the encrypted data key and sends the decrypted data key to Glacier server.
5. The Glacier server uses the plaintext data key in hypervisor memory to encrypt disk I/O to the volume. The plaintext data key persists in memory as long as the volume is attached to the Glacier server.

Learn more about Amazon Web Services Key Management Service.

### Obfuscation Node

The Obfuscation Node is a temporary cloud based instance that acts as a redirector between the client and the Glacier VPN endpoint server. The Obfuscation Node is the only server the Glacier client knows about. At any time the Obfuscation Node can be replaced with another Obfuscation Node.

### VPN Endpoint

The VPN Endpoint is a cloud based instance that establishes tunnels with the client device. This instance does not support legacy cipher suites or protocols like L2TP, IKEv1, or RSS.

## Open Source

All components of the Android and iOS apps, along with the Push Notification server are published under open source licenses:

Apps: https://github.com/GlacierSecurityInc
Push Server (Conversations Push Proxy): https://github.com/iNPUTmice/p2

## Additional Resources

OMEMO: https://conversations.im/omemo/
Signal Protocol: https://github.com/signalapp/libsignal-protocol-c
Unofficial Objective-C wrapper for the Signal Protocol: https://github.com/ChatSecure/SignalProtocol-ObjC