Session: A Model for End-To-End Encrypted Conversations With Minimal Metadata Leakage

Kee Jefferys[*] Maxim Shishmarev[†] Simon Harman[‡]

February 11, 2020

Abstract

Session is an open-source, public-key-based secure messaging application which uses a set of decentralised storage servers and an onion routing protocol to send end-to-end encrypted messages with minimal exposure of user metadata. It does this while also providing common features of mainstream messaging applications.

1 Introduction

Over the past 10 years, there has been a significant increase in the usage of instant messengers, with the most widely-used messengers each having amassed over 1 billion users [1]. The potential privacy and security shortfalls of many popular messaging applications have been widely discussed [2]. Most current methods of protecting user data privacy are focused on encrypting the contents of messages, an approach which has been relatively successful.

This wide deployment of end-to-end encryption (E2EE) does increase user privacy; however, it largely fails to address the growing use of metadata by corporate and state-level actors as a method of tracking user activity. In the context of private messaging, metadata includes the IP addresses and phone numbers of the participants, the time and quantity of sent messages, and the relationship each account has with other accounts. Increasingly, it is the existence and analysis of this metadata that poses a significant privacy risk to journalists, demonstrators and human rights activists [3].

Session is, in large part, a response to this growing risk; it attempts to build robust metadata protection on top of existing protocols, including the Signal protocol, which have already been proven to be effective in providing secure communication channels.

[*] CTO - Loki Project : kee@loki.network

[†] Software Engineer - Loki Project : maxim@loki.network

[‡] Director - Loki Foundation : simon@loki.foundation

Session works to reduce metadata collection in several ways:

Firstly, Session does not rely on central servers, instead using a decentralised network of thousands of economically incentivised nodes to perform all core messaging functionality. For those services where decentralisation is impractical, like storage of attachments and hosting of large group chat channels, Session allows users to self-host infrastructure, or rely on built-in encryption and metadata protection to mitigate trust concerns.

Secondly, Session ensures that IP addresses cannot be linked to messages sent or received by users. This is accomplished by using an onion routing protocol called onion requests.

Thirdly, Session does not ask or require users to provide a phone number or email address when registering a new account. Instead, it uses pseudonymous public-private key pairs as the basis of an account's identity.

2 Threat Model

Before explaining Session's design, it is useful to understand the protections Session provides to users, and the threat model which it is effective in defending against.

2.1 Protections

Session aims to provide the following protections against attackers within the scope of the threat model:

Sender Anonymity: The long-term identity key of the sender is only knowable to the member(s) of the conversation, and the IP address of the sender is unknown to all parties except the first hop in the onion routing path. However, the first hop does not know the intended destination or contents of the message.

Recipient Anonymity: The IP address of the recipient is unknown to all parties except the first hop in the onion routing path. Again, the first hop does not know the origin or contents of the message.

Data Integrity: Messages are received intact and unmodified, and if messages are modified they appear as corrupted and are discarded.

Storage: Messages are stored and available for the duration of their specified time to live.

End-To-End Encryption: Messages (with the exception of friend requests) maintain the properties of the Off the Record (OTR) messaging protocol, namely Perfect Forward Secrecy and Deniable Authentication.

2.2 In Scope

2.2.1 Service Node Operators — Passive/Active Attacks

Storage of messages in Session is handled by Service Node operators. Since the Service Node network is permissionless (only sufficient stake is required to join), our threat model considers a highly resourced attacker that has limited financial resources and can only run a fraction of the storage network. A dishonest Service Node operator would be able to perform a range of active or passive attacks. Such passive attacks could include passively reading message headers, logging timestamps of when messages were relayed/received, saving the encrypted contents of a message, and assessing the size of a message. Active attacks could include failing to relay messages, failing to store messages, providing clients with modified messages, and refusing to respond to requests for messages belonging to public keys.

Service Nodes also operate the onion request system and thus could also attack it. Active attacks on the onion request system could include dropping arbitrary packets, modifying latency between hops, and modifying packets. Malicious Service Nodes would be able to continue performing these active attacks for as long as they continue to pass inter-Service Node tests. Passive attacks may involve a malicious Service Node collecting and storing all data that passes through it and logging all connections with other Service Nodes.

2.2.2 Network Adversary — Passive Attacks

Session's threat model also considers a local network adversary such as an ISP or local network provider. This adversary can perform passive attacks such as monitoring all traffic it relays, conducting deep packet inspection, or saving relayed packets for later inspection.

2.3 Out of Scope

Attackers who are out of the scope of Session's threat model may be able to break some of the protections Session aims to provide.

2.3.1 Network Adversary — Active Attacks

A network adversary could conduct active attacks including corrupting or rerouting packets, or adding delays. These attacks could compromise the storage and retrieval of messages. This is primarily addressed by encrypting data and using onion requests to store and retrieve messages, making targeted attacks by network adversaries difficult.

2.3.2 Global Passive Adversary

A global passive adversary (GPA) that can monitor the first and last hops in an onion request path could use traffic analysis to reveal the true IP address of a Session client and the destination that Session client is talking to. This potential attack is a property of the onion request system; onion requests are a low-latency onion routing network, meaning that packets are forwarded to their destinations as fast as possible, with no delays or batching. This behaviour, while beneficial for user experience, makes traffic analysis trivial in the case of a GPA.

2.3.3 Out of Band Key Discovery

Session cannot protect users from exposing the pseudonymity provided by the public key-based account system. If a user associates their real world identity with their public key, then other parties will be able to discover if they receive new friend requests.

3 Foundations

At its core, Session is built on the Loki Service Node network, so it is important to understand what this network is, how it functions, and what properties Session derives from it.

3.1 Service Nodes

Many projects have attempted to establish decentralised permissionless networks. These projects have often found themselves struggling with a 'tragedy of the commons' of sorts, wherein public servers, required for the operation of the network, are under-resourced and overused [4]. This inadvertently causes the network to provide poor service to users, which discourages further use or expansion of the network.

Conversely, those projects which are able to create large, public permissionless networks find themselves constantly facing questions about the parties that contribute to running that infrastructure. This can be especially damaging when the operation of that infrastructure can adversely affect the privacy, security, or user experience of an application. For example, the Tor network faces constant questions about evidence of Sybil attacks from unknown parties attempting to run large sections of of the public routing network, which could be used to deanonymise users [5, 6].

Session seeks to sidestep these questions by using a different type of public access network: a staked routing and storage network called the Loki Service Node network. This network is based on the Loki blockchain, which itself is based on the Cryptonote protocol [7]. Through the integration of a blockchain network, Session creates a financial precondition for anyone wishing to host a server on the network, and thus participate in Session's message storage and routing architecture.

Authorisation for a server to operate on the network is attained through the server operator conducting a special staking transaction, which requires that an operator provably lock an amount of Loki cryptocurrency assigned to their node [8](approximately 18,550 Loki coins; equivalent USD 7,420 dollars as of 10/02/2020).

This staking system provides a defense against Sybil attacks by limiting attackers based on the amount of financial resources they have available. The staking system also achieves two other goals which further reduce the likelihood of a Sybil attack.

Firstly, the need for attackers to buy or control Loki to run Service Nodes creates a feedback loop of increasing prices to run large portions of the network. That is, as the attacker buys or acquires more Loki and locks it, removing it from the circulating supply, the supply of Loki is decreased and the demand from the attacker must be sustained. This causes the price of any remaining Loki to increase, furthering the feedback loop of increasing prices. Secondly, the staking system binds an attacker to their stake, meaning if they are found to be performing active attacks, the underlying value of their stake can sharply decline as users lose trust in the network, or could be destroyed or locked by the network, in any case increasing the attackers sunken costs.

The other main advantage of a staked blockchain network is that Service Nodes earn rewards for the work they do. Service Nodes are paid a portion of the block reward minted upon the creation of each new block. This system makes Session distinct from altruistic networks like Tor and I2P and instead provides an incentive linked directly with the performance of a Service Node. Honest node behaviour and the provision of a minimum standard of operation is ensured through a consensus-based testing suite. Misbehaving nodes face the threat of having their staked capital locked, while the previously-mentioned cryptocurrency rewards function as the positive incentive for nodes to behave honestly and provide at least the minimum standard of service to the network.

3.2 Onion Requests

The other foundational component of Session is an onion routing protocol, referred to as onion requests, which enables Session clients to obfuscate their IP addresses by creating a 3-hop randomised path through the Service Node network. Onion requests use a simple onion routing protocol which successively encrypts each request for each of the three hops, ensuring:

- the first Service Node only knows the IP address of the client and the IP address of the middle Service Node,
- the middle Service Node only knows the IP address of the first and last Service Nodes, and
- the last Service Node only knows the IP address of the middle Service Node and the final destination IP address for the request.

Each Session client establishes a path on startup, and once established, all requests for messages, attachments and meta information are sent through this path. Session clients establish a path by selecting three random nodes from their Service Node list (see 4.5), which contains each Service Node's IP address, storage server port and X25519 key. Clients use this information to create an onion, with each layer being encrypted with the X25519 key of its respective service node. This onion is sent to the first Service Node's storage server; this Service Node then decrypts its layer of the onion. When a Service Node unwraps a layer, the destination key for the next node is revealed. The first Service Node decrypts its layer and initialises a ZMQ connection with the specified downstream node. When the onion reaches the final node in the path, that node sends a path build success message backwards through the path, which indicates a successful path built upon its receipt by the client.

Upon receiving the path build success message, the client will encrypt their messages with the X25519 keys of the final destination, be that a Service Node, file server, open group server, or client. The client also includes an ephemeral X25519 key in their request. When the destination server or client receives the request, they decrypt it and generate a response. This response is then sent back down the previously-established path, encrypted for the initial sender's (the client's) ephemeral key, so that the client can decrypt this response upon receiving it.

4 Building on Foundations

Onion requests provide a straightforward anonymous networking layer, and the Service Node network provides an incentivised, self-regulating network of remote servers which provide bandwidth and storage space. A number of services are built on top of this foundation in order to give Session features commonly expected of modern messaging applications.

4.1 Storage

Message storage is an essential feature for any chat application aiming to provide a good user experience. When a user sends a message, they expect the recipient to receive that message even if they turn off their device after the message has been sent. Users also expect the user on the other end to receive the message when their device wakes up from an offline state. Apps that run on decentralised networks typically cannot provide this experience, because of the lack of incentive structures and, consequently, the ephemeral nature of clients and servers on such a network. Session is able to provide message storage through the incentivised Service Node network and its usage of swarms.

4.2 Swarms

Although the Loki blockchain incentivises correct Service Node behaviour through rewards and punishments, these incentive models cannot prevent nodes going offline unexpectedly due to operator choice, software bugs, or data center outages. Therefore, for redundancy, a secondary logical data storage layer must be built on top of the Service Node network to ensure reliable message storage and retrieval.

This secondary logical layer is provided by replicating messages across small groupings of Service

Nodes called swarms. The swarm a Service Node initially joins is determined at the time of that Service Node's registration, with the Service Node having minimal influence over which swarm it joins. This protects against swarms being entirely made up of malicious or non-performant nodes, which is important to maintain the network's self-regulating properties.

Composition of each swarm inevitably changes as the networks evolves: some nodes leave the network and the newly registered nodes take their place. If a swarm loses a large number of nodes it may additionally "steal" a node from some other, larger swarm. In the unlikely event that the network has no swarms to steal from (i.e., every swarm is at $N_{min} = 5$ nodes, the 'starving' swarm (a swarm with fewer than N_{min} nodes) will be dissolved and its nodes will be redistributed among the remaining swarms. Conversely, when a large number of nodes enter the network that would oversaturate existing swarms (i.e., every swarm is already at max capacity $N_{max} = 10$), a new swarm is created from a random selection of $N_{target} = 7$ excess nodes. Note that $N_{min} < N_{target} < N_{max}$ to ensure that a newly generated swarm doesn't get dissolved shortly after and that there is still room for growth.

The outcome of this algorithm is the creation and, when necessary, rebalancing of swarms of around $N_{min} - N_{max}$ Service Nodes which store and serve Session clients' messages. The goal of the swarm algorithm is to ensure that no swarm is controlled by a single entity and that the network is resilient enough to handle both small and large scale events where Service Nodes are no longer contactable, ensuring data integrity and privacy in both cases.

The following set of simple rules ensure that Service Nodes within swarms remain synchronised as the composition of swarms changes:

- When a node joins a new swarm, existing swarm members recognise this and push the swarm's data records to the new member.
- When a node leaves a swarm, its existing records can be safely erased, with the exception of when the node is migrating from a dissolving swarm. In this case, the migrating node determines the swarms responsible for its records and distributes them accordingly.

4.3 Identity and Long-Term Keys

The majority of popular messaging applications require the user to register with an email or phone number in order to use the service. This provides some advantages, including account verification, for purposes of spam protection, and social network discoverability. However, such requirements also create some major privacy and security issues for users.

The use of a phone number as the basis for ownership of an identity key/long-term key pair weakens security against user accounts being compromised, such as in the cases of popular applications like Signal and WhatsApp. This weakness primarily stems from the fact that phone numbers are managed by centralised service providers (i.e. telecommunications service providers) who can circumvent user control, allowing these providers to assume direct control of specific users' numbers. Widespread legislation already exists to compel service providers to take this kind of action. Additionally, methods such as SIM swapping attacks, service provider hacking, and phone number recycling can all be exploited by lower-level actors [9, 10].

Signal and Whatsapp put forward varying degrees of protection against these types of attacks. Signal and WhatsApp both send a “Safety numbers have changed” warning to a user’s contacts if identity keys are changed. In practice, however, users rarely verify these details out-of-band [11, 12].

Both Signal and WhatsApp also allow users to set a “registration PIN lock” [13, 14]. This protection means that an attacker (including a service provider or state-level actor) needs access to both the phone number and the registration PIN code to modify identity keys. However, this feature is off by default, difficult to find in the settings menu, and automatically disabled after periods of user inactivity. These factors all significantly reduce the efficacy of registration PIN locks as a protective measure against the security risks of phone number-linked accounts.

Using phone numbers as the basis for account registration also greatly weakens the privacy achievable by an average user. In most countries, users must provide personally identifiable information such as a passport, drivers’ license or identity card to obtain a phone number — permanently mapping users’ identities to their phone numbers.These identity mappings are kept in private databases that can be queried by governments or the service providers that own them. There are also a number of web scrapers and indexers that automatically scrape phone numbers associated with individuals. These scrapers may target sources such as leaked databases, public social media profiles, and business phone numbers to link people to their phone numbers. Since the only method of initiating contact with a user in Signal, WhatsApp, or similar application is to know the user’s phone number, this immediately strips away user anonymity — a significant concern for whistleblowers, activists, protestors and other such users.

Account systems based on phone numbers also limit the potential for the establishment of multiple identities by a single user. These systems also prevent high-risk users without access to a phone number from accessing these services.

Session does not use email addresses or phone numbers as the basis of its account system. Instead, user identity is established through the generation of X25519 public-private key pairs. These key pairs are not required to be linked with any other identifier, and new key pairs can be generated on-device in seconds. This means that each key pair (and thus, each account) is pseudonymous, unless intentionally linked with an individual identity by the user through out-of-band activity.

4.3.1 Restoration

Because Session does not have a central server to keep records of user identities, the commonly expected user experience of being able to recover an account using a username and password is not possible. Instead, users are prompted to write down their long-term private key (represented as a mnemonic seed, referred to within Session as a recovery phrase) upon account generation. A user can use this backup key to recover their account if their device is lost or destroyed, and the user’s contacts will be able to continue contacting that same user account, rather than having to re-initiate contact with a new key.

4.4 Partitioning Identities

We have described the grouping of Service Nodes into swarms to achieve redundancy for data storage. However, we also require a scheme to balance Session users across these groups — ensuring that swarms share the storage of offline messages on the network in an equal manner. To do this, we divide the user-generated public key space into distinct deterministic groupings, and map each grouping directly to a swarm responsible for storage of messages for users within that grouping.

Each swarm (A) is assigned a 64-bit unsigned integer (A_{ID}) as its identifier. We reserve $2^{64} - 1$ as a sentinel value used to indicate that a swarm identifier is not yet known. For any pair of swarms $A, B\,(B_{ID} > A_{ID})$ we define ($D_{A,B}$) as distance between them. ($D_{A,B}$) is determined as follows:

$$D_{(A,B)} = D_{(B,A)} = min\left(B_{ID} - A_{ID,}A_{ID} - B_{ID} + 2^{64} + 1\right)$$

Note that we wrap the number line around 0 such that if we were to increment $2^{64} - 2$, we would arrive at 0. In the special case where $A \equiv B$, the distance is determined as the number of increments required to go from A_{ID}, wrap around 0 and arrive back at A_{ID}:

$$D_{A,A} = 2^{^{64}} - 1$$

The initial swarm is created as soon as there are enough Service Nodes on the network; it is assigned the identifier of 0 (the choice is arbitrary). From then onwards, whenever a new swarm is created, its identifier is chosen in such a way that would minimise the maximum distance between any two existing swarms.

Let S denote the set of existing swarms. Then for some swarms A and B $(A, B \epsilon S, A_{ID} < B_{ID})$ that satisfy $D_{A,B} = max\left(\left\{D_{(X,Y)} | X, Y \epsilon S\right\}\right)$, the identifier for the new swarm C is calculated as: $C_{ID} = A_{ID} + D_{(X,Y)}$(wrapping around 0 if necessary).

Once the identifier of every swarm on the network is known, we can create a mapping from clients' public keys to swarms. To do so we first reduce a given public key to a 64-bit unsigned integer K to match the number space of swarm identifiers. The key is then assigned the swarm whose ID is the closest on the number line to K using the same definition for distance as we used for two swarms. This approach ensures that adding or removing a swarm disturbs at most two of the neighbouring swarms.

4.5 Bootstrapping

To route messages or use onion requests, each Session client needs an up-to-date list of all Service Nodes. On first launch, a Session client fetches this list over the clearnet from a hardcoded trusted node. After first use, the Session client periodically queries multiple Service Nodes (from its previously-acquired list) for an up-to-date list. Since the Service Node list is deterministically derived from the Loki blockchain, all Service Node lists should be in sync. Session clients can avoid being partitioned onto alternate (i.e. malicious) networks by updating their lists only when all queried Service Nodes are in consensus on a new Service Node list.

4.6 Message Routing

Session follows one of two distinct cases for message routing, depending on the availability of participating clients:

4.6.1 Asynchronous (Offline) Routing

By default, or when either of the participating clients' statuses is determined as offline (see Synchronous Routing 4.6.2 for how client status is determined), Session will use asynchronous routing. In asynchronous routing, the sender determines the recipient's swarm by obtaining the deterministic mapping between the recipient's long-term public key and the currently registered Service Nodes. This information is initially requested from a random Service Node by the sender and updated whenever the client gets an error message in the response that indicates a missing swarm.

Once this mapping is determined, the sender creates the message protobuf and packs the protobuf in an envelope with the information to be processed by Service Nodes: the long-term public key of the recipient, a timestamp, TTL (time to live) and a nonce which proves the completion of the required proof of work (see Attacks — Spam 6.1). The sender then sends the envelope using an onion request to one or more random Service Nodes within the target swarm (in practice, each request is always sent to 3 service nodes to achieve a high degree of redundancy). These Service Nodes then propagate the message to the remaining nodes in the swarm, and each Service Node stores the message for the duration of its specified TTL.

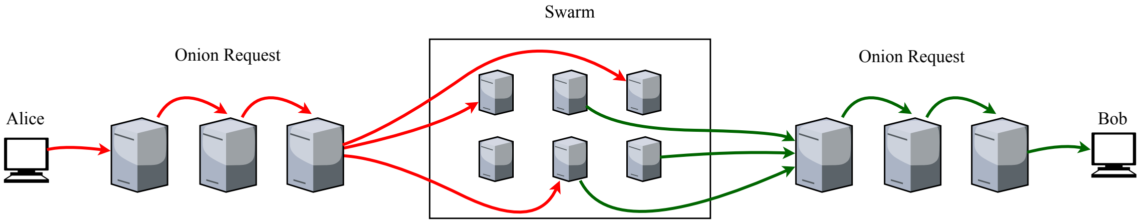

Figure 1: *Alice uses an onion request to communicate with three random Service Nodes in Bob's swarm. Bob then uses an onion request to retrieve said message, by talking to three random Service Nodes in his swarm.*Not shown here is the process of Alice's message being replicated across Bob's swarm.*

4.6.2 Synchronous (Online) Routing

Session clients expose their online status in the encrypted protobuf of any asynchronous message they send. Along with their online status, a sending client also lists a Service Node in their swarm which they are listening to via onion request.

When a Session client receives a message which signals the online status of another client, the receiver sends an onion request to the sender's specified listening node. The recipient also exposes their own listening node to the sender. If this process is successful, both sender and receiver will have knowledge of each others' online status and corresponding listening nodes. Messages may now be sent synchronously through onion requests to the conversing clients' respective listening nodes.

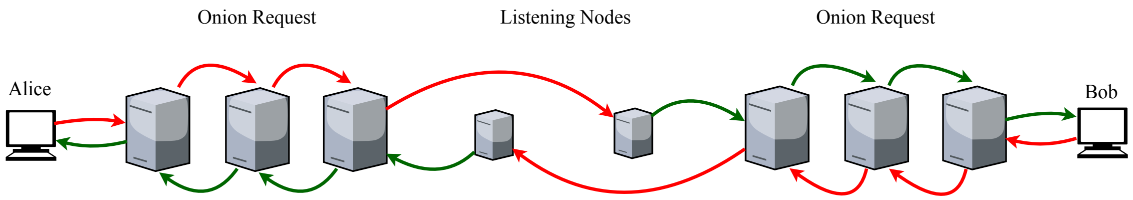

Figure 2: *Alice uses an onion request to send a message to Bob's listening node. Bob receives this message using an onion request, then sends a message to Alice's listening node.*

Messages sent using this synchronous method do not contain proof of work, and listening nodes do not replicate or store messages. To ensure messages are not lost, receiving clients send acknowledgements after receipt of each message. If either device goes offline, this acknowledgement will not be received, and the client which is still online will fall back to using the above asynchronous method of message transmission.

4.7 Encryption and the Signal Protocol

So far, we have discussed both the transport and storage of messages. However, any secure messaging application also requires message encryption in order to preserve user privacy. In order for messages to maintain perfect forward secrecy (PFS) and deniable authentication, we cannot only encrypt messages using the long-term public keys of each Session client. Instead, Session uses the Signal protocol [15].

The Signal protocol allows clients to maintain PFS and Deniable Authentication in an asynchronous messaging context after initially establishing a session using long-term keys. The Signal protocol achieves perfect forward secrecy through an Extended Triple Diffie-Hellman (X3DH) key agreement protocol and the Double Ratchet protocol for deriving message keys.

X3DH works in the following way. Consider clients A and B that want to establish a session. A and B each have a long-term identity key: (IK_a, IK_b), respectively. Additionally, each client holds a key signed with their identity key (SK_a, SK_b), that they update on a regular basis. Finally, each client generates a one-time key (OTK) for every session they want to establish.

Client A can start a session with client B if it obtains a set of $B's$ “prekeys”, consisting of $IK_b(pub)$, $SK_b(pub)$, $OTK_b(pub)$. A then validates the signature on SK_b, generates an ephemeral key EK_a, and performs a series of Diffie-Hellman derivations:

$$
\begin{aligned}
DH_1 &= DH(IK_a(sec), SK_b(pub)) \\
DH_2 &= DH(EK_a(sec), IK_b(pub)) \\
DH_3 &= DH(EK_a(sec), SK_b(pub)) \\
DH_4 &= DH(EK_a(sec), OTK_b(pub))
\end{aligned}
$$

The DH components are then concatenated and passed through a key derivation function KDF to derive a shared secret key K, which is used to initialise the Double Ratchet:

$$K = KDF(DH_1||DH_2||DH_3||DH_4)$$

Client A is now ready to start deriving message keys using the Double Ratchet, and thus start communicating with B. In the first message that it sends, A includes $IK_a(pub)$, $EK_a(pub)$ necessary for B to derive K.

The Double Ratchet uses a chain of key derivation functions KDF, each taking the previous chain key and DH parameters communicated by both clients in each of their messages, and producing the next chain key and the actual message key used for encrypting the next message. Even if some message keys get exposed, only the messages related to those keys would be compromised, and the remaining message history would continue to be hidden (the PFS property) as KDF is a one-way function. Additionally, no future messages would be exposed (the “self healing” property) as the potential attacker would be missing the necessary DH parameters to maintain the ratchet.

The Signal protocol obtains deniability through the same scheme by allowing for all ephemeral keys used in the scheme to be left unsigned by both parties. This allows any user to create ephemeral keys for any other user, combine those ephemeral keys with their own long term and ephemeral keys to produce plausible yet forged transcripts.

The Signal protocol achieves X3DH in an asynchronous environment through the use of prekeys, which contain the required information to asynchronously calculate the ephemeral keys used in the X3DH protocol. In the case of the Signal application, prekeys are stored on a central server, ensuring that these prekeys are available even when a user’s device is offline.

4.8 Modifications to the Signal Protocol

Session does not modify the fundamentals of the Signal protocol. However, in order to avoid using centralised servers, we have made some changes to the sharing of prekey bundles. In Session, the sharing of prekey bundles is conducted through the ’friend request’ system (see below). We also add additional information to each message, for the purpose of routing the message to its desired recipient and verifying that it was created correctly.

Friend Requests

Friend requests are sent the first time a client initiates communication with a new contact. Friend requests contain a short message with a written introduction, the sender's prekey bundle, and meta-information like the sender's display name and public key, which the recipient can use to respond. Friend requests are encrypted for the public key of the recipient using ECDH. When a friend request is received, the client can choose whether to accept it. Upon acceptance, the client can use the prekey bundle to begin a session as per the original Signal protocol, and start sending messages asynchronously.

5 Adding Additional Layers

By building on the above foundational elements — the Service Node network, the onion request system, and the Signal protocol — we now have a simple secure messenger which allows two users to talk to each other both synchronously and asynchronously without using a central server or exposing metadata such as IP addresses or phone numbers. However, there are certain additional features which users expect from modern messengers, the implementation of which requires further extension of Session.

5.1 Group Chats

Instant messaging applications are increasingly becoming places for communities to gather, rather than simply being used for one-on-one conversations. This has led to widespread use of group chats, channels, and similar functionality in messaging applications [16, 17]. Many of the most popular messaging applications support group chats, but the levels of encryption and privacy provided to users in these group chats is often unclear. Group chats in applications such as Telegram and Facebook Messenger only support transport encryption, rather than end-to-end encryption. Even those applications which do support end-to-end encryption in group chats (e.g. Signal and WhatsApp) still use central servers to store and disseminate messages.

There are two key areas to focus on when considering the deployment of encrypted group chats in Session.

5.1.1 Scaling

There are two main approaches to sending messages in a group chat: server-side fanout and client-side fanout. The choice of method can have a significant impact on the scalability of the group chat.

In client-side fanout, the client individually pushes their message to each recipient device or swarm. Client-side fanout is preferable in some cases since it can be done in peer-to-peer networks and does not require the establishment of a central server. However, client-side fanout can prove burdensome on client bandwidth and CPU usage as the number of group members increase — a factor which proves particularly problematic for mobile devices.

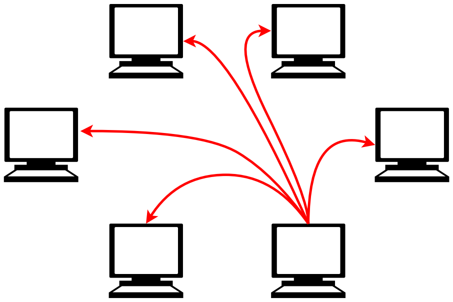

Figure 3: *Client sends message using client-side fanout*

In server-side fanout, the client typically sends their message to a server, from which the message is pushed out to each of the other clients (the other clients may also fetch the message from the server at a later point in time), which is more efficient for larger groups.

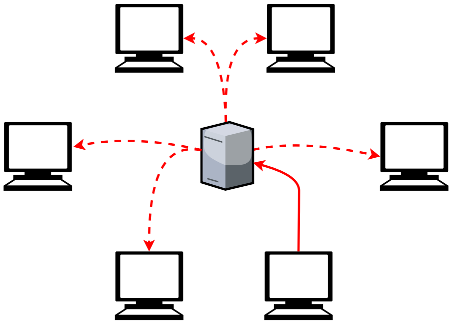

Figure 4: *Client sends message using server-side fanout: Here, the client sends the message to the server (solid red line) and the server then distributes the messages to clients (dotted red lines)*

5.1.2 End-To-End Encryption

Another factor which impacts group chat scalability is the choice of how to implement end-to-end encryption.

The most naive solution to building group chats in Session would be to simply leverage the existing pairwise sessions we can create for one-on-one conversations. To send a message to a group chat, a pairwise session would be started with every member of the group, and each message would be individually encrypted for each participant. This provides the group chat with the same guarantees possessed by standard pairwise communications using the Signal protocol: perfect forward secrecy and deniable authentication. However, this would come at the cost of requiring the payload to be encrypted and stored N times, where N is the number of members in the group. This process could become burdensome for low-powered clients participating in large group chats.

One way to improve group chats is to adopt the "Sender Keys" system used by WhatsApp [18]. This system involves a set of keys (a Chain Key and a Signature Key) that each client generates for each of its groups. These Sender Keys are shared between all group members in a traditional pairwise manner using the Signal protocol. When a client needs to send a message to the group, it derives a message encryption key using its Chain Key and encrypts the message only once. In Session, this would allow only having to generate proof of work exactly once per message, irrespective of the number of members in a group. The same ciphertext can then be decrypted by all other group members, as they can generate the same message key from the senders' chain key. Note that all future keys can be generated this way by all group members, so no further sharing of keys is necessary. However, all Sender Keys in the group will need to be updated whenever a group member leaves or is kicked from the group to ensure that they won't be able to read future messages. Additionally, this approach has the downside of losing the "self healing" property of the traditional Signal protocol provided in pairwise sessions.

The Sender Keys scheme is effective in small- to medium-sized group chats where the membership set changes infrequently. However, it can be impractical in larger groups, where users frequently leave (or are kicked from) the chat as all Sender Keys must be updated and redistributed in each such event. Further improvements to the Sender Keys scheme have been proposed in the draft MLS specification (discussed in Future Work below).

5.1.3 Other Considerations

Group Size

It may be possible to create large encrypted groups that scale well even when members are added and removed frequently. However, the reality of large groups is that as more members are added to the group, it becomes increasingly likely that members will leak or otherwise share the contents of the conversation. Identifying and removing a malicious or compromised group member in a very large group is difficult, and thus, perfect forward secrecy and deniability would be violated in such cases, unless malicious users could be identified and removed.

Proof of Work

A small proof of work must be produced for each new message which is sent offline and stored in a swarm (see Attack - Spam 6.1). In a case where many group members are offline at the same time, the sender must calculate many such proofs of work before their message can be delivered to all members of the chat, this quickly becomes taxing on mobile devices.

Metadata Protection

Information about a group chat, including the public keys of members, administrators, and the IP addresses of users, should be kept private by participants, as public availability of information about the relationships between public keys significantly reduces privacy

5.1.4 Group Type Comparisons

With the above considerations in mind, Session deploys two different schemes for the encryption and scaling of group chats, with scheme selection based on group size.

Closed Groups 3 - 500 Members

To initialise a closed group chat, a user selects a number of users from their contacts list. The user's client sends a control message through a pairwise channel to the selected users. This control message communicates the group name, group members, group avatar, and other relevant data about the group. If the group chat includes users who have not previously communicated with each other, sessions are established between these users in the background.

Using these pairwise channels, the group derives shared ephemeral encryption and signing keys. This ensures messages only need to be encrypted once for the entire group, as per the Sender Keys scheme detailed above. Instead of communicating these encrypted messages to each user in the group individually, the group chooses a random swarm to store non-pairwise messages. This ensures messages are only stored on a single swarm, regardless of group size.

Onion requests are used for transmitting messages to and from the shared swarm, and also used any time pairwise communication is required.

Closed Group Administration

The creator of a closed group becomes the administrator of that group. All users added to the group have rights to add new members, but users can only be kicked from the group by the administrator. This information is shared through pairwise channels when the group is created, and sent via a pairwise channel to new members when they join the group.

Open Groups

Large closed groups run into significant scaling issues when members leave the group, as keys must be re-derived and redistributed to the entire group — an inefficient process when there may be hundreds or thousands of members. Additionally, as previously addressed, the usefulness of end-to-end encryption in very large groups is unclear, since a single malicious group member or compromised device is catastrophic to group privacy, and in large groups this is extremely difficult to protect against, regardless of the degree of encryption deployed. In Session, once group membership reaches the upper bound for closed groups, the administrator is encouraged to convert the group into an open group. open groups revert to transport-only encryption, which protects users against network adversaries but provides comparably weak protection against server-side attacks.

To balance the risk of such attacks, Session's open groups do not use the Service Node architecture. Open groups instead require group administrators to operate their own server, or arrange for a channel to be created on an existing open group server host. The software required to do this is open-source, and a reference implementation is provided [19]. All messages and attachments stored on open group servers are fetched and posted through onion requests using the IP address or domain name of the open group host server, preserving network-layer anonymity for participants.

Open Group Administration

Administration of open groups is comparably more complex than that of closed groups. The open group server operator is the original administrator, and they are able to add new administrators. All administrators have the right to delete messages from the server. Joining rights to open groups falls into one of two categories: whitelist-based groups and blacklist-based groups. Whitelist-based groups require each user's public key to be preapproved (added to the whitelist) by an administrator, and users must be invited before being able to join the open group. Blacklist-based groups can be joined by any user who knows the domain/IP address of the group, but users can be banned if an administrator adds their public key to a list of banned public keys (the blacklist).

5.2 Multi-Device

Modern messaging applications are expected to sync data (message histories, contact lists, etc.) across multiple devices, ensuring that users are able to move between a laptop and a phone (for example) and continue their conversations where they left off. Such multi-device syncing of messages is more difficult in the context of the Signal protocol since key materials are constantly rotated and deleted after a period of time. This means when messaging other users, an out-of-sync device in a multi-device configuration will encrypt using the wrong keypairs. Sesame, Signal's session management algorithm, attempts to resolve this, but must rely on a central server to provide a consistent transcript of messages to multiple user devices [20]. Since Session does not rely upon central servers, Session requires a different solution, which we broadly refer to as 'Multi-Device.'

5.2.1 Device Linking

Once an account has been created on one device (referred to as the primary device), the other device (the secondary device) initiates the linking process, which creates a new public/private key pair. A pairwise channel is established between primary and secondary devices, and the primary device's private keys are shared from the primary device to the secondary device. To the user, it now appears that both primary and secondary devices are using the same public/private key pair.

5.2.2 Friend Requests

Consider user A with two devices (A_1, A_2) who wants to start communicating with user B, who also has two devices (B_1, B_2). User A obtains B's primary public key, i.e., the pubkey of its primary device (B_1), out of band. Using this public key, one of A's devices (A_1) sends a friend request to B's primary device's swarm containing its prekey bundle and the list of its linked devices (A_2). Both primary and secondary devices (B_1, B_2) scan the primary device's swarm periodically, looking for new friend requests. When A's friend request is found, either one of the devices is able to accept the friend request and initiate a session with A_1. The device that accepts the friend request, say B_2, then uses the pre-established pairwise channel to notify the other device B_1 of actions it has taken, and instructs B_1 to request a session with all of A's devices. B_2 also establishes any outstanding sessions with A's devices. Finally, B_2 provides A with the public key of all of its own linked devices (B_1), so A knows to link them. A and B can now begin communicating in a multi-device setting.

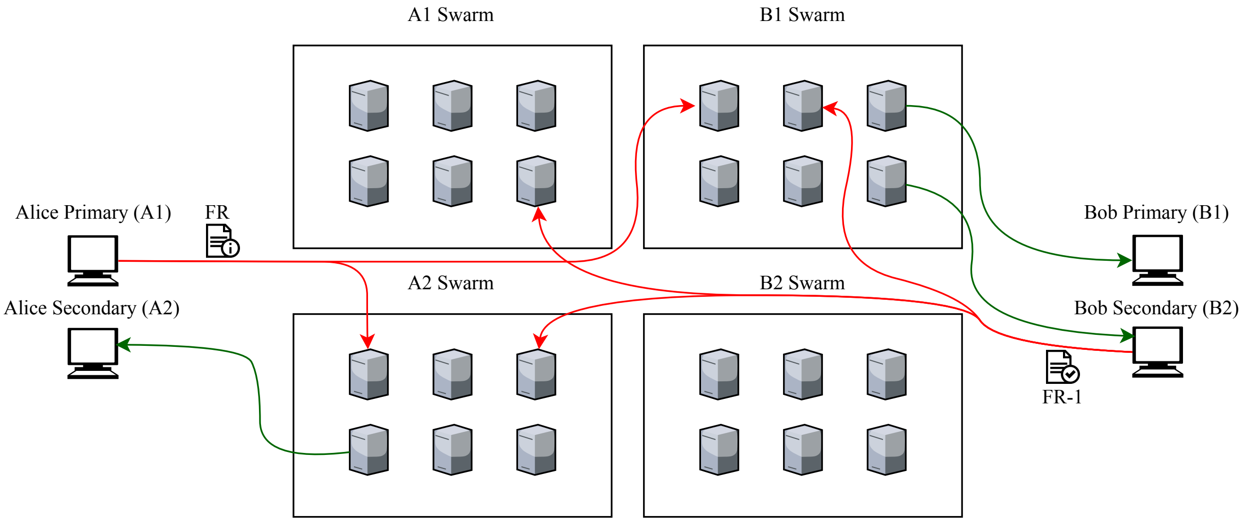

Figure 5: *Alice sends Bob a friend request with multi-device enabled*

5.2.3 Sending and Receiving Messages

After the initial friend request process, the clients can start communicating. When client A needs to send a message to a multi-device-enabled user B, they deliver the message to all of B's linked devices using the established pairwise sessions. Note that secondary devices only poll the primary's device swarm for establishing new sessions; regular data messages are associated with the device's own keys, and thus arrive at the device's own swarm. If user A also has multiple devices, they additionally send a copy of the message to all of their other devices, which appear in those device's message history as having been sent by them, ensuring that their message history is synced across devices.

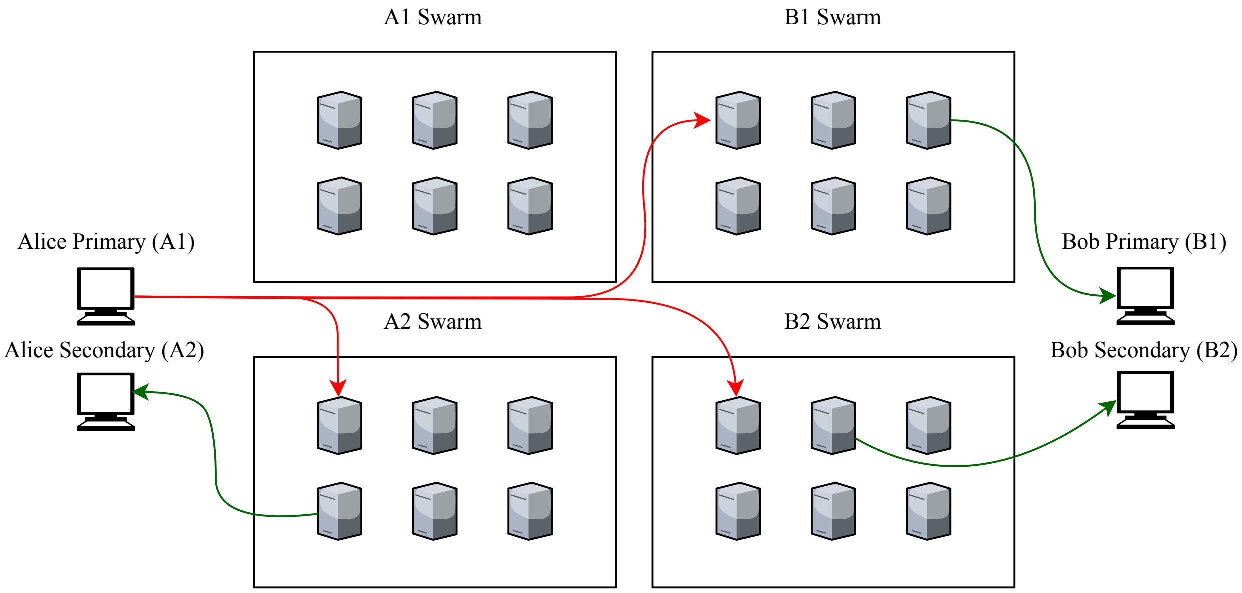

Figure 6: *Multi-device sending: Here, Alice sends a message from her primary device to Bob, who has multi-device enabled. Alice sends a message to Bob's primary and secondary device swarms. She also sends a message to her own secondary device's swarm, for later retrieval.*Not shown here: Onion requests, swarm message replication*

Open Groups

All open group messages are signed by a user's long-term identity key. Recall that secondary devices have access to the primary device's long-term identity private key as a result of the linking procedure. The secondary devices thus can sign all open group messages using that key. This ensures that other users in the open group see messages from both the primary and secondary device as coming from the same user (having been signed by the same key).

5.3 Attachments

Although Service Nodes have the ability to store data on behalf of clients, this responsibility only extends so far. Requiring Service nodes to store attachments, which can easily be orders of magnitude larger than messages (and might need to be stored for longer periods of time) would place an undue burden on the Service Node network.

With this in mind, a logical solution is for Session to interface with an untrusted centralised server that stores data obliviously. As long as the central server cannot know the contents of files, or who is storing and requesting the files, this system does not cause any metadata leakage.

This is achieved by first padding each attachment to fit within a fixed number of constant sizes between 0 and 10 megabytes, then encrypting the attachment with a random symmetric AES key. The sender then uploads the encrypted file using an onion request. In response, the file server provides a link to the piece of content, returned via the onion request path.

Once the sender obtains this link, they then send a message to the recipient via an existing pairwise session. This message contains a link to the content, a hash of the content, and the decryption key. The recipient then uses an onion request to pull the encrypted attachment from the centralised file server and decrypt it locally using the decryption key provided by the sender. The recipient also checks the hash against the attachment, ensuring the file has not been modified in transit.

By default, all Session clients use a Session file server run by the Loki Foundation for attachment sending and storage. Since attachments are not considered a core feature of Session, this design is in keeping with Session's design principles. The file server is fully open-source, with setup instructions provided so that users are able to set up their own file server [21]. Users are able to specify in the Session client which file server they want to use for attachment sending functionality. This is important both for providing users with choice and control, and ensuring the continued usefulness and functionality of Session if the Loki Foundation were no longer able to maintain the default Session file server.

5.4 Client-Side Protections

Secure messaging applications have typically focused their development efforts towards providing protections against network and server level adversaries, which has led to new advances in encryption and metadata protections. However, when interviewing high risk individuals researchers, it has been found that client-side privacy and security protections are some of the most-requested features. High-risk individuals may not be focused on protecting themselves against global adversaries, but instead against a small nation state, or corporate entity [22]. For these individuals, endpoint compromise, device seizure, and forced disclosures are described as the biggest risks. To better mitigate these risks, Session implements a number of client-side protections which allow users to better manage the security of the Session app on their device.

5.4.1 Deletion

Granular message and data deletion controls are important for users who are likely to have their devices physically seised. Session implements standard features like disappearing messages, which are deleted from sending and receiving clients after being viewed, and the ability to fully wipe all client side stored data. However, Session also features additional ways to manage client side security.

5.4.2 Duress Codes

Users may set a PIN or pattern lock to access the Session app, which adds additional security on top of any device-level passcodes. As an additional layer of security, users may also specify a duress code, which if entered in lieu of the standard Session app PIN, will wipe Session app data on the device. This is useful in cases where users are forced to unlock their devices and wish for it to appear as if there was never data to begin with.

5.4.3 Remote Deletion

Remote deletion allows a user to specify a trusted friend and negotiate a shared secret with that friend. Once this secret is generated and stored on the device, the trusted friend can generate a remote deletion message which reveals this prearranged secret. When this message is received by the user's device, it initiates the immediate destruction of their local database.

5.4.4 Pseudonyms

High-risk users such as whistleblowers often need to create accounts which are not linked to any real-world physical identifiers (e.g. phone numbers and email addresses). Session account creation only requires generation of a public-private key pair, making it trivial for users to establish multiple pseudonyms without needing to link their account to pieces of information which could be used to identify them.

5.4.5 Backup and Restore Account States

Border crossings or checkpoints can be an area of significantly increased risk for high-risk users. In these zones, high-risk users may be forced to disclose passwords and surrender devices so device images can be taken. To protect their data, some high-risk individuals have begun implementing a strategy of backing up device and application data, wiping their device to cross a border or pass through a checkpoint, and then restoring that data once it is safe to do so. To ease this process, Session supports encrypted backups to a number of popular cloud services. Backups are encrypted with a symmetric key derived from the user's Session long-term private key, meaning the user only needs knowledge of their 12 word mnemonic seed (recovery phrase) to recover their account after completing the border or checkpoint crossing.

5.5 Testing

Service Nodes are rewarded for providing services to the network in an honest and consistent manner. Consequently, dishonest Service Nodes must be prevented from refusing to store messages for the network while continuing to collect rewards. This is accomplished through Service Node testing, a network-level system of peer policing.

5.5.1 Service Node Testing

Every Service Node monitors the state of the Loki blockchain, which periodically generates blocks whose hashes are relatively unpredictable. On every block, Service Nodes use this blockhash to deterministically derive a pair of nodes within each swarm: one to be tested T and one to perform verification V. Since T and V both belong to the same swarm, they are expected to store the same database records. As a way of testing this, Node V selects a random record from its database (verifying that the message indeed belongs to the current swarm) and sends a test request to node T, containing only the record's hash and the current block height h. T is expected to respond with the record's actual data. Note that the official binaries used by honest Service Nodes do not expose an endpoint for retrieving a record by its hash, so a cheating node would generally have to download the entire database from one of its honest peers to search for the requested record, making cheating impractical.

When T receives a test request, it first confirms that (T, V) is correct for the specified height h and that h is within some reasonable boundary. It then tries to retrieve the requested record from its own database. Note that due to the nature of message propagation, it is possible for V to hold a record for a message before T first receives it, so T will wait for a short time for the message to arrive, and only then respond to the test. When the requested record is obtained, T can respond to the test with the requested record's data. If V does not receive the response within some acceptable time window or it is incorrect, V reports T to the blockchain as having failed the test. In cases of repeated failures, the Service Nodes as a collective might then decide to decommission or deregister T if it consistently fails storage tests as reported by multiple other Service Nodes.

6 Attacks & Future Work

6.1 Spam

Since Session accounts are simply public-private key pairs, they can be generated en masse extremely easily. Client-based Sybil attacks are much easier to execute, making moderation of public chats and even personal conversations challenging. Further, because each identity key pair can store data on the Service Node storage network, an attack which generated large numbers of key pairs and sent junk messages between those key pairs would be an effective way to congest Service Nodes' storage space and deny service to legitimate users.

Session deploys two methods to limit this type of attack.

6.1.1 Proof of Work

Any message that is to be stored on the Service Node network requires an attached proof of work. The goal of this proof of work requirement is to make conducting spam attacks on the Loki storage server network computationally expensive and, thus, less practical. This is specifically accomplished by Session clients attaching a nonce to sent messages that will be stored offline.

In order for a message to be validated and stored on the Service Node network, this nonce must reach a specific difficulty threshold when hashed alongside the payload and a timestamp of the message. This threshold is dynamic, derived from the global proof of work difficulty setting D, the specified time to live of each message τ , and the (non-zero) length of the message in bytes L. Messages with a larger τ (i.e., those intended to be stored for longer) require more proof of work. The parameter D is also dynamic, set according to the load on each swarm: it rises as the number of messages stored by swarms over a certain time period increases.

Let's imagine that a client wants to store an encrypted message C, of size L, on a swarm. It performs the necessary proof-of-work procedure by concatenating bytes of C with the bytes of various routing information to produce payload P:

$$P = \tau || timestamp || pk || C,$$

where *timestamp* is the time of message creation on the client and *pk* is the recipient's public key. The client proceeds by randomly generating nonce (a 64-byte unsigned integer) and hashing it together with P to produce H:

$$H = \mathrm{SHA}_{512} \left(\mathrm{SHA}_{512} \left(P \right), nonce \right)$$

The following equation defines how *threshold* is calculated as a function of D, τ and L:

$$threshold = \frac{2^{64-1}}{D \left(L + \frac{\tau L}{2^{16}-1} \right)}$$

The proof of work is considered complete if $H < threshold$, which is determined by reducing H to a 64-byte unsigned integer to match the number space of threshold and comparing them as regular integers. Note that a smaller *threshold* leads to harder proof of work, so messages with larger τ or L require more computations on the client device.

Finally, once the right *nonce* is found, the client is ready to make a store request on a swarm, which consists of the original message C along with its metadata and the value of *nonce*. Service Nodes in the swarm then validate the proof of work by recomputing H and *threshold*. If they satisfy the requirement $H < threshold$, the message is stored for the duration τ and gets propagated to other nodes in the swarm. If the proof of work is not sufficient, which could happen if, for example, D has changed, the client receives the new value of D so it can repeat the procedure with the updated requirements.

6.1.2 Administrative Controls

The ability for users to rapidly create new accounts without attached, verifiable real-world identifiers creates issues for open group administrators — an administrator may ban a certain public key, only for that user to simply create a new key and rejoin the chat. Whitelisted chats have been implemented as an answer to this problem. As detailed in the open group section above, whitelisted chats require an administrator to either explicitly invite a given public key to the chat, or generate invite links that are either time-limited or limited based on public keys which join using a given link. Additional protections such as requiring an existing trusted member of the chat to 'vouch for' a new member could also be effective protections against these attacks.

6.1.3 Future Spam Resistance

There are a number of issues with the use of proof of work as a spam prevention mechanism. The primary issue is that due to their relative lack of processing power, most consumer-grade mobile devices can only produce extremely low-difficulty proofs of work without placing undue strain on the device, draining its battery, and introducing a long delay before a message is sent, compromising user experience. Conversely, specialised professional or enterprise-grade hardware can produce high-difficulty proofs of work with ease, and thus there is no concern with regards to the user experience of the time taken to send a message.

There are several potential methods which would increase Session's resistance to these types of attack. One such method would be a CAPTCHA-type system where users would have to pass a CAPTCHA check if their client was sending an unusually high volume of messages. Alternatively, Session clients could be required to store encrypted data of other uses before being able to send messages, effectively requiring Session clients to provide services to the network in exchange for the rights to send messages. Adding a micropayment requirement to each message would also prevent this issue, but would severely impact the usability of the application.

6.1.4 Limited TTL

TTL is the time for which a message will be held by its recipients' swarm before being deleted. The maximum Time To Live (TTL) for any message is 96 hours . This is potentially limiting, as some messages may be required to be stored for longer periods of time before the recipient is expected to be able to receive that message. The maximum TTL is set to ensure the network can clear messages progressively and also so that spam attacks can be quickly recovered from. As the network scales, we can investigate how much storage is actually being used on in the Service Node network which will allow us to better understand what impact longer and shorter TTLs will have on Service Nodes.

6.2 Path Selection Algorithms

When making an onion request, a Session client selects routers randomly from the list of available Service Nodes. We could further strengthen Session's protections against traffic analysis attacks by ensuring routers are chosen from distinctly different geographical zones to ensure that a single onion request does not use multiple nodes in a single datacenter or operated by a single provider.

There are also open questions about whether onion requests should subject first hops to any additional requirements beyond the staking requirement already necessary to operate a Service Node. Networks like Tor traditionally set higher requirements for first-hop nodes because these nodes (guard nodes, in Tor's case) have access to users' IP addresses directly. Consequently, Tor's guard nodes are expected to provide additional bandwidth and uptime to better prevent Sybil attacks on the Tor network. However, this is less of an issue for Session's onion requests as each router must already have a significant stake in the network in order to operate.

6.3 Payments

Session is enabled by services provided through the Loki blockchain network. The Loki cryptocurrency is a fundamental part of these services, providing an anonymous way to transfer value between people. A lightweight wallet integrated into the Session app, using keys derived from the user's existing keypair, could allow users to quickly and privately transfer value inside Session.

6.4 Social Network Discovery

As previously discussed, common messaging applications like WhatsApp, Facebook Messenger, and Signal typically require users to provide their phone number or email address upon registration. These apps are then able to hook into users' contact lists and social networks to discover which of a user's phone contacts or social network friends are already using the application, users are then able to quickly and easily initiate contact within the app. This is a powerful feature, as users do not need to spend time establishing connections with other users or building a new social network within the app.

However, this type of contact discovery is not well-suited to Session's design or its philosophy, as this functionality intrinsically requires users to link their accounts to a real-world identity to enable discovery. A better, more privacy-friendly potential approach might be to provide new Session users with the option to send a message through their existing communications channels (such as SMS and email), inviting their contacts to download Session and add them via their public key.

6.5 Loki Name Service (LNS)

Session long-term public keys are 66-character alphanumeric addresses. To add another user as a contact, you must know their long-term public key. Right now, users can only add a new contact by scanning a QR code representing this public key, or by directly receiving the full 66-character key out of band (typically via relatively insecure communications channels such as an unsecured messenger, SMS, or email).

This is not ideal from a user experience or privacy standpoint. Preferably, users would be able to reserve a short, human-readable username and share this username with friends, while still maintaining their security and privacy. LNS will allow users to register mapping between a long-term public key and a shorter, human-readable string of text. This mapping is stored in the Loki blockchain through the creation of a special Loki transaction which burns a small amount of Loki to register a name. This transaction requirement exists to guard against frivolous name registrations.

By storing these mappings in the Loki blockchain — which is propagated to every registered Service Node — we can ensure that each Service Node has a copy of every public key-username mapping, and that those mappings cannot be modified. When a client wants to add a new contact using a username, the client has several options. It can sync the entire Loki blockchain and re-derive the mappings itself, it can connect to a number of random Service Nodes via onion requests and ask for the public key that the username in question belongs to, or they can sync only the Loki blockchain headers and ensure that the returned mappings from Service Nodes include merkle inclusion proofs increasing their security.

6.6 Traffic Analysis

The goal of onion requests and other low-latency onion routing solutions is to forward messages from their origin through to their final destination as fast as possible while traversing a number of routers (hops) in between. Unfortunately, these types of onion routing networks are susceptible to traffic analysis attacks due to their reliance on the internet, which is a highly centralised system [23, 24]. If both ends of an onion request connection between Alice and Bob are monitored by a co-opted ISP, then a path between the two can be discovered by inspecting Alice's outgoing encrypted packets, and correlating those with Bob's incoming encrypted packets. Even if packets are padded to be a constant size, Alice's ISP could introduce a delay (or drop packets) and, with the help of Bob's ISP, watch as that delay or packet loss propagates into Bob's connection.

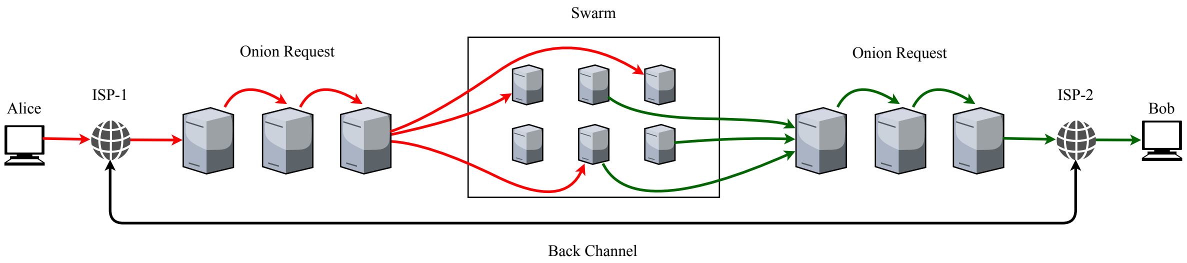

Figure 7: *ISP 1 communicates with ISP 2 through a back channel. ISP 1 gives Alice's encrypted packets to ISP 2, who looks for packets with a similar structure being downloaded through Bob's connection.*

Worse still, state-level actors can directly monitor encrypted packets as they move over the physical infrastructure of the internet, often not requiring the explicit cooperation of ISPs [25].

6.6.1 MixMode

MixMode is a future project attempting to build a traffic analysis resistant layer for Session which operates on top of the existing low latency onion routing infrastructure. In order to achieve this, MixMode will build upon ideas drawn from two existing systems: Bitmessage and mix networks.

Bitmessage

Bitmessage is a public key encrypted messenger with two key points of difference from most encrypted messengers: first, messages are sent using a gossip protocol (all users receive and forward all messages), and second, messages are sent without a destination address [26].

Advantages:

- Traffic analysis (even by a GPA) becomes extremely difficult. It is not possible to know whether a user is receiving a message or simply flooding someone else's message, since Bitmessage clients/users relay all messages they receive – even ones that they successfully decrypt.
- Although all clients are routers, routers can't gain any information about the messages they relay, as messages contain only a small proof of work and no destination address.

However, there are some significant downsides – namely, all messages must have an attached proof of work, all clients must relay and store all messages before their elapsed TTL, and all clients must trial-decrypt every message they relay.

Mix Networks (Mixnets)

Mixnets are systems in which messages propagate through several layers of nodes (mixers) running on an overlay network. Each mixer holds and delays messages until a certain number of messages have been sent, or a set time interval is reached. The mixer then sends a batch of their held messages to another mixer. This makes traffic analysis difficult, as network adversaries are required to track batched messages. This has significant anonymity benefits, as even if only one mixer in a path is honest (not leaking each message's source and destination), malicious mixers (or network adversaries) are unable to effectively track the flow of messages.

Local network adversaries such as dishonest ISPs are typically thwarted by padding messages and encrypting connections between hops. Timing attacks can be defeated using special decoy messages which loop back to the sender, acting as cover traffic.

Mixnets do provide significant benefits, but ultimately their reliance on the generation and registration of users with federated providers is suboptimal for implementation in Session.

Goals

MixMode should protect user privacy from GPAs able to monitor all user-to-Service Node and Service Node-to-Service Node network traffic. MixMode should also allow for an adversary to own up to $N - 1$ of the set of nodes queried (where N is the total number of nodes queried) without compromising the privacy of the query.

Sending

MixMode would involve a few minor changes to message sending. However, these changes still allow for normal and MixMode clients to communicate without needing prior knowledge of whether MixMode has been turned on by the recipient. The major change to message sending is that all clients — even those who do not opt in to MixMode — would start producing cover traffic messages.

Cover messages are dummy messages periodically sent to users in your contacts list to obfuscate real network traffic. They are not visible to users, and exist only for obfuscatory purposes. In order to generate the proof of work to send cover messages after successfully calculating a nonce for a primary message, a client will (for a short time) continue to calculate proof of work, searching for a new nonce which hashes to the output of a cover message. By adjusting the duration for which a client continues to search after they produce the required proof of Work for their primary message, we can effectively tune the amount of cover traffic on the network.

Receiving

Changes to the message receiving procedure would only affect users who enable MixMode. Primarily, these changes provide message fingerprint anonymity and decrease the chances of passive data collection by dishonest Service Nodes.

Receiving clients who have enabled MixMode would periodically send a fixed-size private information retrieval query to three Service Nodes. These three nodes would respond with a fixed-size response, regardless of whether a message is retrieved or not. With this data, the recipient can successfully recover either a dummy message, or a real message addressed to their public key. This process should be ongoing and occur in the background as long as the client is alive.

Private Information Retrieval

Private Information Retrieval (PIR) schemes are protocols which allow a user to query a database (in our case, a Service Node) without the database learning what information was selected or returned to the user. PIR schemes improve upon the efficiency of a naive method in which the client simply requests and downloads all of the messages the Service Node stores, then sorts its messages from others locally.

Since Session has multiple Service Nodes in a single swarm holding the same data, we would likely employ an information-theoretic PIR scheme. An information-theoretic PIR scheme holds that as long as a single server (out of the multiple servers queried) is honest — i.e., does not collude and share the user's query or the returned response — the privacy of the query and the response is maintained.

There are a few libraries which implement information theoretic PIR, like Percy C++ [27]. We will be testing a number of schemes to find the most efficient overall implementation to include in Session.

Performance

MixMode will introduce some performance overheads, including increased bandwidth usage and increased time taken to receive messages (due to the regular periodic schedule at which clients request messages from their swarms under MixMode). There are also some additional bandwidth and processing overheads imposed on both clients and Service Nodes through the use of a PIR scheme, when compared with directly requesting messages. These overheads will need to be taken into account when considering MixMode's implementation.

6.7 Open Groups and MLS

Message Layer Security (MLS) is a new interoperable standard being developed for encrypted messaging on the internet, similar to TLS, with the key difference being that MLS specifically focuses on secure group messaging [28].

The primary goal of MLS is to provide increased efficiency for large group chats when a group is updated and pairwise rekeying needs to occur. MLS improves upon the Sender Keys method by using binary trees, ensuring that when rekeying occurs, key information only needs to be encrypted for the nodes along the path to the root instead of each user individually. This reduces the overhead for kicking a member and rekeying the group from $O(N^2)$ in Sender Keys to $O(logN)$ in MLS.

MLS could be applied to further expand E2E encryption in closed group chats. However, as previously discussed, MLS cannot protect and does not aim to protect against a malicious member or compromised device in a large group.

6.8 Destination Privacy

To ensure messages are delivered to the correct swarm, a sender must insert the public key of the recipient unencrypted in the header of the message. This is not ideal, because if a user's public key is linked to their real-world identity out of band, then Service Nodes in that user's swarm can work out when there is a new message destined for them.

Instead of using long-term keys in message headers, sender and recipient should negotiate ephemeral tags to attach to the header of each message. These ephemeral tags can identify the destination swarm without referring to the recipient's long-term public key. When receiving a message, a client polls their swarm and asks for messages related to the ongoing tags they are following. This ensures recipient long-term keys cannot be associated with messages (beyond friend requests) sent on the Service Node network, regardless of a user's handling of their anonymity out of band.

6.9 Multi-Device Scalability

The current methods used for multi-device syncing incur significant overheads for each sent message. In a worst-case scenario where both sender and receiver have two devices, each message between these users must be sent to three different swarms. Ideally, this overhead could be reduced by relying more on direct synchronous connections between primary and secondary devices when both are online, or establishing common swarms between devices.

6.10 Open Group Discoverability and DNS

When establishing an open group server, an operator must have access to a domain name for the server. This complicates the setup process, and also makes Session's open group transport encryption reliant on the certificate authority (CA) network. It would be preferable if, when joining an open group, a user was given an IP address and the ED25519 public key of the server. This allows us to remove the need to trust CAs and simplify the process of setting up an open group server.

However, this also creates a usability issue: under this system, it becomes difficult to quickly communicate the information required to join an open group in person without resorting to QR codes or insecure backchannels. We could optionally allow open group operators to buy LNS mappings to map IP and key combinations to human-readable names; these mappings would be stored in the Loki blockchain in a similar fashion to the usernames in LNS. This would allow us to achieve a high level of security while maintaining human-readability. Using this scheme would also allow open groups to be discovered by downloading a list of registered open group names from the Service Nodes.

7 Conclusion

The internet is under ever-increasing surveillance by local, state, and corporate actors. With this mounting invasion of privacy in mind, it is more crucial than ever that people have access to tools and applications which allow them to communicate online, whilst hiding both the contents of their conversations, and the very existence of those conversations.

By combining some of the strongest privacy-enhancing technologies available today, including the Signal protocol, onion routing, and decentralised message storage and retrieval, Session provides users with minimal metadata leakage and strong message encryption.

Session provides access to these privacy tools in the form of a user-friendly application which also provides features which users expect from a modern messenger, including multi-device, attachments, and group chats. Future work on the Session application will provide users with additional features, in pursuit of greater levels of security and privacy.

8 Addendum

This paper does not describe Session as it exists in its initial release — some of the features described in this paper will still need additional work before being deployed to the live Session application. The most significant deviations from this paper and the Session application as of February 11th, 2020 are mentioned below:

- Onion requests are not yet implemented in the initial release, and are replaced with Service Node proxy routing, which provides only a single hop, instead of the three hops which will be provided by onion requests.
- In the initial release, proxy routing applies when communicating with Service Nodes and Loki File servers, but not Loki open group servers.
- The proof of work difficulty threshold is controlled centrally, so that it can be manually raised if Service Nodes are being spammed.
- Mappings between devices for multi-device are stored encrypted on the Loki File server instead of being communicated when adding a friend or joining an open group.
- Some client-side protections, including duress codes, remote deletion, and backup and restore, are not yet implemented.
- Service Node testing related to Session functionality is implemented and running, but not yet enforced.
- Closed groups in the initial version of Session are implemented using the naive pairwise channel approach, and capped to a maximum of 10 users.

References

[1] *Most popular global mobile messenger apps as of October 2019, based on number of monthly active users* (November 20, 2019), https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/.

[2] *Privacy and Data Protection in Smartphone Messengers* (2019), https://publications.sba-research.org/publications/paper_drafthp.pdf.

[3] *Federal police accessed the metadata of journalists nearly 60 times* (2019), https://www.smh.com.au/politics/federal/federal-police-accessed-the-metadata-of-journalists-nearly-60-times-20190708-p52598.html.

[4] *The Ethereum Classic GAS System and The Miner Tragedy of the Commons*, https://etherplan.com/2019/09/22/the-ethereum-classic-gas-system-and-the-miner-tragedy-of-the-commons/8789/.

[5] *Honey Onions: a Framework for Characterizing and Identifying Misbehaving Tor HSDirs*, http://www.ccs.neu.edu/home/amirali/publications/HOnion_CNS_2016.pdf.

[6] *Identifying and Characterizing Sybils in the Tor Network*, https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_winter.pdf.

[7] *Cryptonote 2.0*, https://cryptonote.org/whitepaper.pdf.

[8] *Loki Documentation Loki Service Node Staking Requirement*, https://docs.loki.network/ServiceNodes/StakingRequirement/.

[9] *Large-scale SIM swap fraud*, https://securelist.com/large-scale-sim-swap-fraud/90353/.

[10] *WhatsApp FAQ - Phone Number already in use*, https://faq.whatsapp.com/en/android/24068052/.

[11] *Is that you, Alice? A Usability Study of the Authentication Ceremony of Secure Messaging Applications*, https://www.usenix.org/system/files/conference/soups2017/soups2017-vaziripour.pdf.

[12] *Action Needed! Helping Users Find and Complete the Authentication Ceremony in Signal*, https://www.usenix.org/system/files/conference/soups2018/soups2018-vaziripour.pdf.

[13] *WhatsApp FAQ - Using two-step verification*, https://faq.whatsapp.com/en/android/26000021/.

[14] *Registration Lock*, http://support.signal.org/hc/en-us/articles/360007059792-Registration-Lock.

[15] *Signal Documentation*, https://signal.org/docs/.

[16] *U.S. group chat usage frequency by age 2017*, https://www.statista.com/statistics/800650/group-chat-functions-age-use-text-online-messaging-apps/.

[17] *Most Americans rely on group chats to keep up with family and Friends*, https://today.yougov.com/topics/lifestyle/articles-reports/2018/01/16/most-americans-rely-group-chats-keep-family-and-wo.

[18] *WhatsApp Encryption Overview*, http://www.cdn.whatsapp.net/security/WhatsApp-Security-Whitepaper.pdf.

[19] *loki-project/session-open-group-server*, https://github.com/loki-project/session-open-group-server.

[20] *Signal Specifications The Sesame Algorithm: Session Management for Asynchronous Message Encryption*, https://signal.org/docs/specifications/sesame/.

[21] *loki-file-server*, https://github.com/loki-project/loki-file-server.

[22] *Can Johnny Build a Protocol? Co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols*, https://pdfs.semanticscholar.org/41e4/f623d838ead1a782de46a94e44fb762cdd32.pdf.

[23] *Approximating a Global Passive Adversary Against Tor*, https://pdfs.semanticscholar.org/aa31/f6e8676e6c1a531854e4337649f0556e9280.pdf.

[24] *Recent Attacks On Tor*, http://www.cse.hut.fi/en/publications/B/11/papers/salo.pdf.

[25] *NSA slide shows surveillance of undersea cables*, https://www.washingtonpost.com/business/economy/the-nsa-slide-you-havent-seen/2013/07/10/32801426-e8e6-11e2-aa9f-c03a72e2d342_story.html.

[26] *Bitmessage: A Peer-to-Peer Message Authentication and Delivery System*, http://kevinriggen.com/files/bitmessage.pdf.

[27] *Percy++ / PIR in C++*, http://percy.sourceforge.net/.

[28] *Messaging Layer Security - IETF*, https://datatracker.ietf.org/wg/mls/charter/.