Groove: Flexible Metadata-Private Messaging

Ludovic Barman
EPFL

Moshe Kol
Hebrew University of Jerusalem

David Lazar
EPFL

Yossi Gilad
Hebrew University of Jerusalem

Nickolai Zeldovich
MIT CSAIL

Abstract

Metadata-private messaging designs that scale to support millions of users are *rigid*: they limit users to a single device that is online all the time and transmits on short regular intervals, and require users to choose precisely when each of their buddies can message them. These requirements induce high network and energy costs for the clients, restricting users to communicate via one powerful device, like their desktop.

Groove is the first scalable metadata-private messaging system that gives users *flexibility*: it supports users with multiple devices, allows them to message buddies at any time, even when those buddies are offline, and conserves the user's device bandwidth and energy. Groove offers flexibility by introducing *oblivious delegation*, where users designate an untrusted service provider to participate in rigid mechanisms of the metadata-private communication. It provides a differential privacy guarantee on par with rigid systems like Stadium and Karaoke.

An evaluation of a Groove prototype on AWS with 100 servers, distributed across four data centers on two continents, demonstrates that it can achieve 32s of latency for 1M users with 50 buddies in their contact lists. Experiments with a client running on a Pixel 4 smartphone show that it uses about 100 MB/month and increases battery consumption by 50 mW (+16%) compared to an idle smartphone. These measurements illustrate that Groove's flexibility allows supporting mobile devices, covering an important class of users that prior systems could not support.

1 Introduction

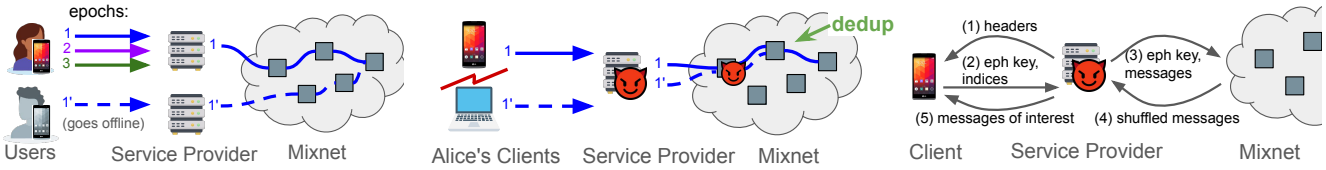
There has been significant recent progress in scalable metadata-private messaging systems. These systems give a strong privacy guarantee by hiding who communicates with whom and can support more users by deploying proportionally more servers, which also allows better privacy to everyone [11]. However, they impose *rigid* requirements on users [18, 21, 22, 28, 29], who need to synchronize messaging into rounds and coordinate precisely in which rounds they would communicate with their buddy. If two buddies are not simultaneously online, they cannot communicate: they cannot store messages for one another to fetch later to avoid a correlation between the messages' age and the users who were online in time to send them. Moreover, prior work allowed users to receive messages only from one buddy per round. It relied on a hefty dialing protocol that allows users to switch between buddies [23, 29]. If the message is short or urgent, dialing may bear an excessive overhead for users.

The rigid requirements turn into high costs for running the systems' clients. Since communicating users must be simultaneously online, an attacker monitoring the network can correlate buddies over time. To combat such attacks, clients submit and poll messages at every round, leading to high bandwidth and energy overhead. This makes running a client on a phone prohibitively expensive and effectively requires the users to have an always-on desktop computer (e.g., at home). However, many users do not have such a computer, e.g., if they use a laptop that they carry with them. Moreover, users may only run a client on one device, or risk revealing which messages they send if their devices partition and accidentally interact with the system in the same round. These limitations stand out compared to traditional messaging systems (without metadata privacy) and hinder adoption.

We present Groove, the first *flexible* messaging system that provides metadata-privacy and scales well to support a large user base. Groove's users can send messages to any of their buddies from any device, and can go offline and retrieve messages sent to them later. Groove provides similar performance to recent metadata-private communication systems under the same global active adversary model [17, 18, 21, 22, 28]. Like these systems, Groove builds on a mixnet where servers shuffle messages in batches to unlink them from their senders. Mixnets are inherently rigid: they require users to submit a message in every round to mix all conversations together, and to receive messages from the mixnet at the same rate regardless of their conversations to avoid correlated traffic patterns between buddies. To handle the mixnet's rigidity, Groove introduces *oblivious delegation* to an untrusted service provider: users interact with a service provider, who participates in the communication protocol on their behalf and synchronizes between their devices. Oblivious delegation ensures that an adversary learns nothing about users' communication metadata by controlling service providers, even if the users' devices go offline or get partitioned.

Achieving oblivious delegation involves three mechanisms:

- **Non-interactive setup** (Figure 1a). Establishing a message channel between buddies requires them to submit just one setup message through the mixnet, without waiting for their



(a) *Non-interactive Setup*. Groove's async setup protocol enables metadata-private messaging between buddies, even if they are not simultaneously online. The service provider buffers setup messages and submits them to the mixnet at the right time.

(b) *Uncoordinated Replacement*. Groove allows multiple clients (on different devices) of the same user to interact with the system safely, even if they cannot coordinate. Clients can replace old messages buffered at the service provider to support new buddies.

(c) *Oblivious Fetch*. Groove's clients avoid retrieving dummy messages stored at the service provider, without revealing which messages they fetch. The servers only process a user's messages once per client, regardless of the number of messages stored.

Figure 1: The three mechanisms that allow oblivious delegation in Groove.

peer's response. A non-interactive setup protocol is crucial since buddies might not be simultaneously online to run the protocol. Groove hides communication metadata even if only one buddy attempts this setup (or their peer's provider discards the other setup message). Message channels persist for long epochs, and this protocol allows Alice to prime setup messages at her service provider well in advance and for many epochs. Service providers submit setup messages to the mixnet at the right time, allowing Alice's buddies to send her messages even she goes offline.

- **Uncoordinated replacement** (Figure 1b). The adversary may partition a user's devices and prevent them from deciding which device communicates at a given round. If several devices of the same user accidentally submit messages to the same buddy in the same round, they can expose the recipient, who receives extra messages. Groove solves the issue with a new message-replacement technique: each device can independently refresh the messages queued at the service provider without coordinating with other devices. Crucially, the protocol ensures that no metadata is revealed to the adversary, even if the provider is rogue and submits all messages (old and new) to the mixnet. This is achieved through path selection and message-tagging mechanisms, which ensure that each buddy receives at most one message from all of the user's devices. The replacement protocol allows clients to update setup messages, and hence to add buddies after priming message channels for future epochs. It also allows users to switch between their devices, like traditional messaging applications.
- **Efficient messaging with many buddies** (Figure 1c). Groove avoids expensive dialing (e.g., as in [23, 29]) by keeping message channels open between a user and her buddies. To minimize client costs when having many message channels, Groove's clients use a submission protocol that avoids sending a dummy message for each idle channel, and a fetch protocol that avoids retrieving dummy messages from idle channels. Crucially, both protocols are oblivious and do not reveal sensitive information to the service provider (e.g., which channels were active and carried real messages from buddies). Finally, Groove minimizes the cost that each message channel induces on the mixnet, with the most crucial improvement over prior work being the memory footprint. This allows supporting 75M parallel message-exchanges with 100 servers (cf., prior work supports 1M-10M parallel message-exchanges in the same deployment [18, 21, 22, 28]).

We analyze Groove's design and show that it achieves differential privacy against an attacker who has complete control over the network and has the power to compromise many servers that make up the system (including all service providers). Through this analysis, we choose the system's parameters that would provide a strong degree of privacy.

To demonstrate that Groove can support a large user-base, we built a prototype and evaluated its performance on AWS servers in two continents. To demonstrate that Groove allows users to run clients on relatively weak devices, we also implemented a mobile client, deployed it on a Pixel 4 phone, and tested Groove in terms of battery and network usage. We show that Groove scales well with the number of servers and using 100 servers, it supports 1–3 million users sending and receiving messages from 50 buddies every 32–80 seconds. The client uses 54 MB–106 MB of network per month and its battery consumption increases by 16% compared to the idle phone (when using cellular data). We recognize that Groove's latency is high compared to traditional messaging apps; nonetheless, it removes the rigid client requirements of previous designs and enables large-scale metadata-private messaging for mobile users.

In summary, our contributions are the following:

- oblivious delegation, an approach for offloading rigid mixnet requirements to an untrusted service provider.
- Groove, a metadata-private messaging system that scales well to support a large user-base and provides flexibility to clients using oblivious delegation.
- an analysis proving Groove's privacy guarantee.

- an implementation of Groove and an experimental evaluation of its performance, showing that it can support a large user base and mobile clients.

2 Related Work

Differential privacy. Groove's approach of providing differentially-private communication using a mixnet is based on prior work [21, 28, 29]. However, these earlier designs impose rigid requirements to their users: First, users must be simultaneously online, and their clients must send and receive messages at every round to communicate. Second, users can only run a client on a single device; thus, prior works avoid the problem of device partitions, which happens when a user's devices cannot coordinate sending messages (this reveals the buddy, who receives more messages as a result). Groove addresses these problems through oblivious delegation.

Private Information Retrieval (PIR). PIR protocols support storing and retrieving information asynchronously from a database, without revealing anything about which message is retrieved [9, 25]. These protocols could provide a good degree of flexibility to clients: they have low client bandwidth requirements and allow the recipient to read a message at any time after the sender deposits it in a database. However, as users' messages accumulate in the database, processing each client's PIR query requires more work from the database server, limiting scalability. For example, Pung [4, 6] supports up to 20k–60k fetches per minute with 1M messages in its database. Express [13] optimizes PIR retrievals, at the cost of making writes more expensive: the system only supports sending tens of messages per second across all clients [13]. In contrast, Groove clients induce the same amount of work independent of how many messages accumulate in the system, enabling millions of users to send and receive messages every minute.

Communicating with many buddies. Metadata-private messaging systems minimize the load on the servers and overhead for the client by limiting users to receive messages from one buddy at a time [4, 6, 18, 21, 28, 29]. They require buddies to run a hefty dialing protocol to agree on when to communicate. For PIR systems, such as Pung and Addra [3,6], relying on dialing is even more essential; since reading is expensive for the servers, clients cannot retrieve messages from each buddy all the time. Many systems [3, 4, 18, 21, 28] propose using Alpenhorn's differentially private dialing [23], which takes about 5 minutes to coordinate between buddies and requires 62 GB of client bandwidth per month [23, §8.2]. Groove mitigates the clients' costs through its oblivious message fetching protocol and minimizes the load each message channel induces on the system, so users can keep channels with many buddies (e.g., 50 buddies per user in our implementation). Dialing may also be seen as a limited form of asynchronous messaging between users. The Vuvuzela [29] and Alpenhorn [23] dialing protocols use 3000×– 4000×more bandwidth on clients than Groove[1].

Metadata-private communication over persistent channels. Groove's users communicate over persistent message channels, similar to Tor's circuits. However, Tor does not protect users from a strong adversary model. An adversary monitoring the network can infer the path of users' messages across relays by correlating incoming and outgoing packets. Hydra [27] uses circuits to connect two endpoints, like Tor's hidden services [7], but does not hide the number of active conversations to the adversary [27, §4.6¶1], which allows for intersection attacks. Yodel [22] uses a mixnet with persistent circuits but requires two communicating users to be online at the same time to coordinate their circuits. Both Hydra and Yodel support only one device per user and have clients send and receive messages every round, two rigid properties that Groove addresses.

Flexible clients through trusted servers. Loopix [26] hides metadata by relaying messages through a mixnet. It addresses the clients' flexibility problem by storing messages for users at trusted service providers. However, these service providers see when their users receive a message, and can thus perform intersection attacks with other users who were online in time to send this message. Thus, users must make a tricky decision about which service providers they trust. This design also gives a single target for learning about the relationships of all of a service provider's users. Pond [19] requires users to run a trusted server themselves. This leads to a simpler design than Groove (e.g., partitioned devices are not a concern) but limits the system to savvy users who can securely operate their own servers.[2] This is problematic for metadata-private communication, which benefits from a large user-base [11]. In Groove, users maintain privacy even if their service providers fall under the attacker's control, so they do not need to operate servers themselves.

3 Overview

Groove allows buddies that share a secret to secretly exchange short text messages (e.g., 100B). Each buddy can run clients on multiple devices and seamlessly switch between them. Service providers bridge between the rigid mixnet protocol and the flexible clients, that can send and receive messages asynchronously. Each user designates a service provider and runs Groove's oblivious protocols for setting up message channels (Figure 1a), replacing messages (Figure 1b), and fetching messages (Figure 1c). The service provider submits messages to the mixnet on the user's behalf and stores messages that the user receives from the mixnet. It also helps

[1] When configured to provide a 1-min messaging latency like Groove.

[2] Running a personal server in the cloud undermines security against a strong adversary that may be able to compel the cloud operator to disclose data or exploit side channels by deploying a VM on the same machine, and operating a physical server requires significant effort and expertise.

the user synchronize her clients across all devices. Users need not trust their service provider for privacy. To anonymize messages, Groove uses a parallel mixnet [14]. Parallel mixnets scale well with the number of servers by offering many parallel routes for processing messages. Groove extends the servers' message processing logic to enable oblivious delegation, and it is otherwise agnostic to the mixnet's internal design choices, such as the server topology or verification protocol (that ensures servers process messages correctly).

3.1 Threat model

Groove aims to hide its users' communication metadata from a *global active attacker* that controls all network links. In particular, this attacker observes when clients (dis)connect from the network and may partition users' devices. The attacker may also control all service providers, and each mixnet server with some probability f, which is provided as an assumption in the system's configuration. A smaller f yields better performance at the expense of a stronger trust assumption. The attacker can also run arbitrarily many clients. Still, an attacker could learn about a user's communication by directly compromising their device or the devices of their buddies [5], although forward secrecy prevents them from learning about the user's communication in the past. Finally, Groove assumes that the attacker is computationally bounded, so standard cryptographic primitives, such as hash functions and encryption schemes, are secure.

3.2 Goals

Privacy. Groove achieves differential privacy [12]. Consider an attacker and their view of the system through all network links and service providers, and the mixnet servers and clients that they operate. Groove ensures that the attacker's view is likely to be the same whether two users, call them Alice and Bob, are buddies or not. More formally, consider the attacker's observations $\mathcal{O}$ and the following two scenarios. In one scenario, Alice and Bob are buddies (and can chat), and in the other, they are not (and cannot chat). Groove ensures the following inequalities for small $\varepsilon, \delta \geq 0$:

$$\begin{aligned} Pr[\mathcal{O}|\mathrm{A} \leftrightarrow \mathrm{B}] &\leq e^{\varepsilon} Pr[\mathcal{O}|\mathrm{A} \not\leftrightarrow \mathrm{B}] + \delta, \\ Pr[\mathcal{O}|\mathrm{A} \not\leftrightarrow \mathrm{B}] &\leq e^{\varepsilon} Pr[\mathcal{O}|\mathrm{A} \leftrightarrow \mathrm{B}] + \delta \end{aligned} \tag{1}$$

In Equation 1, $\mathrm{A} \leftrightarrow \mathrm{B}$ denotes Alice and Bob being buddies, and $\mathrm{A} \not\leftrightarrow \mathrm{B}$ denotes them not being buddies. That is, the probability for any observations the attacker could make is close under both scenarios (up to small constants ε, δ). Informally, Alice being buddies with Bob appears to the adversary almost as likely as them not being buddies. Thus, Alice could plausibly deny being buddies with Bob or claim to be buddies with anyone else. This privacy guarantee holds even if Alice or Bob is the sole honest user of a rogue service provider, the attacker partitions their devices, and observes the system for a long time.

Client flexibility. Groove should not impose strong timing or resource requirements on clients. It should allow Bob to retrieve Alice's message at any time after it reaches Bob's service provider and accommodate clients with network and battery constraints (that need to minimize communication or that might go offline). In particular, Groove should support clients running on mobile devices. At the same time, Groove should allow other clients to connect more often and achieve lower message latency. Finally, Groove should enable users to run clients on multiple devices and switch between them.

Performance. Groove aims to support millions of users with dozens of buddies. Once Alice sends a message, Bob can retrieve it with a latency on the order of a minute. The system should scale, i.e., provide the same performance to a larger user base by deploying proportionally more servers.

Availability. Groove's availability guarantees in the face of failing servers should primarily come from the mixnet and not degrade by its oblivious delegation approach. An overloaded or downed service provider can prevent service to its users, but the system's availability for users of other service providers should not be affected. Since the service provider is untrusted, it can use replication for better availability guarantees without security implications for its users. Moreover, having untrusted service providers also allows users to change to another provider if they suspect their provider is preventing service without risking exposing sensitive information to more parties (the current and new provider). We describe how users can detect and combat such providers in Groove's design (§5).

Today's mixnet designs usually provide weak availability guarantees and risk halting when mix servers go offline, which Groove inherits. Improving this aspect of mixnets is orthogonal to Groove, which will benefit from future improvements to mixnet availability.

4 Background

Groove's users send and receive messages over message channels called *circuits*. A circuit is a fixed route of servers in the mixnet that persists for an *epoch*, which consists of many communication rounds (e.g., an hour to a day's worth). Each circuit connects a user's service provider to a *dead drop*, an ephemeral address where users exchange messages. Two buddies coordinate pseudorandom dead drops using a shared secret. The mixnet ensures that an adversary cannot correlate which service provider connects to what dead drop using noise messages. Groove borrows this communication model from previous systems [7, 21, 22, 27] (see §2), but changes the way circuits establish to support oblivious delegation. We summarize below the existing techniques that Groove uses to simplify the exposition of its new mechanisms in §5.

Like other mixnets (e.g., [17, 28, 29]), Groove's mixnet servers have unique public keys per epoch. Clients know the servers' public keys, e.g., through a transparent public

key infrastructure [20, 24]. The way rounds are kicked-off depends on the mixnet, which Groove abstracts; e.g., many designs use an untrusted coordinator that announces to all mix servers when to start new rounds [17, 18, 21, 22, 28, 29].

Messaging over circuits and dead drops. The circuit setup message is onion encrypted with the public keys of the servers on the route. Each onion layer includes the next server's id, an ephemeral Diffie-Hellman public key, and an authentication code that ensures no earlier server has modified the onion. Each server completes the Diffie-Hellman handshake to derive a shared symmetric key with the client, and uses this key to verify the authentication code. After receiving messages from all servers in the previous hop, the server deduplicates and shuffles messages, and forwards them to the next hop. Servers store their shuffle's permutation and the symmetric keys; later, they use these records to process messages over the circuit.

During a communication round, mixnet servers process one message on each circuit. Each message is onion encrypted with the symmetric keys registered at circuit setup. The mixnet's servers shuffle and decrypt messages they receive using the permutation and symmetric keys they stored earlier. Servers deduplicate messages on the same circuit. If a server does not receive a message on a circuit, it fills in a dummy message to ensure all circuits have one message. The symmetric encryption ensures that any random message that a server fills in is indistinguishable from a real message to other servers along the route. When two circuits connect to the same dead drop, the server hosting its address swaps the messages on these circuits and sends them back through the circuits, which is how buddies exchange messages.

Noise. Previous work shows how mixnet servers can noise messaging [21]; we apply it in the context of Groove's circuits. For each inter-server mixnet link, each server decides on the number of noise circuits to route over that link by drawing from the Poisson distribution. Servers generate two kinds of noise: "doubles" which are a pair of circuits bound to the same dead drop to obscure the number of buddy-relationships, and "singles" which is a circuit terminating at a dead drop without a pair, to obscure the case where one of the buddies does not create their circuit to the shared dead drop. Much like Karaoke, Groove uses Bloom filters to ensure that malicious servers do not drop the noise circuits.

5 Design

Figure 1 illustrates the parties in Groove: users, clients running on different types of devices, service providers, and the mixnet. When Alice and Bob become buddies, they add each other to their address books, and their clients establish a fresh shared secret. This secret allows Alice and Bob's clients to authenticate and encrypt messages (end-to-end) and coordinate dead drops for exchanging messages. The clients might create this secret out-of-band (e.g., by scanning QR codes if users meet in person) or via a metadata-private "add-friend" protocol (like in Alpenhorn's protocol suite [23][3]).

```
while true {
   // Block, waiting for the next wakeup event.
   <-client.Schedule // §5.1

   if oncePerDay {
      client.RefreshCircuits() // §5.2, §5.3 and Fig. 4
      client.ForwardSecrecy() // §5.5
   }

   buddy, msg := client.OutgoingMessageQueue.Pop()
   // If the user has nothing to say, send cover traffic.
   if buddy == nil {
      msg = random.Bytes(MessageSize)
   }
   client.SendMessage(buddy, msg) //§5.4 and Fig. 5
   client.CheckForMessages() // §5.4 and Fig. 6
}
```

Figure 2: The client's main loop. The client refreshes circuits once per day, at that time it also evolves the multidevice and buddy keys for forward secrecy. It sends and receives messages according to the schedule, which can be configured to balance battery life with communication rate.

Users designate a service provider that connects their clients to the mixnet and participates in the rigid mixnet protocol on their behalf. The mixnet operates in rounds, where users exchange messages over the circuits. We envision rounds being relatively frequent, e.g., starting every 30 seconds to a minute. Every round, service providers submit messages to the mixnet. Its servers shuffle messages and ensure each circuit carries precisely one message per round (by deduplicating messages on the same circuit and filling in a dummy message when one is missing, as §4 describes).

In the remainder of this section, we introduce the concept of client schedules, present the protocols for oblivious delegation from Figure 1, and the mechanism for forward secrecy. Our descriptions follow the client's operation, outlined in Figure 2, and its interactions with the service provider via the API depict in Figure 3.

5.1 Client schedules

To achieve Groove's privacy goal, the network communication pattern between a user's client and their service provider must not reveal information about the user's communication with her buddies. In particular, this pattern includes when the client tries connecting to the service provider, which we call the client's *schedule*. The adversary can potentially infer any information that goes into deciding the client's schedule, and hence it should be independent of the user's buddy-relationships, which Groove aims to hide. Groove gives flexibility for clients to operate by their own schedule, independent from other clients. In this manner, it can accommodate clients on low-power devices with lightweight schedules without impacting other clients. A simple safe schedule is to communicate with

[3] Alpenhorn's add-friend protocol differs from its *dialing* protocol, which precedes every conversation – a cost that Groove avoids.

```
var B = MaxBuddies
type Onion = [MessageSize + SymmetricOnionOverhead]byte

rpc BeginTransaction() *Txn
rpc (t *Txn) Commit() error

rpc (t *Txn) GetAddressBook() ([]byte, int)
rpc (t *Txn) SetAddressBook(data []byte, round int)

// RPC to set the circuit setup onions for an epoch.
// Each buddy corresponds to 2 circuits.
rpc (t *Txn) SetEpochOutgoing(epoch int, onions [B][2][]byte)

// RPC to fetch messages from our buddies.
rpc (t *Txn) GetHeaders(epoch int, round int) [B][2]byte
rpc (t *Txn) ShuffleInbox(ShuffleParams) [][]byte
rpc (t *Txn) FetchInbox(DHkey []byte, indices []int) [][]byte

// RPCs to queue a message for a buddy.
rpc (t *Txn) GetRoundOutgoing(epoch int, round int) RoundData
rpc (t *Txn) SetRoundOutgoing(epoch int,round int,rd RoundData)

type RoundData struct {
   // Messages are split into two, one for each buddy circuit.
   Onion [2]Onion
   // Previously sent message for this round.
   OutboundMsg []byte
}
```

Figure 3: Service provider API. Clients use these RPCs asynchronously to setup circuits to their buddies and send/receive messages through them.

the service provider at regular intervals. Clients can use different intervals to trade network and power consumption for communication latency. They can also piggyback on other device wake-ups (like checking for software updates) to interact with the service provider at a relatively low cost.

Clients can change their communication patterns if this change is independent of the user's buddies. For instance, it is safe for Alice's client to skip transmissions due to a network outage or because Alice boards a flight and her phone disconnects from the Internet. It is also safe for users to have correlated schedules, as long as the correlation is not caused by their relationship (being buddies or not). For example, users in the same time zone may prefer their devices to be more conservative during the day, when on battery, but less at night, when near a power outlet. Such correlations do not leak new information to the attacker (i.e., he already observes their IP addresses and can deduce the geographic location). However, changes in the client's network patterns that depend on a user's buddies are unsafe; e.g., if Bob's client stops sending messages whenever Alice's device goes offline, an adversary might infer that Bob is connected with Alice.

5.2 Non-interactive circuit setup

Groove splits time into epochs, which correspond to the circuits' lifetime. Periodically, e.g., once a day, clients call `RefreshCircuits` to generate circuit setup messages and upload them to their service provider (see the client's main loop in Figure 2). The service provider queues these messages and sends them to the mixnet at the appropriate time (sending circuit setup messages for one epoch when the preceding

```
func (c *Client) RefreshCircuits() error {
  epochs := c.serviceProvider.UpcomingEpochs()
  txn := c.serviceProvider.BeginTransaction()

  // Get address book and epoch of last update.
  addressBook, epochUpdated := txn.GetAddressBook()
  addressBookKey := c.MultiDeviceKey[epochUpdated]

  // Merge address books (buddy lists) across devices.
  prevBuddies := Decrypt(addressBookKey, addressBook)
  c.buddies = MergeAddressBooks(prevBuddies, c.buddies)

  // Pad the buddy list so its size doesn't reveal anything
  // to the provider and so we generate noise onions below.
  if len(c.buddies) < MaxBuddies {
      c.buddies = append(c.buddies, GenerateFakeBuddies())
  }
  newBook:= Encrypt(c.MultiDeviceKey[c.currentEpoch],c.buddies)
  txn.SetAddressBook(newBook, c.currentEpoch)

  for epoch := range epochs {
   var onions [MaxBuddies][2][]byte
     for i, buddy := range c.buddies {
        // Devices use the same PRNG to choose circuit
        // routes & tags, enabling deduping setup messages.
        randRouteTag := PRNG(i, epoch, c.MultiDeviceKey[epoch])
        onions[i][0] = GenCircuitSetupMsg(randRouteTag,buddy,0)
        onions[i][1] = GenCircuitSetupMsg(randRouteTag,buddy,1)
     }
     txn.SetEpochOutgoing(epoch, onions)
  }
  return txn.Commit()
}
```

Figure 4: Pseudocode for updating a client's address book and corresponding circuit setup onions for upcoming epochs, which are stored on the service provider. It is safe for multiple devices to run this function concurrently.

epoch nears its end), even if all of the user's clients go offline. Users exchange messages over circuits, so adding or removing buddies only takes effect on epoch boundaries, when circuits are established. The more epochs a client covers in `RefreshCircuits`, the longer the user can go offline and keep receiving messages from her buddies. If all of a user's clients remain offline beyond the epochs they have covered with setup messages when calling `RefreshCircuits`, Groove will eventually not be able to establish circuits with the user's buddies, preventing them from communicating. (Groove noises circuit setup to obscure this case, where a user does not establish circuits with her buddies, and maintains its differential privacy guarantee.) The epoch's duration is a knob that allows Groove to trade less client communication for higher latency in setting up circuits with new buddies.

Figure 4 gives the pseudocode for `RefreshCircuits`. First, the client synchronizes the user's contacts through the service provider, since the user might have added or removed a buddy through another device. The service provider's `BeginTransaction` and `Commit` APIs allow each of the user's clients to retrieve and upload data atomically with respect to the user's other clients, which may simultaneously call `RefreshCircuits` (rogue providers can still break the transactional semantics or deliver different address books to different clients, causing users to update stale address books;

Groove protects against such providers, as we prove in §6). The client retrieves the address book from the service provider, appends new buddies to the first available slots, and pushes the new address book to the service provider. Clients pad the address book to `MaxBuddies` slots and encrypt it under the multidevice key, which hides when users add or remove buddies.

Next, the client primes two circuit setup messages for each buddy and many upcoming epochs (e.g., for the next month) and uploads these messages to the service provider. The reason for creating two circuits per buddy, rather than one circuit, is to allow clients to fake connections to buddies when the user has less than `MaxBuddies` friends, to hide the number of her friends. In this case, the client creates two "dummy circuits" to one dead drop, so there are precisely two circuit setup messages to all dead drops a client uses (regardless of the number of buddies the user has). The client learns the epoch number from the service provider and relies on it to submit circuit setup messages at the right epoch; however, the provider can cheat. Thus, the client writes the epoch number in each onion layer, so the mixnet servers can discard circuit setup messages the provider sends at the wrong time. As one epoch nears the end, the mixnet runs circuit setup for the next epoch, so circuits are ready at all times to route messages between buddies. The mixnet's servers then process the setup messages as in previous works (§4).

Previous works, however, required buddies to run an interactive protocol before they could chat. In this protocol, clients submit circuit setup messages, then check their circuits were established, and only then can notify buddies of the circuits they created so the buddies can connect [22]. Without this coordination, the attacker could learn which users communicate. Groove cannot coordinate between users at circuit setup time since users might submit setup messages for the next month and go offline (as illustrated in Figure 1a). Therefore, it must ensure privacy when one user establishes a circuit and their buddy does not. Groove handles this challenge by noising the circuit setup step with dummy circuits that its mix servers create. The dummy circuits ensure that, regardless of whether Alice and Bob are buddies, the attacker observes the same traffic pattern (on the network and to dead drops). Groove applies Karaoke's noising technique [21], of creating "single" and "double" dead drop accesses, to circuit setup messages (summarized in §4).

Circuit setup messages in Groove are acknowledged to the clients, which then learn whether anyone dropped their circuit or the circuit from their buddy. This allows users to detect active attacks and, as in previous systems [21,22], provision a tighter privacy budget when choosing the system parameters (and thus, achieving better performance) compared to systems that do not detect active attacks (like [28,29]). To achieve this in Groove, the content of circuit setup messages is a pseudorandom ID that each user derives from the secret they share with their buddy (or the multidevice key if the circuit is dummy). When the circuit setup message reaches the dead drop, the server hosting that dead drop swaps the content of messages (see §4) and returns it through the mixnet to the user's service provider. The next time the client connects to the provider and learns the current epoch, it downloads the returned IDs of the circuits from previous epochs and checks they are correct by deriving the IDs the buddies would use. Correct IDs acknowledge to clients that the circuit setup message from them and their buddy had propagated to the dead drop, so all servers in the route shuffled these messages with the other setup messages.

5.3 Oblivious replacement

Priming circuits for future epochs allows supporting users that go offline for a long time. However, users might add buddies after submitting circuit setup messages. Groove allows updating circuit setup messages stored at the service provider, so users can communicate with new buddies soon after adding them to their address books. Each client performs this replacement periodically, according to its schedule (Figure 2); if there are no changes in the address book, the client uploads fresh circuit setup messages pointing to the same dead drops. The key challenge in performing this replacement is that, without coordination across the user's devices, several of her clients might establish circuits to the same dead drop. This will create a distinct access pattern (i.e., not the single- or double- dead drop access patterns covered by the noise). An attacker controlling the dead drop's hosting server can then associate that dead drop with the user.

Groove introduces circuit tagging, which enables safe replacement of old setup messages without relying on communication between a user's devices, as illustrated in Figure 1b. When choosing the circuit's path, `RefreshCircuits` seeds a pseudorandom number generator (`randRouteTag` in Figure 4) for each epoch with the multidevice key and the buddy's slot number in the user's address book. This pseudorandom number generator is the same across all of the user's devices. The clients then use it to choose the route for each address book slot and include a pseudorandom tag derived from this generator in each layer of the circuit's setup message. Honest servers on the route deduplicate circuit setup messages according to this tag. Although the routes are the same, each client submits different-looking messages to the service provider since the onion encryption scheme is randomized.

This route and tag selection procedure ensures that she submits circuit setup messages with the same route and tags across all her devices for each buddy in the user's address book. Even if all of Alice's clients upload circuit setup messages and a malicious service provider submits all of them, the first honest server along the (identical) route observes the duplicated tags and discards the redundant messages. This ensures the user's circuits do not access a dead drop an unusual number of times.

```
func (c *Client) SendMessage(buddy, msg string) error {
   epoch := c.currentEpoch
   round := c.nextRound

   txn := c.serviceProvider.BeginTransaction()
   rd := txn.GetRoundOutgoing(epoch, round)

   prevMsg, prevBuddy :=
      Decrypt(c.MultiDeviceKey[epoch], rd.OutMsg)
   if IsRealMessage(prevMsg) {
      err = "refusing to overwrite user-typed message"
      // Re-encrypts previous content.
      msg, buddy := prevMsg, prevBuddy
   } else {
      msg = MsgHeader(epoch, round, buddy.key[epoch]) ++ msg
   }

   msg1, msg2 := SplitMessage(msg)
   ix := GetBuddyIndex(buddy)
   onion1:= EncryptSymOnion(epoch.circuits[ix][0].keys,msg1)
   onion2:= EncryptSymOnion(epoch.circuits[ix][1].keys,msg2)

   txn.SetRoundOutgoing(epoch, round, RoundData{
      Onion:  {onion1, onion2},
      OutMsg: Encrypt(c.MultiDeviceKey[epoch], {buddy, msg}),
   })

   return txn.Commit(), err
}
```

Figure 5: Client pseudocode for sending messages.

5.4 Efficient messaging

Groove uses oblivious protocols for efficiently submitting and fetching messages over many concurrent circuits.

Sending messages. Groove maintains circuits to `MaxBuddies` friends. Submitting a message per buddy every time the client's schedule triggers (Figure 2) might create a deployment limitation (e.g., for mobile devices with high bandwidth costs). Groove avoids this issue. When a client's schedule triggers, it submits just one message (split into two parts, leveraging both circuits per buddy) and does not reveal the designated buddy to the service provider. Instead, the service provider broadcasts it on all of the user's circuits. Messages are encrypted end-to-end, so only the intended recipient can decrypt them. If a rogue provider does not broadcast a message, the first honest mix on each path will fill in a dummy message to ensure buddies keep receiving messages at the same rate (§4). Figure 5 gives the client's pseudocode for sending messages.

Fetching messages. The recipient's service provider receives messages from the mixnet and stores them for the clients to fetch later. (It receives one message per circuit per round, as ensured by the mixnet's servers, see §4.) Clients should avoid fetching dummy messages to minimize their bandwidth and energy costs (e.g., messages that mixnet servers fill in or that are intended for another buddy), and at the same time, hide which circuits carry real messages to hide when someone messages the user. PIR protocols also allow this functionality, albeit at a high cost from the service provider (since a client performs PIR for every message it downloads, and each PIR grows expensive as messages accumulate; see §2). Instead, Groove's fetch protocol relies on mixing messages, where a set of servers processes the messages from the provider just once regardless of the number of messages a client retrieves. We describe it following the illustration in Figure 1c and the pseudocode in Figure 6.

```
func (c *Client) CheckForMessages(epoch int) ([]int,[][]byte){
   round := c.GetNextRound()
   mixers := RandomMixnetPath(11) // Path of length 11

   hdrs := c.serviceProvider.GetHeaders(epoch, round)

   // Identify the indices of real messages from buddies
   indxs = []
   for i, buddy := range c.buddies {
      if hdrs[i] == MsgHeader(epoch, round, buddy.key[epoch]){
         indxs = append(indxs, i)
      }
   }

   // Shuffle user's inbox with a fresh key
   pk, sk := GenerateDHKeypair()
   nonces := c.serviceProvider.ShuffleInbox(ShuffleParams{
      Epoch:          epoch,
      Round:          round,
      PublicKey:      pk,
      Mixers:         mixers,
   })

   // Predict the indices after mixing step in ShuffleInbox
   shuffledIxs := PredictPositions(sk, mixers, nonces, indxs)

   // Ensure we fetch a constant number of messages
   PadWithFakeRequests(shuffledIxs)

   // Fetch messages at the (shuffled) indices
   onions := c.serviceProvider.FetchInbox(shuffledIxs)

   // Remove onion encryption from ShuffleInbox's mixing step
   msgs := DecryptOnions(onions, sk, mixers, nonces)

   // Map the messages back to the correct buddy
   Unshuffle(msgs, sk, mixers, nonces)

   return indxs, msgs
}
```

Figure 6: Client pseudocode for oblivious fetch. Clients first fetch headers from the service provider and identify indices of interest. Finally, they request the shuffled indexes corresponding to the result of the mixing step of the oblivious fetch.

When Bob's client calls `CheckForMessages` from its main loop (Figure 2), it first retrieves from the service provider a short header for each message it stores for the user (e.g., two bytes). The header acts as a pseudorandom flag, shared between the two buddies: when Alice sends a message to Bob, her client derives the header's value from the current round and the key that Alice and Bob share, and sends along with the content (inside the onion). Bob's client derives the same headers and compares them against the headers from Alice's circuits. If they match, his client knows to fetch the corresponding message next. To avoid revealing messaging rates between buddies, the service provider must not learn from which circuits Bob fetches messages. Groove hides this by mixing Bob's messages again, as follows.

`CheckForMessages` also instructs the service provider to submit all messages pending for Bob to a "fetch mixchain," which is a sequence of mixnet servers chosen by the client. The client also supplies a new Diffie-Hellman public key, which the service provider relays to the first server in the sequence. The first server uses its secret key to complete the Diffie-Hellman handshake and derives a shared secret with the client; it then chooses a fresh nonce and hashes it with this shared secret to derive an ephemeral symmetric key. The server derives the shuffle permutation from this ephemeral key and encrypts the messages. The server passes its nonce, the client's public key, and the list of the remaining servers to the next mixchain server, which continues in the same fashion. The nonces ensure that a mixchain server's output looks freshly random, even if a rogue service provider replays an old request. The last server passes the shuffled messages and servers' nonces to the recipient's service provider. The service provider gives the mixchain servers' nonces from the client. The client then derives the symmetric key for each server and computes the (shuffled) position of the messages it should fetch. Finally, it fetches messages at the shuffled indices directly from the service provider; the freshness of the client's Diffie-Hellman key ensures that it accesses random-looking locations each time it runs the protocol.

Clients download a small fixed number of messages for every round; say, download six messages per round to support up to three buddies simultaneously messaging the user. This way, the number of messages clients retrieve does not reveal information about messaging rates to the attacker. The oblivious fetch mechanism also lets clients retrieve a different number of messages to quickly catch up after being offline for an extended time; for example, fetch 500 messages per day. Clients may also run this procedure every day to catch up at relatively low cost in case their user receives a burst of messages from many buddies in a few rounds during that day.

5.5 Forward secrecy

Groove achieves forward secrecy for both the message contents as well as metadata, meaning that an adversary that compromises a user's device cannot retroactively decrypt messages, determine who the user communicated with, or who was in the user's address book. Note, however, that an adversary that compromises a user's device does get to see the device's current address book and messages.

The challenge in achieving forward secrecy in Groove is that user devices may be partitioned from one another, and thus cannot refresh their keys by coordinating over the network. To achieve forward secrecy in this setting, Groove deterministically evolves secret keys based on the epoch: for each passing epoch, clients hash the keys (as described below) and erase the pre-image. Clients evolve the multidevice key, ensuring that an adversary compromising a device cannot track old circuit routes from the user to a dead drop, and cannot decrypt old copies of the user's address book (metadata forward secrecy). Clients also evolve shared secrets across buddies to ensure that the adversary cannot decrypt old messages (data forward secrecy).

Deterministically evolving keys could allow an adversary that obtains old keys (e.g., by compromising a user's old powered-off phone) to derive all future keys, thereby compromising data and metadata privacy for all epochs since the stale compromised key, a form of post-compromise insecurity [10]. To avoid this vulnerability, Groove involves the servers in computing the hash function for evolving keys, and honest servers refuse to compute this hash function for epochs in the past or that are more than T epochs in the future. This limits the vulnerability window by ensuring that an adversary with access to older keys cannot evolve them forward to decrypt newer user data or metadata. It also lets the client have keys for T epochs in the future to allow priming circuits (Figure 4).

Realizing this approach is challenging because it requires combining secrets from two parties, without either party learning the other's secret: the client wants to hash a key without giving it to the server, and the server must not reveal its secret that would allow hashing arbitrary values in the future. We address this by using an oblivious pseudorandom function (OPRF), specifically, the verifiable DH-OPRF construction from [15]. Groove's clients run the verifiable DH-OPRF protocol with each server (evolving their keys with each server in turn). This ensures that the keys are evolved with at least one honest server's secret key. Clients verify the DH-OPRF result against the server's public key for that epoch (§4). Verifying this result is critical here, since it ensures that an adversary cannot cause two of a user's devices to diverge in their multidevice key, which would cause them to create different circuit paths and thus leak metadata.

The client evolves keys every day (see Figure 2), deleting keys older than T epochs from the newest key. The client keeps keys for T epochs in the future, which allows priming circuits. If the device is off for longer than T, the client's newest key becomes stale, the mixes will refuse to roll it forward with their old keys. Thus, the duration T is a tradeoff between security and user convenience: after T epochs offline, a device must be set up again manually (e.g., by copying keys from another device). If the provider lies about committing the address book, then the evolved key on the device is also useless since it cannot decrypt the address book.

5.6 Provider availability and switching providers

Groove's service providers are *not* trusted for privacy (as we prove in §6). A service provider can still block communication for its users. However, Groove clients can periodically send messages to themselves (on empty buddy circuits) and detect provider malfunction in case these messages often do not route back intact (the provider cannot tell which message is for a buddy and which would route back to the user). In this case, the client notifies the user

to switch providers. Such self-addressed messages were proposed in prior work to detect attacks [26]. Keeping providers untrusted for privacy, however, simplifies dealing with such availability attacks compared to prior work since users can switch providers without risking exposing their communication metadata to more parties. Moreover, this property also protects against attacks that steer users towards corrupt providers and contrasted against systems with trusted providers (§2).

Users can also submit copies of messages to multiple providers to ensure availability when all but one provider fail. This is safe since Groove ensures privacy even if rogue providers duplicate messages (by deduplicating messages in the mixnet, §5.3). One detail when using multiple providers is that Groove delivers only one message copy (to defend against malicious providers submitting multiple messages). Malicious providers can thus actively try to prevent service by submitting corrupt messages, hoping their copies will prevail. Users can detect this intervention and switch providers (as discussed above).

6 Privacy analysis

Clients send and receive messages through the user's service provider. They communicate according to a schedule that is independent of their buddy relationships (§5.1); thus, the clients' network pattern does not leak sensitive information about the user.

Since it places the users' trust in the mixnet's servers as a collective rather than their service provider, our analysis focuses on reducing the security of Groove to the security of the mixnet (§6.1). We analyze each oblivious protocol and show that a more restricted attacker, who controls the network and the same mixnet servers but not the service provider, can obtain the same information. Groove's design uses a parallel mixnet as a black box that hides the sender of messages. The dead drop-based message exchange provides differential privacy for every epoch (since users can change their communication patterns by setting up circuits on epoch boundaries), similar to prior work [21, 28, 29]. The advanced composition theorem [12, 3.20] allows to compute Groove's privacy guarantee after multiple epochs, as we do in §7.

6.1 Oblivious delegation

We now prove that the security of Groove reduces to the mixnet; controlling the service providers does not enable an attacker that already controls the network and some mixnet servers to learn new information. In particular, this implies that it is safe to be the sole user of a rogue service provider.

Theorem 1. Consider Groove's attacker, who controls the users' service providers, the network, and a portion of the mixnet servers, and his observations about the users' communication. A restricted attacker, who only controls the network and the same mixnet servers (but not the service providers), can obtain the same observations.

Proof. We consider all the ways a user's clients interact with her service provider. These interactions take part in three oblivious protocols (non-interactive circuit setup, oblivious replacement, and efficient messaging). We analyze each protocol in §6.1.1 – §6.1.3 and show that Groove's attacker cannot learn any information that the restricted attacker could not obtain (e.g., by dropping network packets). □

6.1.1 Non-interactive setup

The user's clients synchronize address books through one service provider. Clients always update the user's address book before priming circuit setup messages (§5.2). On each update, the client uploads the address book under fresh encryption, padded to a fixed length (`MaxBuddies`), so the service provider cannot tell whether it has changed. The service provider may give stale address books to clients; this is our focus in §6.1.2.

Clients also retrieve the current epoch number from their service provider before priming circuit setup messages. A malicious service provider can lie about the epoch number or submit the circuit setup messages to the mixnet at the wrong epoch. The client writes the epoch number in every onion layer of the setup message, and honest mix servers discard onions with the wrong epoch number. Thus, if an honest server exists en route, this service provider's attack is equivalent to a network attacker simply dropping the user's circuit setup messages. If there is no honest server on the route, then even the restricted attacker can learn everything about the circuit by observing the setup message going from one malicious mix server to the next. (Groove mitigates this risk by using sufficiently long mixnet routes, §7.1.)

6.1.2 Oblivious replacement & device partitions

A rogue service provider may interfere when clients replace circuit setup messages, or collect and submit setup messages from multiple devices. Since the attacker controls the network, it might partition devices and prevent them from communicating. The risk with partitioned devices is that a rogue service provider submits circuits from different devices and creates distinct dead drop access patterns (i.e., a dead drop getting more than two accesses, which is not obscured by the "single" and "double" noise). Groove solves this problem with its mechanism for choosing circuit routes and tagging circuit setup messages (§5.3), as we prove next.

Consider the user Alice with two partitioned devices, d and d', and a circuit they establish for the buddy at slot $i \in [1, \texttt{MaxBuddies}]$ in their respective address books for the same epoch. Both devices submit setup messages for circuits with the same route and tag: they derive them from the buddy's slot number i, the multidevice key, and the epoch number, which are all the same for both devices, even if their address book differs and have different buddies for the same slot (see Figure 4). (The multidevice key is identical on all devices for the same epoch. If two devices use different epochs, the

mixnet will discard the circuit setup message from the device using the wrong epoch, as described above.) If there is no honest server on this route, then the attacker can trace Alice's messages through the malicious mixnet servers to the dead drop, regardless of controlling her provider.

Otherwise, an honest mix server exists on the circuits' route. It will de-duplicate the two circuit setup messages and ensure that only one circuit will be established. There are two cases regarding the dead drops at the end of these circuits. First, the circuits from d, d' reach the same dead drop. In this case, since the honest server drops one message, the attacker's observations will be precisely the same as if the devices could coordinate and only one device submitted setup messages. The second case is that device d submits a circuit setup message to a different dead drop than device d'. A device-partitioning attacker can cause this, e.g., by giving one device an old address book where a now-occupied slot was free. In this case, one of the circuits' dead drops receives one less circuit, i.e., becomes a "single"-access dead drop, which is covered by Groove's noise. The attacker could obtain the same view by dropping one circuit setup message from Alice on the network (even if Alice had just one device).

6.1.3 Efficient messaging

The sender's client submits one fixed-length, onion-encrypted message to the service provider. However, it does not contain any information about the intended recipient and, therefore, cannot teach the provider about the user's communication (the service provider broadcasts it on all circuits, §5.4).

The service provider serves messages to the recipient's client by routing them through the fetch mixchain, which the client chooses when calling `CheckForMessages` (see §5.4). The choice of mixchain servers is independent of the user's buddy-relationships, so it leaks nothing about them. Since mixchain servers choose fresh nonces when shuffling messages, they uses different ephemeral keys for processing every fetch. Thus, the messages output from the fetch mixchain always appear random to the recipient's service provider. Furthermore, the client chooses a new ephemeral key each time it calls `CheckForMessages`, so it always fetches messages from random-looking locations (even if the service provider replays old headers). Therefore, having one honest server on the fetch mixchain ensures that the client's pattern of retrieving messages appears random every time.

7 Implementation

We implemented a prototype of Groove in Go on top of Yodel's mixnet framework [22] in 20k lines of code. The mixnet has a full-mesh server topology [21, 22], which allows Groove to scale with the number of servers. The prototype uses ChaCha20 for symmetric onion encryption, the NaCl box primitive to generate circuit setup onions, and the Blake2b hash function. To implement forward secrecy with DH-OPRF, we use BLS12-381 in the CIRCL library [8].

Communications between clients, service providers and mixnet servers all use TLS 1.3 for transport security. We use gRPC between the service providers and the mixnet, and, to save bandwidth, custom RPCs between the client and the service provider. The service provider uses BadgerDB to implement atomic transactions and manage user state.

Our implementation includes two types of clients, for desktop and mobile devices. The desktop client is a command-line program, and the mobile client is built for Android using the `gomobile` tool.

Memory usage. A prominent challenge in implementing Groove has been minimizing the circuits' memory footprint to allow users receive messages from any of their buddies in parallel. With 3M users, 100 circuits per user, 100 mix servers, and 14 mixnet hops, each server needs to keep track of at least 42 million pieces of cryptographic state per epoch:

$$100\,\text{circuits} \times 3\,000\,000\,\text{users} \times \frac{1}{100\,\text{servers}} \times 14\,\text{hops} = 42\text{M}.$$

Initially, we used AES-GCM to implement Groove's circuits, but its state is 512 B, resulting in at least 20 GB of memory usage per mixnet server. To reduce memory, we replaced AES-GCM with ChaCha20, which requires only 32 B per state, reducing this memory usage to 1.3 GB per server, but increasing CPU usage due to lack of hardware acceleration.

7.1 Parameter selection

We set Groove to resist $f = 20\%$ malicious mixnet servers, and the mixnet path length to be 14 hops (similar to previous works with the same mixnet topology [21, 22]). As shown in these works, this topology requires two honest servers in a circuit's route, and the probability that this holds for all four circuits two buddies use to communicate in Groove is under $f = 20\%$ is $\geq 1 - 4 \cdot 10^{-8}$. In addition, we set the fetch mixchain length (§5.4) to be 11, which provides a smaller chance of error compared to the circuit route length (the fetch mixchain requires only one honest server and is, therefore, shorter). Users send 128-byte messages, split over the two circuits they create per buddy. Out of the 128 bytes, 2 bytes are reserved for the pseudorandom header indicating whether the message is real or cover traffic (see §5.4), 12 bytes are reserved for the end-to-end authentication code, and 12 bytes are reserved for a nonce. Thus, recipients get a 102 byte encrypted text message per buddy per mixnet round.

Noise in practice. We configure Groove to tolerate 245 epochs of active attacks and 9 600 epochs of passive observations . These parameters provide comparable configuration to Karaoke [21], which resists the same number of active attacks, and sustains passive attacks for a year (we assume Groove's epochs are at least 1-hour long, so 9 600 epochs are longer than a year).

Specifically, with 100 mixnet servers, each honest server creates 88 000 noise circuits on average in every epoch to

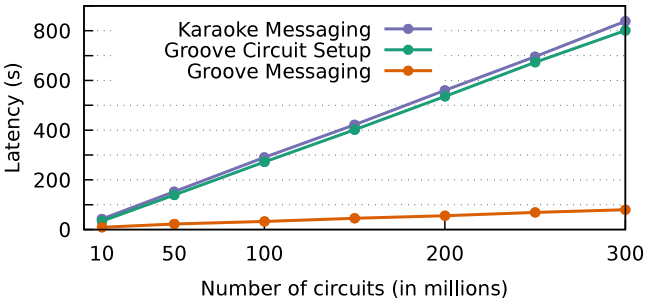


Figure 7: Latency of circuit setup and messaging rounds with respect to the number of circuits. The figure also compares with Karaoke's messaging round as a baseline (1 user relationship in Karaoke equals 2 Groove circuits) [21]. The mixnet has 100 servers.

provide ($\varepsilon = \ln 2, \delta = 10^{-4}$)-differential privacy (the same ε, δ as Vuvuzela's implementation, and better than Karaoke's $\varepsilon = \ln 4, \delta = 10^{-4}$ and Stadium's $\varepsilon = \ln 10, \delta = 10^{-4}$ [21, 28, 29]). The more servers there are, the less noise each server contributes (e.g., a mixnet with 50 servers requires each of them to create 125k noise circuits for the same privacy level).

8 Evaluation

We use the prototype to evaluate Groove's performance and costs. Our experiments answer the following questions:

1. What throughput and latency can Groove achieve?
2. How does Groove scale with the number of servers?
3. What are Groove's deployment costs?
4. What are the costs for a mobile client in terms of battery consumption and network usage?
5. What is the cost for a client catching up to old messages after having been offline?
6. How does Groove's performance compare to prior work?

First, we focus on the Groove's back-end, i.e., the service providers and mixnet, and then on the mobile client.

8.1 Back-end performance and costs

Setup. We test Groove with 25–150 mixnet servers that we deploy evenly split across 3 EC2 regions across the U.S. and one in Europe: `us-east-1`, `us-east-2`, `us-west-2`, `eu-west-1`. Each server is an `r5.8xlarge` VM with an Intel Platinum 8000 3.1 GHz CPU with 32 cores, 256 GB of memory, and a 10 Gbit/s network link. We evaluate with a single service provider on `us-east-1`: since service providers only buffer and relay messages, but do not participate in processing messages through the mixnet, the performance of Groove's back-end does not depend on the number of service providers. Only clients connected through an overloaded service provider will experience performance issues.

We simulate hundreds of millions of circuits by having mixes create extra circuits. Assuming each user has up to 50 buddies, and hence requires 100 circuits, the system load corresponds to relationships between millions of users. Although clients do not use these circuits, they correspond to real conversation load on the back-end. We set Groove's system parameters as described in §7.

Throughput and latency. We measure back-end latency for a given circuit load in deployment of 100 mixnet servers. To measure the latency on the back-end, we measure the time since the service provider submits a message until the mixnet completes the round and returns the result to the service provider, plus the time needed for the fetch protocol (§5.4). Users will experience additional latency depending on their schedules and the network RTT to their service provider. Groove is not tied to one configuration of buddies per user, so we use the number of circuits to quantify Groove's load (each buddy requires two circuits). Since Groove ensure there is one message delivered on every circuit in every round §4, the number of circuits sets the load on its servers.

In Figure 7, we observe that the Groove's messaging round latency is 32.4s, 55.9s and 79.83s for 100M, 200M and 300M circuits, respectively. The measured latencies have two components: The first, larger, component is mixing the users' messages and routing them from the source to the destination service provider; this represents the majority of the duration of Groove's messaging round (28.7s, 50.8s and 71.3s for 100M, 200M and 300M circuits). The second, smaller, component ($\approx 11\%$ of the total duration) is the time spent on running Groove's fetch protocol (§5.4). For this second part, we model an average load of clients running the fetch protocol: we simulate fetches at the end of each round, and we wait for all messages of a round to be fetched before moving on to the next round. Thus, assuming the average user sets up 100 circuits (to communicate with up to 50 buddies), Groove could support 1M-3M users with these latencies.

The duration for setting up the circuits in Groove is 13.3 minutes for 300 million circuits (Figure 7). Since circuit setup is relatively infrequent (e.g., once a day), it can run in the background and stretch up to an epoch's duration. Groove's circuit setup is similar to a messaging round in Karaoke [21], which also offers differential privacy, though there is a difference in the payload size (Groove's payload is about 200 bytes smaller than Karaoke). For the same setup, where the system processes 300M messages per round, a communication round in Karaoke takes 14 minutes (Figure 7).

Scalability. We test how Groove scales with the number of users by measuring the latency for varying deployment sizes of 25-150 mixnet servers and keeping the number of circuits per server constant at 1M (so the load on the system increases proportionally to the number of servers). Figure 8 shows that Groove scales well: it can support additional users at almost the same latency by proportionally increasing the number of servers. We attribute the slight latency increase to the fact that shuffling messages together requires each server, in every hop along the mixnet route, to wait for inputs from all other servers that processed messages at the previous hop.

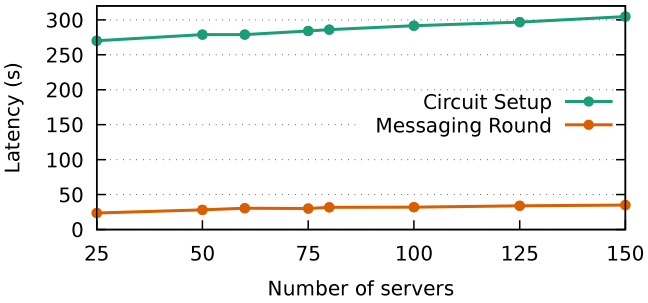


Figure 8: Scalability of the back-end. The mixnet server load is constant (1 million circuits per server). This experiment shows that by having more mix servers, Groove can support more clients with the same communication latency.

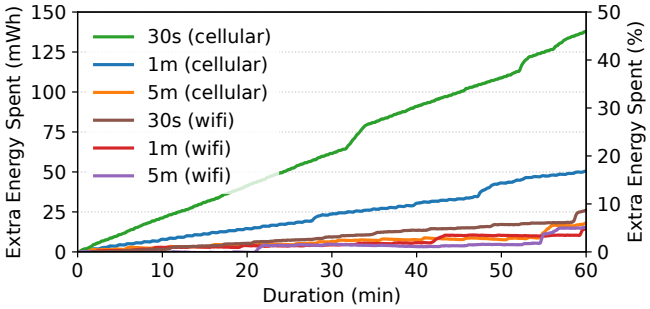


Figure 9: Client's energy consumption on a Pixel 4 phone for different schedules and network types. The graph shows how much more energy is spent running a schedule compared to leaving the phone idle. The bumps in some of the lines correspond to wake-ups to service other running apps (as we verified by reading device logs).

Deployment costs. With 300 million circuits, each of the 100 mixnet servers sends at about 2 Gbps at peak usage, for a network usage of 13.4 GB per messaging round. In this deployment, setting up circuits for one epoch uses 47.5 GB of network data per mixnet server.

Service providers buffer messages for their users. For a user who generates circuit setup messages for the next 30 epochs (i.e., the next month if epochs last a day), the storage requirement for circuit setup is 2.1 MB. Further, if messaging rounds happen every minute, then storing a month's worth of received messages amounts to 264 MB per user.

Forward secrecy. On the server, computing OPRF incurs low overheads. A single mix can answer 12k DH-OPRF requests per second, or 10^9 per day. A client evolves keys with all their buddies and the multidevice key every epoch (§5.5), creating 51 requests/day with day-long epochs. Servers can run the OPRF computations in the background and load-balance client requests throughout the day, allowing servers to easily support 1–3M users as in the earlier experiments. On the client-side, it takes 2.03s to run `ForwardSecrecy` with 100 servers (primarily due to the network latency, then to the two pairing operations used in the verification), and the bandwidth usage is 150 KB/day.

8.2 Mobile clients

We evaluate Groove's mobile client, which represents an important class of clients enabled by its flexibility. We focus on two metrics: the impact on battery life and network usage. We run the mixnet with 32s messaging round time, which correspond to the latency with 100M circuits and 100 servers (Figure 7). We evaluate the mobile client on a Pixel 4 cellphone running the stock Android 10 OS. Our tests include cellular (3G with HSDPA) and WiFi networks.

Battery usage. We explore the impact of different schedules on battery life. To evaluate battery consumption, we connected the mobile device to a USB power meter (UM25C) and collected energy consumption roughly every 1 second.

We force-enable *doze mode* to reduce noise from apps running in the background; this is the battery-saving optimization that typically runs when the device's USB port is unplugged. Yet, running Groove's app in this mode implies that the client's transmissions may change without adhering to the schedule. Therefore, we excluded our app from doze mode (Android's `AlarmManager` provides APIs to avoid it).

First, we measure the baseline energy consumption when the phone is fully charged, idle, the screen is turned off, and Groove's client is not installed. We observe that after an hour, the idle phone consumes about 310 mWh of energy, both when the phone uses WiFi and cellular networks. Then, we run Groove's client with different schedules. At the beginning of the experiment, the client uploads circuit setup messages for the next 30 epochs. Then, the client sends a message and downloads pending messages according to its schedule (following Figure 2).

Figure 9 shows the energy consumption over an hour for different schedules compared to the baseline. It is apparent from the figure that as the schedule becomes more frequent, the energy consumption increases, especially when using cellular networks. A mobile client following the mixnet's $\approx$ 30s round-schedule would have increased battery usage by 47% over the idle phone, but flexibility allows the mobile client to use lighter schedules. On a 1-minute schedule, the client on the cellular network uses an extra 50 mWh compared to the baseline, a 16% increase. Moreover, the energy cost from running the client is substantially reduced further when the client runs on a 5-minute schedule (about 6% over idle). We hypothesize that this allows the phone to hibernate and save power. The energy consumption is more modest when using WiFi, which is more energy-efficient [16].

Network usage. We monitored the link between the client and service provider to quantify the client's network usage. It uploads one message to the service provider every time the schedule triggers and downloads a message per round (see Figure 2). Consequently, different schedules affect the message upload volume, which ranges from 33 KB/h with a 5-minute send schedule to 77 KB/h with a 1-minute schedule. The download volume is 39 KB/h on the 5-minute schedule

and 69 KB/h on the 1-minute schedule. In addition, the client sends about 110 KB for the circuit setup per epoch; for a month's worth of primed circuits, this phase costs 3.2 MB. These measurements show that Groove's client uses a total of 54 MB to 106 MB of bandwidth per month, depending on the schedule. We believe that Groove's moderate network usage is compatible with mobile data packages.

Catch-up. Groove's oblivious fetch protocol filters dummy messages, allowing clients to quickly catch up on messages that buddies send their users after being offline for an extended time (§5.4). Consider a client that catches up on a month by downloading 500 messages per offline day (15k total); we evaluate it requires 11 MB of bandwidth.

8.3 Comparison with prior works

Groove provides asynchronous messaging with many buddies, whereas recent mixnet-based works, Karaoke, Stadium, and Yodel [21, 22, 28], provide synchronous communication with one buddy. PIR-based approaches, like Pung [6], could allow asynchronous messaging but with a significant performance cost compared to [21, 22, 28] (owing to stronger privacy guarantees and weaker trust assumptions); see discussion in §2. Pung also limits users to communicate with one buddy at a time due to the high server's cost when retrieving messages.

Thus, these prior systems rely on a hefty dialing protocol (Alpenhorn [23]) to synchronize between buddies before they can communicate. Groove outperforms prior designs when users talk with multiple buddies since dialing through Alpenhorn adds about 5 minutes of latency when users switch between buddies. On the other hand, if users communicate with just one buddy who is simultaneously online, they can avoid frequent dialing, and in this case, Yodel and Karaoke outperform Groove.

In more detail, Stadium, Karaoke, Yodel, Pung, and Groove were evaluated on a similar 100 server configuration, which we use to compare. We assume that users in Stadium, Karaoke, Yodel, and Pung only chat with one buddy that is simultaneously online. Karaoke supports 1M users with 7s-8s of latency (Figure 6 in the Karaoke paper [21]), while 1M Groove users can communicate with 50 buddies with a latency of 32s. The increase in latency is only 4× that of Karaoke, despite Groove supporting all 50 buddies to message the user at the same time, since Groove establishes circuits through the mixnet (allowing for more efficient symmetric onion encryption rather than the public-key onion encryption used in Karaoke). Stadium induces latency on the order of minutes (see Figure 9 in the Stadium paper [28]) largely because of its use of zero-knowledge proofs to ensure that mixnet servers process messages correctly. Pung's latency increases quadratically with the number of users [4, 6]; with millions of users, latency is over 30 minutes (as we interpolate from Figure 8 in the Pung paper [4] assuming that a 100-server Pung cluster performs 100× better than one server). This performance gap grows with the size of the user base. Yodel builds on its interactive circuit establishment protocol to directly connect (not through the mixnet) each user to its buddy's dead drop. Groove avoids this direct connection to protect the user's communication metadata-privacy if the buddy is offline (hiding the user was trying to connect with an offline buddy). For 1M users, Yodel's latency is 750ms (Figure 10 from the Yodel paper [22]), 42× better than 1M Groove users that can communicate with 50 buddies simultaneously (corresponding to the 100M circuits data-point in Figure 8). If we limit Groove to allow one buddy per user, then it needs to support only 2M circuits per 1M users (as in Yodel), and Groove's latency shrinks to about 4× that of Yodel. This remaining performance gap is primarily due to Groove connecting both buddies to dead drops through the mixnet (above) and Groove's fetch protocol that conserves client bandwidth at the expense of routing the messages buffered at the recipient's service provider again through the mixnet.

Another significant difference is the client bandwidth. Dialing through Alpenhorn requires clients to receive 62 GBytes/month (§2), on top of the overlying system's bandwidth requirements (such as Karaoke, Stadium, etc.). In contrast, Groove's oblivious messaging protocols (§5.4) allow communication with many buddies while keeping clients' bandwidth costs 1000× lower (see §8.2).

9 Conclusion

Groove removes the rigid requirements that prior designs forced on clients. It allows users to have asynchronous text-message chats with multiple buddies, seamlessly switch between devices, and run clients on relatively constrained mobile devices. At the same time, it provides the same scalability and privacy guaranty as prior rigid differentially-private messaging systems. Groove achieves this advancement by introducing protocols for oblivious delegation that allow users to have an untrusted service provider participate in the rigid messaging protocol on their behalf. Our evaluation of a prototype of Groove shows that it can support a large user-base with latency on the order of a minute. Our experiments with a Pixel 4 smartphone demonstrate that Groove can accommodate mobile clients' network and power constraints, unlike previous rigid systems. In that, Groove steps metadata privacy closer to standard messaging apps and broad adoption.

Acknowledgments

The authors thank Gil Segev and Bryan Ford for helpful discussions, and our shepherd, Natacha Crooks. This work was supported, in part, by the Alon fellowship, the Hebrew University cybersecurity research center, and gifts from Microsoft and Google.

References

[1] *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.

[2] *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[3] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *OSDI*, 2021.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society, 2018.

[5] Sebastian Angel, David Lazar, and Ioanna Tzialla. What's a little leakage between friends? In *Proceedings of the Workshop on Privacy in the Electronic Society*, pages 104–108, 2018.

[6] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* [1], pages 551–569.

[7] Alex Biryukov, Ivan Pustogarov, Fabrice Thill, and Ralf-Philipp Weinmann. Content and popularity analysis of Tor hidden services. In *ICDCS Workshops*, pages 188–193. IEEE Computer Society, 2014.

[8] Circl - bls12-381, 2021. github.com/cloudflare/circl/ecc/bls12381.

[9] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, February 1998.

[10] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178. IEEE, 2016.

[11] Roger Dingledine and Nick Mathewson. Anonymity loves company: Usability and the network effect. In *Workshop on the Economics of Information Security*, 2006.

[12] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[13] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *Proceedings of the 30th USENIX Security Symposium*, Vancouver, Canada, August 2021.

[14] Philippe Golle and Ari Juels. Parallel mixing. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 220–226, Washington, DC, October 2004.

[15] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and t-pake in the password-only model. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 233–253. Springer, 2014.

[16] Goran Kalic, Iva Bojic, and Mario Kusek. Energy consumption in Android phones when using wireless communication technologies. In *MIPRO*, pages 754–759. IEEE, 2012.

[17] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* [2], pages 406–422.

[18] Albert Kwon, David Lu, and Srinivas Devadas. XRD: Scalable messaging system with cryptographic privacy. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, February 2020.

[19] Adam Langley. Pond, 2016. https://github.com/agl/pond.

[20] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962, RFC Editor, June 2013.

[21] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 711–726, Carlsbad, CA, October 2018.

[22] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: Strong metadata security for real-time voice calls. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, October 2019.

[23] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proceedings of the 12th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI) [1], pages 571–586.

[24] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security Symposium*, pages 383–398. USENIX Association, 2015.

[25] Femi Olumofin and Ian Goldberg. Privacy-preserving queries over relational databases. In *Proceedings of the 10th Privacy Enhancing Technologies Symposium*, Berlin, Germany, July 2010.

[26] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *Proceedings of the 26th USENIX Security Symposium*, pages 1199–1216, Vancouver, Canada, August 2017.

[27] David Schatz, Michael Rossberg, and Guenter Schaefer. Hydra: Practical metadata security for contact discovery, messaging, and dialing. In *ICISSP*, 2021.

[28] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* [2], pages 423–440.

[29] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 137–152, Monterey, CA, October 2015.