Alpenhorn: Bootstrapping Secure Communication without Leaking Metadata

David Lazar and Nickolai Zeldovich
MIT CSAIL

Abstract

Alpenhorn is the first system for initiating an encrypted connection between two users that provides strong *privacy* and *forward secrecy* guarantees for metadata (i.e., information about which users connected to each other) and that does not require out-of-band communication other than knowing the other user's Alpenhorn username (email address). This resolves a significant shortcoming in all prior works on private messaging, which assume an out-of-band key distribution mechanism.

Alpenhorn's design builds on three ideas. First, Alpenhorn provides each user with an address book of friends that the user can call. Second, when a user adds a friend for the first time, Alpenhorn ensures the adversary does not learn the friend's identity, by using *identity-based encryption* in a novel way to privately determine the friend's public key. Finally, when calling a friend, Alpenhorn ensures forward secrecy of metadata by storing pairwise shared secrets in friends' address books, and evolving them over time, using a new *keywheel* construction. Alpenhorn relies on a number of servers, but operates in an anytrust model, requiring just one of the servers to be honest.

We implemented a prototype of Alpenhorn, and integrated it into the Vuvuzela private messaging system (which did not previously provide privacy or forward secrecy of metadata when initiating conversations). Experimental results show that Alpenhorn can scale to many users, supporting 10 million users on three Alpenhorn servers with an average dial latency of 150 seconds and a client bandwidth overhead of 3.7 KB/sec.

1 Introduction

To achieve privacy in a communication system, it is not enough to just hide the contents of the messages sent or received by a user. It is also important to hide who the user is communicating with, at what time they are communicating, and whether they are communicating with anyone at all; we refer to such information as *metadata*. For instance, researchers have shown that they can learn significant amounts of sensitive information by looking at just what phone numbers a person called [33], who the person emailed, their IP address, or social network connections [18]. Similarly, NSA officials have also said that metadata is crucial for surveillance [38].

Recent work shows that it is possible to build private messaging systems that hide metadata at scale [10, 15, 19, 28, 29, 41, 43]. Unfortunately, these systems do not provide users with a convenient way to bootstrap communication without leaking metadata in the process. This impedes practical deployment and precludes any end-to-end metadata privacy guarantees.

Alpenhorn is the first system to address this problem. Functionally, Alpenhorn allows users to initiate a conversation: that is, Alice can use Alpenhorn to call Bob, and Alpenhorn will ensure that Bob knows that Alice is calling, and that Alice and Bob agree on a fresh cryptographic key, called a session key, to protect their conversation. Alpenhorn is purely a bootstrapping protocol: the actual conversation can take place through one of the systems mentioned earlier. Crucially, Alpenhorn provides *privacy* and *forward secrecy* of metadata. This means that an adversary cannot determine who, if anyone, a user might be calling at any given time, and even if the adversary later compromises a user's computer, they will not be able to tell what calls the user made or received in the past.

To understand the challenges faced by Alpenhorn, consider the traditional approach for establishing a session key between users, which works in two steps. First, users learn of each other's long-term public keys, through some public key infrastructure (PKI) system. In the second step, users run a key exchange protocol, such as Diffie-Hellman, to establish a fresh session key, and they use their long-term keys to confirm each other's identity. These two steps correspond to the two challenges faced by Alpenhorn:

First, looking up a user's public key can leak metadata in itself. For instance, if Alice asks a key server for Bob's public key, and the adversary learns about this request, the adversary now knows Alice is about to call Bob. This violates Alpenhorn's goal of achieving *privacy* for metadata, and most existing PKI systems operate in this manner.

Second, even if users somehow manage to obtain each others' public keys, long-term public keys are a poor fit for metadata *forward secrecy*. Specifically, key exchange protocols like Diffie-Hellman authenticate participants by signing messages, which makes it obvious to an adversary who the participants are. A strawman solution is to encrypt these messages using the other user's public key, and to broadcast these messages, so that an adversary cannot tell who the intended recipient is. Even ignoring the performance overheads, this strawman fails to provide forward secrecy, because any adversary that later com-

promises the recipient's computer will obtain that user's long-term private key, and will be able to learn about all past incoming calls received by that user, by decrypting those messages.

Alpenhorn addresses these challenges using three ideas. First, instead of using long-term public keys to encrypt key exchange messages, Alpenhorn maintains an address book on each user's computer, containing a pairwise shared secret for each of that user's friends. This helps ensure forward secrecy because there is no long-term encryption key for an adversary to compromise.

Second, to allow users to add friends to their address book, Alpenhorn uses identity-based encryption (IBE) [7, 17, 39]. IBE is different from traditional public-key cryptography, in that a user's *public* key is purely a mathematical function of their username, such as an email address, together with a master public key from some server.[1] This allows Alpenhorn to compute a friend's public key without leaking the friend's identity. As described in §4, Alpenhorn extends IBE to handle server compromises and to ensure forward secrecy.

Finally, Alpenhorn must also provide privacy and forward secrecy for the metadata involved in actually initiating a conversation. To do this, Alpenhorn uses a novel *keywheel* construction, which continuously evolves all shared secrets in a user's address book, so as to provide forward secrecy while still ensuring that, at any given time, two friends have the same secret value in their address books.

Alpenhorn relies on two sets of servers: the IBE servers, mentioned above, and a set of mixnet servers, whose job is to hide the source of every message. Both the IBE and mixnet servers operate in the *anytrust* model, requiring just one honest server for security. The use of a trusted server allows Alpenhorn to achieve good performance, compared to purely cryptographic approaches like private information retrieval that do not trust any servers at all. §3 describes Alpenhorn's precise guarantees and assumptions.

To evaluate Alpenhorn, we implemented a prototype in Go, and integrated it with the Vuvuzela private messaging system, which did not previously provide privacy or forward secrecy for bootstrapping conversations. Integrating Alpenhorn into applications is straightforward; modifying Vuvuzela to use Alpenhorn required changing 200 lines of code. Alpenhorn's performance scales well with the number of users: 3 Alpenhorn servers can support 10 million users with 5% of them initiating a conversation every 5 minutes, with a modest client-side bandwidth cost of 3.7 KB/sec. The client-side overhead in particular is ~10× less than that of the equivalent dialing protocol in Vuvuzela (which fails to provide the same privacy guarantees).

In summary, the contributions of this paper are:

- Alpenhorn, the first system for establishing session keys that provides privacy and forward secrecy for metadata;
- a novel way of using IBE in an anytrust setting to achieve metadata forward secrecy;
- the keywheel construction, which allows the Alpenhorn client to establish fresh session keys with low latency and low bandwidth overheads;
- a prototype implementation of Alpenhorn; and
- an experimental evaluation of Alpenhorn that demonstrates it can scale to 10 million users.

2 Related work

A common way to bootstrap private messaging is to assume that users have exchanged keys or secrets out-of-band. For example, the Ricochet [13] private messaging system requires a user to know the other person's Ricochet ID (a public key) to start a conversation. The Pond [29] private messaging system uses a protocol called PANDA [2] to establish relationships between users that have previously shared a secret. In contrast, Alpenhorn allows two users to start a conversation without knowing each other's public keys or having a shared secret. Alpenhorn can be used to bootstrap PANDA (see §8.5).

Both Ricochet and Pond use Tor's hidden services [20]. Alpenhorn's privacy guarantees are stronger than those of Tor's hidden services in two ways. First, hidden services do not protect against traffic analysis. This is because Tor has many ways for an adversary to infer information based on traffic patterns. Alpenhorn uses techniques from Vuvuzela [41] to defeat traffic analysis (achieving differential privacy). Second, hidden services have a weaker adversary model for protecting metadata: e.g., an adversary that compromises the rendezvous point of a hidden service learns when that user is receiving calls. In contrast, Alpenhorn provides metadata privacy under an anytrust assumption (any $N-1$ out of N servers can be compromised).

DP5 [10] solves a related problem of *online presence*. It enables users to query their friends' online status (and learn additional information, such as their current IP address) without revealing metadata. DP5 assumes that every user already has a list of all of his friends and their public keys. This is precisely the problem that Alpenhorn is designed to address: to allow users to add new friends without knowing their public key, and to inform a user that someone wants to add them as a friend (or wants to call them).

[1] To achieve this, the user's *private* key must be generated by a server holding the corresponding master secret key.

Identity-based encryption (IBE). Alpenhorn uses IBE to exchange keys between two users for the first time. IBE typically assumes a trusted server known as the private key generator (PKG) that distributes private keys to users. To avoid trusting a single server, Boneh and Franklin [7] proposed using a distributed key generation (DKG) scheme to distribute the master secret key among multiple PKGs. Recently proposed DKG schemes require $3t + 1$ or $2t + 1$ servers to tolerate t dishonest servers, depending on the communication model [25]. Our Anytrust-IBE approach to distributing the PKG requires only 1 honest server, but this comes at the expense of availability (Alpenhorn provides no fault tolerance). In future work, we hope to explore whether more sophisticated cryptographic constructions can further improve Alpenhorn's performance [16].

Private information retrieval (PIR) could, in principle, provide an alternative to IBE for privately obtaining a user's public key [24]. To ensure forward secrecy, each user would need to periodically generate fresh keys, and upload them to a central database. Each user would also need to perform fake PIR queries even if they are not interested in looking up a key, to avoid leaking at what times the user is starting conversations. In practice, state-of-the-art PIR implementations cannot handle tens of millions of users each performing a query on a database containing tens of millions of records. Most implementations require quadratic [1, 28], or nearly quadratic [31] cost to handle N queries on a database containing N items. In contrast, Alpenhorn's design achieves a total server cost that is linear in the number of users, which enables it to support tens of millions of users. Alpenhorn's overall design also addresses an important challenge not faced by PIR: informing a user that someone wants to call them, and minimizing the bandwidth required for this notification.

Forward secrecy. IBE can achieve forward secrecy by having users generate a different key for each day (e.g., by concatenating the date with their username [7], or by using more efficient constructions [6]). A user can erase old private keys so that they are not disclosed when an adversary compromises the user's computer. However, this assumes that the IBE PKG server is not compromised: a compromised PKG could re-compute all old user private keys. In contrast, in Alpenhorn's design, even if an adversary compromises *all* PKGs, the adversary cannot decrypt past messages.

Binary tree encryption (BTE) [14] also allows users to forget old private keys to achieve forward secrecy. BTE does not require any interaction or trusted servers, but it also does not address two of the key problems faced by Alpenhorn: obtaining public keys without leaking metadata, and informing a user that someone has added them as a friend. Another downside of BTE is that the keys are much larger than in traditional public-key encryption.

Functions provided by the Alpenhorn library

```
// Initialize an Alpenhorn account
func Register(email string)

// Get your long-term key to share with friends
func MySigningKey() PublicKey

// Send friend request to the given email address.
// Their public key for extra verification is optional.
func AddFriend(email string, theirSigningKey *PublicKey)

// Call a friend; returns a shared secret known only
// to you and the friend. The call's intent is optional.
func Call(email string, intent int) SessionKey
```

Callbacks that must be implemented by the application

```
// This function is called when the client gets a friend
// request. Return true to accept the friend request.
func NewFriend(email string, theirSigningKey PublicKey) bool

// This function is invoked when client receives a call.
func IncomingCall(email string, intent int, key SessionKey)
```

Figure 1: Simplified Alpenhorn API.

The double ratchet algorithm [36] used by Signal and WhatsApp [42] continuously rotates session keys between users for forward secrecy, similar to Alpenhorn's keywheel. The key difference is that the ratchet ensures forward secrecy for *data*, whereas the keywheel produces dialing tokens, which Alpenhorn uses to ensure forward secrecy of *metadata*. Off-the-record messaging (OTR) [9] similarly rotates keys to achieve forward secrecy for data but does not hide metadata. Alpenhorn uses Bloom filters [5] to encode the dialing tokens produced by the keywheel, similar to the approach taken by AnoNotify [37].

3 Overview

To use Alpenhorn, the developer of a messaging application must integrate their application with the Alpenhorn client library, and specify a set of Alpenhorn servers that the library should use.[2] The API provided by the client library is shown in Figure 1. The API allows applications to perform two main tasks: to add a friend (for when the user asks the application to add a friend to their address book), and to initiate a call with a friend (for when the user asks the application to call a friend). Alpenhorn uses email addresses to identify users; §4 discusses what happens if an email server is compromised.

When a user starts the messaging application for the first time, the application calls `Register()`, passing in the user's own email address. `Register()` will generate a long-term signing key for the user, and register it with Alpenhorn. The user will have to prove their identity to the PKGs through confirmation emails.

Users can add friends by invoking the `AddFriend()` function. For example, if Alice and Bob are both us-

[2]For simplicity, this paper assumes the application developer sets up the Alpenhorn servers. Multiple applications can also share a set of Alpenhorn servers, but this paper does not discuss the issues that are involved in doing so.

ing an Alpenhorn-based messaging application, and Alice wants to add Bob as a friend, the application will call `AddFriend("bob@gmail.com", nil)`. In this example, Alice did not have any prior knowledge of Bob's long-term signing key, so the second argument is `nil`. However, Alice must know Bob's email address ahead of time; otherwise, there is no way for Alice to tell Alpenhorn whom she wants to add as a friend.

On the other side of the world, Bob's application receives a callback from the Alpenhorn library, `NewFriend("alice@gmail.com", "e27scvh08m...")`, and displays the request to Bob. If Bob had out-of-band knowledge of Alice's signing public key, he could verify it before accepting the request; an application can obtain the user's own long-term signing key by calling `MyPublicKey()`. Bob doesn't know Alice's long-term signing key, but he knows that she recently registered her email address with Alpenhorn, so he accepts the request knowing that the Alpenhorn servers have validated her identity.

The application returns `true` from the `NewFriend` callback to indicate to the Alpenhorn library that Bob accepted the friend request. Internally, this causes the library to send a friend request back to Alice to confirm the request.

After some time, Alice gets back the friend request from Bob, which confirms that she is now friends with him. At this point, Alice and Bob's Alpenhorn libraries have internally agreed on a shared secret, stored in their keywheels, and the Alpenhorn library continuously rolls forward this shared secret; however, this secret value is not directly exposed to the application.

The next day, Alice opens a chat window for Bob in her messaging app, which causes the application to invoke `Call("bob@gmail.com", 0)`. The second argument, `0`, is an application-specific *intent* that is passed along to the application on the other side; we discuss intents more in §5.3. In Alice's client, the Alpenhorn library returns a fresh shared key that Alice's application should use for the conversation, such as `"3xdq9t7vP0..."`. Shortly afterward, Bob's Alpenhorn library invokes the `IncomingCall("alice@gmail.com", 0, "3xdq9t7vP0...")` callback, and the application tells Bob about an incoming call from Alice. If he accepts, Alice and Bob can start talking to each other through the application's private messaging protocol, using the fresh key `"3xdq9t7vP0..."`.

3.1 Overall design

Figure 2 shows the major components of Alpenhorn. Each Alpenhorn client maintains a long-term signing key, described above, and an address book, consisting primarily of a keywheel table, which stores and rolls forward shared secrets with each of that user's friends. In addition to the client library, Alpenhorn relies on two sets of servers:

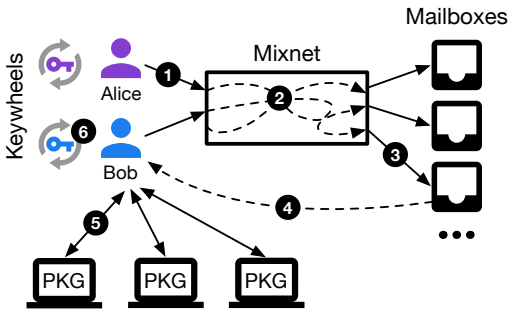

Figure 2: Overview of what happens when Bob adds Alice as a friend using Alpenhorn's add-friend protocol.

a set of *private-key generator* (PKG) servers, used for identity-based encryption, and a set of *mixnet* servers, used to hide which client submitted which request.

Alpenhorn consists of two protocols: the *add-friend* protocol for adding a friend to an address book, given their email address, and the *dialing* protocol for establishing a new conversation with a friend, which we describe in more detail in §4 and §5, respectively. This split allows Alpenhorn to achieve good performance. The add-friend protocol uses public-key cryptography, which is necessary to bootstrap communication with a new friend, but is relatively expensive (and thus has a higher latency). The dialing protocol uses symmetric-key cryptography, which allows existing friends to perform low-latency key exchanges. Figure 2 shows the add-friend protocol, which we will now describe; the dialing protocol is similar.

Alpenhorn clients send requests in periodic *rounds*, which are coordinated by the first mixnet server. Each client submits a fixed-size request to the mixnet in every round, shown by step 1, even if they don't want to add a friend at that moment. This provides *cover traffic*, so that an adversary cannot learn anything about who the user might be communicating with from the fact that a client is sending messages to Alpenhorn servers.

Requests are encrypted for the intended recipient, so that an adversary cannot decrypt the request's contents without the recipient's private key. The caller obtains the recipient's public key using identity-based encryption, which allows the client to obtain a given recipient's public key by simply computing it, without having to query any server for it. To ensure forward secrecy, the recipient's public key changes each round, and the recipient's client deletes each round's private key at the end of the round.

In step 2, the mixnet shuffles the requests for a given round, and adds additional *noise* to mask any statistical information that an adversary might learn at the end of the mixnet. The mixnet operates in an anytrust model; just one honest mixnet server is sufficient to provide security. Alpenhorn uses the Vuvuzela mixnet design [41], which adds enough noise to achieve differential privacy.

At the end of the mixnet, shown by step 3, client requests are distributed into *mailboxes* based on the intended recipient of the request. The request includes the destination mailbox ID in plaintext form for this purpose; it is computed by the client as the hash of the recipient's email address modulo the number of mailboxes; many users share the same mailbox. A special mailbox ID is used for cover traffic, so that it need not be processed further.

Each client then downloads their mailbox, in step 4. In step 5, the client contacts every PKG server to obtain its private key for this round. Alpenhorn combines the private keys from all PKG servers to ensure security as long as just one of them is honest. Then in step 6, the client tries to decrypt every request in the mailbox using the private keys for this round. If the decryption succeeds, the Alpenhorn client processes the incoming add-friend request, adds the resulting key to its keywheel, and sends an acknowledgment back (as another add-friend request). If the decryption fails, the request must have been intended for someone else, or was noise.

The dialing protocol works similarly, but significantly reduces the size of the mailbox using a Bloom filter [5] to efficiently encode a set of values submitted by clients. §5 describes the dialing protocol in more detail.

3.2 Security goals

Alpenhorn's security goals are motivated by the private messaging applications that Alpenhorn is aiming to support, such as Vuvuzela [41], Pung [1], and Pond [29]; specifically, Alpenhorn's guarantees should meet or exceed those of the application itself. Alpenhorn focuses on privacy, and does not achieve fault tolerance (a single server can make Alpenhorn unavailable). Specifically, Alpenhorn's guarantees are as follows:

Authenticated key exchange. A powerful adversary, capable of compromising servers and tampering with traffic, must not be able to learn the session keys generated by Alpenhorn. Alpenhorn must also prevent the adversary from impersonating other users, meaning the adversary should not be able to send friend requests or calls on behalf of an email address the adversary does not own.

Privacy for metadata. Alpenhorn should not reveal metadata about friends or calls (i.e., whom, if anyone, you call or add as a friend, or who, if anyone, calls you or adds you as a friend) even after the application has been running for a long time. Specifically, Alpenhorn provides differential privacy for this metadata, as formalized in Vuvuzela [41].

Forward secrecy for metadata. If the secret state of a server or client is compromised, the adversary must not be able learn metadata or the contents of messages sent in the past. An adversary can store all past traffic, in the hope of one day acquiring the private key of a server or client, so providing forward secrecy means that encryption keys must be short-lived and erased quickly after use.

An adversary that compromises a user's computer can, of course, obtain the contents of the address book from the user's chat application. This would allow the adversary to learn about a set of friends that the user may have talked to. If the user is concerned about this, they can remove a friend from their address book, at which point Alpenhorn's guarantees would prevent the adversary from determining if these two users were or were not friends in the past.

Worst-case security. If all servers are compromised, Alpenhorn is unable to offer privacy or forward secrecy for metadata. Nonetheless, Alpenhorn provides at least the same security guarantees as existing key-exchange protocols, even if all servers are compromised; specifically:

- If users have out-of-band knowledge of each other's public keys, Alpenhorn's API can use them to defeat man-in-the-middle attacks, as in existing protocols.
- Alpenhorn's client uses an SSH-like trust-on-first-use (TOFU) approach if out-of-band keys are not provided, by remembering the friend's long-term signing key from their first add-friend request. If two users called `AddFriend` when at least one server was honest, then a later compromise of all servers does not allow an adversary to mount a man-in-the-middle attack.
- In the absence of out-of-band keys, Alpenhorn could require each user to register their public key in a verifiable ledger (such as Keybase [26] or Namecoin [35]), and to send a proof to new friends that their key is registered in such a ledger. Depending on the scenario, this can prevent man-in-the-middle attacks or allow a user to detect that someone is impersonating them; we have not implemented this in our Alpenhorn prototype.
- If a client's state is compromised, then future interaction with that client is compromised. The user can recover by revoking all of his friendships and sending a new `AddFriend` request to each of his friends.

3.3 Threat model

Alpenhorn assumes an adversary that controls all but one of the Alpenhorn mixnet servers and all but one of the PKG servers (users need not know which ones), controls an arbitrary number of clients, and can monitor, block, delay, or inject traffic on any network link. Alpenhorn assumes that the client machines of legitimate users are not compromised, and that the client software properly implements the Alpenhorn protocol. Alpenhorn does not protect against malicious servers mounting denial-of-service attacks, but it is resilient to client denial-of-service attacks.

For forward secrecy, Alpenhorn assumes that the Alpenhorn client can irrevocably delete data from memory or disk (e.g., a cryptographic key or an address book entry). Forward secrecy guarantees could be subverted on short time scales by cold boot attacks [22], and on longer time scales by a storage system on the user's computer that allows recovering previously erased data (e.g., an SSD that does not overwrite data in place). Alpenhorn servers must also be able to securely erase memory, but they never store encryption keys on disk.

We make standard cryptographic assumptions like secure public and symmetric key encryption, Diffie-Hellman key-exchange, signature schemes, and hash functions. We also assume the security of pairing-based cryptography[3] which we use for identity-based encryption in §4.

We assume that the long-term signing public keys of the Alpenhorn servers are known to all users. These keys can be distributed in the Alpenhorn software package, similar to how web browsers ship with a list of CA keys.

Alpenhorn uses email addresses to identify users, and thus relies on the user's email provider for bootstrapping user identity. This boils down to two assumptions. First, when user A adds user B as a friend, A should know B's email address, and should be sure that B successfully registered for an Alpenhorn account under B's email address. Second, each user must periodically connect to Alpenhorn (at least once every 30 days) to prevent their Alpenhorn account from being reset by an adversary that may have compromised the user's email account. §4.6 describes Alpenhorn's use of email in more detail.

4 Add-friend protocol

When Alice adds Bob as a friend using the API shown in Figure 1, her client constructs a friend request that contains her email address, her public key, some signatures, and other sensitive data (as shown in Figure 3). Since the friend request contains sensitive information, Alice needs to encrypt it so that only Bob can read it. However, Alice does not have Bob's public key, and she can't ask a server to look it up, because that would leak metadata.

The add-friend protocol uses identity-based encryption (IBE) to enable Alice to encrypt her friend request using Bob's email address as the public key, as explained in §4.1. Since Alice already knows Bob's email address, this approach does not require any directory lookup and thus leaks no metadata. However, IBE traditionally assumes a trusted server to distribute private keys, which does not align with Alpenhorn's goals. We describe how we distribute the trust among multiple servers in §4.2.

Using IBE, Alice proceeds to encrypt the request, and sends it to a shared mailbox, which is a publicly known memory location on one of the Alpenhorn servers. The contents of this mailbox are visible to the servers and available for all clients to download, so Alpenhorn must ensure that Alice's client does not reveal any metadata in the process of placing the request in the mailbox.

First, Alpenhorn must ensure that the encrypted friend requests in the mailbox do not reveal the email addresses of recipients for which they are encrypted. This property is known as *ciphertext anonymity* and is discussed in §4.3. Second, the keys used to produce the encrypted friend requests must be destroyed quickly. Otherwise, an attacker will keep a copy of the mailbox contents indefinitely, in hopes of one day compromising the private keys. This property is known as *forward secrecy* and is discussed in §4.4. Finally, we must prevent the adversary from learning who sent the friend requests, even in the face of sophisticated attacks like traffic analysis or tracking patterns in the number of messages in a mailbox. We borrow techniques from prior work on metadata privacy to ensure that an adversary does not learn who is friending whom, by adding noise messages to the mailbox [41], as discussed in §6.

Bob's client eventually downloads the mailbox from the server, containing encrypted friend requests. Bob also obtains his private key for that round from the private key generators (PKGs). Using his private key, Bob's client attempts to decrypt each friend request in the mailbox. When the client succeeds in decrypting one, the request is validated using the protocol in §4.5, and Bob is prompted to accept the request. If Bob accepts, his client sends a friend request back to Alice as an acknowledgment.

4.1 Identity-based encryption

Identity-based encryption (IBE) is a relatively new cryptographic primitive in which any username string (such as an email address) can be used as a public key. Typically, IBE assumes a single trusted party that knows everyone's private key, but we will see shortly that Alpenhorn distributes trust among many independent servers and that only one of these servers must remain honest.

For now, suppose there is a trusted server known as the private key generator (PKG). The PKG has a master public key M_{pub} known to all, and a master secret key M_{priv} known only to itself. An IBE scheme provides the following functions:

- Encrypt($M_{\text{pub}}, \mathit{identity}, \mathit{msg}$) $\rightarrow \mathit{ctxt}$
  which encrypts a message to some identity string (e.g., a username, email address, or other unique identifier).
- Decrypt($\mathit{identity}_{\text{priv}}, \mathit{ctxt}$) $\rightarrow (\mathit{msg}, \mathit{ok})$
  which decrypts a ciphertext using the private key corresponding to some identity string.
- Extract($\mathit{identity}, M_{\text{priv}}$) $\rightarrow \mathit{identity}_{\text{priv}}$
  which computes the private key corresponding to some identity string.

[3] Chen et al [16] discusses the assumptions behind pairing-based cryptography; it has been deployed in systems such as Zerocash [4].

The PKG verifies the identities of users and issues to them private keys corresponding to their identities. For example, suppose the PKG identifies users by their email address. When Bob asks the PKG for the private key corresponding to "bob@gmail.com", the PKG can send a random nonce to that email address. If Bob can produce that nonce, the PKG gives him the private key for that email address. In practice, Alpenhorn uses a more secure scheme to authenticate email addresses, described in §4.6.

IBE's power comes from the fact that anyone with M_{pub} can encrypt a message to another user without any directory lookups, as long as they know the recipient's identity. Avoiding communication for looking up the recipient's public key avoids the possibility of that communication being intercepted by an adversary to learn metadata.

4.2 Distributing trust

Using a single trusted PKG means that if it were compromised, the adversary would be able to compute the IBE private keys of every user. To avoid this, Alpenhorn uses multiple independent PKGs. A naïve approach would be to onion-encrypt a message using the master public key of each PKG in turn. For example, suppose there are n PKGs with master public keys $M^1_{\text{pub}} \ldots M^n_{\text{pub}}$. To encrypt an add-friend message for Bob, Alice could compute:

$$\begin{aligned}&\text{Encrypt}(M^1_{\text{pub}}, \text{“bob@gmail.com”},\\ &\quad\text{Encrypt}(M^2_{\text{pub}}, \text{“bob@gmail.com”}, \cdots\\ &\qquad\text{Encrypt}(M^n_{\text{pub}}, \text{“bob@gmail.com”}, msg)\cdots))\end{aligned}$$

To decrypt this ciphertext, Bob must obtain the private key for his email address from each of the n PKGs. Now, even if many PKGs are compromised, the ciphertext stays private as long as one of the master secret keys (and Bob's corresponding private key) is unknown to the adversary. Although this would achieve Alpenhorn's security goals, it is inefficient because each layer of encryption adds additional space overhead, and because Bob's decryption takes time proportional to the number of PKGs.

Alpenhorn introduces a more efficient scheme, called Anytrust-IBE, that achieves the same goal of distributing the trust among n PKG servers, by adding together the master public keys in the Boneh-Franklin IBE scheme [7]:

$$\text{Encrypt}(\sum_{i=1}^{n} M^i_{\text{pub}}, \text{“bob@gmail.com”}, msg)$$

Bob can decrypt this ciphertext by adding his private keys:

$$\text{Decrypt}(\sum_{i=1}^{n} identity^i_{\text{priv}}, ctxt)$$

This scheme is efficient: once Bob obtains and adds up his private keys, neither the ciphertext size nor the decryption time depend on the number of PKGs. A technical report [30: §A] provides more details and a proof of security for Anytrust-IBE.

```
type FriendRequest struct {
    SenderEmail  string
    SenderKey    SigningKey
    SenderSig    Signature
    PKGSigs      MultiSignature
    DialingKey   DiffieHellmanKey
    DialingRound int
}
```

Figure 3: Alpenhorn friend request.

4.3 Ciphertext anonymity

To avoid leaking metadata, it is important that encrypted friend requests (produced by the IBE Encrypt function) do not reveal the intended recipient. This property is known as *ciphertext anonymity* [11], and it is not generally true of IBE schemes [12]. Alpenhorn deliberately uses the Boneh-Franklin IBE scheme [7] because it is one of the few IBE schemes with this property. Ciphertext anonymity is also necessary to generate noise messages in the mixnet (§6).

4.4 Forward secrecy

The encrypted friend requests created by the add-friend protocol eventually become public, so it is crucial that the keys to the ciphertext are destroyed quickly to limit the possibility of compromise. For IBE ciphertexts, there are two keys we worry about: the identity private keys held by users and the master secret keys held by the PKGs. When both sets of private keys are destroyed, ciphertexts created with the corresponding public keys become useless to the adversary. Better yet, in Anytrust-IBE, only one (rather than all) of the PKGs must destroy its master secret key to achieve forward secrecy.

The add-friend protocol operates in rounds to achieve forward secrecy. Every round, the PKGs create new master keys, and broadcast the public keys for that round to the users. Users must then obtain their private keys for that round from each PKG. After a preconfigured amount of time or after all users have obtained their private keys, each PKG deletes its master secret key for the round. Users also delete their identity private keys after downloading and scanning their mailbox for friend requests.

4.5 Authenticating requests

Figure 3 shows the structure of a friend request, obtained after decrypting a ciphertext in the add-friend mailbox. The request includes the sender's email address (e.g., "alice@gmail.com"), but how does the Bob the recipient verify that the friend request really came from Alice?

To authenticate the request, Alpenhorn's includes two signatures in the friend message. First, the `SenderSig` is a signature over the entire friend request using the sender's long-term signing key, `SenderKey`. If Bob happens to somehow know Alice's long-term key (e.g., because he got Alice's business card, which lists her signing public key), he can verify the authenticity of the message by

verifying `SenderSig` using the key he obtained out-of-band.

If Bob does not know Alice's long-term signing key, Bob's client can rely on the PKGs to authenticate Alice. Specifically, when a user's client acquires the user's IBE private key, each PKG also responds with a signature of the user's long-term key and email address. The friend request includes a multi-signature [8], `PKGSigs`, which combines the signatures from all PKGs into a single compact value. Bob can check `PKGSigs` to ensure that `SenderKey` belongs to `SenderEmail`, as long as at least one PKG is honest.

4.6 Registering email addresses

Every round, users must authenticate to the PKG in order to obtain their private keys. To avoid manual user involvement at every round, Alpenhorn splits authentication into two steps: first, a manual account registration step, and second, an automatic private-key-generation step. The second (key-generation) step is straightforward: each PKG keeps track of the long-term signing key for every registered email address, and users can obtain their IBE private key for a round by signing a request with their long-term signing private key. The first registration step, however, is more complicated because it involves trusting the user's email account provider to bootstrap the user's identity.

When using Alpenhorn for the first time, Alice registers her email address with her long-term signing key at each PKG. Each PKG sends Alice a confirmation email containing a secret token, which Alice must send to the PKG to finish the registration.[4] After registration, each PKG locks the user's email address to that user's long-term signing key, to prevent anyone else (e.g., a malicious email provider) from re-registering the address.

There is no *quick* way to reset an account; otherwise an attacker could perform a man-in-the-middle attack just by compromising Alice's email address. To deal with a situation where the user's long-term private key is no longer available (e.g., due to a disk failure), Alpenhorn institutes a lockout policy: if 30 days pass without a legitimate attempt to acquire the user's IBE private key, a PKG allows re-registering that email address with a new long-term signing key, using email verification as described above.

An adversary with access to a user's email account may register that email address before the legitimate user has a chance to do so himself. This poses a risk when a user adds a friend: is that friend's account registered by the friend, or by someone else that compromised their email account? To address this issue, it suffices for a user to learn one bit of information from their friend: namely, whether they successfully registered for an account in Alpenhorn. This bit can be conveyed informally (e.g., by announcing "contact me using Alpenhorn"), so as to minimize the need to exchange information out-of-band prior to using Alpenhorn. Once a user successfully registers for an account on all PKGs, and connects at least once every 30 days, Alpenhorn's lockout policy ensures that a compromised email account cannot be used to take over the user's Alpenhorn account.

[4] To avoid receiving a separate confirmation email from each PKG, Alice could send a single DKIM-signed email message [23] containing her long-term signing key, which each PKG could independently verify.

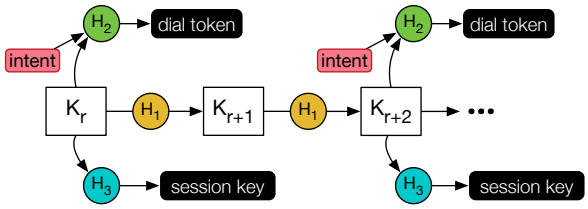

Figure 4: Overview of keywheel operations. K_r is a shared secret key in the keywheel at round r. H_i is a keyed family of cryptographic hash functions (such as HMAC-SHA256), with subscript i denoting the key.

4.7 Computing a shared secret

Once two clients have exchanged add-friend messages through Alpenhorn, they can compute a shared secret using the standard Diffie-Hellman key-exchange protocol. Specifically, the `DialingKey` in the add-friend message represents the public part of an ephemeral public-private key pair generated by each client for that request. Upon receiving the other party's `DialingKey`, a client combines its private key with the other party's public key to compute a secret key known only to these two clients. The `DialingRound` value helps the two clients synchronize their keywheels, as described in the next section.

In summary, Alpenhorn's add-friend protocol allows two clients to establish a shared secret. It achieves privacy for metadata by using distributed IBE with ciphertext anonymity; achieves forward secrecy by using short-lived IBE keys; and achieves authentication through its email registration protocol, PKG signatures on user keys, and optional out-of-band key distribution. The client pseudocode for the add-friend protocol is shown in Algorithm 1.

5 Dialing protocol

Once the add-friend protocol establishes a shared secret between two clients, the dialing protocol allows clients to repeatedly establish conversations and obtain fresh, ephemeral keys for these conversations. The dialing protocol faces two challenges: providing forward secrecy, and providing low latency (compared to the add-friend protocol). The dialing protocol addresses these challenges using a *keywheel* construction, shown in Figure 4. A keywheel stores a shared key, and performs two operations on it.

Algorithm 1 Add-friend round: client

Consider a user Alice, with email address id_{Alice} and signing key pair $(pk_{\text{sign}}^{\text{Alice}}, sk_{\text{sign}}^{\text{Alice}})$. Each mixnet server i $(1 \le i \le n)$ has a short-lived public encryption key pk_{enc}^{i}. Each PKG server j $(1 \le j \le N)$ has a long-term signing key pk_{sign}^{j}, and a short-lived IBE master key pk_{ibe}^{j}. K is the total number of add-friend mailboxes for this round. Alice's client takes the following steps for each round r:

1. **Acquire private keys** (assuming Alice already registered her email address): Alice uses $sk_{\text{sign}}^{\text{Alice}}$ to authenticate to each PKG server. Each server, if authentication succeeds, returns private key $sk_{\text{ibe}}^{j,\text{Alice}}$ and signature σ^j of $(id_{\text{Alice}}, pk_{\text{sign}}^{\text{Alice}}, r)$ using pk_{sign}^{j}.

2a. **Sign and encrypt request** (if Alice wants to introduce herself to Bob, whose email address is id_{Bob}): Create the request $m = (id_{\text{Alice}}, \sum_{j=1}^{N} \sigma^j, pk_{\text{sign}}^{\text{Alice}}, \sigma, pk_{\text{dh}}^{\text{Alice}}, w)$, where $pk_{\text{dh}}^{\text{Alice}}$ is a freshly generated Diffie-Hellman key to be used in dialing round w, and $\sigma = \text{Sign}(sk_{\text{sign}}^{\text{Alice}}, (id_{\text{Alice}}, pk_{\text{dh}}^{\text{Alice}}, w))$. The mailbox is $b = H(id_{\text{Bob}}) \bmod K$. Using IBE, encrypt the request to get $e_{n+1} = (b, \text{Enc}_{\text{IBE}}(\sum_{j=1}^{N} pk_{\text{ibe}}^{j}, id_{\text{Bob}}, m))$.

2b. **Construct fake request** (if Alice does not want to introduce herself to anyone this round): Set $e_{n+1} = (K, 0^{\ell})$ where ℓ is the length of an IBE-encrypted request as above.

3. **Onion wrap the request and send it to the mixnet**: Encryption happens in reverse, from server n to server 1, as server 1 will be the first to decrypt the request. For each server i, generate a temporary keypair (pk_i, sk_i). Then, re-encrypt e_{i+1} with $s_i = \text{DH}(sk_i, pk_{\text{enc}}^{i})$ to get $e_i = (pk_i, \text{Enc}(s_i, e_{i+1}))$.

4. **Download and scan mailbox**: Download the mailbox $H(id_{\text{Alice}}) \bmod K$. For each ciphertext c in the mailbox, attempt $(m, ok) = \text{Dec}_{\text{IBE}}(\sum_{j=1}^{N} sk_{\text{ibe}}^{j,\text{Alice}}, c)$. If decryption succeeds, then $m = (id, \sigma_{\text{servers}}, pk_{\text{sign}}^{id}, \sigma, pk_{\text{dh}}^{id}, w)$.

   Let $ok_1 = \text{Verify}(\sum_{j=1}^{N} pk_{\text{sign}}^{j}, \sigma_{\text{servers}}, (id, pk_{\text{sign}}^{id}, r))$ and let $ok_2 = \text{Verify}(pk, \sigma, (id, pk_{\text{dh}}^{id}, w))$. If $ok_1 \wedge ok_2$, then notify the user of the friend request from id.

5. **Compute shared secret**: If id is a new friend and Alice accepts the request, generate a fresh Diffie-Hellman keypair $(pk_{\text{dh}}^{\text{Alice}}, sk_{\text{dh}}^{\text{Alice}})$, and send an add-friend request with $pk_{\text{dh}}^{\text{Alice}}$ to id in the next round. Otherwise, id is a friend Alice added in a previous round with keypair $(pk_{\text{dh}}^{\text{Alice}}, sk_{\text{dh}}^{\text{Alice}})$, and now the friend is confirmed. In either case, compute shared secret $s = \text{DH}(sk_{\text{dh}}^{\text{Alice}}, pk_{\text{dh}}^{id})$ and add (id, s, w) to the keywheel table.

First, in every round of the dialing protocol, the keywheel updates the key, by hashing it with a cryptographically secure hash function. This is represented by the evolution of k_r into k_{r+1} and so on in Figure 4. By updating the key, a client ensures that an adversary that compromises a client at some time will be unable to obtain any keywheel state from prior rounds. Alpenhorn clients securely erase the old key when performing keywheel updates.

Second, the keywheel can generate *dial tokens* that the user will send out to signal their intention to call a friend. Dial tokens are generated by applying a different hash function to the current round's key. Figure 4 shows the client generating dial tokens in rounds r and $r + 2$. A dial token is a 256-bit value; this is much shorter than the size of the request in the add-friend protocol, and allows the dialing protocol to be efficient. Since the 256-bit dial token is pseudo-random, an adversary that does not know the keywheel state of two friends cannot predict what dial token they might use in a given round. An additional *intent* is hashed along with the key, as we will describe shortly.

Alpenhorn clients call each other by sending dial tokens to a mailbox through the Alpenhorn mixnet. To call a friend, an Alpenhorn client simply computes the dial token for a given round, with an application-supplied intent value, and sends it through the mixnet. To check if a friend is calling, a client downloads the list of dial tokens for that round, and computes all possible dial tokens that each of its friends could have sent in that round. Since two friends have the exact same keywheel state in a given round, a client can easily compute all of the possible incoming dial tokens, by enumerating all of its friends, and all possible intent values; this is cheap to do because hashing is fast and the number of intents is typically small.

If a client finds a dial token from a friend in the mailbox, the client invokes the `IncomingCall` callback to inform the application of the incoming call. The session key for the conversation is computed by hashing that round's key from the keywheel with a different hash function, as shown in Figure 4. The use of this hash function is a precaution, so that if an application inadvertently leaks a session key, it does not compromise future keywheel states.

Finally, Alpenhorn encodes the set of dial tokens in a mailbox using a Bloom filter to reduce the client bandwidth required to download the contents of a mailbox.

5.1 Keywheel synchronization

An Alpenhorn client maintains keywheels for each friend, as shown in Figure 5, consisting of a shared key and a round number. When two friends establish a shared secret through the add-friend protocol, this shared secret is added to the clients' respective keywheel tables. It is important for the clients to agree on the round number corresponding to this initial key; Alpenhorn uses the `DialingRound` field from the add-friend request for this purpose.

To maintain forward secrecy, an Alpenhorn client must update the keywheel state over time and discard old keys. However, the client needs to be able to generate dial tokens for the current round, and to check for incoming dial tokens. Thus, the client advances its keywheels to round $r + 1$ as soon as it has both sent any possible dial requests

Alice's Keywheel table at round 25

Friend	Secret Key	Round
bob@gmail.com	gZbkHyECIhQJ0XaQcKm	25
joanna@foo.edu	s1lJ5kRWp73M4WEMI09	25
chris@hotmail.com	W9uoocTsoYToW1A7nH7	28

↓

Alice's Keywheel table at round 26

Friend	Secret Key	Round
bob@gmail.com	AUuJw64TXCAFabdbCGp	26
joanna@foo.edu	z3XukuxRR4dUnkrWpYr	26
chris@hotmail.com	W9uoocTsoYToW1A7nH7	28

Figure 5: Evolution of a client's keywheel table. The keywheel entry for `chris@hotmail.com` has a round number higher than the current round because it was recently established through the add-friend protocol, and Chris's client supplied a `DialingRound` value of 28.

for round r, and checked the mailbox from round r for possible incoming dial tokens.

If a client cannot download the mailbox for some round, it keeps retrying; the mailbox contents is public state and is maintained by the Alpenhorn servers for a relatively long time. After some time (e.g., a day), the Alpenhorn client gives up trying to fetch the mailbox for an old round, and advances the keywheels to preserve forward secrecy.

5.2 Bloom filter encoding

Alpenhorn optimizes the dialing protocol by encoding the mailbox contents as a Bloom filter [5], which reduces the size of the mailbox that the clients must download, and in turn enables the dialing protocol to run more frequently. The encoding is done by the last server in the mixnet, which is responsible for choosing the optimal parameters to encode a given number of dial tokens into a single Bloom filter.

A Bloom filter allows clients to determine if a dial token is present in the mailbox, with no false negatives, and a small probability of a false positive. No false negatives means that Alpenhorn never misses an incoming call. A false positive translates into the Alpenhorn client invoking the `IncomingCall` callback even though the friend did not initiate a call. Alpenhorn tunes the Bloom filter to provide a low false positive rate of 10^{-10} (which roughly translates into one phantom incoming call in over a decade), using 48 bits per element in the Bloom filter. This is a significant savings over the 256-bit size of the dial token.

5.3 Intents

Alpenhorn's target messaging applications have relatively high overheads associated with setting up and tearing down conversations, and may have limits on how many active connections a user may have at a time. For instance, Vuvuzela allows a user to be active in only one conversation at a time, so if a user receives an incoming call, they may need to drop one conversation to start a different one.

To convey additional information, Alpenhorn allows an application to pass a small integer along with a call, to help the recipient decide how to respond to the incoming call, before a conversation is established. For example, the following might be useful intents for a messaging application to inform the recipient of the nature of the call: (1) let's chat right now; (2) let's chat soon; (3) call me back when you're free. An application informs the Alpenhorn client ahead of time how many intents it plans to use, so that the client can enumerate all possible incoming dial tokens.

6 Sender anonymity

Alpenhorn uses the Vuvuzela mixnet design [41] to ensure that an adversary cannot determine which client sent any given request in a mailbox, and cannot correlate a user's requests with mailbox activity over time (more precisely, Alpenhorn achieves differential privacy, as formalized for a messaging protocol by Vuvuzela).

The mixnet works by arranging a fixed, small number of servers in a chain (e.g., three servers). Each server receives all of the requests for a round, decrypts them using its private key, re-orders them randomly, and sends them to the next server in the mixnet chain. Each server also adds a number of noise messages destined to each mailbox, chosen according to a Laplace distribution with a configurable mean amount of noise μ. As long as one server in the mixnet chain is honest (i.e., does not reveal either its private key or its random re-ordering), an adversary cannot determine which incoming request (if any, due to noise) corresponds to a particular outgoing request.

Achieving good performance requires striking a balance in terms of the number of mailboxes. If there are too few mailboxes, each mailbox will contain a large number of requests, and clients will have to download a lot of data each round. If there are too many mailboxes, the servers will be overwhelmed by noise requests, since *each* mailbox receives the same average amount of noise, regardless of how many mailboxes there are. Alpenhorn aims to strike a good balance by ensuring there is a roughly equal amount of noise and real requests in each mailbox.

7 Implementation

To evaluate Alpenhorn's design, we implemented a prototype in approximately 10,000 lines of Go code:

https://github.com/vuvuzela/alpenhorn

Our implementation of IBE uses the BN-256 elliptic curve [3] (implemented in AMD64 assembly [34]), which targets the 128-bit security level. Recent improvements in cryptanalysis that were published after we built our prototype suggest that BN-256 actually provides less than 96 bits of security [27]. We hope to adopt a more suitable curve in the future to address this, but we do not expect this to have a dramatic impact on our performance results.

Our prototype implements an *entry server*, which is separate from the mixnet and IBE servers. The entry server's job is to manage a large number of WebSocket connections from clients, announce when a new round is starting, and aggregate client requests into a single batch that is sent to the Alpenhorn servers. The entry server is not trusted.

Finally, to distribute the add-friend and dialing mailboxes to many users, our prototype relies on a content distribution network (CDN), such as Akamai.

8 Evaluation

We quantitatively answer the following questions:

- What is the latency for adding a friend and initiating a conversation through Alpenhorn, and what is the client overhead imposed by Alpenhorn? (§8.2)
- Can Alpenhorn support a large number of users, and how does it scale when adding more servers? (§8.3)
- How does Alpenhorn handle skewed workloads, where some users are highly popular? (§8.4)
- How much effort is required to integrate Alpenhorn into a private messaging application? (§8.5)
- How would Alpenhorn's performance be impacted if new weaknesses are discovered in the pairing-based cryptography that Alpenhorn's IBE relies on? (§8.6)

The results suggest that Alpenhorn can provide acceptable performance for private text messaging applications that tolerate latency on the order of minutes, such as Vuvuzela.

8.1 Experimental setup

To answer some of the above questions, we ran experiments on Amazon EC2. Each server ran on a `c4.8xlarge` virtual machine running Linux 4.4 with 36 Intel Xeon E5-2666 v3 CPU cores, 60 GB of RAM, and 10 Gbps of network bandwidth. We compiled the code with Go 1.7.

Unless otherwise specified, our experiments used a chain of three servers, each corresponding to one VM. Each of these servers also ran a PKG. We used one additional VM to run the entry server. The first server in the chain and the entry server were located in the Virginia EC2 region. The second server was located in Ireland and the third server in Frankfurt, Germany. For experiments with more servers, we used the same three regions in a cycle.

Clients were simulated on five `c4.8xlarge` VMs in Virginia (each individual client was limited to using at most 4 cores). To avoid establishing millions of TCP connections, we opened 1,000 connections from each client VM to the entry server, and assigned multiple clients to each TCP connection. We did not use a CDN in our experiments for distributing mailboxes; instead, only a small number of clients downloaded their mailbox in each round (enough to report sound measurements). Each round, 5% of generated requests were real (not cover traffic). For example, to simulate one million users in the add-friend protocol, we generated 50,000 `AddFriend` requests, and 950,000 cover traffic messages. For dialing experiments, each client had 1,000 friends in their address book, and the maximum number of intents was 10. Unless otherwise noted, our experiments assume that all users are equally popular.

Each mixnet server adds an average of $\mu = 4,000$ noise messages to each add-friend mailbox and $\mu = 25,000$ noise messages to each dialing mailbox. With Laplace b parameters of 406 and 2,183 respectively, each protocol achieves ($\varepsilon = \ln 2$, $\delta = 10^{-4}$)-differential privacy for 900 add-friend requests and 26,000 calls (e.g., 7 calls per day for 10 years). For our experiments, we set $b = 0$ to reduce the variance in the results. The Vuvuzela paper discusses the implications of differential privacy parameters in detail.

8.2 Client performance

Deploying Alpenhorn requires the application developer to decide how frequently to run the add-friend and dialing rounds. The main consideration is a trade-off between latency and client bandwidth: more frequent rounds reduce latency but require clients to download mailboxes more frequently (the rest of this section quantifies this trade-off). We expect that Alpenhorn would be used in settings where users do not expect an instant response to friend requests (similar to adding a friend on Facebook), and where users do not add new friends very often. In this setting, the latency of the dialing protocol is more important than the add-friend latency, since the add-friend protocol needs to happen only once between pairs of users, and all further attempts to communicate use the dialing protocol.

Bandwidth. We compute the latency and bandwidth requirements of add-friend and dialing. The crucial parameters that affect latency and bandwidth are round duration (how long Alpenhorn waits to process the next batch of requests for the next round), and the number of active users, which affects the number of requests processed in a round.

The duration of add-friend and dialing rounds are parameters to the system that can be used to adjust client bandwidth usage. Figure 6 shows the total client-side bandwidth requirement of the add-friend protocol as a function of the round duration. The bandwidth is mostly spent on downloading add-friend mailboxes. In Figure 6, with one million users, each add-friend mailbox contains around 12,000 friend requests from users and around 12,000 friend requests from noise (4,000 per server, on average), for a total of 24,000 requests. At 308 bytes per request, the add-friend mailbox is around 7.4 MB every round. As described in §6, when the number of requests goes up, the mixnet increases the number of mailboxes,

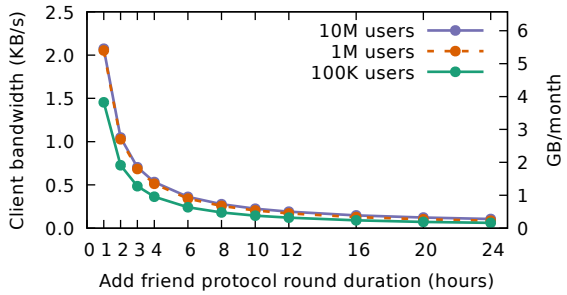

Figure 6: Required client-side bandwidth for the add-friend protocol when varying the round duration.

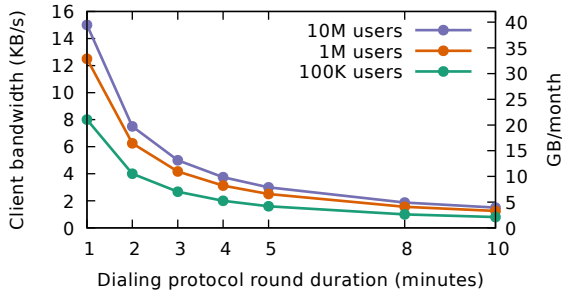

Figure 7: Required client-side bandwidth for the dialing protocol when varying the round duration.

thus ensuring that the size of the mailbox stays roughly constant. (With 100K users, the number of messages each round is less than 12,000, so the single mailbox is smaller than 7.4 MB.)

The dialing protocol analysis is shown in Figure 7. Nearly all of the client bandwidth is spent on downloading the Bloom filter that is scanned for dial tokens. With 1M users and 5% active, Alpenhorn uses one Bloom filter to encode the 125,000 received dial tokens. At 48 bits per token, the Bloom filter is 0.75 MB each round. With 10 million users and 500K active, Alpenhorn distributes the incoming dial tokens into 7 distinct Bloom filters (mailboxes). Each Bloom filter has a roughly equal amount of noise and user data (75,000 each), so the total Bloom filter size is 0.9 MB per user per round. If Alpenhorn uses a dialing round duration of 5 minutes, then the average client bandwidth is 3 KB/s, or 7.8 GB per month (manageable for a cellphone with occasional WiFi connectivity). With a round duration of 5 minutes, the average end-to-end latency for `Call` requests is about 2.5 minutes.

CPU. We measured the client-side CPU usage of both protocols. Our implementation of IBE can compute 800 decryptions per second per core. Thus, to scan a mailbox with 24,000 friend requests takes 8 seconds on a 4 core machine. The CPU cost of dialing is tiny in comparison. One core can compute 1 million hashes per second. Even if a user has 1,000 friends in their keywheel (much more than the average number of friends on Facebook [21, 40]), and the application uses 10 intent values, the Bloom filter can be scanned in less than a second.

Key extraction. We measured the time it takes a client to obtain the combined identity private key for a single round from the PKGs, with a varying number of PKGs running in the same EC2 region as the client. With 3 PKGs, the median latency was 4.9 msec (ranging from 4.7 msec to 7.6 msec) over 100 runs. With 10 PKGs, the median latency was 5.2 msec (ranging from 4.7 msec to 10.8 msec). This suggests that, for a client, there is almost no cost to additional PKGs aside from the network latency of contacting the servers.

8.3 Server performance

To evaluate whether Alpenhorn can support a large number of users, we measured the time it takes for the servers to complete a round, varying the number of users and servers. Specifically, we measured the latency of `AddFriend` and `Call` requests, assuming the client sends the requests at the optimal time just before the round closes, and measuring time until the client downloads the mailbox and finishes scanning all requests. Thus, our latency measurements do not include artificial delays imposed by the servers to reduce bandwidth costs (in an actual deployment, servers would be idle most of the time, because the interval at which new rounds start is much higher than the time it takes to complete a round, to keep client bandwidth reasonable). We also measured the throughput of PKG servers generating users' IBE private keys.

Add-friend. Figure 8 shows the median, minimum, and maximum observed latencies as we varied the number of users, for different numbers of servers. With 10 million users, the median 3-server round latency is 152 seconds. Adding more servers increases the latency due to the additional processing that each server must perform, and due to the additional noise introduced by additional servers.

Dialing. Figure 9 shows the latency for dialing as we varied the number of users. The graph shows that Alpenhorn can support 10 million users on 3 servers with a latency of 118 seconds. The latency increases with more servers much as with the add-friend protocol.

PKG servers. The PKG servers in Alpenhorn have to extract a private key for every user in every round, which can place a lower bound on how frequently add-friend rounds can run. In our experiments, a PKG takes 232 seconds to respond to 1 million user key extraction requests (4310 requests per second). This suggests that, even with 10 million users, each PKG can extract the keys of all users in well under an hour.

8.4 Skewed popularity

To evaluate Alpenhorn's performance under a skewed workload, we measured the latency of `AddFriend` and `Call` rounds, as above, with a varying user popularity

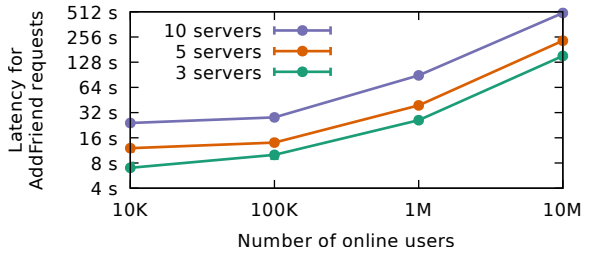

Figure 8: Performance of Alpenhorn's add-friend protocol when varying the number of users online.

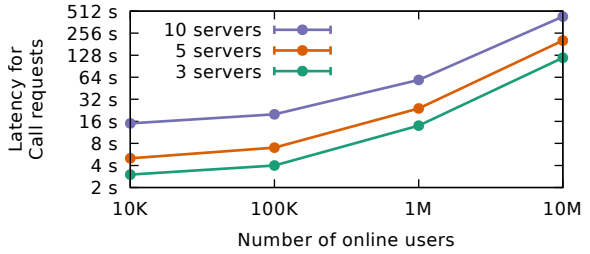

Figure 9: Performance of Alpenhorn's dialing protocol when varying the number of users online.

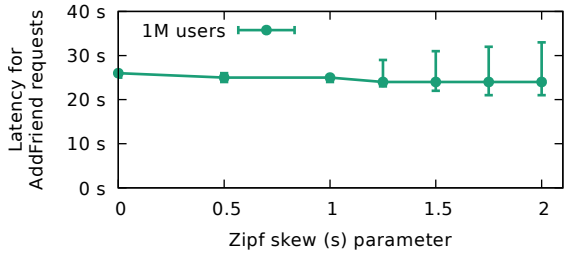

Figure 10: Latency for Alpenhorn's add-friend rounds when varying the skew of the user popularity.

distribution. In particular, instead of choosing the recipient of `AddFriend` or `Call` uniformly at random, in this experiment the recipient is chosen according to a Zipf distribution; that is, the probability of picking some user i, from 1 to N (the number of users) is proportional to i^{-s}.

Figure 10 show the results for the add-friend protocol for 1M users and 3 servers. The median latency stays constant even as user popularity becomes highly skewed (e.g., at $s = 2$, the top 10 users receive 94.2% of all requests). However, as the skew increases, the maximum latency increases (and the minimum decreases) because some mailboxes contain more messages (if they happen to correspond to highly popular users), and other mailboxes become smaller. Even for highly skewed distributions, the effect is not dramatic because Alpenhorn mailboxes already contain a significant amount of noise (about half of the messages, on average) regardless of where users choose to send messages that round. With 1M users and $s = 2$, the largest mailbox is 14.95 MB and the smallest is 4.15 MB.

Dialing is less affected by skew because the client CPU time to scan a mailbox is negligible. With 10 million users and $s = 2$, the minimum and maximum latencies are 119 and 120 seconds respectively, and the minimum and maximum mailbox sizes are 231 KB and 1.39 MB.

8.5 Application integration

To evaluate whether Alpenhorn fits with private messaging applications, we integrated it with Vuvuzela and Pond.

Vuvuzela had its own dialing protocol for starting a conversation (which assumed out-of-band public key distribution and did not provide forward secrecy), which we replaced entirely with Alpenhorn. We had to change 200 lines of code; this does not include deleting all of the code from the old dialing protocol. We had to tweak the Vuvuzela conversation protocol, since it expected a public key as input, rather than a shared secret (as provided by `Call`). Our changes also added two new commands to the Vuvuzela client, `/addfriend` and `/call`, which tie directly into the Alpenhorn API. All other Vuvuzela components remain unchanged. The resulting Vuvuzela client that uses Alpenhorn provides the same security guarantees as Vuvuzela (including differential privacy), with the addition of forward secrecy for bootstrapping conversations (which Vuvuzela's original dialing protocol did not provide).

Pond also provides its own bootstrapping protocol called PANDA [2]. PANDA assumes that pairs of users have a shared secret, and provides a GUI for entering that secret. We built a standalone Alpenhorn client that lets users friend and call each other from a basic command-line interface. The client prints the resulting shared secret to the screen, which the users can then paste into PANDA. This eliminates the need to generate a shared secret out-of-band.

8.6 Cryptographic strength

In light of recent attacks on the BN-256 curve [27], which Alpenhorn uses for IBE, it may be necessary to switch to a different curve or IBE construction to maintain Alpenhorn's security guarantees in the future. Since we cannot predict what scheme may provide the best alternative in the future, this section analytically evaluates the impact of such a switch on Alpenhorn's performance.

The IBE construction impacts three aspects of Alpenhorn's performance: CPU cost on the PKG for generating identity private keys, CPU cost on the client for decrypting the add-friend mailbox, and bandwidth for downloading the add-friend mailbox. PKG and client CPU costs would be directly proportional to the respective CPU costs of any new IBE construction. The bandwidth impact, on the other hand, is a bit more subtle. Alpenhorn's current add-friend request is 244 bytes plus the size of an IBE ciphertext (encrypting a symmetric key that encrypts the rest of the request); the IBE ciphertext is 64 bytes in our prototype. This suggests that changes to the curve or IBE scheme used by Alpenhorn should result in linear or sub-linear impacts on Alpenhorn's performance.

9 Discussion and Limitations

Client compromise. If an adversary compromises an Alpenhorn client (i.e., obtains the user's long-term signing private key and the user's keywheel state), the user must generate a new long-term signing key and new keywheels to re-establish security, as we now discuss.

Registering the new long-term signing key faces two complications. First, the adversary can keep using the stolen signing key, thereby preventing the user from re-registering the same email address (since the PKGs implement a 30-day lockout policy). To address this problem, the user should issue a *deregister* command to the PKGs signed by their old key. The second issue is that, once an account is deregistered, an adversary may be able to register his own key under the user's email address, since they likely got access to the user's email account when they compromised the user's machine. We address this issue by placing the account into a 30-day lockout period after deregistration. This way, if the user can re-establish access to their email account within 30 days of the compromise (e.g., through out-of-band authentication with the email provider), they can regain their Alpenhorn account.

Establishing new keywheels with friends requires the user to simply re-run the add-friend protocol with each friend. To guard against the possibility of an adversary corrupting the list of friend long-term signing keys stored on the user's computer, we recommend that the user keep an offline backup of long-term signing keys of friends, and restore from backup to recover from a compromise. On the other hand, we discourage users from backing up their keywheel, since that is bad for forward secrecy.

Lost client state. If the state of an Alpenhorn client is lost (e.g., because the user physically lost their laptop), the user should follow the steps described above for recovering from a compromised client. The only difference is that the user no longer has access to the long-term signing private key, so the user cannot explicitly deregister. However, the user can simply wait for the same lockout period until re-registering their account through email validation.

DoS attacks. A malicious group of users might attempt to cause a denial of service attack by sending friend or dialing requests in every round (rather than cover traffic) in order to fill mailboxes. This can in turn increase client bandwidth, and, since Alpenhorn will create additional mailboxes to compensate for the extra load, cause the mixnet servers to incur a higher CPU cost to generate noise for the extra mailboxes. To address this, Alpenhorn servers could issue a limited number of blinded signatures to each user every day, and reject any requests that don't have a valid unblinded signature. Since the signatures are blinded, this approach would not leak metadata.

Users going offline. Alpenhorn does *not* assume that users stay online all the time (Alpenhorn avoids intersection attacks [32] by using constant-rate cover traffic to and from all client machines and by using noise to ensure differential privacy of observable mailboxes). However, Alpenhorn does assume that the user's observable activity, which includes going online and offline, is not highly correlated with any confidential metadata they want to keep private. A straightforward way to achieve this is to keep Alpenhorn running all the time, but this may not be practical for users with mobile devices.

An example scenario that illustrates the above problem would be two users that both close their laptops at the same time after finishing a conversation; an adversary that observes both of them going offline at the same time may infer that both of them could have been talking just before. One approach to address this that we hope to explore in future work is to require the users to stay online for a random length of time after finishing a conversation, and to use differential privacy to precisely reason about what random time intervals would be required.

10 Conclusion

Alpenhorn is the first system for establishing session keys between pairs of users that does not require out-of-band communication aside from knowing a user's Alpenhorn username (their email address), and that provides privacy and forward secrecy for metadata, assuming that at least one server is uncompromised. Alpenhorn achieves this by using identity-based encryption (IBE) in a novel way to determine another user's public key without revealing metadata in an anytrust setting. Alpenhorn ensures forward secrecy for all data by refreshing IBE keys and by storing client-side secrets in a *keywheel*. The keywheel provides a bandwidth-efficient means for calling existing friends and starting conversations. Together, these techniques enable Alpenhorn to bootstrap communication in messaging systems that support 10 million users.

Acknowledgments

Thanks to Tej Chajed, Chucky Ellison, Jon Gjengset, Jelle van den Hooff, Frans Kaashoek, and Malte Schwarzkopf for helping improve this paper, and especially thanks to Vinod Vaikuntanathan for helping us prove the security of Anytrust-IBE. Thanks also to the anonymous reviewers. This work was supported by NSF awards CNS-1053143 and CNS-1413920, and by Google.

References

[1] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[2] J. Appelbaum et al. Going dark: Phrase automated nym discovery authentication, 2013. https://github.com/agl/pond/tree/master/papers/panda.

[3] P. S. L. M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography – SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331, 2006.

[4] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2014.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] A. Boldyreva, V. Goyal, and V. Kumar. Identity-based encryption with efficient revocation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 417–426, Alexandria, VA, Oct. 2008.

[7] D. Boneh and M. K. Franklin. Identity-based encryption from the Weil pairing. In *Proceedings of the 21st Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 2001.

[8] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[9] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 Workshop on Privacy in the Electronic Society*, Washington, DC, Oct. 2004.

[10] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. In *Proceedings of the 15th Privacy Enhancing Technologies Symposium*, Philadelphia, PA, June–July 2015.

[11] X. Boyen. Multipurpose identity-based signcryption: A Swiss army knife for identity-based cryptography. In *Proceedings of the 23rd Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 2003.

[12] X. Boyen and B. Waters. Anonymous hierarchical identity-based encryption (without random oracles). Cryptology ePrint Archive, Report 2006/085, June 2006.

[13] J. Brooks et al. Ricochet: Anonymous instant messaging for real privacy, 2016. https://ricochet.im.

[14] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, July 2007.

[15] D. Chaum, F. Javani, A. Kate, A. Krasnova, J. de Ruiter, A. T. Sherman, and D. Das. cMix: Anonymization by high-performance scalable mixing. Cryptology ePrint Archive, Report 2016/008, Jan. 2016.

[16] L. Chen, Z. Cheng, and N. P. Smart. Identity-based key agreement protocols from pairings. http://eprint.iacr.org/, June 2006.

[17] C. Cocks. An identity based encryption scheme based on quadratic residues. In *Proceedings of the 8th Proceedings of the 8th IMA International Conference on Cryptography and Coding*, Cirencester, UK, Dec. 2001.

[18] C. Conley. *Metadata: Piecing together a privacy solution*. ACLU of California, Feb. 2014. https://www.aclunc.org/publications/metadata-piecing-together-privacy-solution.

[19] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2015.

[20] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Usenix Security Symposium*, pages 303–320, San Diego, CA, Aug. 2004.

[21] Edison Research. Average number of Facebook friends of users in the United States as of February 2014, by age group. Statista - The Statistics Portal, Mar. 2014. https://www.statista.com/statistics/232499/americans-who-use-social-networking-sites-several-times-per-day/.

[22] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th Usenix Security Symposium*, San Jose, CA, July–Aug. 2008.

[23] T. Hansen, D. Crocker, and P. Hallam-Baker. DomainKeys Identified Mail (DKIM) service overview. RFC 5585, Network Working Group, July 2009.

[24] A. Iliev and S. Smith. Privacy-enhanced credential services. In *Proceedings of the 2nd Annual NIST PKI Research Workshop*, Apr. 2003.

[25] A. Kate and I. Goldberg. Distributed private-key generators for identity-based cryptography. In *Proceedings of the 7th Conference on Security and Cryptography for Networks*, Sept. 2010.

[26] Keybase. Keybase, 2016. https://keybase.io/.

[27] T. Kim and R. Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *Proceedings of the 36th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 2016.

[28] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the 16th Privacy Enhancing Technologies Symposium*, Darmstadt, Germany, July 2016.

[29] A. Langley. Pond, 2016. https://github.com/agl/pond.

[30] D. Lazar and N. Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata (extended technical report). Technical report, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Oct. 2016. Also available at https://vuvuzela.io/alpenhorn-extended.pdf.

[31] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *Proceedings of the 19th International Conference on Financial Cryptography and Data Security*, Jan. 2015.

[32] N. Mathewson and R. Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *Proceedings of the Privacy Enhancing Technologies Workshop*, pages 17–34, May 2004.

[33] J. Mayer, P. Mutchler, and J. C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences (PNAS)*, 113(20):5536–5541, 2016.

[34] M. Naehrig, R. Niederhagen, and P. Schwabe. New software speed records for cryptographic pairings. In *Progress in Cryptology – LATINCRYPT 2010*, volume 6212 of *Lecture Notes in Computer Science*, pages 109–123, 2010.

[35] Namecoin. Namecoin, 2016. https://namecoin.info/.

[36] T. Perrin and M. Marlinspike. Double ratchet algorithm, 2016. https://github.com/trevp/double_ratchet/wiki.

[37] A. Piotrowska, J. Hayes, N. Gelernter, G. Danezis, and A. Herzberg. AnoNotify: A private notification service. Cryptology ePrint Archive, Report 2016/466, May 2016.

[38] A. Rusbridger. The Snowden leaks and the public. *The New York Review of Books*, Nov. 2013.

[39] A. Shamir. Identity-based cryptosystems and signature schemes. In *Proceedings of the 4th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 1984.

[40] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the Facebook social graph. *CoRR*, abs/1111.4503, Nov. 2011. URL http://arxiv.org/abs/1111.4503.

[41] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[42] WhatsApp. WhatsApp encryption overview, Apr. 2016. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[43] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.