Anysphere: Private Communication in Practice

Security Whitepaper

Arvid Lunnemark Shengtong Zhang Sualeh Asif
{arvid,stzh1555,sualeh}@anysphere.co

June 30, 2022

Last updated: July 1, 2022

ABSTRACT

We describe Anysphere, a metadata-private communication system deployed in the real world. By using private information retrieval based on homomorphic encryption, our protocol guarantees security even if all of our servers are compromised and any number of the users and network observers are malicious. In this whitepaper, we precisely define our threat model, and show how we achieve security against it both in theory and in practice.

CONTENTS

hi

1 MOTIVATION

2 THREAT MODEL

[**TODO**: add table comparing our threat model with e.g. signal and email] [**TODO**: add an illustration of a walled garden, with the walls containing only your computer and your friends' computers]

[**TODO**: Figure out a better format here. See e.g. Skiff's whitepaper.]

Similar to most PIR schemes(for example [Ahm+21], §2.2), our threat model assumes a global adversary who can compromise the entire communication infrastructure except for the user's and their friends' client end. In particular, we assume the adversary has control over all the servers, and can observe and manipulate all network traffic.

End-user trust is more subtle matter. In [ALT18], Angle, Lazar and Tzialla describes the compromised friend(CF) attack on a general meta-data private messaging system, which shows that perfectly hiding metadata while not trusting the user's friends is computationally prohibitive. In our security model, a user trusts that the devices of themselves and all their friends are uncompromised and running an unmodified copy of anysphere's client-side code. The user assumes that any other end-user device might be compromised.

[**TODO**: Can we assume that only a small number of friends are compromised?]

Finally, we assume the security of the standard cryptography primitives we use, including microsoft SEAL's BFV cryptosystem and libsodium's AEAD cryptosystem.

Denial of Service(DoS) attacks are unavoidable if the adversary controls all our servers. In the case of such attacks, we do not guarantee liveliness of our service, but continue to guarantee metadata security. We also defend against DoS attacks launched by an end-user with no access to the servers.

2.1 Desired Properties

1. **Perfect Metadata Protection**: All contents and metadatas associated with a conversation are only visible to the users involved with the conversation. Even with a compromised server, an adversary should not even be able to find out whether two users are engaging in any conversation.

2. **Forward Secrecy**: [**TODO**: do we provide this?]

3. **Resistant to man-in-the-middle attacks**.

4. **Resistant to social engineering attacks**.

[**TODO**: add more.]

3 CORE PROTOCOL

Alice wants to message Bob over an untrusted network, without leaking any data or metadata to anyone. To hide the message content they just use end-to-end encryption. To hide metadata they employ two key ideas: sending data at a constant rate, and retrieving homomorphically compressed data.

When signing up, each user gets their own *outbox* on the server. This outbox is a dedicated storage space that the user sends messages to. Once every minute, Alice will send exactly 1 KB of data to her outbox on the server. If she has a message to send, she sends the encryption of that message, and if she has no message to send, she sends a random sequence of bytes. With this simple first idea, no one, including the server and any network observers, will know when Alice actually sends a message.

The message needs to be routed to Bob. Now, a traditional messaging system would have Bob download data from Alice's outbox, and then try to decrypt it to see if it was meant for him. But this leaks metadata: the server would know that Alice wrote to outbox x and that Bob read from outbox x, which links the two of them together!

Our solution is for Bob to, once every minute, download *all* outboxes from the server. On his own computer, he can then check Alice's outbox. This way, no one, not even the server, has any way of linking Alice to Bob. All metadata is protected.

This is how Anysphere works. Obviously, Bob cannot download all outboxes every minute — that would be way too much data! — so instead he uses *private information retrieval*, a well-studied cryptographic primitive, as a way of compressing his download size. The following subsections will describe the system in detail.

[**TODO**: Figure: Alice, the server outboxes, and bob on the other side downloading the entire database.]

3.1 Private information retrieval

Bob wants to download outbox i without revealing i to anyone. Viewing the collection of outboxes as a database array db, he wants to retrieve $\mathsf{db}[i]$ privately. This problem was first introduced as *private information retrieval* (PIR) in 1995 [Cho+95], extended in 1997 to our threat model under the name cPIR [KO97], and has been extensively studied since then [Mel+16; Ang+18; Ahm+21].

Our implementation currently uses FastPIR, which is one of the fastest cPIR schemes [Ahm+21]. All cPIR schemes have the same security properties (i.e., they leak zero information), and we are actively researching faster schemes (see Section 6).

All known cPIR schemes use homomorphic encryption [Gen10]. To compute the query q, Bob encrypts i with a homomorphic encryption scheme using a secret key s: $q = \mathsf{HEnc}_s(i)$. The server can then homomorphically evaluate the function $f(i) = \mathsf{db}[i]$, producing the answer $a = \mathsf{HEnc}_s(\mathsf{db}[i])$. Bob can finally decrypt to find $\mathsf{db}[i]$. In practice, $f(i)$ is often defined in terms of a dot product with a unit vector representing i, because the homomorphic scheme being used, BFV [FV12], is particularly good at dot products.

Registration. Server allocates outbox i and generates authentication token tk. Alice receives (i, tk).

Trust establishment. Alice and Bob agree on a shared secret key k. Details in Section 4.

Sending. Exactly once every minute:

- If Alice has a queued message m: she sends $(i, \mathsf{tk}, \mathsf{Enc}_k(m))$ to the server, where k is the key shared with Bob.
- Otherwise: she sends (i, tk, r) to the server, where r is a random sequence of bytes.

The server receives (i, tk, c) and stores c in outbox i if tk is correct.

Receiving. Exactly once every minute:

- Bob sends $q = \mathsf{PIR.Query}(i)$ to the server.
- The server responds with $a = \mathsf{PIR.Answer}(\mathsf{DB}, q)$ and Bob decodes it into $c = \mathsf{PIR.Decode}(a)$.
- Try to decrypt $\mathsf{Dec}_k(c)$ where k is the key shared with Alice.

Figure 1: The simplest version of our core protocol.

3.2 Security proof

The simplest version of our core protocol is shown in Figure 1. In this section, we prove: (1) that Alice and Bob enjoy complete metadata-privacy without having to trust anyone else, and (2) that our protocol is resistant to denial of service attacks from users.

In this section, we formally state and prove metadata security.

[**TODO**: simulation security definition, going offline, friend attack, key privacy because prf.]

3.2.1 Going offline. Users will not always be connected to the internet. At night, most people put their computers to sleep. This means that users will not be sending and receiving exactly once every minute. [**TODO**: how much information does this leak?]

3.2.2 Authentication token. On registration, the server creates a unique authentication token for a new user. This allows the server to restrict access to that user's outbox, preventing denial of service attacks from other users. It should still be noted that, in accordance with our threat model, we do not prevent against denial of service attacks by ISPs or the server itself — fundamentally, a powerful actor can always shut down your internet access. In Section 6 we discuss plans for distributing the server such that, say, only 1 out of 3 servers need to be trusted to provide service.

3.3 Multiple contacts

The simple protocol from before assumed Alice had a single contact Bob. Our system allows many more contacts.

[**TODO**: link the compromised friend attack]

3.4 Chunks and acknowledgements

If Alice wants to send a message longer than 1 KB, she needs to chunk the message up. Here, we have taken inspiration from TCP/IP. We require a message to receive an acknowledgement, shortened ACK, before we send the next message in the sequence.

[**TODO**: describe the separate PIR table for ACKs.]

[**TODO**: Figure: pseudocode for Register, Send, Retrieve (with everything)]

4 FRIEND DISCOVERY

Most existing metadata private messaging systems, such as Pung or Addra, assumes that a key exchange has taken place between the users before the conversation between them starts. In our messaging system, we also need a mechanism to conduct the key exchange itself. In other words, if a user A knows the public key pk_B of user B, then A should be able to send a "friend request" to user B without leaking this request to anyone else. User B must then be able to retrieve this request from the server, and complete a key exchange with user A. We call this process "adding friends".

This problem, known as Oblivious Message Detection(OMD) in [LT21], is very hard in general. The state-of-the-art scheme proposed in [LT21] costs each user $1 per million messages scanned, which is too expensive for our messaging application. Instead of solving OMD in general, we provide three alternative ways of adding friends, which trades some user convenience for better computation cost and security.

4.1 Adding Friends Face-to-face

Our first method assumes that users A and B are able to set up a face-to-face meeting with each other, either in person or over zoom.

We implement face-to-face adding friend using a simple key exchange mechanism. To add friends, A's anysphere client generates a QR code, which containing A's name, public key pk_A, and index in the Addra database. B's client generates a similar key. A and B can then scan each other's QR's code, [**TODO**: Alternatives here? Passphrase? Link?] and derive a common secret by calling a standard key exchange algorithm such as crypto_kx_server_session_keys() [**TODO**: Is this secure?] by libsodium. They also add each other's database index to the list of indices they scan each round. User A and B can now send messages to each other at will.

The advantage of this method is that key exchange can be completed instantly, cheaply, and securely. The disadvantage is that it requires our two users to be able to set up a meeting. While this assumption might seem too restrictive, we note that users must establish that their friends are not malicious before adding them, or else risk getting compromised. Thus, they are likely willing to set up a meeting to establish trustworthiness.

[**TODO**: step-by-step guide?]

4.2 Adding Friends Asynchronously

[**TODO**: mention key privacy]

Our second method targets the opposite use-case, when user B does not know user A's intention to add friend beforehand.

In this method, user A must know user B's public keys. User A composes a friend request m containing A's name, public key pk_A, key exchange material, and index in the Addra database. User A encrypts the request m with B's public key pk_B using an AEAD cryptosystem, and deposits the request into a database for all asynchronous requests. User B periodically downloads the entire database and tries to decrypt each message in the database using their secret key sk_B. If B's decryption succeed, then B can add A as a friend by sending A an ACK message over the usual PIR database containing B's key material, encrypted using A's public key. A and B can now compute a shared secret, add each other's index in the database, and send messages to each other at will.

This method offers convenience on par with most existing messaging platforms. Its main disadvantage is cost and delay: downloading the entire database is expensive and time-consuming for user B. Furthermore, it makes user B more difficult to ascertain that user A is trustworthy, thus compromising B's security. We discourage the use of this method, and allow users to disable it completely. [**TODO**: Decision to figure out later?] [**TODO**: step-by-step guide?]

4.3 Adding Friends Transitively

[**TODO**: Not sure if this is going to be a thing for now.]

5 CLIENT-SIDE SECURITY IN PRACTICE

Our theoretical threat model assumes that the user's local computer is completely trusted, and that it is running a correct implementation of our protocol. If your computer is compromised, or you are running a buggy or intentionally incorrect version of our code, none of our previously outlined theoretical guarantees will apply. This is inherent and unavoidable — no matter the fancy encryption schemes you come up with, nothing will help you if your computer comes with a preinstalled backdoor. While **we fundamentally cannot eliminate the client-side risk**, we can *reduce* it, which we will do in this section.

5.1 Reducing the attack surface

The first step in mitigating security is to reduce the attack surface. To do so, we architected our client to consist of two parts: a UI frontend and a daemon backend, where the daemon backend contains all security-critical code. We sandbox the UI frontend in such a way that it is not allowed to talk to the internet, and let all message sending go through the daemon, which handles the cryptography. That way, even if there are bugs in the UI frontend, or potentially malicious code, there is not much it can do.

[**TODO**: Figure of client architecture with UI and daemon, showing how we cut off internet access.]

We also reduce the attack surface of the daemon itself. In particular, we use C++ instead of other popular languages (Rust, Go, Python), because all other practical languages are significantly more susceptible to supply chain attacks. Our daemon has 4 direct dependencies (Abseil, gRPC, SQLite, Libsodium) and 0 transitive dependencies. A comparable implementation in a language with a package manager

would easily use 100s of transitive dependencies. We elaborate more on our choice of C++ in this blog post [**TODO**: Link.].

5.2 Code distribution

We sign everything.

Maybe we sign everything twice?

Maybe we have a cold-storage and a hot-storage signing key?

Do we store a backup key in cold storage that we can use to revoke a version? And people can disi

[**TODO**: Either understand whether standard OS signing is good enough, or whether we should sign things ourselves.]

5.3 Updates

Every update needs to be signed. In fact, it needs to be signed twice: Arvid holds one key, and Sualeh holds one key. Both signatures must be present for the local client to accept the update.

We implement our own signature check. Many popular frameworks have built-in signature checks, such as AutoUpdater for Electron and Updater for Tauri, but to ensure that we are really certain that updates work the way we want them to, we do it ourselves.

This means that if either of us loses our private key, you would not get any updates. This is by design.

5.4 Protecting against non-privileged local malware

If you've granted administrator access to a malicious program on your computer, there is, unfortunately, nothing to be done. We can, nevertheless, reduce the risk of non-privileged malware.

[**TODO**: Actually implement: allow to encrypt the database, in which case the both the GUI and the CLI need to require passwords (and the GUI may cache the password for some amount of time).]

Again, we do not aim to eliminate the risk here. Non-privileged malware may still gather information from side-channel attacks, and potentially other avenues. Once an attacker has access to your computer, it is very, very hard to shield yourself from them.

6 FUTURE WORK

REFERENCES

[Ahm+21] Ishtiyaque Ahmad et al. "Addra: Metadata-private voice communication over fully untrusted infrastructure". In: *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 2021.

[ALT18] Sebastian Angel, David Lazar, and Ioanna Tzialla. "What's a little leakage between friends?" In: *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*. 2018, pp. 104–108.

[Ang+18] Sebastian Angel et al. "PIR with compressed queries and amortized query processing". In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 962–979.

[Cho+95] Benny Chor et al. "Private information retrieval". In: *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE. 1995, pp. 41–50.

[FV12] Junfeng Fan and Frederik Vercauteren. "Somewhat practical fully homomorphic encryption". In: *Cryptology ePrint Archive* (2012).

[Gen10] Craig Gentry. "Computing arbitrary functions of encrypted data". In: *Communications of the ACM* 53.3 (2010), pp. 97–105.

[KO97] Eyal Kushilevitz and Rafail Ostrovsky. "Replication is not needed: Single database, computationally-private information retrieval". In: *Proceedings 38th annual symposium on foundations of computer science*. IEEE. 1997, pp. 364–373.

[LT21] Zeyu Liu and Eran Tromer. "Oblivious Message Retrieval". In: *Cryptology ePrint Archive* (2021).

[Mel+16] Carlos Aguilar Melchor et al. "XPIR: Private information retrieval for everyone". In: *Proceedings on Privacy Enhancing Technologies* (2016), pp. 155–174.