Anysphere: Private Communication in Practice

Security Whitepaper

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

June 30, 2022

Last updated: June 27, 2022

ABSTRACT

We describe Anysphere, a metadata-hiding communication system deployed in the real world. We discuss the theoretical protocol as well as security-critical implementation details. The goal of this whitepaper is to describe the threat model, and show how Anysphere guarantees privacy against all threats within the model.

CONTENTS

1 INTRODUCTION

When the internet was first established, everything sent over it was public. If A sent a message to B, anyone on their path through the internet could see that such a message was sent, and read the actual message. As of today, many messaging services are end-to-end encrypted, meaning that no one can read the contents of messages. However, for sufficiently powerful adversaries, it is still possible to find out who is talking to whom. Our goal is to hide this metadata: we want to create a system where A and B can send messages to each other over an untrusted network, without anyone knowing that they talk to each other. Such a private communication system would be critically important to protect and expand freedom in the world [Lun21].

Anysphere is a provably secure metadata-hiding communication platform, built for the real world. In this document, we outline how we guarantee theoretical security, as well as how we make sure that the theoretical security translates to good practical security.

2 THREAT MODEL

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 of service. We also defend against DoS attacks launched by an end-user with no access to the servers.

3 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.

3.1 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.].

3.2 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.]

3.3 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.

3.4 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.

4 TRANSMISSION

Transmission of messages is the central component of anysphere. In this section, we assume that

5 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.

5.1 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.

5.2 Adding Friends Asynchronously

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?]

5.3 Adding Friends Transitively

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

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.

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

[Lun21] Arvid Lunnemark. *It Is Time to Move Beyond End-to-End Encryption*. 2021. URL: https://anysphere.co/it-is-time-to-move-beyond-end-to-end-encryption/ (visited on 06/16/2022).