Anysphere: Private Communication in Practice

Security Whitepaper

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

June 30, 2022

Last updated: July 2, 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

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[stzh: route?] through the internet could see that such a message was sent, as well as 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 — hackers, ISPs, government agencies — it is still possible to find out who is talking to whom, as well as when and how often messages are sent. 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 [**arvid**].

Attacker compromises ···	Anysphere	Signal	Skiff	Wickr	Onion
Attacker listens on the internet	✓	✓	✓	✓	
Attacker compromises	✓			✓	✓
Skiff	✓				✓
Wickr	✓			✓	✓
Onion	✓	*1		✓	✓

Table 1: Comparing when

From the start, Anysphere has assumed a single important security principle: *no needless trust*. This simple principle guide Anysphere's development and is the key motivator behind our architecture, our threat model, and our choice of private information retrieval as our core protocol.

Quite simply, we believe that communication must place no needless trust in systems across the internet since many conversations are so meaningful[**TODO**: impactful?] that unwarranted trust can lead to harm. In particular, we currently only trust the local device and your friends. That is it.

Our principles are fundamentally different from Signal, Email, and other platforms and protocols that assume trust in the servers, in the plumbing of the internet, and in networks being so large that no entity can monitor them. We also solve many of the practical problems that plague research prototypes. We are working hard to make Anysphere more secure, and we are working hard to make it extremely security realistic for everyone.

2 Security Context

[**TODO**: Explain why Signal is not enough. Good table.]

[stzh: In this section, we describe what our system achieves beyond end-to-end encrypted messaging applications like Signal and Wickr.]

Signal is great. It is end-to-end encrypted, open source, and run by a trustworthy group. Unfortunately, if their servers are hacked, one of their employees bribed, or you are simply attacked by a network-level powerful actor, there is nothing Signal can guarantee.

[stzh: Furthermore, even with a secure server, it leaks metadata. Using timing attacks, an adversary can figure out when, where and with whom you are talking.]

2.1 Goals

[stzh: Our goals]

2.2 Threat Model

[**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 [**ahmad2021addra**], §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 [**angel2018s**], 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.

2.3 Desired Properties

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

An especially challenging case is metadata protection during trust establishment. This happens when two users wish to add each other as friends for the first time. This feature is not supported even by Addra or Pung. We outline our solution in **??**, which offers users a variety of options to establishing trust.

2. **Resistance to man-in-the-middle attacks**: Due to our use of PIR and end-to-end encryption, no man-in-the-middle without access to a user's private keys can access metadata associated with any conversation the user had.

3. **Resistance to DoS attacks**: 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.

2.4 Non-goals??

1.

3 Core Protocol

[stzh: In this section, we consider two users of our service Alice and Bob.] 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 [stzh: padded] encryption of that message, and if she has no message to send, she sends a random sequence of bytes.[stzh: A random message encrypted with a random public key?] 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.

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 [**chor1995private**], extended in 1997 to our threat model under the name cPIR [**kushilevitz1997replication**], and has been extensively studied since then [**melchor2016xpir**; **angel2018pir**; **ahmad2021addra**].

Our implementation currently uses FastPIR, which is one of the fastest cPIR

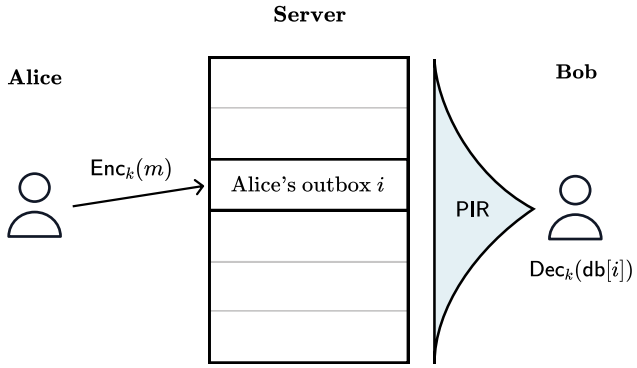

Figure 1: Alice sends to her outbox once every minute. Bob retrieves Alice's outbox using private information retrieval (PIR), which looks the same to everyone else as if he downloaded the entire database. Alice and Bob can use standard symmetric encryption to communicate, and no one, not even the server, will learn anything at all.

schemes [**ahmad2021addra**]. All cPIR schemes have the same security properties (i.e., they leak zero information), and we are actively researching faster schemes (see **??**).

All known cPIR schemes use homomorphic encryption [**gentry2010computing**]. 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 [**fan2012somewhat**], is particularly good at dot products.

3.2 Security proof

The simplest version of our core protocol is shown in **??**. 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

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 **??**.

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 2: The simplest version of our core protocol.

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 **??** 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 Trust Establishment

Most existing metadata private messaging systems, such as Pung or Addra, assume that users know beforehand who they wish to talk to, and had a prior key exchange through another channel. In our messaging application, we need a metadata-secure mechanism for the key exchange itself. In other words, if Alice knows the public key pk_B of Bob, then Alice should be able to send an "invitation" to Bob. Bob must be able to retrieve this invitation from the server, and complete a key exchange with Alice. We call this process "trust establishment".

This problem is known as Oblivious Message Detection(OMD) in [**liutromer2021**]. The scheme proposed in [**liutromer2021**] aims to minimize user download size, but it costs each user $1 per million messages scanned, which is prohibitive for our messaging application. We provide two alternate methods of trust establishment with better computational cost and security.

In the following two methods, we assume that Alice and Bob's daemons have generated a Curve25519 key exchange keypair $kx = (kx^P, kx^S)$. Their goal is to inform each other of their kx^P to derive a shared secret sk.

4.1 Face-to-face Invitations

Our first method assumes that Alice and Bob are able to set up a face-to-face meeting with each other, either in person or over zoom. The trust establishment process is a simple key exchange implemented as follows.

1. Alice encodes the plaintext of her key exchange public key kx_A^P and her allocation index i_A into a human-readable story s_A. Bob similarly encodes his story s_B.

2. Alice and Bob meet. They share their stories, and type the other's story into their own client.

3. Alice decodes Bob's story to obtain kx_B^P and i_B. She computes the shared secret $sk = DH(kx_B^P, kx_A^S)$, and adds i_B to her set of listening indices.[**TODO**: Better name?] Bob does the same. They can now communicate to each other using our PIR transport layer.

Using this method, all interactions between the users do not pass through any third party. Thus, trust establishment can be completed instantly, cheaply, and

securely. Our system do require Alice and Bob to be able to set up a meeting and type each other's story manually. To justify this approach, we believe a privacy-conscious Alice would be willing to set up a face-to-face meeting with Bob before sending him sensitive information.

4.2 Asynchronous Invitation

[**TODO**: mention key privacy]

Our second method targets the case when Bob does not know about Alice's invitation beforehand. For example, Alice can be a sensitive client who wishes to privately reach out to Bob's company.

This method consists of three steps. First, Alice sends an encrypted invitation to the server without indicating its recipient. Second, Bob retrieves this invitation via a full database download. Third, Bob informs Alice of his acceptance via an ACK message.

To send an invitation

1. Upon registration, each user's daemon generates an additional invitation key-pair $ki = (ki^P, ki^S)$. Bob's daemon computes a "public id" id_B, which contains his invitation public key ki_B^P, key exchange public key kx_B^P, and allocation i_B encoded in plaintext. It then displays the id on Bob's GUI. Bob posts id_B on his public profile, such as on twitter or on his company's website.

2. When Alice wishes to send an invitation to Bob, she obtains id_B from Bob's public profile, and decodes it to obtain (i_B, kx_B^P, ki_B^P). She drafts an initial message m_{AB} to accompany her invitation.

3. Alice's daemon periodically sends the key-value pair $(i_A, c_{AB} = \mathsf{Enc}(kx_B^P, id_A | m_{AB}))$ to the server[2], which stores it in a separate **AsyncInvitationDatabase**. When Alice has no invitations, her daemon sends $(i_A, \mathsf{Enc}(kx^P, id_A))$ for a random public key kx^P.

4. As Alice's daemon sends the invitation, it also compute the shared secret with Bob $sk = DH(kx_B^P, kx_A^S)$. It sends Bob a "control message" ctm_{AB} via our PIR transport layer, and listens for Bob's ACK.

To retrieve an invitation

1. Bob's daemon periodically downloads the entire **AsyncInvitationDatabase**. It computes $\mathsf{Dec}(kx_B^S, c)$ over all key-value pairs (i, c). If the decryption fails, Bob's daemon ignores this pair. Now suppose $(i, c) = (i_A, c_{AB})$, and the decryption succeeds. Bob's daemon decodes $i = i_A, id_A, m_{AB}$ from the decrypted data, and displays on Bob's GUI that he received an incoming invitation from id_A with message m_{AB}.

[2]It is important to redo this encryption each round, otherwise adversaries will observe the same message repeatedly.

2. Bob verifies Alice's identity using id_A and m_{AB}, for example by checking Alice's public profile. Bob then chooses to either accept or reject the invitation. If Bob rejects the invitation, no further action is performed.

To accept an invitation

1. If Bob accepts Alice's invitation, Bob's daemon decodes id_A to obtain kx_A^P, and computes the shared secret $sk = DH(kx_A^P, kx_B^S)$. It adds i_A to the set of listening indices.

2. Since Alice is sending the control message ctm_{AB} to Bob using the same shared secret sk, Bob's daemon will read ctm_{AB} from the PIR database. It sends an ACK to Alice's control message.

3. When Alice's daemon reads Bob's ACK to ctm_{AB} from the PIR ACK database, it displays on Alice's GUI that Bob has accepted Alice's invitation. Alice and Bob can now communicate to each other using our PIR transport layer.

This method achieves metadata security. We hide the timing of the invitation by making the daemon send to and download **AsyncInvitationDatabase** on a fixed schedule. We hide the recipient of Alice's request since our encryption scheme is key private([**TODO**: Cite arvid's section]).

This method offers convenient trust establishment on par with most existing messaging platforms. Its disadvantage is that downloading the entire database is expensive. We estimate that a key-value pair in the **AsyncInvitationDatabase** take approximately 200B. If we have 1 million users, then the whole database will have a size of approximately 200MB. Thus, an individual user might wish to only download the database once or twice each day, which saves bandwidth but delays the detection of invitations. On the other hand, a company user could afford downloading the database every second, which takes 200MB/s download bandwidth but can ensure incoming asynchronous invitations get detected almost instantly.

Finally, we note that this method introduces more attack vectors. For example, an attacker might use social engineering to make Alice believe that a fake id_B belongs to Bob, or simply monitor A's traffic to B's public profile. Therefore, we recommend privacy conscious users to use face-to-face invitations whenever possible.

5 Practical Security

The theoretical guarantees in the previous sections — that security is guaranteed without needing to trust the server or the network — all assume that your local computer is completely trusted. This is unavoidable: no matter the fancy encryption schemes you come up with, nothing will help you if your computer comes with a preinstalled backdoor. Nevertheless, while **we fundamentally**

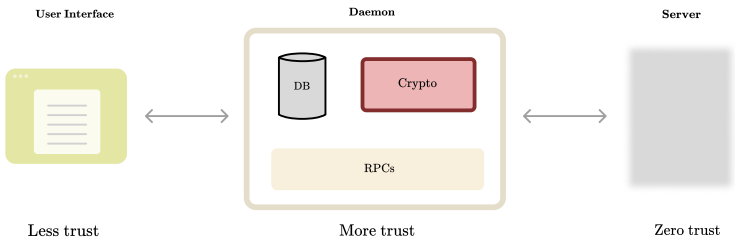

Figure 3: System architecture. The user interface and the daemon run on your local computer and require some trust assumptions. The user interface is cut off from the internet, and can only communicate via the daemon, meaning that it needs lower trust assumptions. The crypto module in the daemon requires the highest level of trust, and uses the widely trusted Libsodium library.

cannot eliminate the client-side risk (no one can), *we can reduce it.* We care about this practical security as much as we care about the theoretical security, and in this section we outline the mitigations we undertake: reducing the attack surface by modularizing and reducing supply chain risk, minimizing bugs by being open source, securing code distribution and updates, and protecting against non-privileged local malware.

Still, we cannot achieve perfect security here, so: **PLEASE DO NOT USE ANYSPHERE FOR SECURITY-CRITICAL USE CASES ON A COMPUTER YOU DO NOT TRUST**. This is especially true while we are in beta, and may have bugs.

5.1 Reducing the attack surface: modularity and supply chain risk

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. This is illustrated in **??**. 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. That way, even if there are bugs in the UI frontend, or potentially malicious code, there is not much it can do.

We also reduce the attack surface of the daemon itself. In particular, we take a lot of care in making sure our dependency chain is small to protect against supply chain attacks. Many of the new and popular languages, including Rust, Go and Python, all have great package managers, which unfortunately has led to an ecosystem where packages generally depend on hundreds of other packages, often transitively. For this reason, we chose to use C++ for essentially all daemon code, depending only on a few well-known packages with long-term support and good security practices (Abseil, gRPC, SQLite, Libsodium). We chose to use Rust for our database code, being extremely careful with our dependency chain.

5.2 Minimizing bugs: open source

All code that is required for trust is open source at https://github.com/anysphere/clientgithub.com/anysphere/c for the daemon and UI code, and https://github.com/anysphere/asphrgithub.com/anysphere/asphr for the code that is shared between the client and our server. Our server is not open source, *because given our threat model it simply does not matter what code is run on our server*. We can run whatever malicious code we want on our server, and your privacy is still completely protected.

5.3 Code distribution and updates

We need to ensure that our code is the code that is running on your computer. For this reason, Anysphere is not a web app — serving security-critical cryptographic code on the web is a huge vulnerability, and should, in our opinion, never be done. The reason is that when you visit a website, you download the cryptographic code every single time, meaning that every single time you use the website, you give attackers an opportunity to serve you malicious code. Instead, Anysphere is a local app: you download it once, and can check the signature and hash after.

Local apps need to be updated. Currently, in our beta version, we use Electron's auto-updater to perform the signature checks for us, but we are actively working on building our own update verification process. In that process, two people from our team need to sign each update separately, and both signatures need to be present for your local app to accept the update. If either of us loses our key, we would not be able to push an update. 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. Our beta version does protect against non-privileged malware right now, but in the future, we are planning to encrypt the local database, require a password to unlock the app, and use OS-level access controls to make sure only certain processes can access the daemon.

Again, we do not aim to eliminate the risk here. Once an attacker has access to your computer, it is very, very hard to shield yourself from them.

6 Related Research

Metadata-private communication has been studied for decades. In 1981, David Chaum introduced so-called mix-nets, which bounce messages between a small number of servers. Using onion encryption, if at least one of the servers is honest, it is impossible to determine the destination of a given source packet. Tor, created in 2002, is one of the most successful privacy-protecting real-world projects, and uses mix-nets [**dingledine2004tor**]. Unfortunately, even aside

from the server trust issue, mix-nets leak timing data which makes it easy for someone with ISP-level control of the network to observe who is talking to whom. In today's world, it is getting easier and easier to amass enough data to perform such correlation attacks, making mix-net based approaches unsuitable for real security [**karunanayake2021anonymisation**].

The 2010s included a flurry of research papers trying out a few different methods of achieving scalable metadata-privacy: so-called DC-nets were tried by Dissent and Riposte [**corrigan2010dissent**; **corrigan2015riposte**], mix-nets with stronger security guarantees were tried by Vuvuzela, Atom, Talek and many others [**van2015vuvuzela**; **cheng2020talek**; **kwon2017atom**], multi-party computation techniques were tried by Clarion, mcMix and Blinder [**alexopoulos2017mcmix**; **eskandarian2021clarion**; **abraham2020blinder**], and function-hiding functional encryption was tried by NIAR [**shi2021non**; **bunz2021non**]. These approaches are either less secure than the PIR-based approach we are using (all but NIAR), or are impractical at scale due to computation time (NIAR). The PIR line of work, started by Angel with Pung [**angel2016unobservable**; **angel2018pir**] and continued by Addra [**ahmad2021addra**], is the only one that promises both perfect security and reasonable scalability.

While there has been a lot of research focusing on the theoretical problem of message transmission, there has been less attention on everything else that needs to exist for a communication system to be useful in practice: initiating connections, handling arbitrary failures, and distributing code securely, to name a few. We draw on the knowledge of the past for message transmission and invent novel protocols for the rest, some of which may be of interest to the research community.

7 What's Next?

Anysphere has just begun operations, and we are devoted to truly free communication. We aim to supervise and help build technologies that allow internet communication without unfair security assumptions. There are several important milestones that we have set for ourselves to reach to make this genuinely possible; some that are essential in the short-term and others that we want to achieve over a longer time horizon with time and effort.

7.1 Short term milestones

- **Group-chats through broadcasting**: An essentially important problem to solve is to allow groups of people to broadcast messages to each other, without depending on the presense of any specific user. We understand that this brings risks in itself to users because it increases the overall risk surface that a user has to trust but we believe it is crucial for large-scale pragmatic adoption.
- **Files and Images**: We hope to allow the sending of small files and images,

through our current protocol. Larger files and images are tricky because they expload in the number of chunks needed to deliver them, but we will tackle that problem in due-time.

- **Denial of Service Protection**:
- **Forward Secrecy**
- **Public Key Infrastructure for Anysphere**

7.2 Long-term milestones

- **Calls**
-