# Privomega: A Privacy-Preserving Random Stranger Chat Protocol (Sketch)

2018-11-22T22:00:00-05:00

## Contents



:warning: This is a work-in-progress sketch.

### Introduction

Omegle is the one of the first stranger chat services. To quote from their website:

> Omegle *(oh · meg · ull)* is a great way to meet new friends. When you use Omegle, we pick someone else at random and let you talk

> one-on-one. To help you stay safe, chats are anonymous unless you tell someone who you are, and you can stop a chat at any time. [1].

Well, it is not exactly accurate:

- It is not really **anonymous** as the server learns your IP address and browser fingerprint, and whatever else it can use to identify you.
- It learns all the messages you sent and can **attribute** those messages to you.
- Users have no **participation privacy** as the server learns who you are chatting with.
- The service cannot prove that it actually **randomly** pairs users.
- There is no **end-to-end encryption** (E2EE) and server sees all your chat messages.

Fortunately, with advances in cryptographic research in recent years, it is possible to have a random stranger chat service that is **privacy-preserving** and **provably random**, and users are guaranteed privacy and anonymity with strong cryptography.

In this post I will present one possible (and not so efficient) construction of such privacy-preserving Omegle, call it *Privomega*, that has the following properties:

- Server does not learn who matched who or who is talking to who.
- Server does not see plaintext chat messages.
- When Alice matches Bob randomly, Alice can be assured that this Bob is randomly chosen from pool of all online users.
- When Bob receives the first message from Alice, he can be assured that Alice has randomly picked him from pool of all online users. (Assuming no collusion between Alice and server, more on that later)
- Server learns nothing other than the total number of users online, users' (one-time) public keys, and which users are posting messages (without learning the recipient of the messages).

## Background and context

### Roles

In this scheme we define 3 roles:

- Alice: a user who is using the service; the initiator of random chat.
- Bob: a random user that Alice has chosen; the random target of the chat. Note that this is not a specific "Bob" but rather a random Bob.
- Server: a server providing key directory service (with key transparency and PIR), a key-value mailbox service (with PIR), and a random beacon service.

### Cryptographic primitives

In this section we outline the cryptographic primitives we will use to achieve our goal of *Privomega*:

- **Transparent logs** will be used on key directory service to ensure consistency of the bindings.
- **Private information retrieval** (PIR) will be used on key directory and mailbox services to allow user to retrieve a piece of data from server without server knowing which piece is retrieved.
- **Verifiable randomness** will be used so Alice can prove to Bob that she has found Bob by random chance.
- **Private end-to-end messaging** will be used so Alice and Bob can communicate with each other while preventing any third party from learning the transcript of the conversation. It is an umbrella term and includes schemes like **digital signatures**, **key exchange**, and **authenticated encryption**.

### Transparency log and key verification

In order for Alice to send an encrypted message to Bob, Alice must first know Bob's public key.

For a user to discover some other user's key for end-to-end encryption, users typically rely on a centralized directory of public keys, maintained by the service provider. Such key servers are vulnerable to hackers and spy agencies, or service provider themselves could be tempering with the data. If Alice asks the key server for Bob's key, and an attacker replaces Bob's key with attacker's key, the attacker could perform man-in-the-middle attack between Alice and Bob.

Existing methods of protecting users against server compromise require users to manually verify recipients' accounts in-person. This simply does not work under the anonymous setting of *Privomega*.

The key directory must be *consistent*, that is given an *authenticated binding* issued by the key server for the name *alice*, anyone can verify that this is the same binding for *alice* that every other party observed [2]. Key transparency allows the auditing of such directories [3]. If server attempts to *equivocate* by issuing multiple bindings for a single username, clients will detect the equivocation promptly with high probability.

In *Privomega*, we use a key directory with key transparency for users' public keys, which can in practice be *CONIKS*. If user Alice detects that her certificate has been changed by the server, or server is carrying out an equivocation attack, she can immediately drop offline and disconnect from the service.

### Private information retrieval

Let server host a mailbox model for message delivery, which is really a key-value map, key is *mailbox ID* and value is a *queue of messages*. Server allows any user to put any message into any mailbox. The messages are of course encrypted so that only the intended recipient could read it. Alice and Bob can secretly agree on the location of some mailbox where Alice is going to drop off messages to Bob. Bob later goes to that mailbox and makes a copy of every encrypted mail, trying to decrypt every one of them until he finds one that he is able to decrypt.

In the real world, if there is no camera or other surveillance around that mailbox, and Bob does not get seen when he retrieves the mail, Alice can be fairly confident that no one knows who she is writing mails to, and Bob can also be confident that no one learns that he is the indented recipient of the mails.

Unfortunately it is not so easy to implement such anonymous mailboxes over the internet. Even if the messages are encrypted, the metadata which includes the identity of participants, the duration of the chat, etc, can be sensitive and potentially leaked. The mailbox service provider may disclose users' information and their interaction with the mailboxes, either to sell users' data for profit or under the coercion of spy agencies, or simply as a result of a database breach.

A private information retrieval (PIR) protocol is a protocol that allows Alice to retrieve an item from a database server without the server learning which item is retrieved. A trivial PIR scheme is Alice simply downloads the entire database.

The security of PIR schemes usually fall under those two categories: information-theoretic security and computational security. Typically computational-secure PIR schemes are more efficient than information-theoretic-secure ones. Every PIR protocol aims to reduce the amount of computation on both Alice and the database server, as well as the traffic between Alice and the server.

In *Privomega*, we need a PIR scheme that guarantees users' privacy when they **retrieve mails** from the mailboxes. It is not required to give user privacy when **putting mails**. In practice, we could deploy *Pung* [4] or a similar efficient computational PIR protocol.

### Verifiable random functions

A verifiable random function (VRF) is a pseudo-random function (PRF), where the prover holding the secret key can produce a non-interactive proof that the PRF output was correct given the input. Only the holder of the private key can compute the hash, but anyone with public key can verify the correctness of the hash.

Informally, the VRF functionalities we are interested in are:

$$(r, \pi) \leftarrow VRF(sk, \alpha)$$

That is, given a secret key $sk$ and a VRF input $\alpha$, the VRF would produce the pseudorandom output $r$ and a proof $\pi$.

$$\{0,1\} \leftarrow VRFVerify(pk, \alpha, r, \pi)$$

During verification, the verifier receives the public key $pk$, the input $\alpha$, the output $r$, and the proof $\pi$, the function returns 1 if the proof is valid, 0 if proof is invalid.

In *Privomega* we would need a VRF so when Alice matches Bob, Alice will include the VRF proof in her matchmaking request with Bob. This way, Bob can verify that Alice has picked him randomly. In practice, we could deploy EC-VRF over some curve, say Curve25519 [5].

#### Private end-to-end messaging

In *Privomega* Alice uses integrated encryption scheme to encrypt her matchmaking request to Bob. Once Bob accepts the matchmaking request, Alice and Bob derives a shared secret. From this point we can use any E2EE messaging protocol to encrypt the messages between Alice and Bob, for example Signal's Double Ratchet [6], if forward secrecy or backward secrecy is important. It could use a simple Authenticated Encryption with Associated Data (AEAD) protocol with no forward or backward secrecy, for example AES-GCM, if we assume the chat sessions are short and identities are not reusable.

## The Privomega privacy-preserving random stranger chat protocol

#### Pseudo-anonymous identity registration

Alice first generates a key pair

$$(sk_a, pk_a) \leftarrow GenKey$$

Alice registers her public key $pk_a$ on the server. Server upon receiving Alice's public key, server creates and signs a certificate for Alice, then adds her certificate to the key directory:

$$(cert_a, id_a) \leftarrow Register(pk_a)$$
$$sig \leftarrow Sign(sk_s, pk_a||timestamp)$$
$$cert_a = pk_a||timestamp||sig$$

The certificate contains Alice's ID $id_a$, the timestamp of the registration, and Alice's public key. $sig$ is server's signature over the Alice's public key and the registration timestamp.

Alice's user ID $id_a$ is not signed. The ID is used to lookup Alice's key from the key directory. It will not change Anyone would be able to retrieve Alice's certificate like this:

$$cert_a \leftarrow GetCert(keydir, id_a)$$

Where $keydir$ is the merkle tree of the key directory. Here we assume the server uses key transparency, so Alice would verify that her ID is correct, and her certificate is included in the key directory.

**Random matchmaking**

Server periodically creates and signs a random nonce, broadcasts the nonce and server's signature over the nonce, $(nonce_t, sig)$, during every epoch $t$. This random nonce can be included in the key directory under a special index - this is to allow all users to observe the same nonce, and should reasonably prevent equivocation attack when key transparency is used.

Alice is the initiator of a random chat. She retrieves the $nonce$ from server for the current epoch $t$, and generates her random number.

$$(r_a, \pi) \leftarrow VRF(sk_a, nonce_t)$$

Where $r_a$ is Alice's random number that server does not know, and $\pi$ is the proof of the VRF output.

Alice then finds the random matchmaking target:

$$id_b \leftarrow r_a \ mod \ len(keydir_t)$$

Where $keydir_t$ is the merkle tree of the key directory at epoch $t$. Because key transparency is used, all users would observe the same $keydir_t$. Here we use $id_b$ for the ID of Alice's pseudo-randomly matched user, call the user Bob.

Alice then retrieves Bob's certificate using a PIR scheme:

$$cert_b \leftarrow GetCert(keydir_t, id_b)$$

The server should not learn whose certificate Alice retrieved. (A trivial way to do this is to simply have Alice download the entire key directory at epoch

$t$) Once she gets Bob's certificate, she extracts Bob's public key $pk_b$ from the certificate, and checks if the *timestamp* is within range for the epoch $t$.

Alice then creates the plaintext matchmaking request, which includes her public key, the server nonce at epoch $t$, her VRF proof $\pi$, and the epoch index $t$.

$$req_{ab} \leftarrow pk_a||nonce_t||r_a||\pi||t$$

Alice then randomly samples a key pair, and encrypts the message using an integrated encryption scheme:

$$(sk_{ephem}, pk_{ephem}) \leftarrow GenKey$$
$$k_{req} \leftarrow DH(sk_{ephem}, pk_b)$$
$$c_{req} \leftarrow AEEncrypt(k_{req}, req_{ab})$$
$$p_{req} \leftarrow pk_{ephem}||c_{req}$$

Here *AEEncrypt* is the encryption function in an authenticated encryption scheme. Alice would later transmit $p_{req}$ to server.

**Submitting matchmaking request**

Server maintains a queue of matchmaking requests. Alice simply puts her matchmaking request into the queue. Server would learn that Alice has sent a matchmaking request to someone, but server does not learn who she has matched with.

**Verifying matchmaking request (by Bob)**

Bob is waiting on the other end for someone to match him. He polls server's matchmaking queue at short intervals.

He will attempt to decrypt every matchmaking request in the queue until he finds one $p_{req}$ that can be correctly decrypted (i.e. decryptable, and correctly authenticated under the authenticated encryption scheme):

$$(pk_{ephem}, c_{req}) \leftarrow p_{req}$$
$$k_{req} \leftarrow DH(sk_b, pk_{ephem})$$
$$req_{ab} \leftarrow AEDecrypt(k_{req}, c_{req})$$

Once Bob has decrypted the request and gets $req_{ab}$, he verifies if Alice's VRF output is correct:

$$1 \stackrel{?}{=} VRFVerify(pk_a, nonce_t, r_a, \pi)$$

Additionally Bob also needs to verify that:

- Alice's public key is included in a certificate in key directory in epoch $t$. He can use a PIR protocol to query the key server, or he downloads the entire key directory at $t$ and check for the certificate.
- Alice's certificate has the correct timestamp for epoch $t$
- Server's $nonce_t$ is the same one he sees at $t$

If everything checks out, Bob accepts Alice's matchmaking request. Otherwise Bob ignores the matchmaking request.

### Key exchange

Once Bob accepts Alice's matchmaking request, both parties derive a shared secret from their public identity keys:

$$sk_{ab} \leftarrow DH(sk_a, pk_b)$$
$$sk_{ab} \leftarrow DH(sk_b, pk_a)$$

Then the shared secret is passed into a KDF to derive 3 more keys:

$$(sid_a, sid_b, k_e) \leftarrow KDF(sk_{ab})$$

Where $sid_a$ is Alice's sending mailbox ID, i.e. Alice puts encrypted messages to Bob in a mailbox whose ID is $sid_a$. Similarly $sid_b$ is Bob's sending mailbox ID. The $k_e$ is the shared secret for E2E messaging, used to encrypt and authenticate the messages.

Note that Alice and Bob would also get an implicit receiving mailbox ID $rid$ from the key exchange respectively, which is the ID of the mailbox they will use to receive messages from the other party.

$$rid_a = sid_b$$
$$rid_b = sid_a$$

This key exchange is forgeable. It allows Alice or Bob to forge a key exchange with the other, providing **message repudiation**. However do note that the matchmaking request is not repudiable - Bob can prove to a third party that Alice has matched with him (he cannot prove Alice has actually sent him a message), therefore the participation repudiation is weak.

**Sending and receiving messages**

Alice and Bob can calculate the associated data $AD$ that contains identity information for both parties:

$$AD \leftarrow Encode(pk_a)||Encode(pk_b)$$

The message $m$ can be encrypted as:

$$c_m \leftarrow AEADEncrypt(k_e, AD, m)$$

Where $k_e$ is the E2E messaging key from key exchange. Here $AEADEncrypt$ is the encrypt function under an authenticated encryption with associated data scheme.

The message can later be decrypted by:

$$m \leftarrow AEADDecrypt(k_e, AD, c_m)$$

> It is possible to integrate with state-of-the-art E2EE protocols instead of a simple AEAD, if chat sessions are assumed to be long-lived.

To send a message as Alice, she would put her messages to Bob in mailbox with ID $sid_a$. Bob would periodically retrieve mails, using a PIR scheme, from mailbox $sid_a$. To send message as Bob, he would put his message to Alice into mailbox $sid_b$, which Alice would also periodically check for mails from, using a PIR scheme.

$$Alice : PutMail(sid_a, c_m)$$
$$Bob : PutMail(sid_b, c_m)$$

Server would learn that Alice and Bob are both sending messages to someone, and the IDs of the mailboxes they put mails in, but it cannot learn if Alice is sending messages to Bob, or if Bob is sending messages to Alice.

At this point, to chat with each other, each party would only need to store

$$AD, k_e, sid_a, sid_b$$

and they can safely delete all other keys.

## Constructions

### Integration with Double Ratchet

Use $SK = k_e$ and $AD = Encode(pk_a)||Encode(pk_b)$ and Bob's identity key $sk_b$ as Bob's initial ratchet key.

Because in *Privomega* the identity keys are not long lived, both parties can safely delete their $sk_a$ or $sk_b$ after sending the first message, users get forward secrecy from Double Ratchet.

## Potential attacks and countermeasures

### Sybil attack

- Alice sybil attack - Alice generates many keypairs to get a VRF output to match Bob. Counter with expensive key registration
- Bob sybil attack - Bob generates a lot of identities to get a higher probability of being matched.
- Server sybil attack - Server inserts lots of fake keys in key directory. Counter by requiring embedment of POW in certificate.

### Mailbox corruption

An attacker can attempt to corrupt mailboxes by putting large nonsense data into mailboxes with random ID. Targeted attacks are less likely, as the send IDs in a chat session are only known to the two participants and server.

## Future research

### Against collusion during random matchmaking

If server and Alice collude, they can deterministically choose Bob while producing a fake VRF proof to convince Bob that he has been randomly chosen by Alice. Bob will not be able to detect such attack.

### Participation repudiation

- Initiator Alice cannot deny that she has talked to Bob if Bob presents the VRF proof to a third party.
- Bob enjoys participation repudiation since he does not sign anything
- Message repudiation is always guaranteed, as the transcript can be forged by either of the participants.

## References


1. Omegle. Omegle.com https://www.omegle.com/
2. Marcela S. Melara and Aaron Blankstein and Joseph Bonneau and Edward W. Felten and Michael J. Freedman. CONIKS: Bringing Key Transparency to End Users. *IACR* https://eprint.iacr.org/2014/1004
3. Ryan Hurst and Gary Belvin. Security Through Transparency. *Google Security Blog* https://security.googleblog.com/2017/01/security-through-transparency.html
4. Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. *Microsoft Research* https://www.microsoft.com/en-us/research/wp-content/uploads/2016/10/pung-osdi16.pdf
5. Sharon Goldberg. Verifiable Random Functions (VRFs). *IETF* https://tools.ietf.org/id/draft-goldbe-vrf-00.html
6. Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm https://signal.org/docs/specifications/doubleratchet/