Privacy Preserving Contact Tracer

Jonathan Logan*

2020-03-28 (draft)

Abstract

This contact tracer design has the goal to help battle infectious diseases by notifying contact networks of positively tested persons and/or places in a privacy preserving way.

Contents

*anontrace.jonathan@cryptogroup.net

1 tl;dr: Results and Privacy Aspects

1. Low data use. Depending on tuning, the system generates 120kB, 340kb or at most 10MB traffic between each client and the backend system, per day. The bandwidth requirements are **not** dependent on the amount of contacts, or the amount of new, global notifications, or the number of users. The amount of traffic roughly corresponds with the storage required on the user's mobile phone.
2. The system can easily be implemented for mobile applications, or mobile applications and cheap dedicated wearables.
3. The calculations required are between 2304 and 207360 scalar multiplications, and the hashing and symmetric encryption of the traffic generated (see 0.), per day. On mobile phones this takes between 4 and 400 seconds . The exact amount of calculations is dependent on privacy settings of the user. Modern devices need between 1 and 100 seconds. The processing requirements are **not** dependent on the amount of contacts, or the amount of new, global notifications, or the number of users.
4. In addition, per contact address received via broadcast, one scalar multiplication and < 100 bytes of hashing and symmetric encryption is required. For sending the broadcast, no cryptography is required. This realistically allows the implementation of a dedicated, offline, bluetooth only data logger and broadcaster on cheap hardware (ESP32). Only about 100byte per contact received need to be stored. Only 32byte

per 5 minute timeslot are required for sending (though this can be dramatically optimized in such a way that an embedded device only needs a single setup and only about 64byte of storage total for all its broadcast packets for its lifetime).

5. A user cannot be tracked through his broadcasted contact addresses: They are single-use, not linkable, and allow exactly one notification to be sent without client authorization. The notification significance can be suppressed by dummy traffic.

6. The system cannot link contact-addresses to a client.

7. The system, if cooperating with an evil device, cannot identify a client.

8. No client can identify the start of a notification chain (the "infected person", target) unless it can prevent any other person from receiving its own notifications **and** prevent the target from receiving notifications from others. Since the infection chain length cannot be determined by an attacker, a notification can be triggered by the target, or one of the target's contacts. Furthermore the attacker cannot determine who from all receivers of its own contact address triggered the infection chain.

2 Introduction

Three models exist:

- Track the location of everybody, at all times, and match co-presence with infected persons centrally. Problem: A privacy nightmare.
- Track all potentially infectious persons around oneself by recording their ever-changing temporary ID, and get notified if any of them tests positive by verifying all matching IDs to one's own record. Problem: The required bandwidth quickly becomes enormous with a lot of infected people.
- Track all persons that oneself can potentially infect. The number of contacts is very limited and they only have to be contacted **if** oneself is tested positive. Problem: Requires an anonymous way of contacting. But... this is easily solveable.

2.1 Bird's-view description

1. User A registers with the app. This generates a number of contact addresses that are *blind*, *unlinkable* and *untraceable* (see below). The contact addresses are stored on the backend.
2. User A now continously broadcasts his unblinded contact addresses. One per timeslot. Each address is only broadcasted in one timeslot and never again. Broadcasting is implemented on top of Bluetooth.
3. User B uses the app as well. His device listens for broadcasted contact addresses. On receiving such an address, the app encrypts it to the global backend public key, annotates it with the time, and stores it in a log (contact-log) on the device.
4. Each installation of the app contineously removes entries from the contact-log if they are older than a configured duration.
5. In case of postitive test for an infection, user B receives a one-time token from his healthcare provider. It includes the disease code. User B then authorizes the app to upload his contact log, the backend system authenticates the upload with the one-time token.
6. The backend decrypts the contact log, and sends a unique case-ID and a one-time-token to each of the registered addresses.
7. User A periodically checks his contact addresses for new messages. When receiving a case-ID and one-time-token, the app notifies him and asks to upload his contact log as well.
8. After user authorization, the encrypted contact-log, **one** of user A's own contact addresses, case-ID and the one-time-token are uploaded to the

backend.

9. The backend verifies that the contact address was part of the genesis case, verifies the one-time-token and decrypts the contact log.

10. The extension of the notification chain can be controlled by the backend and is determined by the original case-ID.

2.2 Key concepts

The system relies on a Mix-Network, and specifically Single-Use Reply Blocks (SURB). A mix network is a set of servers (mixes) that can decrypt, reorder and forward messages among each other. A sender can use a string of data, called the Single-Use Reply Block, to send a message through a mix-network in such a way that neither the sender, nor any of the mixes, can determine who sent which message to whom. Only if all participants cooperate can this information be determined. This design makes use of a slimmed down mix-network design that is optimized for very small pieces of data, but is hence less flexible. Each mix in the network makes sure that every SURB is used exactly once, by recording its hash and refusing to process a message with the same SURB hash again. The second concept is the use of a nym-server. This server stores an encrypted SURB until it is requested and decrypted with a presented key, or overwritten by another SURB after authentication.

2.3 Cryptographic algorithms required

- A strong cryptographic hash function with at least 32byte output size. For example, SHA256.
- A strong symmetric block cipher. For example AES256 or XChaCha20.
- Optional: Poly1305 one-time message authentication code.
- An elliptic curve over a prime field. For example x25519 or secpk256r1.

2.4 Cryptographic constructs required

- HMAC
- A wide-block cipher. For example LIONESS (using the symmetric block cipher and hash from above).
- An authenticated encryption mode, like AES-GCM or XChaCha20-Poly1305.

2.5 Database requirements

- A fast key-value database.

3 Details

3.1 Components

Users: Humans that control devices that run the client application.

Clients: The client application. It broadcasts and receives contact addresses, and executes the client-side protocol. The client-side protocol can be split and distributed over multiple devices. For example, a minimal embedded device can only execute the broadcasting and receiving operations, while not implementing the registration, refresh or discovery operations.

Health-Authority: An organization and technical tool to issue one-time tokens to authorize the notification operation on the nymserver.

Nymserver: Stores encrypted SURBs and executes the notification operation on request. It is also a mix.

Mailbox-Server: Receives, stores and makes accessible messages sent to addresses. It receives messages from the mix network and makes them available to clients. Multiple mailbox servers can and should exist.

Mixes: Servers that relay messages according to the protocol. They must be operated by independent organizations. All Mailbox-Servers and Nymservers are also mixes (but not vice versa).

Discovery: Serves cryptographic public keys of the mix-servers to clients on request.

At most 255 entities can exist in the system (sum of nymservers, mailbox-servers and mixes).

3.2 Mailbox-Server

The mailbox server operates a simple key-value store. It receives messages that consist of an address and a payload. On receiving, the mailbox server appends the payload to any data already stored for the address. Furthermore it returns all data stored for an address if presented with an address. Addresses and their data may be deleted 30 days after the address was first seen on the system.

Methods:

- Receive(address, payload): If address does not exist, create address, record address+deletion data in log, store payload for address. If address exists, append payload to data stored for address.
- Serve(address): If address exists, return all data stored for it.

In addition the Mailbox-Server **also** serves as a mix.

3.3 Nymserver

The nymserver operates a key-value store. For each key it stores an expiration date, a list of one or more messages, a "used" flag, and an encrypted blob.

Methods:

- Send(key, message): The nymserver looks up hash(key) from the database, if an entry is found and the used-flag is false, use key to decrypt the blob. The blob contains the information for sending a notification (see below). If the used-flag is true, append the message to the queue of messages for that key.
- Swap(key, blob, authentication): Look up key from the database, if the key does not exist, store blob, key, and set the expire date. If the key does exist, verify that the given authentication equals the hash of the current blob, and replace the current blob with the new blob. No operation (or error) if failure.

In addition the Nymserver **also** serves as a mix.

3.4 Mixing

Mixes receive messages, check them for uniqueness, decrypt them, delay them, and then send them to the next mix as specified in the message.

A *message* consists of four fields: Header(Key Select, Key, Address) and Payload. Key: The key is a point on an elliptic curve (public key) and is used together with the Mix's private key to create a shared secret that is required for further decryption operations on the message. Length: 32 or 33 byte, depending on used curve.

Key Select: Since a Mix can have more than one private key, the key is selected by the Key Select. It is 1 byte long.

Address: The address is a string of data. It contains encrypted information and an authentication tag. The mix decrypts the address with a key derived from the shared secret and verifies its integrity. The address length depends on the length of the chain of mixes through which a message may be sent. The address is 36 bytes long, plus 18 bytes for each hop. Contained in the address is either: The Key Select and the address (number) of the next Mix, OR the address on which to store the payload on the mailbox-server. If it contains instructions to forward the message to another mix, the message is delayed by a random number of seconds and then forwarded. If it contains instructions to store it in the mailbox-server, the message is delayed by a random number of seconds and then the payload is stored for the address.

Payload: On each hop, the payload is decrypted with a key derived from the shared secret. The decryption operation uses LIONESS and a block cipher. A payload has a fixed length that is 98byte + length of the header. In this system, the payload can contain either a case-notification, OR a swap operation for

the nymserver. The meaning of the payload depends on its last hop: If the last hop is a nymserver, then the payload is interpreted as a swap operation. If it is a mailbox-server, then the payload is used in a Receive operation.

Uniqueness test: Each mix calculates and stores the hash of the Address. If a address is presented again, the message is silently dropped.

For a mix-chain of 4 hops, the header is 141 or 142 bytes long, depending on curve. A payload is 239/240 bytes long, a message is 380 or 382 bytes long. The payload can contain 238/239 bytes of transmittable data (1 byte for padding information).

3.5 Mix public keys

Each mix creates one keypair per day, 30 days into the future. Each key expires 30 days after creation. The public keys are distributed via the Discovery server. This allows the mix to rotate the keys to both limit the amount of storage required for the uniqueness test, as well as preventing messages that are too old to pass the system. Each mix independently enforces the expiration of data in the system. For security reasons, each mix should also maintain a long-term keypair that is used to sign its key list. This way an attacker cannot easily trick a client into using a mix of his choosing.

3.6 Blind and untraceable contact addresses

The client randomly selects one or more mailbox addresses. These will be used to receive notifications, and the client will poll them occasionally. To maintain privacy, the client will also replace mailbox addresses regularly. For each mailbox address, the client creates a SURB message:

- A mix header that contains the client's mailbox address as delivery target. The mixes in the chain are randomly selected. For each mix, a public key is selected that will remain valid for a sufficient period (which can be determined by the client).
- MessageKey: A random key, 32 byte.
- A contact address, 32 byte random value (or shorter).

To create a contact address, the client creates a Mix message containing a swap operation for the nymserver. It contains:

- key:=hash(contact address)
- Authentication value, for new addresses it is all zero.
- Payload, encrypted with contact address: SURB. The client then sends the

mix message through the network. As a result, the nymserver now contains a key that, if the right contact address is presented, allows the decryption of a SURB. The nymserver can then encrypt a payload using MessageKey, and send it through the mix network with the help of the SURB.

The client occasionally checks the mailbox-server for any messages received at his mailbox address. If a message is received, he can decrypt it with the help of the MessageKey and the key material created when the SURB was constructed. If the message contains a notification from the nymserver, the client must create a new SURB message for the same contact address and send it to the nym server. The nym server can only send exactly **one** message to each contact address without the clients cooperation. This prevents easy statistical tagging if combined with client-generated dummy traffic (optional, see below). Each contact address creation requires 380/382 bytes of data sent to the mix network (plus IP protocol headers).

A daily supply of contact addresses equals one contact address per 5 minutes (288 contact addresses), or 110 kB. If each contact address points to a single mailbox address, 288 mailboxes must be polled, which can be done with less then 10kB of data **if** no notifications have been received. For each received notification another 238/239 bytes of traffic is generated.

3.7 Making contact addresses available

The client broadcasts one of his contact addresses per time period (5 minutes). The receiving client encrypts the contact address with the public key of the nymserver. It also annotates the result with the current time and stores both in a log. Occasionally the client will remove entries older than 30 days from the log.

In case of notification, the log and a one-time authentication token provided by a health authority is sent to the nymserver. The nymserver can now decrypt the log and lookup up hash(contact address) in his database. If a match is found, the attached SURB can be decrypted with the contact address and used to send a notification.

The nymserver should store a list of contact addresses associated with the case-id, and send the case-id and optional data to the decrypted SURBS.

A notified client can decrypt the payloads (yielding the case-id), associate them to the contact address used, and send the case-id, one contact address, and the its own log to the nymserver to extend the notification chain. This is only done for a single contact address for maximum privacy, or for multiple addresses to transmit exposure severity data to the server.

3.8 Client-generated dummy traffic

The mailbox-server can:

1. Track the identity of a client by matching the pattern of mailbox addresses it polls.
2. Track if a user is part of an infection chain by seeing if an address has a payload.

This can be easily prevented by introducing two kinds of client-dummy traffic:

1. contact-address replacement: The client regularly updates the SURB of his contact addresses to point to new mailbox addresses. After a waiting period the client does not have to check the mailbox address again.
2. mailbox-dummies: The client regularly sends message to his own mailbox addresses. Since the payloads are encrypted, the mailbox server cannot determine which client is linked to an infection chain.

Both kinds of dummy traffic can be combined and also used to devalidate SURBs by presenting an address at one or more mixes used by previously generated SURBs. This finally breaks the link between contact address and mailbox address. Assuming that a client devalidates both address types every 24h, an additional 330kB of data is required.

Ideal update scheme:

1. Replace SURB in contact-address to point to a new mailbox address.
2. Wait until all messages in flight must have arrived (this is a few hours).
3. Send dummy message to mailbox that intersects at one or more mixes with same address. This can have two effects: The original SURB was used for a notification, and the dummy is discarded, or there was no notification, the dummy is delivered, and any later notification would be discarded (hence step 1+2).
4. Wait until all messages in flight must have arrived.
5. Check mailbox for notifications, contains either:
    1. 1 message that is a dummy.
    2. 1 message that is a notification, in which case any follow-up notification will be sent to the new mailbox address.