ROBERT: ROBust and privacy-presERving proximity Tracing

PRIVATICS team, Inria, France
Fraunhofer AISEC, Germany

May 31, 2020
v1.1

The main goal of proximity tracing mobile applications is to notify people that they have been in close proximity of COVID-19 virus carriers in the last X days even though the carriers who did not have symptoms and were not even tested at the time of interaction. Ensuring the highest data protection and security standards will encourage a quick and broad adoption by citizens.

Any proposed solution should preserve user privacy, but should also be robust against attacks that aim at decreasing the performance or reliability of the system. A system that does not protect user privacy will not be accepted neither by the citizens nor by the Data Protection Authorities. A system that does not provide reliable results will just be useless. The challenge is to build a solution that is privacy-preserving and robust against malicious users and "honest-but-curious" authorities at the same time.

Although it might seem attractive in term of privacy to adopt a fully decentralized solution, such approaches face important challenges in term of security and robustness against malicious users [7]. This document proposes a ROBust and privacy-presERving proximity Tracing (*ROBERT*) scheme that relies on a federated server infrastructure and temporary anonymous identifiers with strong security and privacy guarantees.

The *ROBERT* scheme is the result of a collaborative work between Inria and Fraunhofer AISEC, and a candidate proposal for the Pan European Privacy-Preserving Proximity Tracing (PEPP-PT) initiative. This proposal is not final and is subject to discussions, modifications and improvements.

1. Introduction

The COVID-19 virus is hard to trace because many people can be carriers – and hence contagious – without knowing it and before experiencing any symptoms. Therefore, COVID-19 virus carriers may transmit it to many other people in their vicinity. Successfully containing the COVID-19 pandemic closely depends on the ability to quickly and reliably identify those who have been in close proximity to a contagious individual. In this context, a mobile application that uses a short-range communication in Bluetooth (that is available in all modern smartphones) is able to detect when two individuals are in close proximity. Such applications can later allow users to know that they have been exposed to COVID-19 virus carriers. Such an alerting system would be very beneficial to reduce the infection rate [4, 5]. Ensuring the highest data protection and security standards will encourage fast and a broad adoption from citizens. This work presents *ROBERT*, a robust and privacy preserving proximity tracing scheme.

1.1. Why is Proximity Tracing Useful?

The main goal of proximity tracing mobile applications is to notify people that they have been in close proximity to COVID-19 virus carriers.

We use the following terminology:

- A user is *infected* if she is a carrier of the COVID-19 virus.
- A user is *diagnosed* if she has been tested and diagnosed COVID-positive.

- An infected user can be contagious without knowing it and before experiencing any symptoms. We call the *contagious period* (CT) an upper-bound on the number of days a user that has been diagnosed positive has been contagious[1].
- A user is qualified as "*at risk of exposure*" if she has been in the vicinity of a certain number of infected user(s) during the past CT days. Further criteria such as proximity to the infected user or the duration of a contact can be set by the health authority and can be used to parameterize the risk scoring function.

Figure 1 illustrates an example where an infected user C transmits it to user A before being tested and isolated. User A infects in turn user B, who infects D.

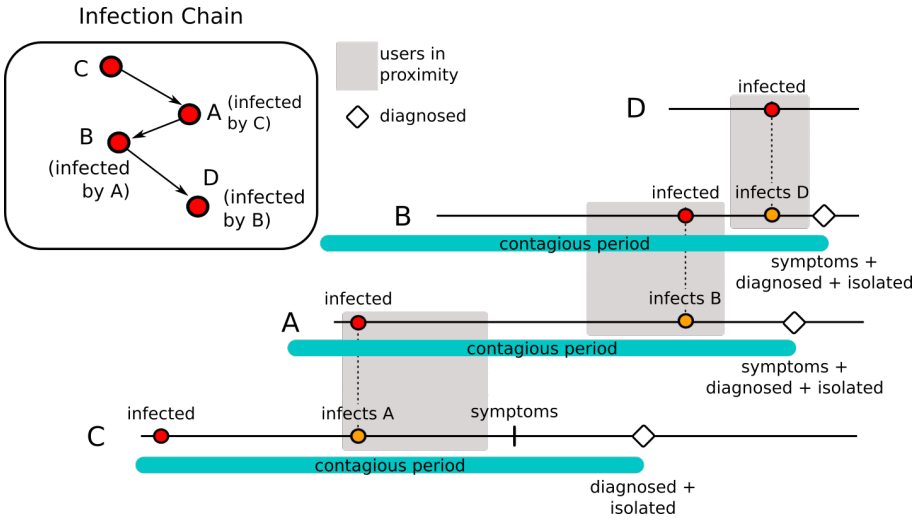

Figure 1: Example of the infection process without proximity tracing App.

Figure 2 illustrates the effect (and benefit) of proximity tracing. C is infected and diagnosed COVID-positive. User A gets notified and become "at risk" users. A gets tested and is diagnosed COVID-positive. B is in turn notified, becomes "at risk" and is confined before meeting D. Consequently, B and D do not get exposed, and therefore infected, anymore.

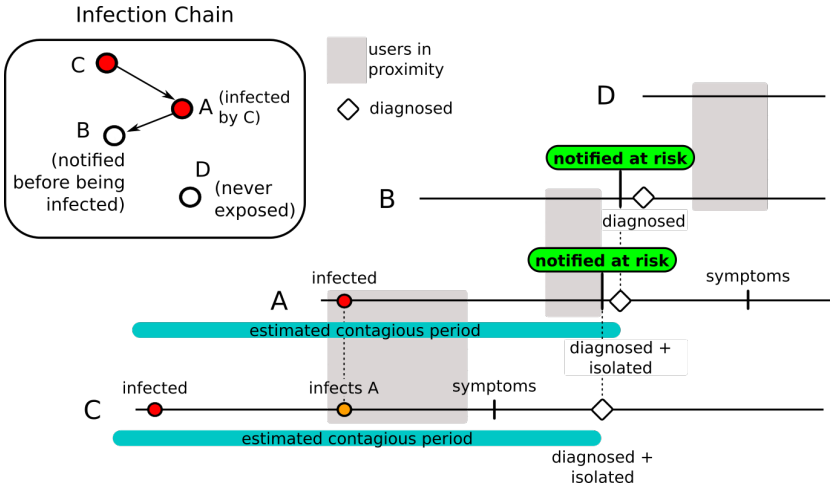

Figure 2: Example of the infection process with proximity tracing App that starts with person C being infected.

1.2. Design Goals

Our scheme has been designed to reliably notify individuals of past collocation with an infected person while minimizing the impact on the individuals' privacy. Our scheme provides the following goals as detailed in [3]:

[1]For simplicity, we consider that CT is fixed and the same for all users. In practice, this value could be adapted to each infected user by doctors.

- **Open participation.** Participants are free to join or leave the system at any time.
- **Simple and transparent.** The system is simple to use and understand.
- **Easy deployment.** The scheme is easy to deploy and requires only minimal infrastructure.
- **Anonymity.** The smartphone App as well as the back-end server database do not collect or store any personal data.
- **Federated infrastructure.** The system must scale across countries, ideally worldwide. In order to preserve countries' sovereignty, a trusted federation of infrastructures is necessary.

1.3. Security and Privacy Requirements

Before describing our proximity tracing scheme, we highlight the main security and privacy considerations (these considerations are extracted from Section 3 of [3]).

- **Accuracy and reliability of proximity data.** Though users are free to join or leave the system, the scheme should not miss an exposure to an infected user. Moreover, the scheme should be robust against adversarial attacks from both the users or the database server.
- **Anonymity of users from other users.** The proposed scheme should guarantee that no user can learn any information that cannot be inferred just from knowing whether they collocated with another user (diagnosed as COVID-positive or not). Similarly, any colluding users should learn only what can be deduced from the individual information that each colluding member is entitled to know.
- **Anonymity of users from a central authority.** The central authority should not be able to learn information about the identities or locations of the participating users, whether diagnosed as COVID-positive or not.

1.4. Adversarial Model

We assume the following adversarial model:

- **Users** can be **malicious**. They can, for example, perform active and passive attacks, eavesdrop, inject bogus messages, modify messages, pollute users' contact lists and develop their own applications.
- The **authority** running the system, in turn, is **"honest-but-curious"**. Specifically, it will not deploy spying devices or will not modify the protocols and the messages. However, it might use collected information for other purposes such as to re-identify users or to infer their contact graphs. We assume the back-end system is secure, and regularly audited and controlled by external trusted and neutral authorities (such as Data Protection Authorities and National Cybersecurity Agencies).

1.5. Design constraints

To ensure compatibility between different mobile operating systems and due to restrictions in the Bluetooth advertisement, the application can only broadcast 16-byte messages (128 bits) over the device Bluetooth interface.

2. High-Level Description and Assumptions

2.1. Architecture considerations

Our proximity tracing scheme relies on an App installed on each mobile phone, supported by a back-end server. In a such distributed architecture, two considerations are important (1) where the data are stored and (2) where the status of the user (at risk or not) is verified. In our scheme, the data to be stored is shared between the App and back-end server. The data collection of proximity contacts is performed and stored locally on each App. This proximity contacts are never revealed to the server except when a user is diagnosed COVID-positive. In this specific case, upon agreement from the user and authorisation from the health authority, the App shares, anonymously, with the server the proximity contacts that it has collected during the estimated contagious period, typically the last 14 days. This data is used by the back-end server to compute an exposure risk score for each of the individuals, defined as anonymous identifiers, who have been in contact with this infected user. Users periodically probe the server to know whether their risk score indicates that they are at risk. As a result, users only learn one bit of information from the server ("at risk" or not "at risk"). They don't get any information

about other users and, in particular, they don't learn who potentially exposed them. The back-end server only maintains the list of exposed users (through anonymous pseudonyms as no personal information are maintained on the server) with their risk scores. These risk scores can easily be adapted according the evolution of the pandemic or the evolving knowledge of the epidemiologists on the COVID-19 virus.

Other, qualified as "decentralised", schemes broadcast to each App an aggregate information containing the pseudonyms of all the infected users [1]. This information allow each App to decode the identifiers of infected users and verify if any of them are part of its contact list. Our scheme does not follow this principle because we believe that sending information about all infected users reveals too much information. In fact, it has been shown that this information can be easily used by malicious users to re-identify infected users at scale [7]. We claim that infected user re-identification must absolutely be avoided since it could lead to stigmatisation. Instead, we chose to securely store this information on a central server.

Although *ROBERT* is proposed as a "proximity-tracing" protocol, *ROBERT* is actually a framework to assess the risk exposure of its users in order to fight pandemics. In our proposal, and as opposed to decentralized schemes, users do not get any information about the status of their contacts. In particular, they do not learn how many of their contacts are infected, nor which of them are. Instead, users get informed about their exposure level only upon the computation of a risk score by the server. The risk score may be based on proximity information, but also on other parameters that epidemiologists will define and adapt according to the evolution of the pandemic. Finally, the risk score algorithm will also depend on how the App will actually be used by the health authority. In fact, the algorithm will probably be different if the App is used to notify users that need to get tested in priority or if it is used to decide who should stay/go into confinement.

2.2. System Overview

All the notations used in this paper are summarized in Table 1. The proposed system is composed of users who install the Proximity Tracing Application, *App*, and a back-end server under the control of the authority. We assume that the back-end server is configured with a well-known domain name, certificate and is highly secured.

Apps interact with the system through the four following procedures:

- **Initialization:** When a user wants to use the service, she installs the application, *App*, from an official App store (Apple or Google). *App* then registers to the server that generates a permanent identifier (ID) and several Ephemeral Bluetooth Identifiers ($EBIDs$). The back-end maintains a table, *IDTable*, that keeps an entry for each registered ID. The stored information are "anonymous" and, by no mean, associated to a particular user (no personal information is stored in *IDTable*).

- **Proximity Discovery:** After registering to the service, *App* broadcasts *HELLO* messages over its Bluetooth interface and collects *HELLO* messages from other devices, running the same application[2], in the vicinity. These *HELLO* contain several fields, and in particular, an Ephemeral Bluetooth Identifier. The collected *HELLO* messages are stored, together with the time of reception (and possibly other information such as the strength of the Bluetooth signal or the user's speed) into a local list of the application, the *LocalProximityList*.

- **Infected User Declaration:** When an individual is tested and *diagnosed* COVID-positive, and after an explicit user consent and authorisation (from the medical services), her smartphone's application uploads its *LocalProximityList* to the authority server, Srv. Srv then flags as "exposed" all IDs of *IDTable* of which at least one $EBID$ appears in the uploaded *LocalProximityList*. It is important to note that:
  1. The server does not learn the identifiers of the *infected* user's *App* but only the $EBIDs$ contained in its *LocalProximityList* (list of Ephemeral Bluetooth Identifiers she was in proximity with).
  2. Given any two random identifiers of *IDTable* that are flagged as "exposed", the server Srv can not tell whether they appeared in the same or in different *LocalProximityList* lists (the proximity links between identifiers are not kept and, therefore, no proximity graph can be built).

- **Exposure Status Request:** *App* queries (pull mechanism) the "exposure status" of its user by probing regularly the server with its $EBIDs$. The server then checks how many times the *App*'s $EBIDs$ were flagged as "exposed" and computes a risk score from this information (and possibly other parameters, such the exposure duration or the user's speed/acceleration during the contact). If this score is larger than a given threshold, the bit "1" ("at risk of exposure") is sent back to the *App* and her account is deactivated, otherwise the bit "0" is sent back. Upon reception of this message, a notification is displayed to the user that indicates the instructions to follow (e.g., go the hospital for a test, call a specific phone number, stay in quarantine, etc.).

[2]Federation is also considered for users who are traveling abroad. See Section 8.

Time-related Assumptions: This *ROBERT* protocol assumes that all the smartphones and the server *Srv* are loosely time-synchronized (thanks to NTP or any other time synchronisation mechanism like cellular mobile phone network information, or GPS time information, etc.). Time is expressed as the NTP "Seconds" value, which represents, for era 0, the number of seconds since 0h January 1st, 1900 UTC [3].

Time is discretized into epochs (e.g., of 15 minutes)[4]. We define as *epoch_duration_sec* the duration of an epoch in seconds.

2.3. Risk Scoring Considerations

Specific and effective risk scoring is out of scope of this paper. We assume, in this proposal, that (1) the server returns a binary reply to its users if they are at risk or not and (2) the reply is only based on a calculated risk score value. These two assumptions need to be discussed. It might be useful, for several reasons, to return a probability value instead of a binary reply. Furthermore, adding some randomness in the query reply mechanism could improve privacy (see Section A).

We further assume that it is the responsibility of the health authorities and epidemiologists to provide input to such an algorithm. This algorithm should be public and may be adapted over time. Modifications and actionable advice based on the risk scoring must be done in accordance with epidemiology research.

In our design, scores are computed on a trusted server and are used to notify users. While this offers more flexibility to adapt the scoring algorithms as needed and leads to more effective systems, it also increases the resilience of the systems against attackers that aim at identifying infected users: In order to be notified, an attacker must inject his own *HELLO* messages into a victim's *App* in such a way that the risk scoring algorithm in the back-end selects him for notification. This makes such an attack more difficult as it requires an attacker to use invasive tools or put himself at risk, consequently reducing the scalability of such an attack. We consider especially the latter property to be a rather strong deterrent.

In contrast, processing the risk of a contact on the phone upon reception of a notification inherently reduces the system to tracing, even for very brief encounters, between users and infected. This has major implications on privacy as using contextual metadata makes it trivial to identify infected users [7]. The system's design would be based on the fact that all users which at any point ever saw an infected user's *HELLO* can now use contextual metadata (such as a meeting date and time) to identify the infected users.

3. Initialisation

This section describes the initialisation process on the server and the application.

3.1. Server Set Up

The back-end server is initialized at the beginning of the proximity tracing official period under the control of the Authority. In order to be able to determine in which period and epoch the system is, the server stores $T_{ptsstart}$, the time when the proximity tracing service has been started in the country. From this timestamp, the server maintains a counter i, initialized to 0, representing the current epoch number coded by 24 bits.

We also assume that the server is configured with:

- **-K_S ("Server Key"):** a L-bit long key, with $L >= 128$ to be defined, only known by the server[5].
- **-K_G ("Federation Key"):** a L-bit long key, with $L >= 128$ to be defined, shared between all servers in Europe (used for federation, see Section 8).
- **-($\mathsf{sk}_S, \mathsf{pk}_S$) ("Registration key-pair") :** an asymmetric-key pair, whose private key sk_S is known only from the server, and public key part is known to every App. This key-pair is defined over the elliptic curve NIST-P256, with $pk_S = sk_S.G$, where G is the base point of prime order on the curve defined by the specifications[6].
- **-CC_S ("Country Code"):** a 8-bit long value that codes the country where the server provides service (used for federation, see Section 8). These country codes should be known to all international systems where federation is possible. For example, France could use the code "33", Germany the code "49", ...

[3] `https://en.wikipedia.org/wiki/Network_Time_Protocol`

[4] This value of 15 minutes is the rotation period of random address recommended in the Bluetooth v5.1 specification [6, Vol 3, Part C, App. A]). We assume that these epochs and rotation periods are synchronized.

[5] Note that this key can be renewed every few epochs for better security. In this case, the server needs to store all the keys that were generated during the last CTK epochs, where $CTK = 86400 * CT/epoch\_duration\_sec$.

[6] See Annex D.1 of NIST FIPS186-4, `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`

Note that the server relies on a security module to store and manipulate its secrets and the secrets shared with mobile applications.

In addition, the server maintains a local database, *IDTable* (see Section 3.2 for details).

3.2. Application Registration (Server Side)

3.2.1. The $IDTable$ database

For each registered application App_A, belonging to an anonymous user U_A, the server Srv keeps a record in a local database *IDTable* with the following information:

-**K_A^{auth} ("Authentication Key for A"):** a L-bit long key, with $L \geq 128$ to be defined, shared with App_A, that is used to authenticate App_A messages.

-**K_A^{enc} ("Encryption Key for A"):** a L-bit long key, with $L \geq 128$ to be defined, shared with App_A, that is used to encrypt sensitive information sent from the server to A.

-**ID_A ("Permanent IDentifier for A"):** a 40-bit identifier that is unique for each registered App, and generated randomly (random drawing process without replacement to avoid collisions). This permanent ID is only known to the server.

-**UN_A ("User A Notified"):** this flag indicates if the associated user has already been notified to be at risk of exposure ("true") or not ("false"). It is initialized with value "false". Once set to "true", the user is not allowed to make any additional status request. The flag can be reset if the user can prove that she is not at risk anymore (for example by proving that she got a test and the result was negative).

-**SRE_A ("Status Request Epoch"):** a 24-bit value that indicates the last epoch when U_A has sent a "Status Request".

-**LEE_A ("List of Exposed Epoch(s)"):** each time one of A's EBID appears in the proximity list of an infected user, the epoch j when the encounter happened between the infected user and U_A is added to this list. This list of epochs reflects the exposure of the user (temporal and frequency information) and is used, together with other information (e.g. the duration and proximity of a contact to an infected person), to compute the risk score. Note that a given epoch i can appear several times in this list if several *HELLO* messages sent by App_A appeared in the proximity list(s) of one of several infected users during that epoch.

3.3. Application Registration (Application Side)

A user U_A who wants to install the application on his device must download it from the official Apple or Android stores. After installing the application App_A, U_A needs to register to the back-end server. During this registration phase:

- A proof-of-work (PoW) system (like a CAPCHA) is implemented in order to avoid automatic registrations by bots or denial of service attacks (the details of this PoW system are out of scope of this document).
- App_A, the application installed on U_A's device, is configured (over a TLS channel) with:
  - The current epoch value, i.
  - The duration of an epoch, $epoch\_duration\_sec$.
  - The starting time of the following epoch, $T_{ptsstart} + (i+1) * epoch\_duration\_sec$ (This is required for synchronization with the server).
  - The keys K_A^{auth} and K_A^{enc}, shared with the server.
  - An initial list of T $(EBID_{A,i}, ECC_{A,i})$ pairs (see Section 4 for more details).

The keys K_A^{auth} and K_A^{enc} are exchanged by means of a key establishment procedure:

- App_A generates an ephemeral key pair $(ske_A, pke_A = ske_A.G)$ and transmits the value pke_A to the server in its registration message
- A shared secret $SharedSecret$, the x-coordinate of the point $(ske_A.sk_S).G$, is obtained by App_A and the server as follows:
  - App_A computes $ske_A.pk_S$

- The server computes $sk_S.pke_A$

- App_A and the server derive K_A^{auth} as $K_A^{auth} = HMAC\_SHA256(SharedSecret, \text{"authentication key"})$

- App_A and the server derive K_A^{enc} as $K_A^{enc} = HMAC\_SHA256(SharedSecret, \text{"encryption key"})$

4. Generation of the Ephemeral Bluetooth Identifiers

During registration and then regularly, i.e. every M epochs (value to be defined), each registered user U_A connects to the server in order to obtain a list of the M $(EBID_{A,i}, ECC_{A,i})$ pairs for the M following epochs. For efficiency, this request can be performed together with an *Exposure Status Request* (see Section 6.2).

Upon receiving such a request, the server generates and sends to App_A an encrypted list of T $(EBID_{A,i}, ECC_{A,i})$ pairs for the upcoming T epochs[7], where:

-EBID$_{A,i}$ ("Ephemeral Bluetooth IDentifier for A"): a 64-bit identifier generated for the epoch i as:

$$EBID_{A,i} = ENC(K_S, i \mid ID_A)$$

where ENC is a 64-bit block cipher, for example, $SKINNY-64/192$ (a block cipher of block size 64 bits and key size 192 bits) [2].

-ECC$_{A,i}$ ("Encrypted Country Code"): an 8-bit code that indicates, in an encrypted form, the country code of $EBID_{A,i}$. This field is used for federation purposes and can only be decrypted by back-end servers that have federation agreements. More specifically. $ECC_{A,i}$ is the country code CC_A encrypted using $AES-OFB$ with key K_G and IV $EBID_{A,i}$, i.e.,

$$ECC_{A,i} = MSB(AES(K_G, EBID_{A,i} \mid 0^{64})) \oplus CC_A$$

The encryption of the list is performed with the authenticated encryption algorithm $AES-GCM$, using key K_A^{enc} with a random 96-bit IV and a 128-bit tag.

5. Proximity Discovery

In this phase, App_A performs two operations simultaneously:

- ***HELLO* Message Broadcasting** and,

- ***HELLO* Message Collection**.

5.1. HELLO Message Broadcasting

App_A continuously broadcasts *HELLO* messages on its Bluetooth Low Energy interface. These *HELLO* messages are composed of 4 different fields and have a total length of 128 bits (16 bytes) (see Figure 3). The *HELLO* messages broadcast at epoch i, $HELLO_{A,i}$, are defined as:

$$M_{A,i} = [ECC_{A,i} \mid EBID_{A,i} \mid Time]$$

$$HELLO_{A,i} = [M_{A,i} \mid \text{MAC}_{A,i}]$$

where:

-ECC$_{A,i}$: the encrypted country code (8 bits).

-EBID$_{A,i}$: Ephemeral Bluetooth ID at epoch i, generated by the back-end (64 bits).

-Time: 16-bit timestamp (to encode the fine-grain emission time). It contains the 16 less significant bits of the current NTP "Seconds" timestamp of App_A (which represents, for era 0, the number of seconds since 0h January 1st, 1900 UTC). Since it is truncated to 16 bits, it covers a bit more than 18 hours, what is much larger than the epoch duration. This field is used to mitigate replay attacks.

-MAC$_{A,i}$: a $HMAC-SHA256(K_A^{auth}, c_1 \mid M_{A,i})$ truncated to 40 bits (c_1 is the 8-bit prefix "01"). This field is used to prevent integrity attacks on the *HELLO* messages.

Note: The *HELLO* contains a "country code" that is encrypted and is therefore not visible by other devices. It is used in case of federation by the foreign back-end servers (see Section 8 for more details).

[7]Note that the server does not store these pairs.

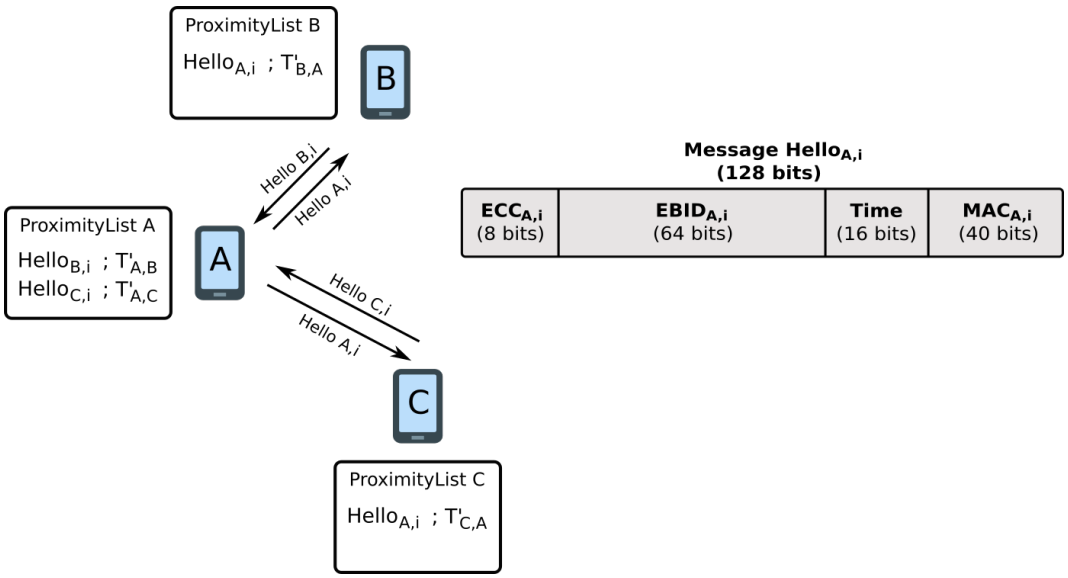

Figure 3: Proximity Discovery: the App continuously broadcasts *HELLO* messages and collects *HELLO* messages sent by nearby devices running the same application.

5.2. HELLO Message Collection

App_A continuously collects *HELLO* messages sent by nearby devices running the same application.

Upon receiving $HELLO_{A,i}$, App_A:

1. parses[8] $HELLO_{A,i}$ to retrieve ecc_A (8 bits), $ebid_A$ (64 bits), $time_A$ (16 bits) and mac_A (40 bits).
2. obtains a 32-bit NTP "Seconds" timestamp, $time'_A$.
3. checks that:
$$|time_A - TRUNC_{16}(time'_A)| < \delta$$
where δ is a configurable time tolerance (typically a few seconds, value to be defined)[9].
4. stores, in its *LocalProximityList* the following pair[10]:
$$(HELLO_{A,i}, time'_A)$$

Note: Entries in *LocalProximityList* are automatically deleted after CT Days (the value CT needs to be defined with the health authority).

6. Infected User Declaration

6.1. Upload by the Application

Upload Authorization Procedure: If user U_C is tested and diagnosed COVID-positive at a hospital or medical office and it is estimated that she was contagious from $ContStart_C$ to $ContEnd_C$ (expressed in seconds using the standard NTP "Seconds" system), she is proposed to upload to the server each $(HELLO, Time)$ pair of her *LocalProximityList* that satisfies:

$$ContStart_C < Time < ContEnd_C$$

In this document, we do not detail the interactions between App_C and the health authority. In particular, we do not present the security/authorization procedure that verifies that only authorized and positively-tested users are allowed to upload their *LocalProximityList*[11] (note that during this upload, App_C does not reveal any of its $EBIDs$ to the server).

[8] In this paper, we use lowercase for variables that result from a parsing operation.

[9] Note that since $time_A$ is only 16-bit long, this check is not enough to detect replay attacks of *HELLO* messages that were generated more than 2^{16} seconds (i.e., a bit more than 18 hours) before $time'_A$. In case U_A is diagnosed positive on COVID-19, an additional test is performed by the server (Section 7) to further detect these attacks.

[10] Other information that could be useful to compute the risk score, such as the message's reception power or the user's speed/acceleration during the contact, could be added.

[11] One solution under study is to consider that the user obtains an authorization token from the hospital or the medical office when it is diagnosed COVID-positive. The User can then use this authorization token to validate its *LocalProximityList* upload.

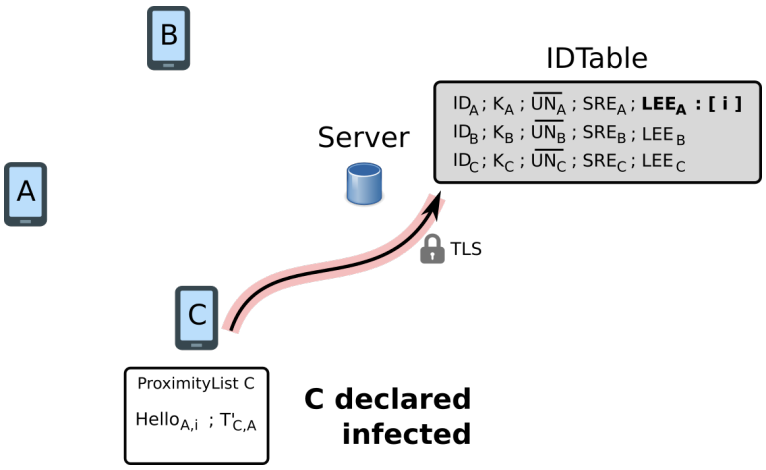

Figure 4: Infected Node Declaration: upon agreement, a user tested COVID-positive uploads her *LocalProximityList* to the server.

Upload Mechanism: A *LocalProximityList* contains the $EBIDs$ of the devices that the infected user has encountered in the last CT days. This information together with the timing information associated with each $HELLO$ message could be used to build the de-identified social/proximity graph of the infected user. The aggregation of many such social/proximity graphs may lead, under some conditions, to the de-anonymization of its nodes, which results in the social graphs of the users.

Would that be a concern, it is necessary to "break" the link between any two $EBIDs$ contained in the *LocalProximityList* to prevent the server from getting get this information. Therefore, instead of uploading the *LocalProximityList* our scheme uploads each of its elements independently.

Different solutions can be envisioned to achieve this goal:

- The $(HELLO, Time)$ pairs of the *LocalProximityList* are sent to the server one by one using a $Mixnet$[12]. Upon reception of these messages, the server won't be able to associate them with a *LocalProximityList* if the upload is spread over a long period of time.
- The *LocalProximityList* is uploaded on a trusted server (for example at a hospital or health organization) that mixes all the $(HELLO, Time)$ pairs of all infected users' *LocalProximityList*. The back-end server has only access to the exposed entries via a specific API provided by the trusted server.
- The back-end server is equipped with some secure hardware component that processes the uploads of the *LocalProximityList*. The back-end server has only access to the exposed entries via a specific API provided by the secure hardware module.

6.2. Server Operations

Upon reception of a $h_A = (HELLO_A, Time_A)$, the server:

1. parses h_A to retrieve ecc_A (8 bits), $ebid_A$ (64 bits), t_A (16 bits), mac_A (40 bits) and $Time_A$ (32 bits).
2. checks that:
$$|t_A - TRUNC_{16}(Time_A)| < \delta$$
where δ is a configurable time tolerance (typically a few seconds, value to be defined). h_A is rejected if this test is not satisfied.
3. decrypts ecc_A, using K_G, to recover the message country code, cc_A. If cc_A is different from the server's country code, CC_S, and corresponds to a valid country code, h_A is forwarded to the back-end server managing this country using the federation procedure in place (See Section 8). Otherwise it continues.
4. computes $ENC^{-1}(K_S, ebid_A)$ to retrieve $i_A \mid id_A$, where i_A is the epoch number corresponding to $ebid_A$ and id_A is the permanent identifier[13].

[12]Since all mobile telecom operators are using NAT, it should be studied whether the use of a Mixnet or proxy is really needed.

[13]In the case where K_S renewal is implemented, K_S is selected based on $Time_A$. In case the procedure fails, and if $Time_A$ is close to a boundary between the validity period of two K_S values, the step 4 is performed again with the previous/next value of K_S .

5. verifies that id_A corresponds to the ID of a registered user, otherwise h_A is rejected silently[14].
6. checks that $Time_A$ corresponds to epoch i_A[15]:

$$|(Time_A - T_{tpsstart})/\text{epoch\_duration\_sec} - i_A| \leq 1$$

(note that a difference of 1 may happen if the *HELLO* message is sent at the very end of the epoch due to transmission and processing times). h_A is rejected if this test is not satisfied.

7. retrieves from *IDTable*, K_A, the key associated with id_A.
8. verifies if the MAC, mac_A, is valid as follows:

$$mac_A == HMAC - SHA256(K_A, c_1 \mid ecc_A \mid ebid_A \mid t_A)$$

If mac_A is invalid, h_A is rejected silently[16].

9. adds i_A in LEE_A[17].
10. erases $(HELLO_A, Time_A)$.

7. Exposure Status Request (ESR)

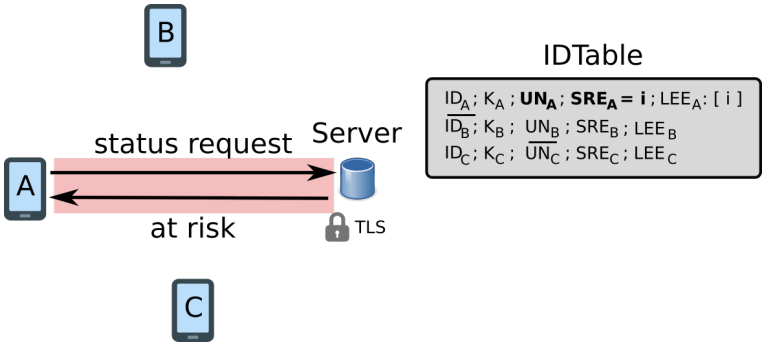

Figure 5: Exposure Status Request: the App regularly requests the server to know if any of its EBID appeared in any *HELLO* messages collected by an infected users.

In order to check whether user U_A is "at risk", i.e. if she has encountered infected and contagious users in the last CT days, application App_A regularly[18] sends "Exposure Status" Requests ($ESR\_REQUEST$) to the server Srv for ID_A. The server then computes a "risk score" value, derived in part from the list LEE_A corresponding to ID_A. The server replies with a ESR_REPLY message that is set to "1" when the user is "at risk" (i.e. if the "risk score" is larger than a threshold value) or to "0" otherwise.

Application processing: App_A queries the server by sending the following request over an TLS channel:

$$ESR\_REQUEST_{A,i} = [EBID_{A,i} \mid i \mid Time \mid MAC_A]$$

with

$$MAC_{A,i} = HMAC - SHA256(K_A^{auth}, c2 \mid EBID_{A,i} \mid i \mid Time)$$

where:

- c_2: Fixed prefix "02" (8 bits).
- $EBID_{A,i}$: Ephemeral Bluetooth ID of the epoch i (64 bits).
- i: epoch of validity of $EBID_{A,i}$, either the current epoch if App_A has an $EBID$ for the current epoch, or the latest epoch for which App_A has an $EBID$ (24 bits).
- $Time$: 32-bit timestamp in seconds, corresponding to the transmission time.

[14] In the case where two K_S values are possible (see footnote 13) and h_A is rejected, the other key should be tested in Step 4.
[15] In the case where two K_S values are possible (see footnote 13) and the following test fails, goto step 4 to test the other key value.
[16] In the case where two K_S values are possible (see footnote 13) and mac_A is incorrect, goto step 4 to test the other key value.
[17] Note that at each epoch i, the server cleans up the LEE_A list of each registered ID_A by removing the "expired" entries. More precisely, all epochs j of LEE_A such that $(i - j) >= (ct * 24 * 3600/epoch\_duration\_sec)$ are deleted.
[18] The queries are sent regularly and at most every T epochs. If a user is allowed to perform N queries per day, T is defined as $T = 86400/(N * epoch\_duration\_sec)$.

Server processing: Upon the reception of a request, $ESR\_REQUEST_{A,i}$, at epoch i, the server:

1. parses $ESR\_REQUEST_{A,i}$ to retrieve $ebid_A$, i_A, $time_A$ and mac_A.
2. verifies that $time_A$ is close to its current time.
3. retrieves the permanent identifier id_A and epoch i'_A by decrypting $ebid_A$, as
$$i'_A \mid id_A = ENC^{-1}(K_S, ebid_A)$$
4. verifies that $i_A == i'_A$, otherwise the message is rejected.
5. uses id_A to retrieve from *IDTable* the associated entries: K_A^{auth}, UN_A, SRE_A, LEE_A.
6. verifies that
$$mac_{A,i} == HMAC - SHA256(K_A^{auth}, c2 \mid ebid_A \mid i_A \mid t_A)$$
If mac_A is incorrect, the message is silently rejected;
7. verifies that UN_A is "$false$" (User Notified) in order to check that the user has not already received a "At Risk" notification. If the user has already been notified, the message is silently rejected.
8. verifies that $(i - SRE_A) \geq T$, where T is minimum number of epochs between two consecutive $ESR\_REQUEST$. Otherwise, the message is silently rejected.

If the $ESR\_REQUEST_{A,i}$ is valid, the server:

1. updates SRE_A with the current epoch number, i, in *IDTable*.
2. computes a "risk score" value, derived in part from the list LEE_A.
3. Two situations are then possible:
   - If the computed score indicates that the user is at risk of exposure, the server sets UN_A at "$true$". It means that App App_A cannot perform any new request until user U_A is tested and her status updated. An $ESR\_REPLY_{A,i}$ message set to "1" (at risk of exposure) is then returned to the user.
   - If the computed score does not indicate any significant risk, an $ESR\_REPLY_{A,i}$ message set to "0" is returned to the user.

Application processing: Upon receiving the $ESR\_REPLY_{A,i}$ message:

1. App_A replies with an ACK message.
2. If $ESR\_REPLY_{A,i}$ is set to "1":
   - App_A keeps broadcasting *HELLO* messages but stops sending $ESR\_REQUEST$ requests to the server.
   - User, U_A, receives a notification from App_A with some instructions (for example to go to the hospital to get tested, to call a specific number or go on quarantine).

8. Federation

In order for our system to be an effective tool, it must operate across neighboring states. This is especially true in the case of Europe, where freedom of movement is a core value.

We therefore propose the use of a distributed, federated architecture where countries may operate their own back-ends and develop their own Apps. This is also practical as we can expect each instance to require country-specific options with respect to health system integration and localization.

A detailed specification of a federation protocol remains to be defined in agreement with all other partners of the PEPP-PT initiative[19]. However, the proposed structure of the *HELLO* message allows some elegant solutions. We propose a standard format for all *HELLO* messages as follows[20]:

$$ECC_{Country}(8bits) \mid EBID(64bits) \mid t(16bits) \mid MAC(40bits)$$

[19] `https://www.pepp-pt.org`

[20] Actually the adopting countries should only agree to use the 8 first bits of the *HELLO* message as the "Encrypted Country Code" that is encrypted using the following 64 bits as an IV of a stream cipher, as detailed in Section 4.

Given this simple definition, a User *Bernard* from a given country, let's say France, could visit another country, let's say Germany, and still be able to use his national application. The protocol works as follows:

- When *Bernard* goes to Germany, his App broadcasts, at each epoch j, $HELLO_{FR,j}$ messages as defined in Section 4.
- When *Bernard* meets a German User, let's say Uta, at epoch i:
  - Uta stores the ($HELLO_{FR,i}$, $time$) pair in her *LocalProximityList*.
  - *Bernard* stores the ($HELLO_{DE,i}$,$time'$) pair in his *LocalProximityList* (where $HELLO_{DE,i}$ is the $HELLO$ message broadcast by Uta at epoch i).
- If Uta is later tested and diagnosed COVID-positive:
  - Uta uploads her *LocalProximityList* on the German back-end server.
  - The German back-end server obtains the ($HELLO_{FR,i}$,$time$) pair and processes it as follows:
    - It parses $HELLO_{FR,i}$ to retrieve ecc_{FR} (8 bits), $ebid_X$ (64 bits), $time'_X$ (16 bits) and mac_X (40 bits).
    - decrypts ecc_{FR}, using K_G, to recover the message country code, CC_{FR}. Since CC_{FR} is the country code for $France$, the ($HELLO_{FR,i}$,$time$) pair is forwarded to the French back-end server.
    - The French server processes it as described in Section 6.
- Similarly, if *Bernard* is later tested and diagnosed COVID-positive in France:
  - *Bernard* uploads his *LocalProximityList* on the French back-end server.
  - The French back-end server obtains the ($HELLO_{GE,i}$,$time'$) pair and processes it as follows:
    - It parses $Hello_{GE,i}$ to retrieve ecc_{GE} (8 bits), $ebid_X$ (64 bits), $time_X$ (16 bits) and mac_X (40 bits).
    - decrypts ecc_{GE}, using K_G, to recover the message country code, CC_{GE}. Since CC_{GE} is the country code for $Germany$, the ($HELLO_{GE,i}$,$time'$) pair is forwarded to the German back-end server.
    - The German server processes it as local pair.

This concept allows us to successively on-board other back-ends with potentially different EBID schemes. It further allows us to improve upon the format without requiring changes to the federation protocols.

A. Towards Probabilistic Notifications

As described in previous work [3, 7], all proximity-tracking schemes are vulnerable to the "one entry" attack. In this attack, the adversary has only one entry, corresponding to $User_T$, in her *LocalProximityList*[21]. When the adversary is notified "at risk", she learns that $User_T$ was diagnosed COVID-positive. The $ROBERT$ scheme, however, mitigates this attack by:

- (1) Requiring to all users to register (anonymously) to the server.
- (2) Not allowing a user that receives an ESR_REPLY message set to 1 to query the server anymore.

As a result, a registered user can only perform the attack once and then will be blocked by the system. She is therefore limited to one victim.

We argue that the only way to prevent this attack is to use *probabilistic* notifications in order to introduce some "deniability". In such a scheme, the server that receives a $ESR\_REQUEST$ message would reply:

- "0" (i.e. not at risk) if the User's ID is not in the list of exposed IDs.
- "1" if the User's ID is in the list of exposed IDs **or** if it is **randomly selected** by the server (the server selects additional users to receive a "1" reply with probability p).

[21] This attack can easily be achieved by keeping the Bluetooth interface off, switching it on when the adversary is next her victim and then switching it off again.

As a result, if the user receives a "1" back, she does not know whether it is because she has been exposed or whether she has been randomly selected by the server. Since the user cannot query the server anymore (as it already received a reply "1" back), she cannot send additional requests to refine his attack. We acknowledge that this attacks remains possible by n colluding nodes that target one user. In this case, the n colluding nodes will all get a "1" back and will find out the exposure status of their victim. The scalability of the attack is however reduced since it now requires n adversaries to target one victim.

The side effect of this proposal is that it introduces some false positives, i.e. some people will be notified whereas they are not really "at risk" (at least according to the proximity risk score). Is this acceptable or not? There are several elements of answer to this question. First, we need to acknowledge that proximity tracing is not perfect, and that there will be anyway false positives or false negatives. In this context, is it really problematic to add 5% or 10% more false positives? Second, the answer to this question may also depend on what the application is used for. If the App is used to target users that should get tested, we believe that testing 5 or 10% more users randomly should be quite acceptable. If the App is used to notify users to go in quarantine, false positives could be more problematic...

B. Server Security Considerations

This proposal assumes that the back-end server is correctly secured, implementing the best state-of the art counter-measures and deploying the required security measures to prevent intrusions [22].

Its security will be audited, tested and validated by the competent national bodies. Different measures are already considered like:

- A secured logging systems (in order to allow regular audits of the operations for secured or privacy checks).
- The use of hardware or at least software security modules for secure cryptographic processing, key generation and protection.

C. Authenticated Requests

ESR Request is an authenticated request, enabling App_A to make the server perform an operation on the data associated to user A. Other types of authenticated requests can be defined in a similar manner to trigger other operations, for example to unregister a user.

Any type of authenticated request is built in the same manner as an ESR request (see Section 7), with the exception that the constant c used in the MAC computation must be different.

For example, the following values of c could be used:

Code	Request Type
1	Hello message
2	ESR Request
3	Unregister
4	DeleteHistory

The server processing part is identical up to step 6 (included). These steps validate the request. In following steps the operations specific to the request are implemented.

References

[1] Decentralized privacy-preserving proximity tracing.

[2] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The skinny family of block ciphers and its low-latency variant mantis. In *Proceedings, Part II, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9815*, page 123–153, Berlin, Heidelberg, 2016. Springer-Verlag.

[3] Ran Canetti, Ari Trachtenberg, and Mayank Varia. Anonymous collocation discovery: Harnessing privacy to tame the coronavirus, 2020.

[4] Joel Hellewell et a. Feasibility of controlling covid-19 outbreaks by isolation of cases and contacts. The Lancet, 2020. `https://www.thelancet.com/journals/langlo/article/PIIS2214-109X(20)30074-7/fulltext`.

[22] See for example: `https://www.ssi.gouv.fr/entreprise/bonnes-pratiques/poste-de-travail-et-serveurs/`

[5] Luca Ferretti, Chris Wymant, Michelle Kendall, Lele Zhao, Anel Nurtay, Lucie Abeler-Dörner, Michael Parker, David Bonsall, and Christophe Fraser. Quantifying sars-cov-2 transmission suggests epidemic control with digital contact tracing. *Science*, 2020.

[6] Bluetooth SIG. *Bluetooth Core Specification v5.1*. 2019. Accessed: 2019-08-30.

[7] Serge Vaudenay. Analysis of DP3T. Cryptology ePrint Archive, Report 2020/399, 2020. `https://eprint.iacr.org/2020/399`.

Name	Description
App	Mobile application implementing our scheme
App_A	Mobile Application installed by user U_A
BLE	Bluetooth Low Energy
CT	An upper-bound on the number of days a user that has been diagnosed positive has been contagious. (for example 14 days)
CC_S	Country Code used for routing within the federation
$EBID$	Ephemeral Bluetooth IDentifier
$EBID_{A,i}$	Ephemeral Bluetooth IDentifier of user U_A at epoch i
ECC	Encrypted Country Code
$ECC_{A,i}$	Encrypted Country Code used by U_A at epoch i
$epoch\_duration\_sec$	Duration of an epoch in seconds
$ESR\_REQUEST$	Request sent by the App to query the user status
ESR_REPLY	Answer sent by the server to users to notify their status
$HELLO$	Message broadcast by an App via its Bluetooth Low Energy interface
ID	Permanent and anonymous identifier associated to each registered user
ID_A	Permanent and anonymous identifier of user U_A, stored by the server
$IDTable$	Database maintained by the back-end server
K_A^{enc}	Shared key between the server and the App_A, used for encryption of sensitive information.
K_A^{auth}	Shared key between the server and the App_A, used for authentication of App_A messages.
K_G	Federation Key (shared between the servers of all countries with a federation agreement)
K_S	Server Key (used by a server to generate $EBIDs$), stored by the server
LEE_A	List of exposed epochs of user U_A, stored in $IDTable$
$LocalProximityList$	Local list on an App where the $HELLO$ messages received by nearby devices are stored
M	Number of epochs between 2 consecutive requests by an App to the back-end server to obtain its list of $(EBID, ECC)$ pairs for the following epochs
(sk_S, pk_S)	Registration key-pair, an asymmetric key-pair of the server, used during registration.
Srv	The back-end server
SRE_A	Variable that indicates the last epoch when U_A has sent a "Status Request" to the server, stored in $IDTable$
T	The minimum number of epochs between 2 consecutive $ESR\_REQUEST$
$Tpststart$	The time when the proximity tracing service has been started
UN_A	Flag indicating if U_A has already been notified to be at risk of exposure. UN_A is stored in the $IDTable$

Table 1: Glossary of terms and variables used in this paper