CrowdNotifier Technical Specification

Release v0.1.5

Wouter Lueks (SPRING Lab, EPFL)
Linus Gasser (C4DT, EPFL)
Carmela Troncoso (SPRING Lab, EPFL)

May 18, 2021

CONTENT

1 Introduction **1**
- 1.1 Authors and Contributors . . . 1
- 1.2 License . . . 2

2 Terminology **3**

3 Cryptographic Building Blocks **4**
- 3.1 Basic Primitives . . . 4
- 3.2 Symmetric-key Encryption . . . 4
- 3.3 Public-key Encryption . . . 4
- 3.4 Identity-Based Encryption . . . 5

4 Basic CrowdNotifier **6**
- 4.1 Global Setup . . . 6
- 4.2 Setting up a Health Authority . . . 6
- 4.3 Setting-up a Location . . . 6
- 4.4 Visiting a Location . . . 8
- 4.5 Initiating Presence Notification . . . 11
- 4.6 Presence Tracing and Notification . . . 13

5 Managed CrowdNotifier **15**
- 5.1 The Idea . . . 15
- 5.2 Organization Setup . . . 16
- 5.3 Setting-up a New Location . . . 16
- 5.4 Initiating Presence Notification . . . 17

6 Server-Based CrowdNotifier **18**
- 6.1 Implications of Automatic Triggering . . . 18
- 6.2 Overview of Server-Based CrowdNotifier . . . 19
- 6.3 Setting up the Backend Server . . . 19
- 6.4 Setting-up a Location . . . 19
- 6.5 Visiting a Location . . . 20
- 6.6 Initiating Presence Notification . . . 20
- 6.7 Presence Tracing and Notification . . . 21
- 6.8 Security and Privacy Analysis . . . 22
- 6.9 Privacy enhancements . . . 24

7 Identity-based Encryption **25**
- 7.1 Mathematical description . . . 25
- 7.2 Implementation of IBE in CrowdNotifier . . . 26

Bibliography **28**

Index **29**

CHAPTER

ONE

INTRODUCTION

This document provides the technical specification for CrowdNotifier. Its goal is to help implementers implement a presence tracing system based on CrowdNotifier or its variants. This document repeats some of the concepts presented in the CrowdNotifier White Paper [LGV+] but should be seen as a companion rather than as a replacement.

This document describes in technical detail three variants of CrowdNotifier:

- *The basic CrowdNotifier scheme* is the same scheme as described in the CrowdNotifier White Paper [LGV+]. The version provides strong abuse resistance by requiring cooperation of both the *Location Owner* and *Health Authority* to trigger tracing. Records stored on the phone are private: they can only be decrypted if and only if these parties trigger tracing.
- *A managed version of CrowdNotifier* that enables an organization to manage many locations (for example, meeting rooms) at the same time without the overhead of storing different tracing QR codes for each of them. This scheme has the same properties as the basic CrowdNotifier scheme.
- *A server-based version of CrowdNotifier* that doesn't require cooperation of the *Location Owner* to trigger notifications, and can instead send notifications based on records uploaded by index cases. As a result, abuse resistance is weaker – the health authority can trigger locations on its own. However, it is fully compatible with the basic CrowdNotifier scheme so that clients, if they want, can still enjoy full privacy protection of records on the phone.

None of these schemes reveal which *Locations* are notified to adversaries that didn't visit these locations (nor colluded with somebody that did). We refer to the academic paper for a thorough analysis of requirements and security proofs [LGV+21].

1.1 Authors and Contributors

This technical specification was written by:

- Wouter Lueks, SPRING Lab, EPFL
- Linus Gasser, C4DT, EPFL
- Carmela Troncoso, SPRING Lab, EPFL

This document benefited from feedback by:

- Fabian Aggeler, Ubique
- Johannes Gallmann, Ubique
- Matthias Felix, Ubique
- Simon Roesch, Ubique

This document also benefited from earlier feedback on the white paper CrowdNotifier White Paper [LGV+].

1.2 License

CHAPTER

TWO

TERMINOLOGY

Backend Server A backend for server-based CrowdNotifier. It replaces the *Health Authority Backend* in the *server-based CrowdNotifier scheme*.

Health Authority The public health-authority that determines which visitors of which locations to notify. Usually, there is only one health authority.

Health Authority Backend A backend server operated by the *Health Authority*.

Location A semi-public location (for example, bar, restaurant, religious building, events venue, or meeting place) or an event which takes place in one or more locations (for example, an exhibition).

Location Owner The owner or manager of a location. We assume each location has one owner. We call an owner that manages multiple locations an *Organization*.

Organization A party that manages several locations. For example, a larger company that manages several meeting rooms.

Visitor Somebody who visits a location. Sometimes called user. There can be many visitors.

CHAPTER

THREE

CRYPTOGRAPHIC BUILDING BLOCKS

A short overview of the cryptographic primitives used by CrowdNotifier.

3.1 Basic Primitives

- SHA256 - The usual hash function
- HKDF - Hash-Based Key Derivation Function based on HMAC and SHA256.

3.2 Symmetric-key Encryption

CrowdNotifier uses symmetric encryption to send data that only visitors of (notified) locations can read. For this, we use a authenticated encryption scheme given by the algorithms AE.Enc and AE.Dec. We construct these using XSalsa20 (as stream cipher) and Poly1305 (as MAC).

As implementation, we use the following algorithms from libsodium:

- AE.Enc: crypto_secretbox_easy
- AE.Dec: crypto_secretbox_open_easy

3.3 Public-key Encryption

CrowdNotifier relies on a regular CCA2 secure public-key scheme given by the algorithms KeyGen, Enc, and Dec with the usual semantics:

- $(pk, sk) \leftarrow \mathsf{KeyGen}()$. Generates a public-private key-pair.
- $\mathsf{ctxt} \leftarrow \mathsf{Enc}(pk, m)$. Given a public key pk and a message m outputs a ciphertext ctxt.
- $m \leftarrow \mathsf{Dec}(sk, \mathsf{ctxt})$. Given a private key sk and a ciphertext ctxt outputs a message m or a failure symbol $\perp$.

We construct these using X25519 for key-exchange and XSalsa20-Poly1305 for the subsequent symmetric encryption.

As implementation, we use the following algorithms from libsodium:

- KeyGen: crypto_box_keypair
- Enc: crypto_box_seal
- Dec: crypto_box_seal_open

3.4 Identity-Based Encryption

CrowdNotifier relies on an identity-based encryption scheme to provide its most important properties. In an identity-based encryption scheme, messages can be encrypted against identities without requiring a specific key-pair to be generated for each identity. Instead, a trusted authority – in our case usually a location owner or organization – generates a master public key mpk and a corresponding master private key msk by running the $\mathsf{IBE.KeyGen}$ algorithm. We emphasize that each location has its own corresponding public key mpk.

To encrypt a message m against an identity id under the public key mpk, a party (in our case a visitor) runs $\mathsf{ctxt} \leftarrow \mathsf{IBE.Enc}(\mathsf{mpk}, \mathsf{id}, m)$. To decrypt this ciphertext, the trust authority (e.g., location or organization) first computes the corresponding identity-based decryption key $\mathsf{sk_{id}} \leftarrow \mathsf{IBE.KeyDer}(\mathsf{mpk}, \mathsf{msk}, \mathsf{id})$. Given the identity-based decryption key $\mathsf{sk_{id}}$ a visitor (user) can then decrypt a ciphertext ctxt encrypted under an identity id by running $m \leftarrow \mathsf{IBE.Dec}(\mathsf{id}, \mathsf{sk_{id}}, \mathsf{ctxt})$.

We refer to the *Identity-based Encryption* section for the full details. For completeness, we introduce the full syntax that we use:

- $\mathsf{pp} \leftarrow \mathsf{IBE.CommonSetup}(1^\ell)$. Generates the common parameters pp. Typically these parameters are part of the implementation.
- $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{IBE.KeyGen}(\mathsf{pp})$. Generates a master public-private key pair.
- $\mathsf{sk_{id}} \leftarrow \mathsf{IBE.KeyDer}(\mathsf{mpk}, \mathsf{msk}, \mathsf{id})$. On input of a master public key mpk, a master private key msk, and an identity id; outputs private decryption key $\mathsf{sk_{id}}$ corresponding to this identity.
- $\mathsf{ctxt} \leftarrow \mathsf{IBE.Enc}(\mathsf{mpk}, \mathsf{id}, m)$. On input of a master public key mpk, an identity id, and a message m, outputs a ciphertext ctxt.
- $m \leftarrow \mathsf{IBE.Dec}(\mathsf{id}, \mathsf{sk_{id}}, \mathsf{ctxt})$. On input of an identity id, a private key $\mathsf{sk_{id}}$, and a ciphertext ctxt, either outputs the decryption m of ctxt, or $\perp$ if decryption fails.

We implement the concrete *Identity-based Encryption* scheme over the BLS12-381 curve. As instantiation we use the mcl library. We refer to *later sections for more information on the implementation*.

CHAPTER

FOUR

BASIC CROWDNOTIFIER

The basic CrowdNotifier scheme equals the scheme presented in the white paper. The scheme we present here has some modifications to incorporate some new insights that simplify the QR codes.

In this scheme, generating notifications requires cooperation of both the *Location Owner* and the *Health Authority*.

4.1 Global Setup

This document fixes specific algorithm choices for the cryptographic schemes as well as the cryptographic parameters of these schemes. We refer to *the section on Cryptographic Building Blocks* for the details on these instantiations. We assume these algorithms and parameters are public.

4.2 Setting up a Health Authority

The *Health Authority* generates a public-private encryption key pair pk_H, sk_H by running KeyGen of the public-key encryption algorithm. The health authority publishes pk_H and privately stores sk_H at the *Health Authority Backend*.

This key-pair provides an extra layer of privacy protection for encrypted visits stored on a user's phone. To decrypt these visits, an attacker would need to obtain both *skH* as well as the private information stored by the *Location Owner*.

Since we use `libsodium`, the health authority simply runs `crypto_box_keypair` to generate this key-pair.

4.3 Setting-up a Location

To set up a *Location* the *Location Owner* runs the setup program. This program will output two QR codes: a public QR code $\mathsf{QR}_{\mathsf{entry}}$ and a private QR code $\mathsf{QR}_{\mathsf{trace}}$. For security reasons, this setup program must run client-side. We propose to use client-side JavaScript to statelessly generate the PDFs containing the QR codes.

The *Location Owner* provides a description of the location (e.g., name, address, type). Setup then proceeds as follows. It computes two IBE key pairs (one for the location, and one for the health authority)

$$(\mathsf{mpk}_{\mathsf{L}}, \mathsf{msk}_{\mathsf{L}}) \leftarrow \mathsf{IBE.KeyGen}(\mathsf{pp}),$$
$$(\mathsf{mpk}_{\mathsf{HA}}, \mathsf{msk}_{\mathsf{HA}}) \leftarrow \mathsf{IBE.KeyGen}(\mathsf{pp})$$

and computes $\mathsf{mpk} = \mathsf{mpk}_{\mathsf{L}} \cdot \mathsf{mpk}_{\mathsf{HA}} \in G_2$. It encrypts the master secret key $\mathsf{msk}_{\mathsf{HA}}$ for the health authority by creating the ciphertext $\mathsf{ctxt}_{\mathsf{HA}} = \mathsf{Enc}(pk_H, \mathsf{msk}_{\mathsf{HA}})$. It deletes and does not store the value $\mathsf{msk}_{\mathsf{HA}}$. Finally, it picks a random 32-byte seed seed.

In code, the setup script computes the following values:

```
// Input: the public key pkha of the Health Authority

// Generate IBE key pairs
const [mpkl, mskl] = IBEKeyGen();
const [mpkha, mskha] = IBEKeyGen();

// Compute resulting master public key
const mpk = mcl.add(mpkl, mpkha);

// Compute encrypted master secret key for health authority
const ctxtha = crypto_box_seal(mskha.serialize(), pkha);

// Generate seed
const seed = randombytes_buf(32);
```

The setup program then encodes these values into two QR codes:

1. The entry QR code QR_{entry} containing the description of the location, mpk and the seed seed.
2. The tracing QR code QR_{trace} containing the same values as QR_{entry} and additionally msk_L and $ctxt_{HA}$.

To provide strong abuse-prevention properties, it is essential that the setup procedure does not store the master private key msk_{HA}. By only storing the encrypted version $ctxt_{HA}$, both the data stored securely in the tracing QR code QR_{trace} as well as cooperation of the *Health Authority* (to decrypt $ctxt_{HA}$) are needed to compute tracing keys. As a result, the *Location Owner* is protected against coercion attacks.

4.3.1 Entry QR Code Format

> **Warning:** The precise QR code format might see some minor updates in the near future.

CrowdNotifier adopts a standard payload format to encode data into the QR code QR_{entry} that is scanned by visitors. We use the following standard protobuf format:

```
syntax = "proto3";

message QRCodePayload {
  uint32 version = 1;
  TraceLocation locationData = 2;
  CrowdNotifierData crowdNotifierData = 3;

  // Country-specific location information
  bytes countryData = 4;
}

message TraceLocation {
  uint32 version = 1;
  // max. 100 characters
  string description = 2;
  // max. 100 characters
  string address = 3;

  // UNIX timestamp (in seconds since Unix Epoch)
  uint64 startTimestamp = 5;
  // UNIX timestamp (in seconds since Unix Epoch)
  uint64 endTimestamp = 6;
}

message CrowdNotifierData {
  uint32 version = 1;
  bytes publicKey = 2;
  bytes cryptographicSeed = 3;
  uint32 type = 4; // exact semantic tbd
}
```

The setup program includes the description of the location in the `TraceLocation` structure. It adds potential other country-specific information to the `countryData` field. The `startTimestamp` and `endTimestamp` denote the validity time of this QR code.

The `CrowdNotifierData` structure encodes the CrowdNotifier elements as described above. The `publicKey` field encodes the master public key mpk as a byte array. We use mcl's serialization format. See *the section on serialization for more details*. The `cryptographicSeed` fields encode the 32-byte seed seed as a byte array.

Note: The QR code format does not include a country code. Instead, apps should use the URL embedded in the QR code to deduce the corresponding country.

Finally, the payload protobuf must be encoded into a QR code. There are different methods for doing this. The most obvious approach is to encode a URL in the QR code that includes the encoded payload protobuf.

NotifyMe

The NotifyMe app encodes the following URL in the QR code:

```
https://qr.notify-me.ch?v=3#<base64-encoded-protobuf>
```

Users scan this QR code either directly with the corresponding app, or with their camera application. When the app is not installed, phones open this url in the browser. Including the payload after the anchor tag ensures that it is not sent to the server. Ensuring that the server doesn't learn which locations the user is visiting.

4.4 Visiting a Location

Upon entering a *Location*, the user uses their app to scan the corresponding entry QR code $\mathsf{QR}_{\mathsf{entry}}$ and obtain the values encoded therein. The app shows a description of the location based on the information in the `locationData` and `countryData` fields. Then the app asks for confirmation that the user wants to check in.

At this point the app stores the check-in time. After a while, the app learns that the user left the *Location*. Several mechanisms are possible:

- That app sends a reminder to the user after at time chosen during check-in
- The QR code contains a default time, and checks the user out automatically.

In both cases, it might be helpful if apps allow users to adjust the check-in and check-out times to reflect the actual time in the *Location*. So that the app can store the correct records, even if the user only remembers to checkout later.

Given the arrival time `arrival time` and departure time `departure time`, as well as the master public key mpk and seed seed encoded in the `CrowdNotifierData` part of the payload, the app proceeds as follows:

1. The app uses the QR code payload payload to compute the notification key notifykey and the time-specific identities id using *the process detailed in the next section*.
2. The app encodes a record m capturing the arrival and departure times, as well as the notification key notifykey:

$$m = (\texttt{arrival time},\ \texttt{departure time},\ \mathsf{notifykey})$$

3. For each identity id computed in step 1, the app computes the ciphertext

$$\mathsf{ctxt} \leftarrow \mathsf{IBE.Enc}(\mathsf{mpk}, \mathsf{id}, m)$$

and stores it together with a label for the current day. The app does not store any of the other data computed as part of this process.

The following code-block shows an example.

```
// Calculate the MessagePayLoad m as a JSON string
const messagePayload: MessagePayload = {
  arrivalTime: arrivalTime,
  departureTime: departureTime,
  notificationKey: venueInfo.notificationKey,
};
const msgPBytes = from_string(JSON.stringify(messagePayload));

// Encrypt the record m using the IBE scheme
const ctxt = enc(masterPublicKey, identity, msgPBytes);
```

Devices automatically delete any entry older than 10 days.

4.4.1 Computing Identities and Keys

As part of the process to visit a *Location*, the app computes time-specific identities corresponding to the user's visit to this *Location*. These time-specific identities correspond to time intervals that overlap with the user's visit. Currently, these intervals are all exactly 1 hour long, corresponding to a interval length of `intervalLength = 60*60 = 3600` seconds. But the following specification supports different interval lengths.

1. The app derives three 32-byte values $\mathsf{nonce}_{\mathsf{preid}}$, $\mathsf{nonce}_{\mathsf{timekey}}$, and notifykey from the QR code using HKDF:

$$\mathsf{nonce}_{\mathsf{preid}} \,\|\, \mathsf{nonce}_{\mathsf{timekey}} \,\|\, \mathsf{notifykey} = \mathsf{HKDF}(96, \mathsf{payload}, \texttt{""}, \texttt{"CrowdNotifier\_v3"})$$

where the input key material payload is the raw protobuf (e.g., after base64 decoding, but before parsing), the salt is empty, and `"CrowdNotifier_v3"` is the info string.

The 32-byte `cryptographicSeed` in the payload ensures that the input key material has sufficient entropy. By using the entire payload as input key material rather than only this seed, we ensure maximal entropy, even if the `cryptographicSeed` is shorter.

2. The app computes a pre-identity for the *Location*:

$$\mathsf{preid} = \mathsf{SHA256}(\texttt{"CN-PREID"} \,\|\, \mathsf{payload} \,\|\, \mathsf{nonce}_{\mathsf{preid}})$$

The input to the hash-function is the concatenation of 3 byte arrays: the ASCII encoded 8-byte string `CN-PREID` for domain separation, then the raw payload, and finally the 32-byte nonce $\mathsf{nonce}_{\mathsf{preid}}$.

3. For each supported interval length intervalLength in seconds (currently only 1 hour, corresponding to 3600 seconds, is used) compute the interval start times intervalStart (in seconds since UNIX epoch) for all intervals of length intervalLength that overlap with the user's visit. The start times must be aligned with the start of the interval, e.g., `intervalStart % intervalLength == 0`.

4. For each interval length intervalLength (in seconds), and interval start time intervalStart (in seconds since UNIX epoch) compute the corresponding identity key timekey:

$$\mathsf{timekey} = \mathsf{SHA256}(\texttt{"CN-TIMEKEY"} \,\|\, \mathsf{intervalLength} \,\|\, \mathsf{intervalStart} \,\|\, \mathsf{nonce}_{\mathsf{timekey}}),$$

where the inputs to the hash-function is the concatenation of the following values:

- The 8-byte ASCII encoding of the string `CN-TIMEKEY`

- The 4-byte big-endian encoding of the value intervalLength (`900 <= intervalLength <= 86400`)
- The 8-byte big-endian encoding of intervalStart
- The 32-byte nonce $\text{nonce}_{\text{timekey}}$.

Next, compute the corresponding time-specific identity

$$\text{id} = \text{SHA256}(\texttt{"CN-ID"} \parallel \text{preid} \parallel \text{intervalLength} \parallel \text{intervalStart} \parallel \text{timekey}),$$

where the inputs to the hash-function is the concatenation of the following values:

- The 5-byte ASCII encoding of the string `CN-ID`
- The 32-byte SHA256 output preid
- The 4-byte big-endian encoding of the value intervalLength (`900 <= intervalLength <= 86400`)
- The 8-byte big-endian encoding of intervalStart
- The 32-byte SHA256 output timekey

In code, the app proceeds as follows:

```
// Calculate the values using the HKDF
const hkdf = require('futoin-hkdf');

const ikm = qrCodePayload;
const length = 96;
const salt = ''; // salt is empty
const info = 'CrowdNotifier_v3';
const hash = 'SHA-256';

const derivedBuffer: Uint8Array = hkdf(ikm, length, {salt, info, hash});
const noncepreid = derivedBuffer.slice(0, 32);
const noncetimekey = derivedBuffer.slice(32, 64);
const notificationKey = derivedBuffer.slice(64, 96);

// Calculate the pre-identity for the Location
const preid = crypto_hash_sha256(
    Uint8Array.from([
      ...from_string('CN-PREID'),
      ...qrCodePayload,
      ...noncepreid,
    ]),
);

// Currently only one intervalLength is supported
const intervalLength = 3600;

// `intervals` contains the `intervalStart` times for the whole duration of the
// visit.
const ids = intervals.map((id) => {
  timekey = crypto_hash_sha256(
        Uint8Array.from([
          ...from_string('CN-TIMEKEY'),
          ...toBytesInt32(intervalLength),
          ...toBytesInt64(intervalStart), // timestamp might use up to 8 bytes
          ...noncetimekey,
        ]),

  id = crypto_hash_sha256(
        Uint8Array.from([
          ...from_string('CN-ID'),
          ...preid,
          ...toBytesInt32(intervalLength),
          ...toBytesInt64(intervalStart), // timestamp might use up to 8 bytes
          ...timekey,
        ]),
    )
```

(continues on next page)

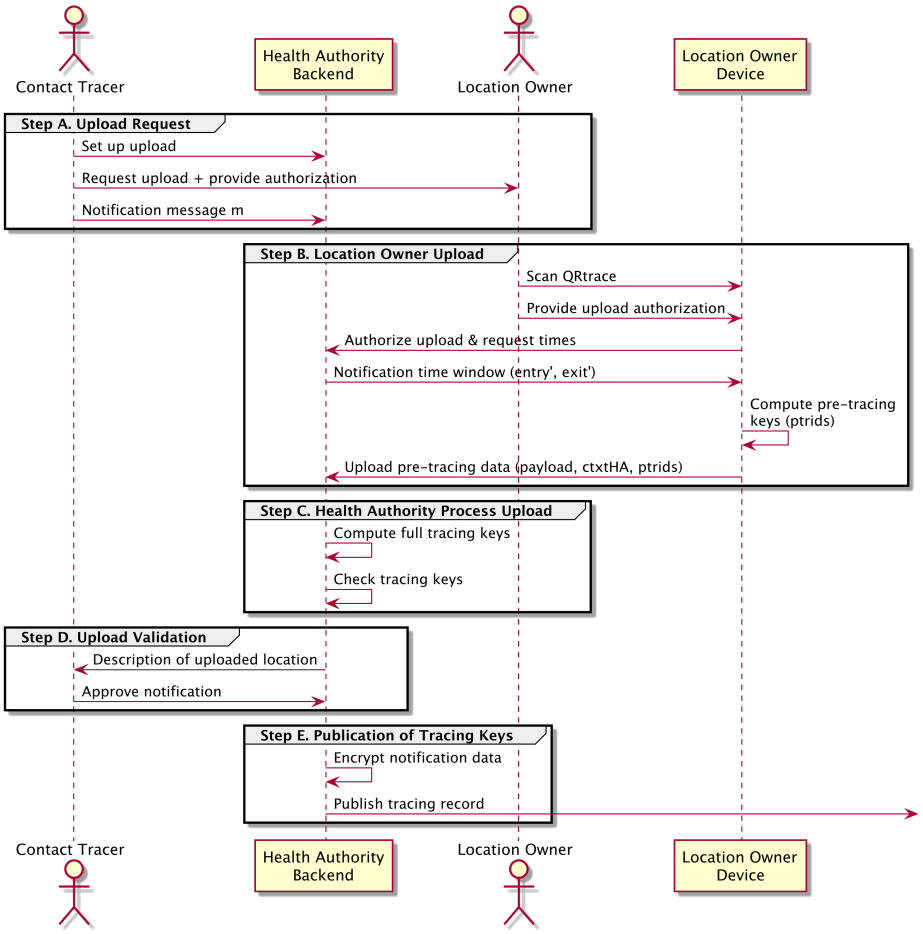

Fig. 4.1: Overview of process to initiate presence notification

(continued from previous page)

```
  return id;
  });
```

4.5 Initiating Presence Notification

After the contact tracing team of the *Health Authority* has determined that a SARS-CoV-2-positive person has visited a location during the contagious period, they proceed as follows. Let entry' and exit' be the times that the SARS-CoV-2-positive person entered and exited this location. See Fig. 4.1 for an overview.

- *Step A (upload request).* The contact tracing team contacts the *Location Owner* of the *Location* and requests an upload of the hour-specific pre tracing keys to the health authority's servers. They also provide the *Location Owner* with a means to authenticate this upload, for example a one-time token. Finally, the tracing team specifies a message m that should be sent to the notified users. They upload this message to the *Health Authority Backend*.
- *Step B (location owner upload).* The location owner scans the tracing QR code $\mathsf{QR}_{\mathsf{trace}}$ with their app to obtain the payload payload of the entry QR code, as well as the values $\mathsf{msk}_{\mathsf{L}}$ and $\mathsf{ctxt}_{\mathsf{HA}}$. The app then proceeds as follows

  1. The app authenticates to the *Health Authority Backend* (e.g., using the one-time token) and obtains the entry' and exit' times corresponding to the index case.
  2. The app uses the QR code payload payload and the times entry' and exit' to compute the

relevant time-specific identities id using *the process detailed in the next section*.

3. For each identity id computed in the previous step it computes the partial identity-based decryption key

$$\mathsf{psk}_{\mathsf{id}}^{\mathsf{L}} = \mathsf{IBE.KeyDer}(\mathsf{msk}_{\mathsf{L}}, \mathsf{id}).$$

Let the pre-tracing key be $\mathsf{ptr}_{\mathsf{id}} = (\mathsf{id}, \mathsf{psk}_{\mathsf{id}}^{\mathsf{L}})$.

4. The app uploads payload, $\mathsf{ctxt}_{\mathsf{HA}}$, and all pre-tracing keys $\mathsf{ptr}_{\mathsf{id}}$ to the *Health Authority Backend*.

- *Step C (health authority process upload).* The health authority's system processes each upload as follows.

1. It decrypts $\mathsf{ctxt}_{\mathsf{HA}}$ to obtain $\mathsf{msk}_{\mathsf{HA}}$.
2. It uses the QR code payload payload and the times entry' and exit' to recompute the relevant time-specific identities id using *the process detailed in the next section*.
3. It parses the pre-tracing keys $\mathsf{ptr}_{\mathsf{id}}$ as $(\mathsf{id}, \mathsf{psk}_{\mathsf{id}}^{\mathsf{L}})$ and discards any entries with identities id it did not compute in the previous step. Next, it computes its part of the identity-based decryption key

$$\mathsf{psk}_{\mathsf{id}}^{\mathsf{HA}} = \mathsf{IBE.KeyDer}(\mathsf{msk}_{\mathsf{HA}}, \mathsf{id}).$$

and computes the final identity-based decryption key

$$\mathsf{sk}_{\mathsf{id}} = \mathsf{psk}_{\mathsf{id}}^{\mathsf{L}} \cdot \mathsf{psk}_{\mathsf{id}}^{\mathsf{HA}}.$$

Let $\mathsf{tr}_{\mathsf{id}} = (\mathsf{id}, \mathsf{sk}_{\mathsf{id}})$.

4. It validates the computed tracing key $\mathsf{tr}_{\mathsf{id}} = (\mathsf{id}, \mathsf{sk}_{\mathsf{id}})$. To do so, it picks a random message m of sufficient length and computes the ciphertext $\mathsf{ctxt} \leftarrow \mathsf{IBE.Enc}(\mathsf{mpk}, \mathsf{id}, m)$, and verifies that $\mathsf{IBE.Dec}(\mathsf{sk}_{\mathsf{id}}, \mathsf{id}, \mathsf{ctxt}) = m$. If the check fails, it removes the upload.

- *Step D (upload validation).* The contact tracing team checks that the uploaded tracing-keys correspond to the expected *Location*. To do so, they compare the description of the location in the supplied payload with the expected location. This validation step could happen automatically.

- *Step E (publication of tracing keys).* To publish the tracing keys, the *Health Authority Backend* proceeds as follows:

1. The server formats creates notification data notificationdata that contains the message m (specified in step A) as well as information about the notification interval. This package also contain extra machine-readable information such as the severity of the warning:

$$\mathsf{notificationdata} = (\mathsf{entry'}, \mathsf{exit'}, m)$$

see *the definition of AssociatedData below* for a concrete ProtoBuf based instantiation.

2. Finally, the *Health Authority Backend* uses payload to recompute the notification key notifykey and computes $c_{\mathsf{notify}} = \mathsf{AE.Enc}(\mathsf{notifykey}, \mathsf{notificationdata})$, the encrypted message for the notified visitors, and for each relevant timeslot it publishes a record $(\mathsf{tr}_{\mathsf{id}}, \mathsf{day}, c_{\mathsf{notify}})$, where day corresponds to the day (since the UNIX epoch) on which this timeslot begins. See *the definitions of ProblematicEvent and ProblematicEventWrapper below* for concrete ProtoBuf based instantiations.

Note: As for DP3T-based application, the *Health Authority Backend* should publish tracing records via a CDN to facilitate downloading by millions of clients.

4.5.1 Data Formats

The CrowdNotifier uses the following standard encrypted payload format to encode notificationdata:

```
syntax = "proto3";

message AssociatedData {
  int32 version = 1;
  string message = 2;

  // UNIX timestamp (in seconds since Unix Epoch)
  int64 startTimestamp = 3;
  // UNIX timestamp (in seconds since Unix Epoch)
  int64 endTimestamp = 4;

  bytes countryData = 5;
}
```

This data structure follows the same structure as *the entry QR code format* defined above. The `countryData` can be used to insert other country-specific information. It is currently not used.

The per time-slot event data $(\mathsf{tr}_{\mathsf{id}}, \mathsf{day}, c_{\mathsf{notify}})$ is wrapped in a `ProblematicEvent`:

```
syntax = "proto3";

message ProblematicEventWrapper {
  int32 version = 1;
  repeated ProblematicEvent events = 2;
}
message ProblematicEvent {
  int32 version = 1;
  bytes identity = 2;
  bytes secretKeyForIdentity = 3;

  // UNIX timestamp corresponding to day start (in seconds since Unix Epoch)
  int64 day = 4;

  bytes encryptedAssociatedData = 5;
  bytes cipherTextNonce = 6;
}
```

Recall that $\mathsf{tr}_{\mathsf{id}} = (\mathsf{id}, \mathsf{sk}_{\mathsf{id}})$. In the `ProblematicEvent` protobuf, `identity` encodes id, and `day` encodes the start of day (in seconds since UNIX Epoch).

The field `secretKeyForIdentity` encodes $\mathsf{sk}_{\mathsf{id}}$ *serialized as per the specification in the building blocks section*. The ciphertext c_{notify} is encoded by `encryptedAssociatedData` and `cipherTextNonce`. The ciphertext follows the `libsodium` encoding, see *the cryptographic building blocks* for more details.

A sequence of such tuples is wrapped in a single `ProblematicEventWrapper`.

4.6 Presence Tracing and Notification

The user's app regularly (say, every few hours) performs the following checks:

1. The app downloads all $(\mathsf{tr}_{\mathsf{id}}, \mathsf{day}, c_{\mathsf{notify}})$ tuples that were published since the last time it checked.
2. For each tuple downloaded $(\mathsf{tr}_{\mathsf{id}}, \mathsf{day}, c_{\mathsf{notify}})$ let $\mathsf{tr}_{\mathsf{id}} = (\mathsf{id}, \mathsf{sk}_{\mathsf{id}})$. The app proceeds as follows.
    1. For each record ctxt recorded on a day corresponding to day, the app tries to decrypt it using $\mathsf{IBE.Dec}(\mathsf{id}, \mathsf{sk}_{\mathsf{id}}, \mathsf{ctxt})$. The app selects records where decryption succeeds (i.e., those not equal to $\perp$).
    2. For all selected records ctxt, the app uses the plaintext of ctxt to recover the arrival time, departure time and the notification key $\mathsf{notifykey}$.

3. The app then uses notifykey to decrypt c_{notify} and recover:

$$\text{notificationdata} = (\text{entry'}, \text{exit'}, m)$$

   if decryption fails, it moves on to the next matching tuple.

4. The app checks if there is an overlap between the user's time at the location and what is indicated by entry' and exit'. If there is an overlap, the app uses m to notify the user.

CHAPTER

FIVE

MANAGED CROWDNOTIFIER

In this section we introduce a managed version of CrowdNotifier where an *Organization* manages many *Locations* at once without having to store a tracing QR code for each.

The design in the white paper assumes that each *Location Owner* manages a single location, and can thus store the single tracing QR code containing the tracing information. In reality, a single restaurant may consist of several different rooms, or locations. And a large company might want to manage many (meeting) rooms at the same time. In these cases, storing tracing QR codes for each of these locations becomes cumbersome. Instead, we show how a single tracing key can be used to manage all rooms and locations under control of the same entity.

5.1 The Idea

In the *basic CrowdNotifier scheme*, the *Location Owner* creates one master public key and corresponding private keys per location. In this section we show how the same master key can be used for *all Locations* managed by the same *Organization*. Thereby drastically simplifying key management.

The *Organization* still needs to store information to facilitate tracing. We assume the *Organization* has a local database in which it keeps track of the following values:

- The master public key mpk for this *Organization*
- The ciphertext $\mathsf{ctxt}_{\mathsf{HA}}$ for the *Health Authority*
- For each *Location*, a copy of the QR code $\mathsf{payload}$.

To initiate notifications, the *Organization* also needs the master secret key $\mathsf{msk}_{\mathsf{O}}$. The security of tracing information hinges on keeping the master secret $\mathsf{msk}_{\mathsf{O}}$ secure. In the following, we describe how this key can be derived from a passphrase. As a result, no security-critical information needs to be stored in the local database. The passphrase itself can be stored in a password management system.

Given the local database, and the master secret key, the manager can recover all information needed for tracing, and compute the per location tracing keys. In the following, we describe these steps in more detail below.

5.2 Organization Setup

Initializing an *Organization* proceeds as follows:

1. The system generates a strong passphrase of at least 256 bits of entropy. The operator should store the passphrase securely, for example in a password management system. The passphrase is the only security-critical component and is only needed to initiate tracing of rooms. It is not needed to add new rooms to the system.
2. First, setup computes the organization's master secret key as using the passphrase:

$$\mathsf{msk_O} \leftarrow H(\mathsf{passphrase}) \mod p,$$

   where p is the group order and $\mathsf{mpk_O} = g_2^{\mathsf{msk_O}}$ the corresponding *Organization*'s public key.

   Ideally, the output of H should be much longer than the bit length of p. For example, using SHA512 for p of 256 bits. Alternatively, $\mathsf{msk_O}$ can be directly computed by an appropriate method for hashing to the field $\mathbb{Z}_p$ if provided by the cryptographic library.
3. Setup then proceeds as in the original QR code generation process to compute the health authority key-pair $\mathsf{mpk_{HA}}, \mathsf{msk_{HA}}$; the encrypted master secret key $\mathsf{ctxt_{HA}}$ for the health authority; and the master public key $\mathsf{mpk} = \mathsf{mpk_O} \cdot \mathsf{mpk_{HA}}$. See the *Setting-up a Location in the basic CrowdNotifier scheme*.
4. The system then stores mpk and $\mathsf{ctxt_{HA}}$ in the local database. It does not store any of the other generated values. In particular, as in the *basic CrowdNotifier scheme*, it is essential for abuse resistance that the setup process does not store $\mathsf{msk_{HA}}$.

In code, the setup script computes the following values:

```
// Input: the public key pkha of the Health Authority
// Input: strong passphrase pp

// Generate mskO, mpkO from passphrase pp
const mskO = new mcl.Fr();
mskO.setHashOf(from_string(pp));
const mpkO = mcl.mul(baseG2(), mskO);

// Compute IBE key-pair for health authority
const [mpkha, mskha] = keyGen();

// Compute resulting master public key
const mpk = mcl.add(mpkO, mpkha);

// Compute encrypted master secret key for health authority
const ctxtha = crypto_box_seal(mskha.serialize(), pkh);
```

5.3 Setting-up a New Location

To add a new *Location*, the *Organization* supplies the information describing the new *Location* (e.g., name, address), see also *the entry code format*. The system then proceeds *as in the basic scheme*, except that it uses the master public key mpk from the database:

1. Retrieve the master public key mpk from the local database.
2. Pick a random 32-byte seed seed
3. Construct the entry QR code *as in the basic scheme* by including mpk, seed, and the description of the location.
4. Store the resulting QR-code payload in the local database.

5.4 Initiating Presence Notification

To initiate tracing, the *Health Authority* contacts the *Organization* and specifies the room/location for which it wants to notify the visitors. The operator uses the passphrase generated initially to recover the information that would normally be in the tracing QR code as follows:

1. The *Organization* enters the $\mathsf{passphrase}$ into the local system. The system recomputes the organization's master secret key msk_O from the passphrase.
2. The system retrieves the master public key mpk and the encrypted master secret key $\mathsf{ctxt}_\mathsf{HA}$ of the health authority from the database.
3. For the location (e.g., a room) specified by the *Health Authority*, retrieve the stored payload $\mathsf{payload}$ from the database.
4. Proceed as in *initiating presence notification of the basic scheme* where $\mathsf{msk}_\mathsf{L} = \mathsf{msk}_\mathsf{O}$.

CHAPTER

SIX

SERVER-BASED CROWDNOTIFIER

In the basic CrowdNotifier scheme, triggering notification requires cooperation of both the *Health Authority* and the *Location Owner* (or *Organization*, when using *the managed version of CrowdNotifier*). The need for two parties strengthens the abuse resistance of the solution. It ensures that the *Health Authority* on their own cannot trigger notifications. Additionally, it enables users to receive notifications based on encrypted data stored on the phone, providing strong privacy protection of phone records.

An alternative design choice can be to prioritize speed of notification, by choosing what venues should trigger notifications without relying on decisions by the *Health Authority*. Instead, the decision to trigger can be made automatically, either at the server, e.g. [RBC21], or on the phone, e.g., the initial proposal by the CoronaWarnApp team[1]. In this approach, notifications are sent without explicit approval by health authorities, and without explicit approval by venue owners.

Visitors' phones store clear text records of (possibly ephemeral) identifiers of visited *Locations* together with the visit times. The *Health Authority* provides index cases with an authorization to upload these identifiers and times to the backend server. Other phones download these identifiers and times and compare them against their own records. If there is an overlap, the phone notifies the user.

In this section we propose a variant of CrowdNotifier, called Server-Based CrowdNotifier. This variant facilitates automatic notifications mechanism based on uploads by index cases, while being interoperable with *the basic CrowdNotifier protocol* and its clients. The basic and server-bases approaches are interoperable in that clients are always notified regardless of scheme:

- Basic CrowdNotifier clients can operate in regions that deploy Server-Based CrowdNotifier. They scan QR codes, they store encrypted records with strong privacy protections as before, and they can determine exposure and notify their users just like Server-Based CrowdNotifier clients.
- Server-Based CrowdNotifier clients can operate in regions that deploy the basic CrowdNotifier scheme. They scan QR codes, they store records (containing identifiers) as before, and they can determine exposure and notify their users just like basic CrowdNotifier clients in these regions.

6.1 Implications of Automatic Triggering

Automating the triggering decision process brings the following differences with respect to the basic CrowdNotifier.

First, the lack of filtering by a *Health Authority* is likely to result in the system triggering more notifications than if the *Health Authority* was involved. Depending on how many CrowdNotifier locations users visit, there may be an increase in the number of notifications they receive. This can result in notification fatigue, and in users ignoring these notifications.

Second, because in this approach phones need to store clear text records to enable uploads, anyone with access to the phone can learn the identifiers of the places where the user has been. To ensure

[1] https://github.com/corona-warn-app/cwa-documentation/blob/master/event_registration.md

privacy of the records stored on the phone, *Locations* should frequently rotate their identifiers and the corresponding QR codes. If the QR codes remain static, privacy of records is limited. On the contrary, even if the QR codes are static, the basic CrowdNotifier protocol still guarantees privacy of stored records.

Third, if QR codes are not rotated, malicious actors can use identifiers they know (e.g., via crowdsourcing collection) to trigger notifications for locations they did not visit if they also have an upload authorization.

Fourth, the upload of *Locations* visited by the same user and/or the publication of *Locations* visited by a small group of index cases, may enable an adversaries to learn information about user patterns and co-locations. In the basic CrowdNotifier scheme, since the *Health Authority* is the one triggering notifications, neither uploads nor downloads leak any information about users.

We provide an in-depth security and privacy analysis of Server-Based CrowdNotifier at the end of this section.

6.2 Overview of Server-Based CrowdNotifier

The key idea of Server-Based CrowdNotifier is to let the central *Backend Server* replace the roles of the *Health Authority* and *Location Owner* in the basic CrowdNotifier scheme. To this end, the *Backend Server* can generate tracing keys that let basic CrowdNotifier clients decrypt their records after receiving appropriate uploaded information from the index case.

To enable this shift, we build on the *managed CrowdNotifier scheme*, and let the *Backend Server* generate a single master public-private key pair. The corresponding master public key is included in all QR codes in the region managed by this server. The *Backend Server* locally stores the corresponding master secret key.

Index cases upload the information they collected about the locations they visited to the *Backend Server*. The backend uses this information to compute the relevant tracing identities, and uses the master secret key to compute the corresponding tracing keys. Finally, the backend transmits the tracing keys to all clients. We recall that the tracing identity is just cryptographic material. To run this process, the backend does not need to know the venue's real data.

We next detail the steps of Server-Based CrowdNotifier.

6.3 Setting up the Backend Server

The *Backend Server* generates a master public key mpk_S and a corresponding master secret key msk_S by running $\mathsf{IBE.KeyGen}$ of the identity-based encryption algorithm. The server publishes mpk_S and privately stores msk_S.

6.4 Setting-up a Location

To set up a *Location* the *Location Owner* runs the setup program. This process proceeds in much the same way *as in the basic CrowdNotifier scheme*, but skips the key-generation steps. The program outputs one public QR code. In Server-Based CrowdNotifier their is no corresponding private QR code.

For security reasons, the setup program must run client-side. We propose to use client-side JavaScript to statelessly generate the PDFs containing the QR code.

The *Location Owner* provides a description of the location (e.g., name, address, type). Setup then proceeds as follows.

1. It retrieves the master public key mpk_S of the server.

2. It picks a random 32-byte seed.

Setup then generates the public QR code by encoding it into *standard QR-code format*. In particular, it inserts the location information, the seed it just generated, and the server's master public key mpk_S.

6.5 Visiting a Location

When visiting a location basic CrowdNotifier clients *proceed as in the basic scheme*. Server-Based CrowdNotifier clients proceed differently to support uploads.

As before, we assume the app gathers the arrival time `arrival time` and departure time `departure time`. See *the basic scheme for more details*. The app then proceeds as follows.

1. Using *the process detailed for the basic CrowdNotifier scheme* the app derives from the QR code payload: the pre identity preid for the *Location*, the notification key notifykey, and for each interval (intervalLength, intervalStart) that overlaps with the user's visit the time-specific keys timekey and identities id. This process requires only basic cryptographic primitives.
2. The app creates a visit record containing `arrival time`, `departure time`, the pre identity preid, the notification key notifykey, and the time specific tuples:

$$(\mathsf{intervalLength}, \mathsf{intervalStart}, \mathsf{timekey}, \mathsf{id}).$$

3. The app stores the visit record. When extra privacy is required, the app can encrypt the visit record against the public key of the *Health Authority* and additionally store the *basic CrowdNotifier encrypted record* to match notifications.

The pre identity preid and values (timekey, id) are needed to enable the *Backend Server* to compute the location tracing keys. The notification key notifykey is needed decrypt notification messages from the backend, and to enable the Server-Based CrowdNotifier backend to send encrypted tracing data to basic CrowdNotifier clients.

When records are stored in the clear, apps use the computed identities id to recognize tracing keys published by basic CrowdNotifier systems. Otherwise, when storing CrowdNotifier encrypted records, clients proceed as in *the basic CrowdNotifier scheme*.

6.6 Initiating Presence Notification

In Server-Based CrowdNotifier, presence notification is initiated by an index case that has been tested positive for SARS-CoV-2. We assume that the user has an upload authorization and that the *Backend Server* knows the corresponding contagious window.

The app and server proceed as follows:

1. The app sends its upload authorization to the *Backend Server* to obtain the corresponding contagious window.
2. For each record corresponding to this contagious window, the app uploads: the (possibly rounded) arrival and departure times, the pre identity preid, the notification key notifykey, and the tuples

$$(\mathsf{intervalLength}, \mathsf{intervalStart}, \mathsf{timekey}).$$

3. The *Backend Server* validates the uploaded data. In particular, it checks that:
   - All reported visits fall within the user's contagious window as established by the *Health Authority*.

- Individual records are not too long (e.g., at most the maximum duration allowed by the app)
- Validates that the reported tuples $(\mathsf{intervalLength}, \mathsf{intervalStart})$ are correctly formed and the corresponding interval overlaps the reported visit times for the corresponding record.
- That the user does not report being in more than one place at the same time. To do so, the server checks that the time intervals covered by the records do not overlap. Or, in case the app reports rounded interval lengths do not overlap more than what would be allowed because of time grunularity.

Optionally, if the *Backend Server* applies a heuristic to determine when to trigger a *Location* it can store and filter the uploaded data before proceeding to the next step.

4. The *Backend Server* then proceeds as follows for each uploaded (or selected) record.
   1. It uses the pre identity preid, and corresponding tuples
      $$(\mathsf{intervalLength}, \mathsf{intervalStart}, \mathsf{timekey})$$
      to recompute the corresponding time-specific identities id for this record following *the process laid out for the basic scheme*
   2. For each of these identities id it computes the corresponding identity-based decryption key
      $$\mathsf{sk}_{\mathsf{id}} = \mathsf{IBE.KeyDer}(\mathsf{msk}_S, \mathsf{id})$$
      using its master secret key msk_S. Let $\mathsf{tr}_{\mathsf{id}} = (\mathsf{id}, \mathsf{sk}_{\mathsf{id}})$.
   3. The server now proceeds as in *basic CrowdNotifier* step E to compute tuples $(\mathsf{tr}_{\mathsf{id}}, \mathsf{day}, c_{\mathsf{notify}})$ where it instead uses the notification key $\mathsf{notifykey}$ provided by the client rather than recomputing it from scratch.
5. Regularly, the server publishes a shuffled batch of tuples $(\mathsf{tr}_{\mathsf{id}}, \mathsf{day}, c_{\mathsf{notify}})$.

The information that is uploaded to the backend server – the pre identity preid, the notification key $\mathsf{notifykey}$, and the values $\mathsf{timekey}$ – do not reveal to non-visitors any information about the *Locations* they correspond to. The *cryptographic procedure used to compute these* and the presence of a strong cryptographic seed in the QR codes ensures that without knowledge of the seed, these values are pseudo random.

The values $\mathsf{timekey}$ are time-slot specific. As a result, a malicious server can only compute identities id for the time slots reported by the app. The *basic CrowdNotifier scheme* instead relies on the *Location Owner* to validate the requested time slots to protect against malicious servers.

We assume that apps use cover traffic to hide from network observers that a user has been diagnosed with COVID-19. When Server-Based CrowdNotifier is combined with a GAEN-enabled app, this dummy traffic should be aligned so as not to trivially reveal real uploads. We refer to the DP-3T best practices document [DP3TTeam] for more details on how to do this.

6.7 Presence Tracing and Notification

The records published by the server have exactly the same format as in the basic CrowdNotifier scheme. These records will enable apps to decrypt the encrypted records, as they contain the correct identity-based decryption keys corresponding to the QR codes that these clients scanned. So notification will proceed *exactly as for the basic scheme*.

Since Server-Based CrowdNotifier clients store more extensive records, they can avoid the trial decryption step. These apps proceed as follows.

1. The app downloads all $(\mathsf{tr}_{\mathsf{id}}, \mathsf{day}, c_{\mathsf{notify}})$ tuples that were published since the last time it checked. Let $\mathsf{tr}_{\mathsf{id}} = (\mathsf{id}, \mathsf{sk}_{\mathsf{id}})$.

2. The app checks if any of the records it stored contain the identity id. If so, the app uses the stored notifykey in that record to decrypt c_{notify} and recover:

$$\text{notificationdata} = (\text{entry'}, \text{exit'}, m)$$

   if decryption fails, it moves on to the next matching tuple.

3. The app compares the reported visit times entry' and exit' with the visit times it stored. If there is an overlap it notifies the user using the recovered message m.

6.8 Security and Privacy Analysis

We provide an analysis of the privacy properties of Server-Based CrowdNotifier. We refer to the white paper [LGV+] for a detailed description of the properties we refer below as PUX, PLX, or SX.

6.8.1 Privacy of Users

We first consider privacy of users. Like in the Basic CrowdNotifier, there is never any collection of personal data at a location (ensuring PU2). There is no network traffic related to notifications, and thus no adversary can learn who is notified based on network traffic (ensuring PU4). Privacy of positive status is protected from network adversaries by dummy uploads using the methods described in [DP3TTeam] (ensuring PU5).

If records are stored in the clear, as described above, PU3 is not fulfilled. Below, we describe a modification which enables users to store records that do not directly reveal the locations' identifier stored on the phone. This ensures that the Server-Based CrowdNotifier records stored on a user's phone do not reveal a user's visits (ensuring PU3).

Finally, regarding central collection of data (PU1), in Server-Based CrowdNotifier there is no explicit central collection of visitor data. However, some information about users' might be deduced based on the interactions of the system. These leaks are *inherent* to the fact that in Server-Based CrowdNotifier, to enable fast notifications, users upload their visited locations in a single group, and mixed batches of such groups are published without large delays.

For our analysis we separate two adversaries: the *Backend Server* and other users.

Adversarial Backend Server

The *Backend Server* receives all uploaded information from a single positive user in a single group. The *Backend Server* can derive the following information from these uploads.

1. If the *Backend Server* can map identifiers (or QR codes) to real locations, the backend can learn groups of locations visited by positive users. If the system is deployed with a registration service for venues, the backend would know all identifier-location pairs.
2. From the timestamps in the records uploaded, the *Backend Server* can learn temporal patterns about positive users (e.g., whether users work morning shifts or work night shifts).
3. As uploaded location identifiers are shared among users, the server can learn co-locations among positive users at a location.

Whether in the previous attacks the *Backend Server* can map users to real identities depends on whether users communicate anonymously with the *Backend Server*. Re-identification can also happen if the groups of locations can only be associated to one or few users. To reduce the power of this attack, we recommend that users are given the capability to redact the traces they upload to skip compromising or identifying locations.

We discuss below mechanisms to mitigate these attacks.

Adversarial Users

The *Backend Server* regularly broadcasts batches of data uploaded by positive users. An adversarial user (or anyone else) can uses these public batches to try to learn information about the visit patterns of positive users. This adversary cannot associate records in the published batches to individual users because the *Backend Server* anonymizes records before broadcasting them.

Published records consists of a time-specific location identifier and other cryptographic information. Records that the adversary cannot map to real locations, e.g., because the adversary doesn't know the corresponding QR code, provide very little information. At best, the adversary can detect the existence of high-risk events because the same identifier is reported more than once. The adversary, however, cannot associate these repeated identifiers to a location, nor to a specific time slot.

The adversary can learn more information about published batches if it can map records to real locations. To do so, it can use the information contained in that location's QR code. It can obtain these QR codes by visiting these locations either individually, or crowd-sourcing the visits to a group of collaborators.

Given these QR codes, the adversary can try to recover partial location traces. For each batch of released identifiers, it looks up the corresponding visited locations and visit times. The adversary can then use geographic information (where locations are) and timing (when they were visited), to try to reconstruct potential location traces. And from these traces re-identify positive users. These attacks are easier to do when the location traces published in the same batch do not mix (e.g., the batch contains visits from one user that lives in Zurich, and one set of visits from a user in Lausanne).

There are several options to mitigate this attack:

1. Ensure that QR codes of *Locations* are rotated frequently to make collecting QR codes much harder. We expect this to be the case for private events, where QR codes are one-use.
2. Release tracing information in larger batches, to decrease the probability of identifying the underlying location traces. This would delay the publication of traces and therefore the notifications, reducing the advantage of Server-Based CrowdNotifier over the basic protocol.
3. Apply a filter on published location data to only release the urgent (e.g., reported more than once) locations; or only those in which the risk of transmission is high (e.g., release bars, but not a seated dining with social distancing).

6.8.2 Privacy of Locations

The following properties are shared between the Server-Based CrowdNotifier and basic CrowdNotifier schemes with respect to *Locations*: Non-visitors (that do not collude with visitors) cannot recognize the broadcasted information, and thus cannot determine which locations where notified (ensuring PL1). To ensure location privacy with respect to non-contemporary visitors (PL2), locations must frequently rotate their QR code. Server-Based CrowdNotifier does not require a database of locations (PL3 achieved), and does not require uploads by locations (PL4 achieved).

6.8.3 Security

We focus on abuse prevention properties: prevention of fake notification for users (S1) and prevention of notifications targeting a particular location (S2). The basic CrowdNotifier scheme requires cooperation of the *Location Owner* and the *Health Authority* to trigger notifications. Server-Based CrowdNotifier has less strict protections.

First, the *Backend Server* can trigger notifications for any *Location* for which it can obtain the QR code at that *Location*.

Second, when uploaded traces by index cases are not validated, malicious users might add arbitrary visits to their uploads; either by reporting different visit times, or by reporting locations that they

did not visit (using QR codes they obtained elsewhere). This opportunity can be used by malicious users (or the *Backend Server*) to target visitors of particular locations.

To mitigate the second attack, we recommend to sanity check uploads and to limit both the number of reported visits as well as their duration.

6.9 Privacy enhancements

6.9.1 Improving privacy of records in the app

As explained in the CrowdNotifier white paper [LGV+], users might need strong privacy properties of the records stored on their phone. In the variant explained above, an adversary with access to the user's phone and the location records stored therein, and who has access to the QR code of the *Location* visited by that user, can easily determine where users went. In the basic CrowdNotifier scheme this attack does not work.

The Server-Based CrowdNotifier scheme admits an easy modification that strengthens privacy of records on the phone. To do so, clients use the basic CrowdNotifier approach and store an encrypted record for their visit. Only when the server generates correct tracing keys can this record be decrypted.

To enable notification of other users, Server-Based CrowdNotifier requires clients to store other data – the pre identity preid, the notification key notifykey and time-slot specific keys timekey. To protect these, the client could encrypt them against the *Backend Server*'s public key before storing them. This approach comes at the cost of clients not being able to redact these records anymore before uploading them.

We point out that that in some deployment scenarios, this protection is limited. A determined attacker can use the *Backend Server* as a decryption oracle to recover, say, preid, and thus determine a user's location visits after all.

Therefore, we recommend that, if possible, clients store only the encrypted basic CrowdNotifier records for sensitive visits. This enables them to receive notifications at no privacy risk.

6.9.2 Improving privacy of users towards the Backend server

For the privacy attacks on users carried out by the server to be effective, the server needs to be able to map uploads to real identities. A strong defense for users is to use anonymous communications systems when uploading information in order to hide their IP address from the *Backend Server*.

By hiding their network identity, users limit the impact of the attack to cases for which (i) the *Backend Server* has knowledge of the QR codes of more that one of the locations visited by the user (which will rarely include private events whose keys are used only once) and (ii) those locations are enough information to re-identify the user.

To further reduce the re-identification capability, system deployments are encouraged to include redaction to let users remove identifying locations from their upload list.

For co-location attacks, which would be possible even if the adversary does not know the location in which users have been present, the use of anonymous communication renders the attack useless: the *Backend Server* learns that two or more users were at the same location, but not whom.

One could be tempted to use dummy check-ins to try to prevent the *Backend Server* from learning the locations visited by users. However, the use of dummies does not help against an adversary that has access to pairs of real locations and their QR codes. This adversary can use her knowledge to filter out dummy check-ins (the adversary removes check-ins that do not correspond to any known QR). If the adversary cannot associate a check-in to a QR code, then there is no privacy risk to start with as the adversary cannot identify the corresponding location.

CHAPTER

SEVEN

IDENTITY-BASED ENCRYPTION

CrowdNotifier uses a specific identity-based encryption scheme, to ensure all security and privacy properties. In particular, CrowdNotifier uses a slight modification of the FullIdent Boneh-Franklin scheme [BF01] given by the following algorithms. (The only modification is that the randomness r now also depends on the identity id, which is passed to IBE.Dec for verification purposes.)

7.1 Mathematical description

- $\mathsf{pp} \leftarrow \mathsf{IBE.CommonSetup}(1^\ell)$. On input of security parameter ℓ, generate a type III set of bilinear groups G_1, G_2, G_T generated by respectively g_1, g_2, g_T all of prime order p and let $e : G_1 \times G_2 \to G_T$ be the corresponding pairing. Generate the following hash-functions (modeled as random oracles): $H_1 : \{0,1\}^* \to G_1^*$ a hash function mapping points to the group G_1^*, $H_T : G_T \to \{0,1\}^{2\ell}$ mapping group elements from the target group, $H_3 : \{0,1\}^{2\ell} \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^{2\ell}$, and $H_4 : \{0,1\}^{2\ell} \to \mathcal{K}$ mapping into the key-space of the authenticated encryption scheme. Setup outputs $\mathsf{pp} = ((G_1, G_2, G_T, g_1, g_2, g_T, p, e), H_1, H_T, H_3, H_4)$.
- $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{IBE.KeyGen}(\mathsf{pp})$. Pick random $\mathsf{msk} \leftarrow \mathbb{Z}_p$ and set $\mathsf{mpk} = g_2^{\mathsf{msk}} \in G_2$. Return $(\mathsf{mpk}, \mathsf{msk})$.
- $\mathsf{sk}_{\mathsf{id}} \leftarrow \mathsf{IBE.KeyDer}(\mathsf{mpk}, \mathsf{msk}, \mathsf{id})$. On input of a master public key mpk, a master private key msk, and an identity $\mathsf{id} \in \{0,1\}^*$; outputs private key $\mathsf{sk}_{\mathsf{id}} = H_1(\mathsf{id})^{\mathsf{msk}} \in G_1$.
- $\mathsf{ctxt} \leftarrow \mathsf{IBE.Enc}(\mathsf{mpk}, \mathsf{id}, m)$. On input of a master public key mpk, an identity $\mathsf{id} \in \{0,1\}^*$, and a message m, proceed as follows. Check that $\mathsf{mpk} \in G_2^*$. Pick a random key $x \leftarrow \{0,1\}^{2\ell}$ and compute

$$\begin{aligned} c_1 &= g_2^r, \\ c_2 &= x \oplus H_T(e(H_1(\mathsf{id}), \mathsf{mpk})^r), \\ c_3 &= \mathsf{AE.Enc}(H_4(x), m) \end{aligned}$$

  where $r = H_3(x, m, \mathsf{id})$. Return $\mathsf{ctxt} = (c_1, c_2, c_3)$.
- $m \leftarrow \mathsf{IBE.Dec}(\mathsf{id}, \mathsf{sk}_{\mathsf{id}}, \mathsf{ctxt})$. On input of an identity id, a private key $\mathsf{sk}_{\mathsf{id}}$, and a ciphertext ctxt, proceed as follows. Parse ctxt as (c_1, c_2, c_3) and return $\perp$ if parsing fails. Check that $\mathsf{sk}_{\mathsf{id}} \in G_1^*$, and compute $x' = c_2 \oplus H_T(e(\mathsf{sk}_{\mathsf{id}}, c_1))$ and $m' = \mathsf{AE.Dec}(H_4(x'), c_3)$. Return $\perp$ if $m' = \perp$. Finally, compute $r' = H_3(x', m', \mathsf{id})$ and check that $c_1 = g_2^{r'}$. If this check fails, return $\perp$, otherwise, return m'.

7.2 Implementation of IBE in CrowdNotifier

The following is a list of pseudo-codes describing how the above methods can be implemented:

- `IBE.CommonSetup` using mcl.init:

```
mcl.init(mcl.BLS12_381)
```

- `IBE.KeyGen`:

```
msk = new mcl.Fr();
msk.setByCSPRNG();

mpk = mcl.mul(baseG2, msk);
```

- `IBE.KeyDer(msk, id)`:

```
skid = mcl.mul(h1(id), msk)
```

- `IBE.Enc(mpk, id, m)`:

```
x = randombytes_buf(NONCE_LENGTH);

r = h3(x, id, m);
c1 = mcl.mul(baseG2, r);

c2_pair = ht(mcl.pow(mcl.pairing(h1(id), mpk), r));
c2 = xor(x, c2_pair);

nonce = randombytes_buf(crypto_secretbox_NONCEBYTES);

c3 = crypto_secretbox_easy(m, nonce, h4(x));
```

- `IBE.Dec(id, skid, ctxt = (c1, c2, c3))`:

```
x_p = xor(c2, ht(mcl.pairing(skid, c1)));
msg_p = crypto_secretbox_open_easy(c3, nonce, h4(x_p));
// or return _I_ on error

r_p = h3(x_p, id, msg_p);
c1_p = mcl.mul(baseG2, r_p);

if (!c1.isEqual(c1_p)) {
  return _I_;
}

// Check that skid is in G1*
if (!skid.isValidOrder() || skid.isZero()) {
  return _I_;
}

return msg_p;
```

With the following helper methods:

- `h1(id)`:

```
h1 = mcl.hashAndMapToG1(id)
```

- `h3(x_p, id, msg_p)`:

```
h3 = new mcl.Fr();
// The '+' concatenates the binary arrays
h3.setHashOf(x_p + id + msg_p);
```

- `h4(id)`:

```
h4 = crypto_hash_sha256(id)
```

- `ht(gt)`:

```
ht = crypto_hash_sha256(gt.serialize())
```

- `xor(a, b)`:

```
c[i] = a[i] ^ b[i]
```

- `baseG2`:

```
baseG2 = new mcl.G2();
baseG2.setStr(
       '1352701069587466618187139116011060144890029952792775240219908 64' +
       '4239793785735715026873347600343865175952761926303160 ' +
       '3059144344244213709971259814753781636986470325476647558659373 20' +
       '6291635324768958432433509563104347017837885763365758 ' +
       '1985150602287291935568054521177171638300868978215655730859378 66' +
       '5066344726373823718423869104263333984641494340347905 ' +
       '9275536654923324557472019657760378807577401934535929700250279 78' +
       '7939768770026755649809492897279575655754333442195 82');
```

7.2.1 Serialization of Keys

The master public key `mpk` consists of a single element in G_2. We serialize it using the `mcl` serialization function.

BIBLIOGRAPHY

[BF01] Dan Boneh and Matthew K. Franklin. Identity-Based Encryption from the Weil Pairing. In *CRYPTO*. 2001.

[LGV+] Wouter Lueks, Seda Gürses, Michael Veale, Edouard Bugnion, Marcel Salathé, Kenneth G. Paterson, and Carmela Troncoso. CrowdNotifier: Decentralized privacy-preserving presence tracing. White Paper, Version February 5, 2021. URL: https://github.com/CrowdNotifier/documents/blob/main/CrowdNotifier%20-%20White%20Paper.pdf.

[LGV+21] Wouter Lueks, Seda Gürses, Michael Veale, Edouard Bugnion, Marcel Salathé, Kenneth G. Paterson, and Carmela Troncoso. CrowdNotifier: Decentralized Privacy-Preserving Presence Tracing. 2021. Under submission.

[RBC21] Vincent Roca, Antoine Boutet, and Claude Castelluccia. The Cluster Exposure Verification (Cléa) Protocol: Specifications of the Lightweight Version. 2021. https://hal.inria.fr/hal-03146022.

[DP3TTeam] DP-3T Team. Best practices: operational security for proximity tracing. URL: https://github.com/DP-3T/documents/blob/master/DP3T%20-%20Best%20Practices%20for%20Operation%20Security%20in%20Proximity%20Tracing.pdf.

INDEX

B

Backend Server, 3

H

Health Authority, 3
Health Authority Backend, 3

L

Location, 3
Location Owner, 3

O

Organization, 3

V

Visitor, 3