Decentralized Proximity Tracing Interoperability Specification

Release 0.1 (draft)

DP3T Team

May 18, 2020

CONTENTS:

1 Introduction **1**
- 1.1 Objectives . . . 1
- 1.2 Terminology . . . 2
- 1.3 Notation used . . . 2

2 System overview **3**
- 2.1 Setting . . . 3
- 2.2 High Level Design . . . 3
- 2.3 Deploying the System . . . 4
- 2.4 Structure of the Specification . . . 5

3 Proximity Tracing Feeds **7**
- 3.1 Proximity Tracing Feed API . . . 7
- 3.2 Producing a Proximity Tracing Feed . . . 9
- 3.3 Consuming a Proximity Tracing Feed . . . 13

4 Reporting Component **17**
- 4.1 Configuration . . . 18
- 4.2 Operation . . . 18
- 4.3 Deployment Considerations . . . 19

5 Publication Component **21**
- 5.1 Configuration . . . 22
- 5.2 Operation . . . 22
- 5.3 Deployment Considerations . . . 23

6 Interoperable Applications **25**
- 6.1 Configuration . . . 25
- 6.2 Internal storage . . . 25
- 6.3 Tracking Active Regions . . . 26
- 6.4 Storing received Ephemeral Bluetooth Identifiers . . . 26
- 6.5 Upload of report after infection is confirmed . . . 26
- 6.6 Computing Exposure Risk . . . 27

7 Indices and tables **29**

Index **31**

CHAPTER

ONE

INTRODUCTION

This document provides a technical specification for how decentralized proximity tracing systems can interoperate with each other.

Manual proximity tracing aims to determine who has been in close physical proximity to a COVID-positive person in order to warn them so that they can take appropriate precautions. The digital version aims at complementing the manual process by simplifying and accelerating the process of notifying people who have been in contact with an infected person.

In digital proximity users continually run a smartphone app that broadcasts an ephemeral, pseudo-random ID representing the user and also record pseudo-random IDs observed from smartphones in close proximity. Whenever a patient is diagnosed for COVID-19, she uploads data representing the broadcasted pseudo-random IDs from her phone to a central server. This step should only be done with the approval of a health authority and the explicit permission of the individual. Prior to the upload, all data remains exclusively on the user's phone. Other users receive all data published by COVID-positive patients and can use it to locally compute whether the app's user was in physical proximity to an infected person and the risk that an encounter led to a propagation of the virus. In case the app detects the user is at-risk of being infected, it will show a notification to the user.

In this document we describe a technical specification for interoperation designed from the ground up to be secure, scalable and straightforward to implement. The document describes the necessary steps the operator of a proximity tracing system should take in order to interoperate with other decentralized systems which may differ in implementation. The technical specification describes requires for both smartphone applications and central servers provided by operators.

1.1 Objectives

In this document, we specify how inter-operating systems can share proximity tracing information so that users can benefit from proximity tracing regardless of location. We assume the world is divided into geographical regions and on each region there is a contact tracing system. This regions are not necessarily disjoint. Operators of contact tracing systems have three responsibilities:

- Sharing relevant ephemeral identifiers of their users with other systems
- Ensuring availability of ephemeral identifiers to users of other contact tracing systems visiting their region
- Ensuring user's smartphone applications can record each other's Bluetooth transmissions and calculate their risk score correctly.

This document contains a technical specification necessary to perform the above tasks in a secure, scalable, and simple manner.

1.2 Terminology

The following terminology is used throughout this document:

Decentralized Proximity Tracing System A system for performing proximity tracing which performs risk computation entirely on the user's device, rather than on or in conjunction with a central server.

App A smartphone application which is part of decentralized proximity tracing system *users* install on their mobile phones, see *here* for more details.

User A user of the app, who would like to learn about past exposure to COVID-19 positive others.

Tracing Keys Information uploaded by a COVID-19 positive or COVID-19 suspected individual in order to enable proximity tracing to take place.

Proximity Tracing Feed An API endpoint that publishes batches of *Tracing Keys* for consumption by interoperating backends and users' *applications*.

Operator An entity providing a decentralized proximity tracing system to a population of users in a region.

Region A geographic area served by an operator. It may overlap with regions serviced by other operators.

1.3 Notation used

1.3.1 Text Syntax

- `This` is a variable or inline code fragments
- $1 + 2 = 3$ this is a mathematical formula
- The following is a code example

```
{
 "foo": "bar"
}
```

1.3.2 Highlights

Warning: This warning markup is used to highlight security-critical aspects of the implementation.

CHAPTER

TWO

SYSTEM OVERVIEW

2.1 Setting

We assume that one or more *operators* have developed (and possibly deployed) decentralized proximity tracing systems. These operators offer an application to users in a particular region, integrated with local healthcare authorities and public health officials. The operator provides the necessary infrastructure to support the system.

In decentralized proximity tracing systems, COVID-19 positive users provide (authenticated) *tracing keys* which are distributed to all other users. *Operators* provide this functionality for a particular *region*. Users can travel outside their operator's region of service, e.g. due to tourism, business or other considerations. Consequently, operators must interoperate in order to support cross-region and cross-application proximity tracing. This involves three distinct requirements:

1. An operator's application can successfully record proximity information broadcast by applications developed by other operators.
2. *Tracing keys* provided by an operator's COVID-19 positive users can be distributed to users of other contact tracing systems.
3. An operator's users can be informed of relevant *tracing keys* from users of contact tracing systems the user visits without delay.

In this technical specification, we describe how these requirements can be achieved on a technical level. However, we remark that operators that wish to interoperate with each other must agree to do so at a policy level. In particular, for two operators to successfully interoperate, they must mutually trust each other to:

- Authenticate new *tracing keys* from their users, and follow an agreed standard in annotating the diagnosis method used to confirm the Sars-CoV-2 diagnosis.
- Distribute in a timely manner the new *tracing keys* uploaded by their local users to operators of other relevant regions.

These policy considerations are out of scope for this document. We assume that an agreement has been made on the above aspects and describe how such an agreement can be implemented.

2.2 High Level Design

Operators who deploy decentralized proximity tracing systems also deploy applications and backend systems to support them. For *applications to interoperate*, they:

- should provide the means to keep track of which regions a user has visited,
- should monitor for new tracing keys from regions relevant to the user,
- should record Bluetooth transmissions from any proximity tracing application,

- should evaluate risk from exposure to diagnosed users of both their own and other proximity tracing applications.

For backend systems to interoperate, they must satisfy two distinct roles.

- Sharing of local Infected users' Tracing keys with other *operators* if required (see Section *Reporting Component*)
- Polling of relevant keys from other *operators'* Infected users, and distribution to the local *region* (see Section *Publication Component*).

The following diagram illustrates this process:

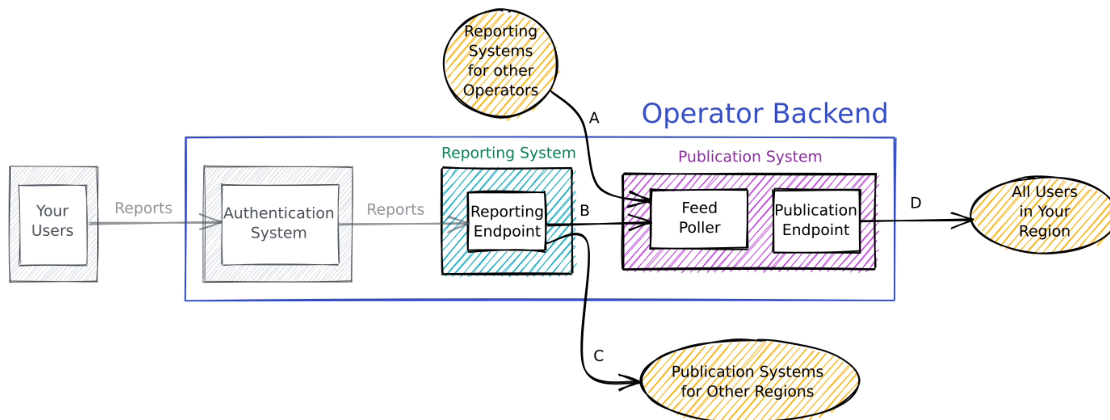

Fig. 1: System Architecture.

The diagram shows the components of an operator's backend systems. We assume the gray components, e.g., report authentication, have been developed and integrated with the local healthcare system according to the policies of that operator's region. In orange circles we highlight entities that may or may not be controlled by the operator. For example, the backends of other operators or the applications deployed by other operators.

An interoperable backend systems works as follows:

1. It collects tracing keys from its own users.
2. The *reporting component* takes these tracing keys and provides them to the relevant other backends (arrow C). If a user indicates to the operator that they visited a region, the reporting component will provide the tracing keys of that user to the corresponding backends of that region.
3. The backend's *publication component* takes as input the tracing keys from its own users (arrow B) as well as the tracing keys provided by other backends of any users that visited the operator's region (arrow A)
4. The backend's *publication component* provides this combined set of tracing keys to all users, both those using the operator's own app, and visiting users using other apps. (Arrow D).

All these information flows, including arrows arrows A, C, and D that cross operator boundaries, transfer *tracing keys* between entities. Therefore, to make interoperability between operators possible, this specification *specifies a minimal API and dataformat to transfer these keys*. We call the mechanism by which these keys are transferred a *proximity tracing feed*.

2.3 Deploying the System

This specification, in so far as is possible, avoids prescribing particular implementation or architecture requirements. Different operators may want to deploy monolithic applications, containerized micro services or completely serverless

solutions. Further, different operators will have dramatically different scaling requirements based on the size of the populations they serve.

We have also made explicit consideration for the performance challenges of contact tracing at scale. In particular, the specification is largely agnostic about the exact contact tracing protocol used, for example, possible differences in key derivation, Bluetooth layer or risk calculation. Where constraints are imposed, they are strictly necessary in order to achieve security, privacy or correctness requirements.

We also highlight some high level considerations that operators should be aware of when configuring aspects of their solution.

2.3.1 Regional Assignments

We largely expect operators to server national populations and work in concert with official organizations. Consequently, it is likely there most systems will consist of one application, serving one country, with one backend system. However, more complex arrangements are supported within our framework. For example, a larger country divided into multiple regions, or several official applications with one country.

Further, we do not require different operators to agree on precise regional boundaries. It is entirely possible for regions to overlap without affecting the correctness of the application. In general, users near regional boundaries (e.g. land borders between two countries) would install the application of supported by their local healthcare provider of choice and would select both countries as their 'default regions' in the application and face no further hassle. They would be able to participate in contact tracing or exposure notification with individuals from either region (e.g. country).

In general, the more population a region covers, the larger the dataset that each user must download. However, even for large populations, e.g. X, this not anticpated to be above Y per user per day. Further, larger regions are easier for users to manage, as they will cross regional boundaries more rarely. Additionally, this reduces the ability of the operator to infer the user's location, as they only see which regions a user has visited, so larger regions provide more privacy. Consequently, we recommend regions typically be assigned to cover whole countries, or up to roughly 100 million users.

2.4 Structure of the Specification

This specification is divided into the following high level parts:

1. We describe how to *implement tracing key feeds*, including the necessary API endpoints and wire formats. Both reporting and publication components expose report feeds. Both publication components and applications consume report feeds.
2. We describe how to *implement a reporting component*. This includes how to handle authorization of uploads of new tracing keys by users, and how to expose this information to publication components.
3. We describe how to *implement a publication component for a region*. It includes how to poll a set of reporting components for new tracing keys, how to aggregate them and how to expose them to user applications.
4. We describe how to *implement an interoperable application*. This includes how apps should keep track of relevant regions, how they poll the publication components of those regions, and how they process the tracing keys that these components provide to compute an exposure score.

CHAPTER

THREE

PROXIMITY TRACING FEEDS

There are two types of proximity tracing feeds: public feeds, to provide tracing information to users of contact tracing systems; and backend feeds, to exchange tracing information between backends. Backend feeds are identical to public feeds but support authorization of downloads via TLS Client Authentication and support notification (via webhooks) to consumers when new data is available.

We first describe the *proximity tracing feed API*, and then describe how to *produce* and *consume* such a feed.

Proximity tracing feeds are produced by *reporting components* and *publication components* and consumed by *publication components* and interoperable *applications*.

3.1 Proximity Tracing Feed API

Proximity tracing feeds necessarily include protocol specific data. To illustrate the inclusion of such data in this specification we use the DP3T low-cost (`dp3t-lowcost`) and the Google/Apple (`gaen`) proximity tracing protocols as examples.

Proximity tracing feeds periodically add new tracing keys, which are added as discrete batches of one or more tracing keys. Each batch is sequentially numbered, so that clients can easily fetch discover and fetch any missing batches. Each proximity tracing feed endpoint is specific to a particular protocol format, denominated `{prot}`. This ensures that consumers of the feed do not parse protocol specific information that they do not understand.

A proximity tracing feed is provided at a particular base URL with the following API:

`GET /v2/{prot}/latest` A pointer to the latest batch and a recommendation of when to query next

`GET /v2/{prot}/exposed/{batchnr}` A batch of tracing keys released as part of the proximity tracing feed

In order to query these endpoints, the client must know the base URL of the service. All endpoints must be secured with TLS, we provide further details later in this section.

Depending on the feed provider's configuration, the feed contents may optionally be signed and the provider may require authentication from clients before they connect. We discuss how providers should *sign feed contents.* and enable clients to *clients to discover signature verification keys.* in self-contained sections. The *authentication of clients is discussed later.*.

3.1.1 Endpoint: `GET /v2/{prot}/latest`

Clients need a way to discover which lists they should download and how often they should query this endpoint. Therefore, for each supported protocol, every backend supplies an endpoint `/v2/{prot}/latest` to provide the index of the latest batch (`latestBatchId`) as well as a recommended time on when to pull next, i.e., the publication time of the next batch (`recommendedNextPollTime` in seconds since UNIX Epoch). These data are provided using the JSON format:

```
GET /v2/{prot}/latest HTTP/1.1
Accept: application/json

{
  "latestBatchId": 123456,
  "recommendedNextPollTime": 1589304661000,
}
```

3.1.2 Endpoint: `GET /v2/{prot}/exposed/{batchId}`

Batches are downloaded from `/v2/{prot}/exposed/{batchId}`. Clients query this endpoint as follows:

```
GET /v1/exposed/<version>/<batchIndex> HTTP/1.1
Accept: application/x-protobuf
```

where `<batchIndex>` is an integer. The server replies with:

```
HTTP/1.1 200 OK
Content-Type: application/x-protobuf

<binary data>
```

Where the binary data is the corresponding protobuf encoded data. The specific format depends on the protocol, but each protocol must follow the following structure:

```
syntax = "proto3";

message GenericExposedList {
  int64 batchReleaseTime = 1
  repeated ProtocolSpecificTracingKey exposed = 2;
}

message ProtocolSpecificTracingKey {
  int64 validBeforeTime = 10;
  optional KeyType type = 11;
  /** Other fields to specify the keys for tracing omitted **/
}

enum KeyType {
  TEST_DIAGNOSED = 0;   // Default value
  DOCTOR_DIAGNOSIS = 1;
  SELF_DIAGNOSED = 2;
  CANCELLED = 3;
}
```

In this format, *batchReleaseTime* is the time (in seconds since UNIX Epoch Time) the batch was released.

It also contains a list of tracing keys. The `validBeforeTime` (in seconds since UNIX epoch) indicates the last time that observations matching these tracing data should be taken into account. The `validBeforeTime` is used to prevent active attacks to cause false at-risk events once tracing data has been made public. The `KeyType` field indicates the type of diagnosis corresponding of the patient corresponding to these tracing data:

- `TEST_DIAGNOSED`: for tracing data from Covid Positive Patients that have officially been diagnosed by a health authority following a laboratory test.
- `DOCTOR_DIAGNOSED`: as for *TEST_DIAGNOSED* but the health authority has determined that this patient was likely infected based on an interview and described symptoms.

- SELF_DIAGNOSED: for tracing data from a patient that self-diagnosed.
- CANCELLED: to cancel any of the previous diagnosis.

Example Dataformat for the low-cost DP3T protocol

The data format for the low-cost DP3T protocol additionally includes information for the validity of keys:

```
syntax = "proto3";

message DP3TLowCostExposedList {
  int64 batchReleaseTime = 1
  repeated DP3TLowCostTracingKey exposed = 2;
}

message DP3TLowCostTracingKey {
  bytes key = 2;
  int64 keyDate = 3;
  int64 validBeforeTime = 10;
  optional KeyType type = 11;
}
```

In particular, keyDate designates the start of the corresponding UTC day (in seconds since UNIX Epoch) from which the supplied key is valid.

Example Dataformat for the Google/Apple protocol

Draft

This format has not yet been finalized

The Google/Apple protocol instead uses per-day keys that have an associated day counter (rollingStartNumber):

```
syntax = "proto3";

message GAENExposedList {
  int64 batchReleaseTime = 1;
  repeated GAENTracingKey exposed = 2;
}

message GAENTracingKey {
  optional bytes key = 2;
  optional uint32 rollingStartNumber = 3;
  int64 validBeforeTime = 10;
  optional KeyType type = 11;
}
```

3.2 Producing a Proximity Tracing Feed

We now describe how to produce a feed. We first describe the required setup and internal state, and then explain how new batches are produced.

Backend feeds are identical to public feeds but support authorization of downloads via TLS Client Authentication and support notification of new data via webhooks.

3.2.1 Configuration

To produce a proximity tracing feed, the server must be configured with the following information:

Format The protocol format. For now only `dp3t-lowcost` and `gaen` are supported.

Publication Schedule A schedule for when to publish the next batch. For example: every 2 hours, starting at UTC midnight.

JWT Signing Keys (when using) "RS256" JWT signing key to sign responses when *using a CDN*. The corresponding verification keys should be *available to clients so they can verify the responses.*

Additionally, when producing a **backend feed** the server must also keep track of:

ClientCerts A list of client certificates or corresponding CA(s) that authenticates clients that are allowed to query the backend feed.

List of Webhook URLs (when using) Backend feeds may additionally be configured with a list of webhook URLs that should be called when a new item is available.

The server producing a backend feed must be configured to require TLS Client Certificate Verification against the list of `ClientCerts`.

3.2.2 Internal state

The server must keep track of the following internal state:

Sequence Counter A monotonically increasing sequence counter of published batches

Tracing Information A list of tracing information. Each item must contain at least the following information:

- `tracingData` the information to enable the tracing
- `validBeforeTime` the time before which this key should be considered valid in seconds since UNIX Epoch
- (Optional) `KeyType` as per the protobuf specification above.

3.2.3 Publishing a new batch

The server publishes a new batch according to the publication schedule. Batches are constructed as follows:

1. Gather all Tracing Information that should be published in this batch. Items for which the `validBeforeTime` is in the future should not yet be published.
2. Sort the tracing information by `validBeforeTime`, from smallest to largest.
3. Set the `batchReleaseTime` to the current time, and set `batchId` to the current sequence counter. Increase the sequence counter.
4. Depending on the configure format, create a protobuf file containing all the tracing information. If the format equals `dp3t-lowcost` use `DP3TLowCostExposedList`, if it equals `gaen` use `GAENExposedList` instead.
5. (When using) *Compute the signature on the protobuf file.* and ensure that the response `Signature` header includes the signature.

6. Publish the file.
7. Update the `/v2/{prot}/latest` endpoint to point to the latest batch and the next release time.
8. (When using) *Notify any webhooks of the newly released batchId.*

3.2.4 Deleting Old Batches

After the maximum tracing window (e.g. 14 days) expires, old batches can safely be deleted from the feed. This is as simple as removing the static the file.

3.2.5 Hosting Endpoints

Feed endpoints must be protected with TLS. Endpoints should reject any connection which does not use TLS. Some endpoints may face significant incoming traffic either from legitimate users or from malicious parties and consideration should be given to scaling and denial of service prevention. We provide more detailed deployment considerations for particular systems later in the specification.

> **Warning:** All connections to the endpoints below must be made through HTTPS with TLS>= 1.2, with AEAD (Authenticated Encryption with Associated Data) cipher suites (CHACHA20_POLY1305, GCM and CCM) or PFS (Perfect Forward Secrecy) ciphers (ECDHE_RSA, ECDHE_ECDSA, DHE_RSA, DHE_DSS, CECPQ1 and all TLS 1.3 ciphers). Public keys should be RSA-2048 bit or stronger, or ECDSA-256 or stronger. Any other configuration or HTTP connection should not be accepted. Server certificates must be signed by a suitable root CA. Endpoints should reject any connection not made over a connection meeting the above criteria.

3.2.6 Verifying TLS Client Certificates (Backend Feeds only)

Backend feeds must authenticate their clients using TLS client certificates to ensure that only authorized clients can access the feed. TLS Client Certificate Authentication is widely supported by both standard hosting software (e.g., Apache and NGINX) and contemporary CDNs (e.g., Cloudflare and Azure).

Typically, Client Certificates are authorized by marking a particular client Certificate Authority (CA) as trusted. It may be that the feed host runs their own client CA and issues certificates to authorized readers. This involves creating the certificate authority and using it to sign TLS client certificates which are issued to other contact tracing systems.

Alternatively, the certificates may be issued by a 3rd party in order to reduce the number of authorization tokens that need to be exchanged. In this case, the third party would run the TLS Client Certificate Authority and sign each operator's certificate. This reduces the administrative burden on each participating feed backends.

It is also possible for a feed backend to mix and match between both systems some consumers are authenticated with the help of a third party and others are issued certificates directly.

3.2.7 Notifying webhooks (Backend feeds only)

Backend feeds may keep track of a list of webhooks to notify every time a new batch is released. When notified, *the client should then pull the batch as normal*. Support for this feature is optional but highly recommended for both feed providers and consumers.

When new information is added to the feed, the provider makes a POST request to the stored webhook URLs and sends the following JSON payload:

```
{
  "batchId": 123456,
  "baseURL": "https://{baseURL}/v2/{prot}/"
}
```

where `batchId` is the ID of the new batch and `baseURL` is the base URL of the publication service. The client should then *retrieve this batch* from `https://{baseURL}/v2/{prot}/exposed/{batchId}`.

3.2.8 Signing Feed Entries (Optional)

When feeds are served directly by the backend server, then standard TLS server-authentication ensures that the lists are authentic. However, for performance reasons we expect these feeds to be frequently served via a CDN or other middle party. In that case, the TLS connection is terminated by the CDN, and the CDN could modify the data to either remove tracing information, or insert non-authentic tracing information.

When using a CDN or other infrastructure not controlled by the operator, the provider should therefore additionally sign the lists. We propose the following mechanism based on standard JSON Web Tokens (JWTs). The server will use RSA with at least a 2048 bits modulus (JWT signing algorithm "RS256"). This algorithm is hard-coded. The public verification key is assumed to be public, and known to the app.

The JWT signs the hash of the response and the endpoint URL. In particular the server proceeds as follows:

1. It computes the SHA256 hash of the exact (binary) response body
2. It creates a JWT with header:

   ```
   {
     "alg": "RS256",
     "typ": "JWT"
   }
   ```

   And body containing a custom claim about the hash of the page's content:

   ```
   {
     "iss": "dp3t",
     "exp": 1590079929000,
     "content-hash": "pCmMHo4tfzaS3Zy1D+u2qSZGGNfPZKzwhI9MA4V/U+g=",
     "url": "https://somedomain.ch/v2/dp3t-lowcost/exposed/123456",
   }
   ```

   Where `url` is the requested endpoint, and `content-hash` is the base64 encoding of the SHA256 hash computed in step 1. The server then signs this header using its private signing key.
3. The server includes the JWT in the `Signature` field of the response.

Warning: The JWT must include an expiry time for resources (such as `latest`) that have a limited validity time.

This signature can be cached and reused for subsequent requests, provided the expiry time is sufficiently far in the future.

3.2.9 Offering signature verification keys (Optional)

When signing is in use by a provider, the signature verification keys must be authentically communicated to clients.

We specify two methods for clients to obtain the RSA verification keys used to create the JWTs: (1) hard-coding the signing keys, (2) retrieving them from a TLS protected endpoint. When using the second case, this domain must be under the direct control of the operator or a closely trusted party, so that TLS guarantees that the data provided by the endpoint is authentic.

The second option has the advantage that backends can update their signing keys. Hosts may choose to add and remove signing keys as it wishes. However, we recommend using at least one active key for signing reports, in addition to a backup key which can be switched over in the event a key is compromised or lost. Further, periodically rotating the signing keys is recommended.

Note that the listed verification keys should be sufficient to validate all available batches. Ideally, verification keys are only removed from this list when they are no longer needed to verify the signatures on any downloadable batches. If a verification key must be removed sooner, the operator should recompute the signatures on all batches that were originally signed with the removed key.

When using a TLS protected endpoint, the returned payload must be a JSON Web Key Set as defined in RFC 7517. Elements of the set are not ordered. Example:

```
{"keys":
  [
    {"kty":"RSA",
     "n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n3oknjhMs
tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbI
SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
w0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw",
     "e":"AQAB",
     "alg":"RS256",
     "use":"sig",
     "key_ops":["sign"],
     "kid":"2011-04-29"}
  ]
}
```

3.3 Consuming a Proximity Tracing Feed

Proximity tracing feeds are consumed by backends, for the purpose of sharing reports between regions, and clients, for the purpose of proximity tracing.

3.3.1 Configuration

Each feed consumer needs to know:

BaseURL The Base URL of the feed. For example: `https://example.com/example-name/`

Format The protocol format.

Verification keys (optional) When the feed signs batches, the consumer needs to know the verification keys to verify the batches. The consumer can either be configured with a list of valid signing keys, or a TLS protected URL from which new signing keys can be retrieved. See *Offering signature verification keys (Optional)*.

TLS Client Certificate (optional) For backend feeds only. The TLS client certificate that the consuming backend must use to authenticate itself to the producer.

A backend feed consumer may optionally provide the feed producer with a webhook URL to call when new information is available.

3.3.2 Internal state

For each feed, the consumer must keep track of the following internal state:

Last Batch Identifier The last batch identifier `lastBatchId` that it successfully retrieved from the feed.

Recommended Next Poll Time The time `recommendedNextPollTime`, denominated in Linux epoch seconds, at which a query should next be performed.

3.3.3 Connecting to Endpoints

Consumers must always connect to endpoints over TLS and verify the presented certificate is authentic for the endpoint's domain. See above, for *technical guidance on recommended ciphersuites*.

Backend feeds intended only for consumption by other backends will typically be protected by TLS Client Authentication. Consumers of Backend feeds must offer a TLS Client Certificate when connecting. This TLS Client Certificate should be issued or authorized via an out of band process. It may be that a third party offers a client certificate issuing service for a group of interoperating contact tracing systems.

3.3.4 Setup

When starting to consume a feed, the consumer must:

1. (Optional) Retrieve the latest set of signing keys.
2. Poll the `latest` endpoint. If using, the consumer must verify the response.
3. Retrieve past batches starting from the most recent batch (see below).
4. When not using webhooks: schedule the next retrieval. So as not to overload the server, add a uniformly random delay of at most 60 seconds to the expected release time.

When using webhooks, the consumer must use an out-of-band process to register the webhook.

3.3.5 Processing webhook notifications (Backend feeds only)

When notified of a new batch using a webhook, the consumer must:

1. Verify that the supplied `baseURL` matches one of the configured feeds, otherwise, discard.
2. Verify that the `batchId` corresponds to the expected batch index, otherwise discard.
3. Schedule retrieval of the new batch with a uniformly randomly delay of at most 60 seconds.

3.3.6 Retrieving a new batch

When the next retrieval time is scheduled (either after notification via a webhook, or because the `recommendedNextPollTime` arrives), the consumer proceeds as follows.

1. It queries the `latest` endpoint. If using signatures, the consumer must verify the response before proceeding.
2. When not using webhooks: schedule the next poll time based on `recommendedNextPollTime`, and again add a uniform delay of at most 60 seconds so as not to overload the server.
3. Let `lastBatchId` be the last retrieved batch identifier and `latestBatchId` be the latest batch identifier retrieved from the server. For each `batchId` such that `lastBatchId < batchId <= latestBatchId` the consumer should:

a. Retrieve batch `batchId`, and verify the signature when available.

b. Process the batch

3.3.7 Verifying Signed Feeds (optional)

When a feed uses signatures, clients must verify them when retrieving a resource. In particular, clients must:

1. Obtain the signature for the given resource and verify the validity of the signature using the `RS256` algorithm against the list of known *valid verification keys for this feed.*.
2. Verify that the `url` claim in the JWT matches the requested resource.
3. Verify that the SHA256 hash of the obtained resource matches the `content-hash` claim in the JWT.

If any of these verifications fails, the client must reject the download.

CHAPTER

FOUR

REPORTING COMPONENT

Reporting components take as input (authenticated) *tracing keys* from their own users and distribute them to the relevant *publication component*. In particular, the tracing keys must be sent to both the operator's publication component, and to any (publication component of) operators of regions that the COVID-19 positive user visited during the contagious period.

The following figure gives a high-level overview.

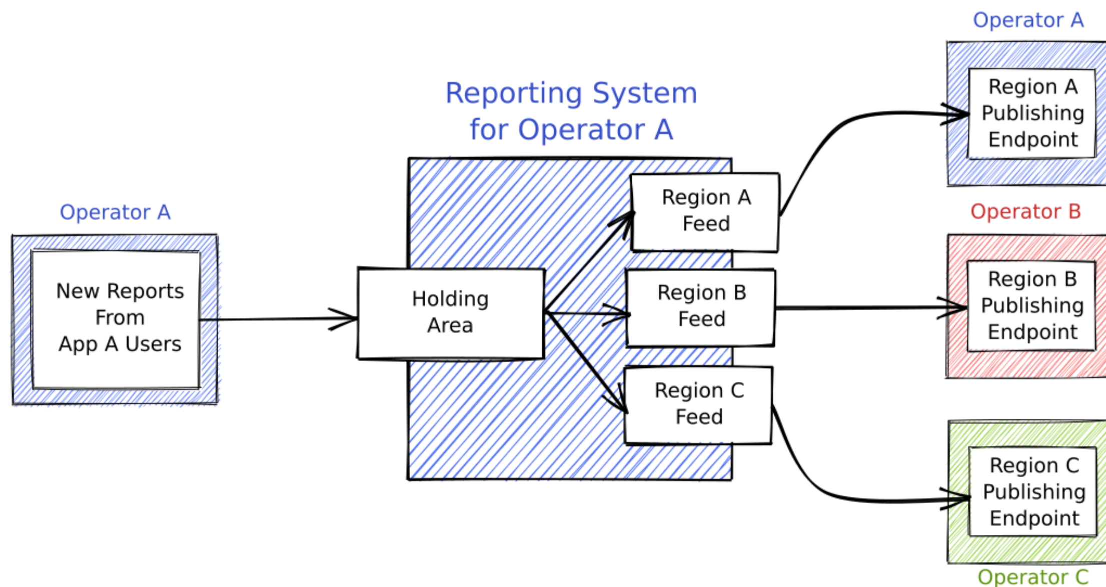

Fig. 1: Information flow of tracing keys. Tracing keys are uploaded by application users, temporarily stored by the reporting component, and then distributed to the *publication components* of all relevant regions using a *publication feed*.

When applications upload *tracing keys* they *inform the operators backend of all the regions that the user visited* during the contagious period. The reporting component uses this list of regions to determine to which operators to forward the keys to.

To send *tracing keys* to other operators, the reporting component provides *a specific feed for each operator*. The publication components of these backends then *consume tracing keys from this feed* and feed them to their own users.

4.1 Configuration

The reporting component, provides a tracing key feed for each backend that it has an interoperability agreement with. As per *the specification of producing a feed* the reporting component must know:

- The TLS Client Certificate of the corresponding backend.
- (Optional) A Webhook URL to call when new reports are available.

To enable the corresponding backend to consume tracing keys published on this feed, the reporting component must, as per *the specification on consuming a feed*, provide the following information:

- The Protocol Report Format
- A Feed URL where the operator is providing the reporting service
- (Optional) A set of signing keys or a URL which describes the signing keys.

Finally, the reporting component must know how to associate regions reported by the app (e.g., because the user visited that region during the contagious period) to corresponding backends/feeds.

4.2 Operation

We discuss here how to secure feeds between backends and how the reporting component determines which keys to attach to which feed. We refer to *the relevant other parts of this document for the details on how to produce a feed*.

4.2.1 Securing feeds to other backends

As discussed before, *feeds between backends should use TLS Client authentication.*.

In the event that the TLS Client Authentication is misconfigured, a TLS Client Certificate Private Key is compromised or a TLS Client Certificate Authority private key is compromised, unauthorized users may be able to access the private regional feeds. This does not endanger the security of the component, it does not allow a malicious party to introduce false information. However, it does entail a minor reduction in privacy, as the the unauthorized party would be able to see which reports corresponded to users of the operator's application. However, the reports are anonymous, so the privacy implications are low.

4.2.2 Publishing Updates to the Regional Feeds

The operator regularly receives new uploads of *tracing keys* from its users. These keys are stored in an operator-specific manner and will be tagged with the regions the user has visited. The operator also assigns a `validBeforeTime` to tracing key. It will not send tracing keys to external backends until after `validBeforeTime` to protect against misbehaving backends. The choice of `validBeforeTime` depends on the protocol, and on how often the backend publishes new tracing keys.

All tracing keys processed by the reporting component originate from users of the same application. They are therefore likely to use the same protocol. If not, the reporting component must provide a feed for each protocol separately.

In order to communicate these reports to the other contact tracing system, they must be added to that region's private feed. Periodically, typically expected to be as frequently as every 10 minutes or as low as every 60 minutes, the reporting component:

The reporting component regularly pushes updates along every feed. We expect updates to the internal publication to be frequent (e.g., every 10 minutes). Updates to external reporting components of other backends are likely to be less frequent, and aligned to the `validBeforeTime` of a batch of keys (for example, every 2 hours).

To create a new batch for a specific feed, the reporting component proceeds as follows:

1. Compile the list of *tracing keys*. When the feed is external (i.e., it is consumed by another operator), exclude all tracing keys for which `validBeforeTime` is after the current time.
2. If this list is non-empty, follow the *instructions on how to produce a batch* to publish the new batch.

As part of these instructions, the reporting component will notify any webhook of the receiving publication component when configured.

4.2.3 Removing Old Entries

Periodically, e.g. daily, the feed should remove old batches that are no longer useful for contact tracing. This is described in X.

4.3 Deployment Considerations

4.3.1 Polling and Publishing Schedule

Reporting components should publish new reports frequently, e.g. at 10 minute intervals to ensure they reach publication components (and thus users) with minimal delay. The use of webhooks to inform publication components of new content is highly recommended.

4.3.2 Data Volume

Reporting components only receive data from the operator's users and need only store it temporarily. It may be safely deleted once consumed by publication components.

As there are only a small number of publication components, e.g. tens or hundreds, outgoing data volume is not a particular concern.

4.3.3 Feed Padding

As ultimately, the report feeds for each country are public, countries may wish to insert 'dummy' reports if report feeds have fewer entries than a chosen limit. This avoids revealing the precise count of individuals that have reported using a particular application and mitigates any attempt to link users who have uploaded multiple tracing keys.

In general, this seems beneficial for user privacy and so we recommend it. There is a slight trade off in increased download sizes for clients, but provided the number of 'dummy' reports is not excessive, this is negligible. There are no impacts for security or correctness and this would not introduce any risk of false positives.

Care must be taken that the 'dummy' reports otherwise look identical to normal user reports, e.g. how various parameters are generated. In particular, they should be 'tagged' with different regional visits in a way that mimics the normal user distribution. Otherwise the 'dummy' reports will be distinguishable from genuine reports.

4.3.4 Use of CDNs

Some operators may wish to use CDNs or other services, e.g. for Denial of Service prevention, even though this is not required for data volume purposes. An alternative mitigation, which should only be deployed in addition to TLS Client Certs, is blocking incoming traffic by default, with exceptions based on the IP addresses of publication servers.

TLS Client Certificate Authorization is not universally supported by CDNs, but several large providers (e.g. Azure, Cloudflare) do support it. This necessarily entails trusting the CDN with client authentication.

As the use of a CDN means that TLS connections from publication components will terminate at the CDN rather than the operator's endpoint, clients cannot rely on the TLS certificates to authenticate the data transmitted by the CDN. Consequently, we recommend *the use of signatures to ensure that CDNs or other parties in between the operator and the relevant users cannot tamper with the downloaded data*.

CHAPTER FIVE

PUBLICATION COMPONENT

Publication components take as input *tracing keys* for visitors of their own region. These tracing keys can come from the operators own users, or from users of other operators. In both cases, they are *provided by a reporting component.* The publication component then provides of feed containing these tracing keys both to its own users, and to all users that visited the region.

The following figure gives a high-level overview.

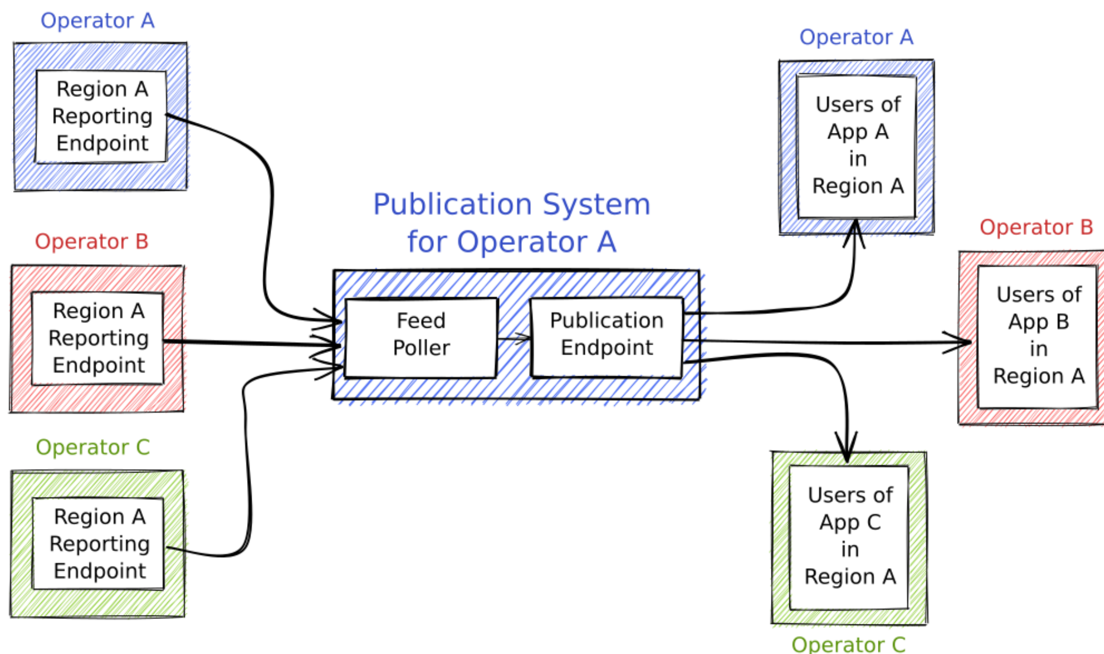

Fig. 1: Information Flow for the Publication Component. The publication component receives tracing keys via feeds provide by (other) operators. It then collates these tracing keys and publishes them so that apps can access the tracing keys to compute exposure scores.

The publication component polls the feeds provided by all operators that it has an interoperability agreement with. Different operators may use different protocols, resulting in the publication component collecting keys from various protocols.

For each of these protocols, the publication component *publishes a feed* that is regularly *queried by any application that visited the operator's region*. Using one feed per protocol, ensures that applications do not have to download information they cannot process.

5.1 Configuration

The publication component consumes a tracing key feed from each backend that it has an interoperability agreement with. As per *the specification of consuming a feed*, the publication component must know, for each feed:

- The Protocol Report Format used.
- A Feed URL where the contact tracing system is providing a reporting service.
- (Optional) A set of signing keys or a URL which describes the signing keys.

To enable the corresponding backend to produce a feed of tracing keys for the publication component, the publication component must, as per *the specfication on producing a feed*, provide the following information:

- A TLS Client Certificate which is trusted by the other contact tracing system.
- (Optional) A Webhook URL to call when new reports are available.

Users visiting the operator's region also need to be informed of how to fetch information from the operator's publication component. Consequently, the operator should ensure this information is available to the other contact tracing systems so that they can distribute it to their users. See the *application configuration* for more details.

5.2 Operation

We specify here the details of how the publication component consumes tracing keys from other systems, and then produces feeds that applications can download.

5.2.1 Fetching New Tracing Keys

The publication component must fetch new tracing keys from reporting components, including the reporting component hosted by this operator and other contact tracing systems. These queries can be triggered in two different ways.

- **Periodically** - The publication component should visit the `latest` url for each regional feed at frequent intervals. The `latest` url advises if there are new reports and when the next update is anticipated.
- **Via Webhook** - If the reporting component has been provided with a webhook for the publication component, the publication component need only look for new urls when it triggers, avoiding redundant queries.

In either circumstance, when a query is triggered, the publication component should follow the *steps to consume a tracing feed* and download all new tracing keys from that feed. These reports are temporarily stored in the publication component.

5.2.2 Data Storage

The publishing component needs to temporarily store new tracing keys before it publishes them to users. The storage method can be selected by the implementation. In practice, the maximum number of reports in temporary storage is equivalent to the maximum number of tracing keys in the time interval between two publications to the feed. The amount of storage required is expected to be small.

5.2.3 Publishing Reports to Clients

The reporting component regularly pushes updates along every feed. The update frequency depends on the specific protocol and regional configuration. Updates can be as frequent as every 10 minutes, or as infrequent as every 3 hours.

After the publishing component has fetched all tracing keys from the various reporting components used by each contact tracing system, it must then publish them to users. It proceeds as follows *for each protocol*:

1. It compiles a list of *tracing keys*. It exlcudes all tracing keys for which `validBeforeTime` is after the current time.
2. If the list is non-empty, follow the *instructions on how to produce a batch* to publish a new batch.

User's applications will *periodically fetch information from the proximity feed(s)* when they have been present in the operator's region.

5.2.4 Removing Old Entries

Periodically, e.g. daily, the feed should remove old batches that are no longer useful for contact tracing. This is described in X.

5.3 Deployment Considerations

5.3.1 Polling and Publishing Schedule

Operator's should ensure they have received all new reports from other contact tracing systems prior to publishing a new batch to clients. This ensures that clients receive new reports in a timely fashion. We recommend a short interval between polls (e.g. 10 minutes) and a longer interval for publication (e.g one or two hours).

Providing webhooks makes this process significantly easier. The publication component needs only poll the feed once the other contact tracing system triggers the relevant web hook. If webhooks are used, operators may want to monitor the last time each webhook was activated in order to detect incorrect configurations.

5.3.2 Data Volume

The publishing component receives incoming data from the reporting servers operated by other contact tracing systems. This data is expected to be relatively small in practice, e.g., on the order of tens of megabytes per day per operator.

However, the publishing component is expected to have significant outgoing data volume. Users of both the operator's system and other contact tracing systems that have visited the operator's region will regularly poll the publishing component for new reports. Consequently, the use of a CDN or other web infrastructure to manage the load is recommended.

5.3.3 Security Issues with CDNs

As the use of a CDN means that TLS connections from users will terminate at the CDN rather than the operator's endpoint, clients cannot rely on the TLS certificates to authenticate the data transmitted by the CDN. Consequently, we recommend *the use of signatures to ensure that CDNs or other parties in between the operator and the relevant users cannot tamper with the downloaded data*.

5.3.4 Protection of Credentials

The publishing component must authenticate with other contact tracing system's reporting servers. As described earlier, *this is carried out using TLS with Client Certificates*. The publishing component must ensure their TLS Client Certificates are stored securely according to best practices. Depending on policy considerations, credentials should be

audited, expired, rotated as appropriate. If the Publishing Component's TLS Client Certificate is compromised, this leads to a small privacy leak for the relevant reporting component.

The publishing component must also ensure that the connection to each reporting component is authenticated. This means that the TLS Server Certificate should be checked. Further, if the reporting server is providing a signed feed, *then the signatures should be checked*.

CHAPTER

SIX

INTEROPERABLE APPLICATIONS

Proximity tracing applications perform the following functions:

1. Generating ephemeral Bluetooth identifiers to broadcast and broadcasting them *(omitted in this specification)*;
2. Storing Bluetooth observations;
3. Computing the exposure risk given the tracing information published by the relevant backend servers; and
4. Uploading tracing information to their backend server once they are Covid Positive to enable exposure risk computation *(largely omitted in this specification)*.

In this interoperability specification, we omit the details of step 1 and step 4 as they are region/country/protocol specific and not relevant for interoperability. We assume that operator's have already designed and/or implemented a particular application and protocol. This document highlights requirements necessary for interoperability.

6.1 Configuration

To facilitate roaming, the app must be supplied with a list of regions (e.g., countries, states, and provinces) and the responsible backends. In particular, for each region it needs to store:

Region Description/Indicator A description or geographical indicator of the region

Base URLs A list of base URLs of the publishing component(s) that are responsible for this region

Formats The formats that each of these publishing component(s) use

Signing Keys (when using) (When using) For each base URL A list of valid signing keys or a trusted URL from which to fetch signing keys for this publishing component.

The operator should provide this information to the application, e.g. through static configuration in the app or dynamically loaded from a configuration server.

6.2 Internal storage

All decentralized contact tracing applications are expected to store some information about previously witnessed Bluetooth transmissions. In addition to recording approximately when a Bluetooth transmission was received, we also expect applications to remember which regions they have been present in.

Contact Events A database of contact events containing contact tuples:

- `EphID`: Received EphID
- `tReceive`: The time of reception. Depending on the specific implementation, this value can be rounded to a later value.

- Other protocol specification information, e.g. proximity or duration estimates.

> **Warning:** Rounding `tReceive` to a coarse value is considered best practice to avoid retroactive privacy attacks from compromise of a user's device. Further, to avoid replay attacks, `tReceive` must never be rounded to an earlier value.

Active regions A list of regions that the user has visited in the past 14 days (or however many days are needed to compute the exposure risk). For each of these regions, the app must store:

- The identifier of the region
- When the user last visited this region.

The app needs to know the active regions so that it can take tracing information from these regions into account when computing the exposure of the user. After 14 days, or the maximum tracing window, the app can delete any regions that were not visited in this window.

Applications will also store other application and protocol specific information.

6.3 Tracking Active Regions

The app will typically ship with a default active region corresponding to the operator's region of operation. The app must provide a mechanism for users to enter the regions that they will travel to, or have previously travelled to. The app should allow users to enter regions retroactively (i.e., after they have already returned from a trip to these regions).

6.4 Storing received Ephemeral Bluetooth Identifiers

Apps need to store received Bluetooth identifiers to enable later exposure risk computation. The details are protocol and app specific. However, for the interoperability specification we assume that for every received EphID the app stores the EphID and a (possibly rounded) receive time `tReceive`.

Specific protocols may have extra requirements on how this data is stored. For example, the DP3T low-cost protocol requires that `tReceive` has a coarse granularity and that observations are stored in random order.

6.5 Upload of report after infection is confirmed

To enable contact tracing, the app must upload tracing information to the operator's backend. The specific tracing information is protocol dependent, but must be sufficient to enable other users to check whether they were in proximity to the reporting user. In addition, information on the user's recently visited regions is also required for interoperability.

At a high-level, the upload process proceeds as follows:

1. Before uploading any data, the app asks the user permission to upload tracing data.
2. Next, the app asks the user to confirm and if necessary amend, the list of regions that the user traveled to during the contagious period.
3. The app uploads to the operators backend the protocol-specific tracing information and a list of regions visited during the contagious period.

The exact format of the upload and other API considerations is specific to a particular application and does not need to be standardized for interoperability. Typically, the backend will require the upload to be verified by a healthcare authority. This process is application and/or protocol specific.

6.6 Computing Exposure Risk

Apps must regularly recompute the user's exposure score. To do so, app must take into account tracing information from any backends responsible for the user's visited regions. Exposure risk computation proceeds in two steps: (1) retrieve all new tracing information from the relevant backends, (2) match any stored EphIDs against this tracing information to compute the risk score.

To retrieve new tracing information, the app must pull tracing information: from each backend of an active region. Recall that active regions are any regions the user has visited in the past 14 days.

To do so:

1. the app looks up the base URL of the backend(s) for the region in its configuration,
2. and follows the instructions *to consume information from a tracing feed*
3. to retrieve all new batches that are at most 14 days old. Apps must take care to retrieve all these batches, in particular for roaming regions that have been retroactively added.

Having received all latest tracing information, apps must:

1. Use the protocol specific cryptography to expand the tracing information into a set of EphID with corresponding `validAfterTime` and `validBeforeTime`.
2. For each of these (`EphId`, `validAfterTime`, `validBeforeTime`) tuples, check the local storage of EphID to see if the recomputed `EphId` matches a tuple (`EphId`, `tReceive`) such that `validAfterTime <= tReceive < validBeforeTime`. If so, incorporate this record into the user's exposure risk calculation.

Optionally, apps may decide to take into account the `KeyType` of the recomputed `EphId`, see *the specific formats for more information*.

> **Warning:** To prevent replay attacks using EphIDs that have been computed from published tracing information, it is essential that apps ignore any matching `EphID` that it received *after* `validBeforeTime` or *before* `validAfterTime`.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- search

INDEX

A

App, 2

D

Decentralized Proximity Tracing System, 2

O

Operator, 2

P

Proximity Tracing Feed, 2

R

Region, 2

T

Tracing Keys, 2

U

User, 2