Best Practices
Operational Security for Proximity Tracing

DP-3T Team

Executive summary

In this document we describe security mechanisms that can be added to Proximity Tracing applications to ensure that the security and privacy properties provided by the protocols are not undermined by other components of the system. In particular we provide recommendations to protect:

- **Sensitive communications between app and server.** We provide guidelines to establish dummy traffic to ensure that network traffic that is associated with sensitive information (COVID-positive status, notified users) cannot be recognized by passive observers.

- **Communications between backend and smartphones.** We analyse the use of attestation as a support mechanism to protect the downstream communication. Attestation must be carefully considered as it increases the dependency on closed-source components and may not be available in all phones.

- **Metadata protection at the server.** We analyse what information could be collected by a server and provide recommendations in regards to logging and storage.

- **Validation of notifications.** We describe a simple mechanism to validate that when a user claims to have received a notification that the user is in possession of a phone for which the app has actually generated such a notification.

Dummy traffic: protecting against network adversaries

Adversaries that can observe network communication can observe network traffic between a user's smartphone and the backend server or servers. Powerful network adversaries might also be able to observe network traffic between backends. Network adversaries can use these observations to try to infer sensitive information about users: whether they received a positive COVID-19 diagnosis, or whether they received a notification of exposure to COVID-19 positive users.

We consider two types of network adversaries:

- Local network adversaries (e.g., a malicious WiFi hotspot).
- Major network operator (e.g., large telecom operators that can see both cellular and domestic internet traffic).

When apps use Google/Apple Exposure Notification protocol (or the DP-3T protocols), they need to communicate to the backend server, and potentially other servers, as determined by the health authority. When this network traffic is associated with sensitive information, it can reveal information to the adversary. Some examples of such sensitive network traffic are[1]:

- COVID-19 positive users' smartphones upload diagnosis keys to the backend server to enable proximity tracing.
- Smartphones of COVID-19 positive users obtain an upload authorization code from a health authority before uploading their keys to a backend server.
- Smartphones of users that have been exposed communicate this to the backend so that the backend can notify these exposed users (e.g., a system in which instead of asking exposed users to call an information number, it is preferred that a person calls those exposed users).
- Smartphones of users that have received an exposure notification provide proof of this notification to a server (e.g., to receive compensation or work leave).

These interactions originate from different **actions** performed by a user or the app. Actions can involve several network connections. For example, the first two are likely both caused by the action of *uploading diagnosis keys*. The third results from an action initiated by the app upon detecting exposure, and the last one corresponds to the action of *verifying notification.*

In this section, we propose a mechanism that provides users with **plausible deniability** with respect to these network interactions — i.e., a user can claim that an interaction is a

[1] At the time of writing, there is no implementation of interoperability between countries in which phones and/or backend servers exchange information. Depending on the implementation, some of these exchanges may also reveal sensitive information and may require the introduction of new mechanisms similar to those described in this document.

fake one generated automatically by the app. To achieve this, the main idea is to have all users regularly performing fake actions that, from the point of view of a network adversary, look like real actions. When observing an action, the network adversary cannot infer sensitive information about the user, as any action could be one automatically generated by the system.

Below, we provide guidelines for mechanisms that provide plausible deniability, including how to create fake actions that are indistinguishable from real actions and how to evaluate the strength of the protection. For each of these elements, we provide example implementation of the techniques in the Swiss app SwissCovid.

Creating fake actions indistinguishable from real actions

This protection mechanism works by (1) producing fake actions that are indistinguishable from real actions and (2) distributing these fake actions over time. As a result, any observed action could, with reasonable probability, be a fake action.

Making fake actions look the same as real ones

Each action can cause multiple observable network events. For example, to perform the *action of uploading diagnosis keys* a smartphone in the Swiss system will:

1. Connect to the health authority backend to validate a Covidcode (a short validation code provided to COVID-19 positive users) and obtain an authorization token.
2. Upload the diagnosis keys using the authorization token obtained in the previous step to the app's backend.

In this example, network observers observe two separate network connections, one for each backend. It will also observe the corresponding DNS queries. For each connection, the observer can see the timing and sizes of network packets.

Fake actions **must be indistinguishable from real actions** from the point of view of a network adversary. This means that fake actions must:

- Make network connections in the same order as real actions, and as a result of this, make DNS queries in the same order as well.
- Ensure that the packets sequences and sizes for each connection follow the same distribution as in real actions.
- Ensure that the packet content follows the same content distribution as in real actions. In practice, this is most easily achieved by using an encrypted connection.
- The timing of package sequences within each connection must follow the same distribution as in real actions. In particular, the server response times must follow the same distribution as for a corresponding real action. This condition is rarely automatically satisfied. We recommend to either perform exactly the same

operations for real and fake actions (including database accesses etc.); or to delay responses up to a known maximum response time for real and fake requests.

- If the action involves more than one connection, the timing patterns between individual requests must follow the same distribution as in a real action.
- Externally accessible information (e.g., published diagnosis keys) should not reveal if a specific request was real or fake.

To achieve these requirements, it might be necessary to modify real requests and how they are handled, to make it easier to make fake requests indistinguishable from them.

Satisfying all these requirements is not always possible. In particular, ensuring that the timing patterns between different requests follow the same timing pattern is very hard when **the user's actions influence the time between real requests.** The time it takes for a user to perform an action is difficult to model, and therefore difficult to reproduce in fake requests. Therefore, we strongly recommend that users' actions should not influence the timing of sensitive requests. In the Swiss app for example, the user's action of entering a Covidcode happens *before any network request is made,* thereby ensuring that the time it takes the user to enter this code does not influence the observations of the network observer.

The SwissCovid smartphone application applies these rules to ensure that fake uploads of diagnosis keys are indistinguishable from real uploads:

1. Requests to both backends use TLS encryption.
2. For each fake upload action, the phone connects to the health authority backend, sends a fake Covidcode and obtains a fake authorization token.
3. The phone uploads fake diagnosis keys together with the fake authorization token to the app's backend.

Both the health authority backend server and the app's backend server have been configured to use the same response delay for fake and real requests. In both cases, the request and the response are the same sizes as in the corresponding real requests. Finally, if any request when performing a fake action fails due to a backend not being available, the phone repeats the request, as for a real action.

The backend receiving the (real) diagnosis keys does *not* publish them immediately. Instead, it only reveals received diagnosis keys every two hours. As a result, the publication timing of keys cannot be used to distinguish real and fake requests.

Scheduling fake action

We propose that Apps schedule fake actions following a Poisson distribution with Poisson rate λ.[2] This rate indicates the number of fake actions per day. We propose a default value of 0.2, i.e., the app will generate on average 1 fake action every 5 days. This parameter choice provides good protection without overloading the server. We show below that the protection provided is independent of the population size.

The app makes a fake request as described above at the instant established by the Poisson distribution. Real actions are made **in addition to these fake actions in order not to incur extra delays.** In particular when uploading diagnosis keys, it is important to upload these keys quickly so that exposed users are notified quickly.

Apps will use the following mechanism to schedule and launch fake actions:

1. Draw a random delay `t_delay` from Exp(λ) (i.e., following an exponential distribution).
2. Schedule a fake action `t_delay` seconds after the last time a fake action was scheduled. If this time is in the past (e.g., when the scheduler is activated later than requested), this event is immediately executed (following step).
3. Once the scheduler activates, the app performs the fake action as described above.
4. Repeat from 1.

Analysis

By construction, real actions and fake actions are indistinguishable when compared. We now analyse whether a network observer can infer the existence of a real action when observing a *sequence of actions* spread over time, by the same user. We make the following **assumptions about real and fake actions**:

1. Real and fake actions are indistinguishable in isolation.
2. Fake actions are scheduled and executed following the Poisson distribution.
3. Real actions are rare. For the purpose of this analysis, we assume they occur at most once in an observation interval (e.g., reporting of a COVID-positive case).
4. The times at which real actions occur are distributed roughly uniformly. In particular, this implies that users who are tested positive for COVID-19 are not all notified at the same time (e.g., between 8:30 and 8:45 in the morning).

[2] The choice of the Poisson distribution is due to its memoryless properties that help provide robust privacy guarantees without complexity.

The last assumption simplifies our analysis. We note, however, that as long as the distribution over time of real actions is known or can be estimated, the parameters for fake actions can be adjusted to provide sufficient cover.

We also make **assumptions on the knowledge of the adversary:**

1. We assume that the adversary knows the prior probability p_{REAL} of an action being real, and that this probability is roughly uniform across the population. For instance, the adversary knows the rate of infection and assumes everyone is equally likely to be infected. This assumption implies that the adversary does not use the fact that some people are more likely to perform a real action because they are at high risk of contracting COVID-19.

2. We assume that the adversary only has information related to internet communication. In other words, the adversary does not collude with other entities to gain access to other events such as telephone calls or SMS messages.[3]

In the following, we make the worst-case assumption that the network observer can associate observed network activity to a persistent (network) identity.

It is always the network adversary's goal to determine if a specific user has (ever) performed a real action. In our analyses, we compute the posterior probability that a user performed a real action given the adversary's observation.

Times between actions

We show that a network adversary that uses the time between successive actions to determine whether a user performed a real action cannot do any better than it already could be determined based on the prior distribution. In other words, the network observer does not learn any more about the user, despite being able to observe the time between successive actions.

Let p_{REAL} be the prior probability that an action is real, and $p_{FAKE} = 1 - p_{REAL}$ be the corresponding prior probability that an action is fake. The prior probability that an action is real is typically small. Let *IAT* be the random variable that indicates the interarrival time between two actions. We use the Bayes' theorem to compute the probability that an observed action is real, given that the previous action happened *T* hours ago:

$$\text{P(ACTION is real} \mid IAT < T) = \text{P}(IAT < T \mid \text{ACTION is real}) * p_{REAL} \,/\, \text{P}(IAT < T).$$

We compute the probabilities on the right hand side as follows:

- $\text{P}(IAT < T \mid \text{ACTION is real}) = \text{CDFExp}(T, \lambda / 24)$, where $\text{CDFExp}(T, \lambda / 24)$ gives the probability that the exponential delay sampled from a rate of $\lambda / 24$ is less than T

[3] We note that telecommunication providers that have access to data other than the internet communication data (calls, SMS) may already be able to infer the status of individuals based on these data (e.g., a user receiving a call from the test center) and therefore the App does not provide additional information to them.

hours, because the probability that at T hours before the receive time of a real action, the user makes a fake action follows the exponential distribution.

- P($IAT < T$ | ACTION is fake) = CDFExp(T, λ / 24), because fake actions are per definition scheduled following an exponential distribution.
- P($IAT < T$) = P($IAT < T$ | ACTION is real) * p_{REAL} + P($IAT < T$ | ACTION is fake) * p_{FAKE} = CDFExp(T, λ / 24), by applying the law of total probabilities and the above identities.

Plugging these in, we find that

P(ACTION is real | $IAT < T$) = p_{REAL},

confirming that the time between actions does not give the adversary more information about whether a real event occurred or not.

Counting number of observed post requests

We now analyse what the network adversary can learn from counting the total number of actions within a time interval. We show that the total number of observed actions influences the posterior probability that a specific user made a real action. Clearly, if the network observer observes zero actions, the observer is now sure that the user did not make a real action. Similarly, if the observer observes many actions, the probability that one of these actions is real goes up somewhat. However, these posterior probabilities remain small, showing that users retain plausible deniability.

We assume the adversary counts events for a duration of D days. Let N be the random variable that indicates the number of actions observed for a specific user. Let p_{REAL} and p_{NOREAL} be the probability that a user makes respectively does not make a real action. We use Bayes' theorem to compute the posterior probability that a user made a real action, given an observation of N actions:

P(did REAL | N ACTIONS) = P(N ACTIONS | did REAL) * p_{REAL} / P(N ACTIONS)

We compute the probabilities on the right hand side as follows:

- P(N ACTIONS | did REAL) = P(N - 1 fake ACTIONS), because if one action is real, then the remaining N - 1 actions observed during these D days must be fake. Since fake actions follow a Poisson distribution, we have that P(N - 1 fake ACTIONS) = $(D\lambda)^{N-1}$*$e^{-D\lambda}$/(N-1)!.
- P(N ACTIONS | no REAL) = P(N fake ACTIONS) = $(D\lambda)^{N}$*$e^{-D\lambda}$/N!, because all actions must be fake.
- P(N ACTIONS) = P(N ACTIONS | did REAL) * p_{REAL} + P(N ACTIONS | no REAL) * p_{NOREAL} using the law of total probabilities.

Plugging these numbers in and cancelling common terms, we find:

$$\text{P(did REAL} \mid N \text{ ACTIONS)} = N * p_{REAL} / (N * p_{REAL} + D\lambda * p_{NOREAL}) .$$

When p_{REAL} is very small, then this equation is approximated by

$$\text{P(did REAL} \mid N \text{ ACTIONS)} = N/\lambda * p_{REAL}/D = N/\lambda * p_{DAY} ,$$

where p_{DAY} is the probability that a real action happens on a given day. Note that this equation only depends on p_{DAY} and not on the length of the interval considered. For small values of N (these are by far the most likely), the posterior probability remains small as well. Larger values of λ, i.e., corresponding to generating more frequent fake actions, reduce the posterior probability. However, a larger value of λ also means that bigger values of N become more likely. As a result, doubling λ does not generally halve the achievable posteriors.

Example: uploading diagnosis keys

As an example we plot the posterior probability for determining whether a user uploaded diagnosis keys or not given a 10 day window, i.e., $D = 10$ (recall from the above that the length of the interval does not really matter). Assuming a diagnosis rate of 25 cases per million inhabitants per day, we estimate $p_{REAL} = 0.00025$ for the 10-day period. In the figure below we plot P(did REAL | N ACTIONS) for realistic values of N. In fact, for $\lambda = 0.2$, we have expect that only one in 50 billion users will produce N = 16 fake actions at least once in a given 10 day period.

The figure below shows that for all realistic values of N the posterior probabilities are very small. In all cases, the probability is more than 99.8% that the observed user did *not* make a real action. This ensures that the actions of a user did make a real action are indeed plausible deniable.

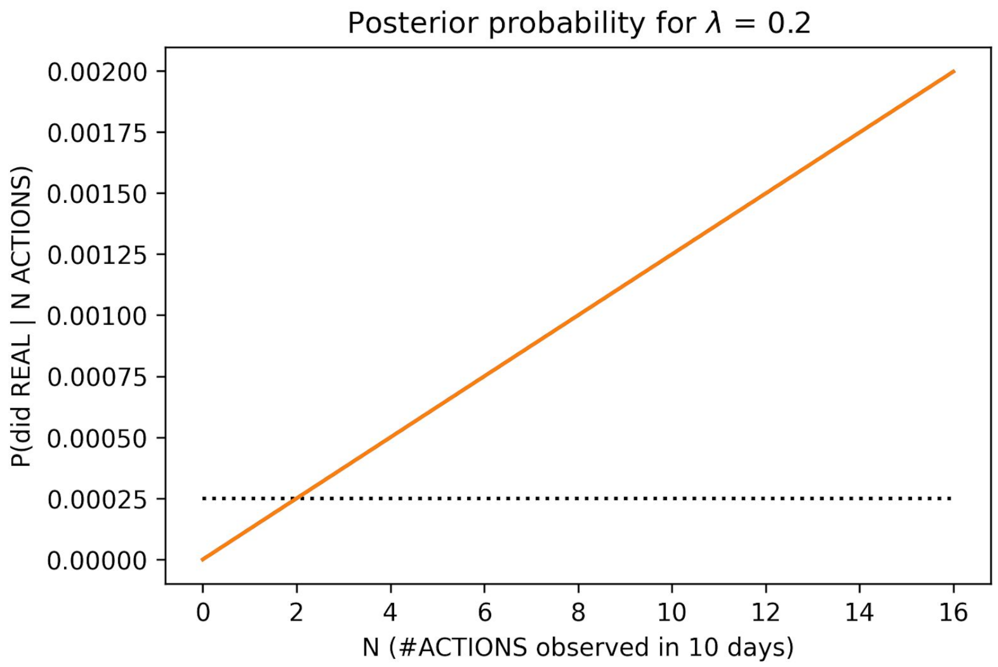
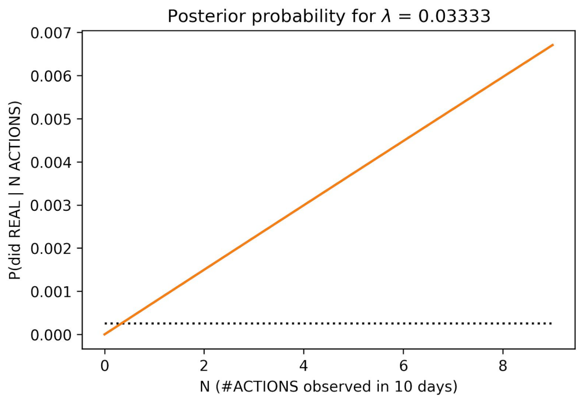


Figure AA: Posterior probabilities of having made a real action given an observation of *N* actions for λ = 0.2 (on average one fake action per 5 days). The dotted line is the prior probability of having made a real action.



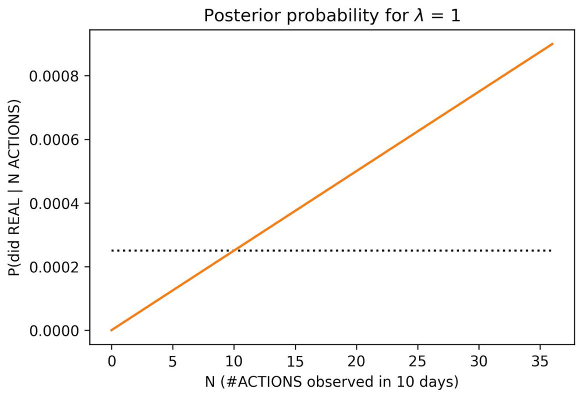


Figure AB: As Figure AA, but with left λ = 0.0333 (on average one fake action per month) and right λ = 1 (on average one fake action per day).

Decreasing the rate λ at which users make fake requests increases the overall posterior probability for the same set of realistic values of *N*, while increasing the rate λ decreases the overall posterior probability. Finally, the posterior increases almost linear in the prior. So if the real event is more likely, a larger value of the rate λ is needed to compensate.

Discussion

The base rate of real actions is extremely small for all the scenarios that might need protection from network adversaries. Therefore, based on the base-rate fallacy, one expects that any conclusions about users making real requests based on observations provide only weak evidence. The above analysis bears this out: the fake action mechanism ensures plausible deniability of real actions.

However, this analysis **hinges on the fact that fake actions are made when scheduled**. Network adversaries might obtain better estimates by exploiting scheduling deviations. Some deviations might be unavoidable on mobile operating systems where the background scheduler checks for scheduled events too infrequently or only at very regular times.

Server-side logging

For security and operational reasons, backend infrastructure is usually configured to log information for each incoming request. This information includes for each incoming request: the time, requested resource, and IP address. Furthermore, backend servers may update databases to record information. Proximity tracing systems, for example, store diagnosis keys of COVID-19 positive users.

While these logs and database records are not necessarily collected at the same place,[4] they are typically under the control of a single entity. Therefore, we recommend a careful study of information stored by different components in the system to ensure that sensitive data about users, e.g., whether they are COVID-19 positive or have been notified of exposure, is protected at all times, even when these logs and database records are combined. We consider a snapshot attacker that accesses the logs and database records at discrete points in time, rather than continuously observing them.

We distinguish two types of requests to the server: non-sensitive and sensitive requests. In decentralized proximity tracing systems, all users regularly retrieve new diagnosis keys and potentially new application configurations. These requests are *non-sensitive*. All users make these requests, and therefore logs of them cannot reveal any sensitive information about users, beyond the fact that these users use a proximity tracing application.

To ensure that the existence of network requests is not correlated to users not having received a positive diagnosis, the app should continue making requests after a COVID-positive user uploads their keys. In particular, the app must continue downloading diagnosis keys, and continue making fake requests.

Requests made by users related to the uploading of diagnosis keys by COVID-19 positive users and requests to confirm notification status of exposed users (see next sections) are *sensitive*. These requests should be treated with care. In the previous section, we described how fake requests can hide sensitive information from network observers. Such fake requests are also **essential to ensure sensitive information cannot be inferred from log files.** For example, without fake requests, the simple fact that a user with a specific IP address made an upload request would reveal that this user is COVID-19 positive.

The use of fake requests is by itself not sufficient. The stored data — logs and database records — should not enable fake requests to be distinguished from real requests. In particular, this means that:

- Request logs should ideally not contain more than a user's IP address, the request time, the requested resource, the request and response sizes, the HTTP status code, and the User-Agent field.

[4] For example, backend infrastructure may consist of web-application firewalls, load balancers, application servers and database servers. All of these can potentially generate and store log entries.

- Request log entries should not reveal whether the request was fake or real. Therefore, fake and real requests should produce identical distributions for all fields in the logs. In particular, logs must not record responses or sensitive request headers.

- Backend applications should not output any logs that distinguish real from fake requests.

- When writing entries to the database, e.g., to store diagnosis keys of a COVID-19 positive user, these entries should either not have a timestamp, or only a coarse-grained one. This requirement ensures that every entry in the database may correspond to a large number of incoming requests, ensuring that the anonymity set of an entry in the database is large.

Finally, any requests not made by the app should be at most weakly correlated in time to real actions by app users. For example, health officials should not request an authorization code seconds before a user uploads diagnosis keys. Without this restriction, time correlation between a health official's request and the user's real request would allow to establish a link between a key upload and an authorization request, revealing that the diagnosis key upload is real.

Analysing the SwissCovid dummy uploads

All communication with the backend originating from the SwissCovid app and health officials is TLS encrypted.

The SwissCovid app regularly retrieves new configuration settings and batches of diagnosis keys from the backend server. Since *all* apps retrieve this information, these requests are non-sensitive. To help handle the load, these requests are served using a content delivery network (CDN). Beyond the fact that the CDN can now *identify users of the SwissCovid app,* this has no privacy implications.

All sensitive requests are made directly to the Swiss backend infrastructure. These requests never traverse a CDN. The sensitive requests are made to a different domain and the corresponding TLS certificate is pinned in the app. The CDN does not have access to this certificate and can therefore not intercept these requests.

For each incoming request, the server only logs allowed data such as the request time, the user agent, the requested resource, and the status code.

The database only holds information of spent COVID codes, spent authentication codes and tokens, and diagnosis keys corresponding to COVID-19 positive users. The records for spent COVID codes do not contain any information on when they were spent. The records on authorization codes only record the day on which they were spent. The diagnosis keys do have a corresponding receive timestamp. However, this timestamp is rounded down, so that it only records in which bucket the diagnosis key should be published. IIn the current configuration, this ensures a 2-hour granularity.

Therefore, based on the logs and database records, every diagnosis key in the database, could have been stored as a result of any upload request made in a 2-hour window. With fake requests, this ensures a large anonymity set.

Finally, the Swiss backend application also logs requests by health officials to create COVID codes. These codes are provided to COVID-19 positive users, and authenticate the upload of diagnosis keys. COVID codes are provided to users either by phone or via postal mail. As a result, there is always at least several minutes of delay between when a COVID code is obtained and when it is used by a user in an upload. This ensures an anonymity set of hundreds of people for the final upload.

Defense in depth: device and app attestation

Modern smartphone operating systems enable backend servers to verify the integrity of the smartphone operating system, as well as the integrity of a specific version of an application. As part of this process, smartphones *attest* the integrity of the OS and an app to the backend server operated by the app provider. Attestation can be used as a defense in depth mechanism to restrict API access to official apps running on an unmodified mobile operating system.

Attestation can be used to increase the security of notification validation and to restrict access to published diagnosis keys. We emphasize that attestation is not likely to withstand all adversaries. A determined adversary may be able to find a way to bypass the attestation check.

Google SafetyNet Attestation API

Google offers the SafetyNet Attestation API on Android. It is provided by Google Play Services, and it is therefore *not part of the open source distribution of Android*. The SafetyNet Attestation API documentation states that "The API should be used as a part of your abuse detection system to help determine whether your servers are interacting with your genuine app running on a genuine Android device."[5]

At a high-level, the communication flow of SafetyNet is as follows. The server sends a nonce to the app, which the app relays to the SafetyNet API. The SafetyNet service on the device verifies the device state, and requests a signed attestation from Google's Attestation API backend (via the internet). The signed attestation is passed back to the app. The app can then send it to the server for verification. The server will verify the attestation against the nonce to convince itself that the smartphone has not been modified, and that the app is the official app.

Observations:

- Standard API limit is 10.000 requests/day, but can be increased on request
- Phone communicates actively with Google. Google's documentation of the protocol is not clear about what is being sent. We expect that at least some device-specific data needs to be sent to ensure that not everyone can simply use the Attestation API backend.
- Requires/depends on Google Play Service which is ***not*** open source. As such, using SafetyNet *increases the dependency on non-transparent code*.

[5] https://developer.android.com/training/safetynet/attestation

- There exist documents that suggest that SafetyNet Attestation can be made to succeed even on rooted phones.[6]

iOS DeviceCheck

Apple offers the DeviceCheck functionality on iOS devices. The documentation states that "The DeviceCheck APIs also let you verify that the token you receive comes from an authentic Apple device on which your app has been downloaded."[7]

At a high-level, the communication flow of DeviceCheck is as follows. Upon request of the app, iOS generates an encrypted token for the device. The app sends this token to the backend server. The backend server can validate the token by making an API call to Apple.

Observations:

- Presumably, if the device has been tampered with, the check will fail. The documentation is not clear on this point.
- Data from the phone is sent to Apple servers by using the app backend as an intermediary.

Comparison and discussion

Both APIs depend on non-transparent and closed-source facilities offered by the mobile operating system. Such closed-sourced dependencies are undesirable as they make it harder or even impossible to verify the app in its entirety.

While it is true that the Exposure Notification APIs also depend on non-open code, it is possible to provide an open and verifiable version of these components, in particular on Android.

The dependency on Google Play Services also prevents a class of Android users from using the app, even if otherwise they could. First, Google Play Services is not available on more open Android phones and recent Huawei models. Second, it prevents privacy-sensitive users from using custom firmware.

Both attestation APIs generate recognizable network traffic that could be picked up by network adversaries. In particular in the case of Android's SafetyNet, a local network adversary can detect the attestation call to Google. Therefore it is essential that the use of attestation is not bound to specific sensitive events such as receiving a positive diagnosis, or receiving an exposure notification.

[6]http://mulliner.org/collin/publications/inside_safetynet_attestation_attacks_and_defense_mulliner2017_ekoparty.pdf although recent news suggest that these work arounds no longer function: https://www.androidpolice.com/2020/03/11/safetynet-improvements-kill-magisk-hide/

[7]https://developer.apple.com/documentation/devicecheck

Encrypting batches of diagnosed keys

The diagnosis keys uploaded by COVID-19 positive users are publicly accessible. An operator may decide to increase the difficulty of executing some attacks against the proximity tracing systems by restricting access to the diagnosis keys. Restricting access makes identification attacks and tracing attacks against COVID-19 positive users harder. We note, though, that identification attacks are an inherent risk in proximity tracing systems.[8] Therefore, restricting access to diagnosis keys is of limited benefit.

Furthermore, we wish to emphasize that these attacks are actually very hard to execute because the current rate of COVID-19 positive diagnoses is low. In countries that are gradually loosening restrictions, the probability of being diagnosed in a given day is less than 1 in 50.000. As a result, an attacker needs to monitor a large number of users to even see a single COVID-19 positive person.

To further harden the system against attacks that use the diagnosis keys, we propose a mechanism that uses app attestation to only provide diagnosis keys to official apps. This approach, however, has all the disadvantages discussed above of attestation. A single device on which attestation fails, enables an attacker to circumvent the protection. If the attacker can read memory on one device for which attestation succeeds, the attacker can also recover the diagnosis keys.

Moreover, making keys available only inside official apps reduces transparency. With this approach, neither the keys nor any auxiliary data that accompanies the keys can be externally verified. Given these downsides and the limited benefits, we do not currently recommend this approach.

App attestation to restrict access to diagnosis keys

The key idea of this mechanism is to publish encrypted diagnosis keys, so that attackers cannot (easily) access the diagnosis keys to execute their attacks. The keys are encrypted using a key that is regularly rotated. Legitimate apps can retrieve the new decryption key after the app attestation is successful.

Let `D` be the duration that a single key is valid, for example 7 days. For each period `d` of `D` days, the backend:

1. Generates a random 128-bits AES key K_d

[8] See "Privacy and Security Risk Evaluation of Digital Proximity Tracing Systems", the DP-3T team, version April 21, 2020. Retrieved from https://github.com/DP-3T/documents/blob/master/Security%20analysis/Privacy%20and%20Security%20Attacks%20on%20Digital%20Proximity%20Tracing%20Systems.pdf

2. To release a batch during period d, the server encodes the batch and then encrypts it using AES-GCM mode with K_d as the key. The backend publishes the ciphertext.

Every D/2 days, smartphones retrieve decryption keys as follows:

1. The phone runs the device attestation protocol (using Android's SafetyNet or iOS' DeviceCheck) with the backend server.

2. If the check passes, the server sends to the smartphone the keys K_d for all relevant past periods (depending on how far in the past the app computes exposures) as well as the next period.

3. The device stores these keys securely, ideally within a special secure storage such as Android's KeyStore.

Smartphones proceed to download batches and then use the appropriate key K_d to decrypt the batch with AES-GCM. Phones run the attestation protocol frequently enough to always have the corresponding decryption keys.

To reduce the load on the backend server and to enable the use of a CDN, we deliberately do **not use attestation for each download**. When using attestation for every download, the backend must process thousands of attestations per second, even for small countries. This requires large amounts of costly infrastructure. It is also unclear whether for example Google would allow attestations at this rate.

Analysis

As long as the keys K_d are secure, no outside attacker can access the diagnosis keys. Thus this countermeasure raises the bar for identification and tracing attacks.

However, the security of the keys has two weak spots. First, the security of the mechanism hinges on the security of the attestation mechanism. As we described above, one device or OS version combination on which attestation succeeds while it should not, give the attacker access to the decryption key. If any such weak setup is present, not even using attestation for each download offers any protection.

The second weak spot is the fact that for efficiency, the decryption keys must be stored on the device. Therefore, the security of the keys is only as strong as the security of the secure storage mechanism in the face of a determined attacker.

Validating the receipt of a notification by the app

Users of the proximity tracing app receive a notification from the app when their exposure to COVID-19 positive users is considered sufficiently high. Users might feel incentivized to falsely claim they have been notified to 'profit' from the system. For example, Switzerland has a policy of economically compensating people of their quarantine is required by the health authorties, or a person may become exempt from an undesired activity (e.g., an exam).

We propose a technical mechanism that enables lightweight remote notification verification to increase the difficulty of making a fake claim. As we explain below, this mechanism can be circumvented by more advanced attackers. To reduce abuse, we recommend that the use of this verification mechanism be combined with legal measures that penalize circumvention.

The proposed mechanism aims to verify that the reporting user has a phone that received a notification. This mechanism satisfies the following requirements:

1. The mechanism should not require changes to the proximity tracing protocol used by Google and Apple.
2. The mechanism should not require changes to the Google/Apple Exposure Notification API.
3. The app should not need additional permissions (e.g., access to the device's location or phone number).
4. The mechanism can be used during a single phone conversation with a call center operator.
5. The mechanism does not introduce additional privacy risks for users. The verifier should not learn any information that is not epidemiologically relevant, such as which COVID-19 positive users contributed to the user's exposure.

Limits of validation mechanisms

We deliberately do not require that this mechanism verifies that the phone that received the notification **belongs to** the user that claims to have been notified. For example, we do not check if a parent makes a false claim based on a notification on one of their childrens' smartphones.

It is likely possible to (weakly) bind a user's device to a user identity and then use this during the verification process. However, we expect that such a mechanism requires the app to request the user to enter personal data manually on first install (e.g., name, date of birth, or phone number). We feel this approach is undesirable, as it gives the perception that the app gathers (and uses) private data, and may introduce data protection compliance requirements for the app.

The above requirements preclude strong cryptographic protection. First, the only information that is exchanged in the Google/Apple EN protocol during a contact event are the EphIDs. The Google/Apple EN protocol (as well as the DP3T low-cost protocol) publishes information from which the EphIDs of COVID-19 positive users can be derived. Therefore, there is **no secret information** that is available only to users that were in actual physical proximity to a COVID-19 positive user. Users that were not in physical proximity will also receive these EphIDs when the COVID-19 positive keys are published.

Second, the Google/Apple Exposure Notification API precludes any access to received EphIDs for privacy reasons. So validation mechanisms cannot access matching EphIDs, and thus cannot exploit the fact that genuinely notified users receive EphIDs *before* they are published.

If modifications to the protocol and API were allowed, cryptographically strong mechanisms would likely be possible. However, we wish to emphasize that these should retain the privacy properties of the original protocol. For example, the mechanism should not reveal edges in the social graph, and thus not reveal *which* EphID matched.

Furthermore, any "proofs" generated by such a mechanism should be small enough to be transferred over a low-bandwidth channel such as a phone call. If the proofs are instead transferred via the internet, the mechanism should take care to provide a dummy traffic mechanism to hide notified-status from network adversaries.

A simple validation mechanism

We present a simple remote validation mechanism. This mechanism can be used during a phone conversation with a hotline operator. The mechanism has been designed to be easy to use by both users and hotline operators.

We make the following assumptions:

a. When the user is notified by the app that their exposure is above the threshold, the app informs the user of the *date* on which the exposure exceeded the threshold.

b. Users inform hotline operators of this *date.*

This assumption is compatible with our requirements. One, the Google/Apple Exposure Notification API returns this date. Two, communicating this date to the hotline is essential for medical reasons. The exposure date is used to determine how long the user should self-quarantine.

We propose the following mechanism:

1. Before the user calls the hotline, the app prominently displays to the user the exposure date T_e and a 6-digit confirmation code, which the device computes as:

$$\texttt{code = TRUNCATE(HKDF(tweak, } T_e \texttt{ || } T_{now} \texttt{))}$$

where `HKDF` uses `SHA256`, `TRUNCATE` reduces the 256 bits output to a 6-digit response code, the tweak `tweak` is a value that is only known to the app and operators. The value T_e is the exposure date encoded as the start of the corresponding UNIX Epoch day in milliseconds since UNIX epoch and T_{now} is the current timestamp when generating the verification code encoded as milliseconds since UNIX epoch and rounded down to a 5-minute multiple.

The app should recommend that the user writes these values down before calling the hotline.

2. When calling the hotline, the user informs the operator of their exposure date T_e and the confirmation code code. The operator enters T_e and `code` into the system.
3. The operator's system computes the confirmation codes for the last half hour and compares them against the supplied code. The system signals the operator if the code is not correct. (Comparing against the last few codes lets the system validate older codes.)

This mechanism satisfies our requirements. It does not require modification to the EN protocols or APIs, does not require extra permissions, works during a single phone call, and does not reduce the privacy of users.

The tweak `tweak` can either be encoded into the app, or retrieved from the backend server after successful attestation and then stored in secure storage. However, we recommend making the value of `tweak` public to ensure verifiability, i.e., that the system uses the same value for all users.

Analysis of the mechanism

As long as the `tweak` value is secret, users only have a small probability of producing the correct validation code `code`. For example, when using 6 digit responses and a half-hour window, this probability is 6 in a million.

It is difficult to unconditionally protect the value `tweak` from tech-savvy users that might decompile the application or circumvent the attestation check. Only providing the (recent) value `tweak` to apps after successful attestation, does make it harder for tech-savvy users to obtain it.[9] Considering the challenges with attestation described above, we do not, at this time, recommend its use.

After obtaining the value `tweak`, tech-savvy users can compute correct responses despite not having been notified.

[9] We deliberately do not advocate for performing an attestation *during* the verification process with the hotline. While a "live" attestation further raises the bar, the attestations are too large to transmit via the phone, and therefore induce yet another network side-channel that must be protected against network adversaries.

By themselves, we expect that ordinary users are not able to compute the correct response, even if they would know the correct value of `tweak`. However, tech-savvy users could set up a validation-service in the form of a website or an app that performs the necessary computation on behalf of the ordinary user.

An offline version

Alternatively, it is possible to use an offline version of the above protocol where the validation codes are *not* verified during the conversation. This has the advantage that the hotline system does not need to know the value `tweak` and does not need to have a system in place to compute verification codes.

Instead, the hotline stores the exposure date T_e, the time of the call T_{now}, and the validation code `code` so that they can be verified at a later time.

Recommendations

The verification mechanism raises the bar somewhat against ordinary users making false claims. However, it is only a small part in a more complex system needed to validate notifications. In particular, we recommend to only deploy it in combination with additional mechanisms to reduce abuse:

1. Use legal measures to penalize fake notification reports.
2. Monitor the availability of services and apps that generate codes.
3. Monitor the number of notifications and compare against predictions based on the number of positive diagnoses.