Coronavirus Contact Tracing App Privacy: What Data Is Shared by the Singapore OpenTrace App?

Douglas J. Leith(✉) and Stephen Farrell

School of Computer Science and Statistics, Trinity College Dublin, Dublin, Ireland
doug.leith@tcd.ie, stephen.farrell@scss.tcd.ie

Abstract. We report on measurements of the actual data transmitted to backend servers by the Singapore OpenTrace app, with a view to evaluating impacts on user privacy. We have three main findings: 1) The OpenTrace app uses Google's Firebase service to store and manage user data. This means that there are two main parties involved in handling data transmitted from the app, namely Google and the health authority operating the OpenTrace app itself. We find that OpenTrace's use of Firebase Analytics telemetry means the data sent by OpenTrace potentially allows the (IP-based) location of user handsets to be tracked by Google over time. We therefore recommend that OpenTrace be modified to disable use of Firebase Analytics. 2) OpenTrace also currently requires users to supply a phone number to use the app and uses the Firebase Authentication service to validate and store the entered phone number. The decision to ask for user phone numbers (or other identifiers) presumably reflects a desire for contact tracers to proactively call contacts of a person that has tested positive. Alternative designs make those contacts aware of the positive test, but leave it to the contact to initiate action. This may indicate a direct trade-off between privacy and the effectiveness of contact tracing. If storage of phone numbers is judged necessary we recommend changing OpenTrace to avoid use of Firebase Authentication for this. And finally, 3) the reversible encryption used in OpenTrace relies on a single long-term secret key stored in a Google Cloud service and so is vulnerable to disclosure of this secret key.

Keywords: Contact tracing · Covid · Privacy · Firebase

1 Introduction

There is currently a great deal of interest in the use of mobile apps to facilitate Covid-19 contact tracing. More efficient and scalable contact tracing might for example, allow the lockdown measures currently in place in many countries to be relaxed more quickly.

Ensuring that contact tracing apps maintain user privacy has been widely flagged as being a major concern. Bluetooth-based proximity detection

N. Park et al. (Eds.): SecureComm 2020, LNICST 335, pp. 80–96, 2020.
https://doi.org/10.1007/978-3-030-63086-7_6

approaches are appealing from a privacy viewpoint since they avoid the need to record or share user location. However, care is needed in the implementation of these approaches to ensure that privacy goals are actually achieved. In addition, independent evaluation of developed apps is important both to verify privacy claims and to build confidence in users that the apps are indeed safe to use. In this report we take a first step in this direction. We measure the actual data transmitted to backend servers by the Singapore TraceTogether/OpenTrace app with a view to evaluating user privacy.

Mobile apps are already being used in several countries to assist with management of Covid-19 infections. Most of these apps are either used to control the movement of people or to assist people to make an initial self-evaluation of their health based on observed symptoms, rather than for contact tracing. A notable example of the former is the Chinese Health Code app [1], and an example of latter is the Spanish self-evaluation app [8]. It is worth noting that some countries use centralised tracking of mobile phone location, which avoids the need for a specialised app, to control movement of people, e.g. Taiwan's "electronic fence" [3]. In South Korea information on locations visited by infected people are made publicly available and this has prompted the development of apps that display this information e.g. the Corona 100m app [5].

In contrast, in the initial stages of the Covid-19 outbreak the Singapore government developed the TraceTogether app [16] and used this to directly assist with contact tracing. The TraceTogether app uses Bluetooth to broadcast beacons while also logging the signal strength of beacons received from neighbouring handsets. Since received signal strength is a rough indicator of proximity, when a person is detected as being infected the data logged on their phone can be used to identify other people that the person was potentially close in the time preceding discovery of their infection. In Europe Bluetooth-based approaches for contract tracing have also been proposed, e.g. Decentralised Privacy-Preserving Proximity Tracing (DP-3T) [9], while Apple and Google have formed a partnership to develop a contract tracing API based on Bluetooth [2]. However, these initiatives are currently at a relatively early stage whereas the Singapore TraceTogether app is already deployed and operational, plus an open source version, referred to as OpenTrace, has now been released [15]. TraceTogether/OpenTrace is therefore currently the focus of much interest.

This work is solely based on the open-source OpenTrace app and the installable app from the Google Play store without any involvement of the developers of the app or any health authority.

The results of our study can be summarised as follows.

We find that the OpenTrace app uses Google's Firebase service to store and manage user data. This means that there are two main parties involved in handling data shared by the app, namely Google (who operate the Firebase service) and the health authority (or other agency) operating the OpenTrace app itself. As owner of Firebase, Google has access to all data transmitted by the app via Firebase but filters what data is made available to the operator of OpenTrace e.g. to present only aggregate statistics.

The OpenTrace app regularly sends telemetry data to the Firebase Analytics service. This data is tagged with persistent identifiers linked to the mobile handset so that messages from the same handset can be linked together. Further, messages also necessarily include the handset IP address (or the IP address of the upstream gateway), which can be used as a rough proxy for location using existing geoIP services. Note that the Firebase Analytics documentation [11] states that "Analytics derives location data from users' IP addresses". Hence, the data sent by the handset potentially allows its location to be tracked over time. Many studies have shown that location data linked over time can be used to de-anonymise: this is unsurprising since, for example, knowledge of the work and home locations of a user can be inferred from such location data (based on where the user mostly spends time during the day and evening), and when combined with other data this information can quickly become quite revealing [20,21].

The Firebase Analytics documentation states that "Thresholds are applied to prevent anyone viewing a report from inferring the demographics, interests, or location of individual users" [11]. Assuming this is effective (note that the effectiveness of de-anonymisation methods is far from clear when applied to location data over time), then the health authority operating the OpenTrace app cannot infer individual user locations. The primary privacy concern therefore lies with the holding of rough location data by Google itself. It is worth noting that when location history can be inferred from collected data then even if this inference is not made by the organisation that collects the data it may be made by other parties with whom data is shared. This includes commercial partners (who may correlate this with other data in their possession) and state agencies, as well as disclosure via data breaches.

In light of this, and since the potential use of a contract tracing app for large-scale tracking of the population is one of the main privacy concerns highlighted in the media, we recommend disabling use of Firebase Analytics in OpenTrace to avoid this regular transmission of messages to Google.

OpenTrace requires users to enter their phone number in order to use the app, and this number is stored on Firebase and visible to the health authority. It is relatively easy to link a phone number to a users real identity (indeed in some jurisdictions ID must be presented when buying a sim [17]) and so this creates an immediate privacy concern. The BlueTrace white paper [6] notes that storage of user phone numbers is optional and push notifications can be used instead. The decision as to whether to ask for user phone numbers (or other identifiers) therefore really depends on the requirements of the health authority for effectively managing contact tracing. For example, due to the approximate nature of proximity tracking via Bluetooth it is likely that OpenTrace data will be only one of many sources of information used in contact tracing and combining data from different sources may require the use of an identifier such as a phone number. The result may be a direct trade-off between privacy and the effectiveness of contact tracing.

Assuming that recording of phone numbers, or similar identifiers, is judged to be necessary then OpenTrace uses the Firebase Authentication service for

this purpose. This use of Firebase Authentication creates an obvious potential conflict of interest for Google whose primary business is advertising based on collection of user personal data. In addition, the data held by Google need not be stored in the country where the app users are located. In particular, the Firebase privacy documentation [4] states that the Firebase Authentication service used by OpenTrace always processes its data in US data centres. Bearing in mind that a successful contact tracing app would be used by a large fraction of the population in a country, use of Firebase Authentication potentially means that their phone numbers may then be available to US state agencies. Such considerations suggest that it might be worth considering changing OpenTrace to make use of backend infrastructure that avoids outsourcing storage of user phone numbers to Google.

2 Threat Model: What Do We Mean by Privacy?

It is important to note that transmission of user data to backend servers is not intrinsically a privacy intrusion. For example, it can be useful to share details of the user device model/version and the locale/country of the device and this carries few privacy risks if this data is common to many users since the data itself cannot then be easily linked back to a specific user [19,22].

Issues arise, however, when data can be tied to a specific user. One common way that this can happen is when an app generates a long randomised string when first installed/started and then transmits this alongside other data. The randomised string then acts as an identifier of the app instance (since no other apps share the same string value) and when the same identifier is used across multiple transmissions it allows these transmissions to be tied together across time.

Linking a sequence of transmissions to an app instance does not explicitly reveal the user's real-world identity. However, the data can often be readily de-anonymised. One way that this can occur is if the app directly asks for user details (e.g. phone number, facebook login). But it can also occur indirectly using the fact that transmissions by an app always include the IP address of the user device (or more likely of an upstream NAT gateway). As already noted, the IP address acts as a rough proxy for user location via existing geoIP services and many studies have shown that location data linked over time can be used to de-anonymise. A pertinent factor here is the frequency with which updates are sent e.g. logging an IP address location once a day has much less potential to be revealing than logging one every few minutes.

With these concerns in mind, two of the main questions that we try to answer in the present study are (i) What explicit identifying data does the app directly request and (ii) Does the data that the app transmits to backend servers potentially allow tracking of the IP address of app instance over time.

3 Measurement Setup

3.1 Viewing Content of Encrypted Web Connections

All of the network connections we are interested in are encrypted. To inspect the content of a connection we route handset traffic via a WiFi access point (AP) that we control. We configure this AP to use mitmdump [18] as a proxy and adjust the firewall settings to redirect all WiFi traffic to mitmdump so that the proxying is transparent to the handset. In brief, when the OpenTrace app starts a new web connection the mitmdump proxy pretends to be the destination server and presents a fake certificate for the target server. This allows mitmdump to decrypt the traffic. It then creates an onward connection to the actual target server and acts as an intermediary relaying requests and their replies between the app and the target server while logging the traffic. The setup is illustrated schematically in Fig. 1.

The immediate difficulty encountered when using this setup is that the app carries out checks on the authenticity of server certificates received when starting a new connection and aborts the connection when these checks fail. To circumvent these checks we use a rooted phone and use Frida [13] to patch the OpenTrace app and Google Play Services (which the app uses to manage most of the connections it makes) on the fly to replace the relevant Java certificate validation functions with dummy functions that always report validation checks as being passed. The bulk of the effort needed lies in deducing which functions to patch since Google Play Services is closed-source and obfuscated (decompiling the bytecode produces Java with randomised class and variable names etc.) plus it uses customised certificate checking code (so standard unpinning methods fail). Implementing the unpinning for OpenTrace is therefore a fairly laborious manual process.

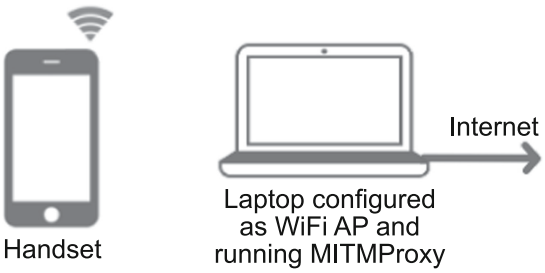

Fig. 1. Measurement setup. The mobile handset is configured to access the internet using a WiFi access point hosted on a laptop, use of cellular/mobile data is disabled. The laptop also has a wired internet connection. When an app on the handset starts a new web connection the laptop pretends to be the destination server so that it can decrypt the traffic. It then creates an onward connection to the actual target server and acts as an intermediary relaying requests and their replies between the handset app and the target server while logging the traffic.

3.2 Hardware and Software Used

Mobile handset: Google Pixel 2 running Android 9. Rooted using Magisk v19.1 and Magisk Manager v7.1.2 and running Frida Server v12.5.2. Laptop: Apple Macbook running Mojav 10.14.6 running Frida 12.8.20 and mitmproxy v5.0.1. Using a USB ethernet adapter the laptop is connected to a cable modem and so to the internet. The laptop is configured using its built in Internet Sharing function to operate as a WiFi AP that routes wireless traffic over the wired connection. The laptop firewall is then configured to redirect received WiFi traffic to mitmproxy listening on port 8080 by adding the rule `rdr pass on bridge100 inet proto tcp to any port 80, 443 -> 127.0.0.1 port 8080`. The handset is also connected to the laptop over USB and this is used as a control channel (no data traffic is routed over this connection) to install the OpenTrace app and carry out dynamic patching using Frida. Namely, using the adb shell the Frida server is started on the handset and then controlled from the laptop via the Frida client.

3.3 Test Design

Test design is straightforward since the OpenTrace app supports only a single flow of user interaction. Namely, on first startup a splash screen is briefly displayed and then an information screen is shown that contains a single "I want to help" button. On pressing this the user is taken to a second screen which outlines how OpenTrace works and which has a single button labelled "Great!!!". On pressing this the user is asked to enter their phone number and again there is only a single button labelled "Get OTP". The user is then taken to a screen where they are asked to enter a 6-digit code that has been texted to the supplied number. On pressing the "Verify" button then if this code is valid the user is taken through a couple of screens asking then to give the app needed permissions (Bluetooth, location, disabling of battery optimisation for the app, access to storage) and then arrives at the main screen which is displayed thereafter, see Fig. 2. This main screen has non-functioning buttons for help and sharing of the app, plus a button that is only to be pressed when the user has been confirmed as infected with Covid-19 and which uploads observed bluetooth data to the app backend server hosted on Firebase.

Testing therefore consists of recording the data sent upon installation and startup of the app, followed by navigation through these screens until the main screen is reached. The data sent by the app when left idle at the main screen (likely the main mode of operation of the app) is also recorded. We also tried to investigate the data sent upon pressing the upload function but found that this functionality fails with an error. Inspection of the code suggests that this upload functionality is incomplete.

Although our primary interest is in the open source OpenTrace app, this app is apparently derived from the closed-source TraceTogether app used by the health service in Singapore. We therefore also tried to collect data for the TraceTogether app for comparison with that generated by the OpenTrace app.

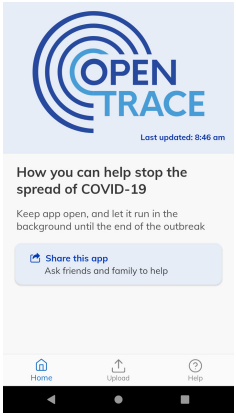

Fig. 2. Main screen displayed by OpenTrace following initial setup.

The latest version (v1.6.1) of TraceTogether is restricted to Singapore phone numbers, in our testing we therefore used an earlier version (v1.0.33) without this restriction in our tests.

3.4 Finding Identifiers in Network Connections

Potential identifiers in network connections were extracted by manual inspection. Basically any value present in network messages that stays the same across messages is flagged as a potential identifier. As we will see, almost all the values of interest are associated with the Firebase API that is part of Google Play Services. We therefore try to find more information on the nature of observed values from Firebase privacy policies and other public documents as well as by comparing them against known software and device identifiers e.g. the Google advertising identifier of the handset.

4 Google Firebase

OpenTrace uses Google's Firebase service to provide its server backend. This means that there are at least two parties involved in handling data shared by the app, namely Google (who operate the Firebase service infrastructure) and the health authority (or other agency) operating the OpenTrace app itself. As owner of Firebase, Google has access to all data transmitted by the app via Firebase but filters what data is made available to the operator of OpenTrace e.g. to present only aggregate statistics.

OpenTrace makes use of the Firebase Authentication, Firebase Functions, Firebase Storage and Firebase Analytics (also referred to as Google Analytics for Firebase) services. The app has hooks for Crashlytics and Firebase Remote Config, but the version studied here does not make active use of these two services.

The Firebase Authentication service is used on startup of the app to record the phone number entered by the user and verify it by texting a code which the user then enters into the app. The phone numbers entered are recorded by Firebase and linked to a Firebase identifier.

Firebase Functions allows the OpenTrace app to invoke execution of user defined Javascript functions on Google's cloud platform by sending requests to specified web addresses. The OpenTrace app uses this service to generate tempIDs for broadcast over bluetooth and for upload of logged tempIDs upon the user becoming infected with Covid-19. Firebase Storage is used to hold uploaded tempIDs. The tempIDs are generated by reversible encryption (see below) using a key stored in Google Cloud's Secret Manager service and accessed by the OpenTrace getTempIDs function hosted on Firebase Functions.

Figure 3 shows an example of the Firebase Functions logging visible to the operator of the OpenTrace app. This fine-grained logging data shows individual function calls together with the time and user making the call (the uid value is the user identifier used by Firebase Authentication and so can be directly linked to the users phone number).

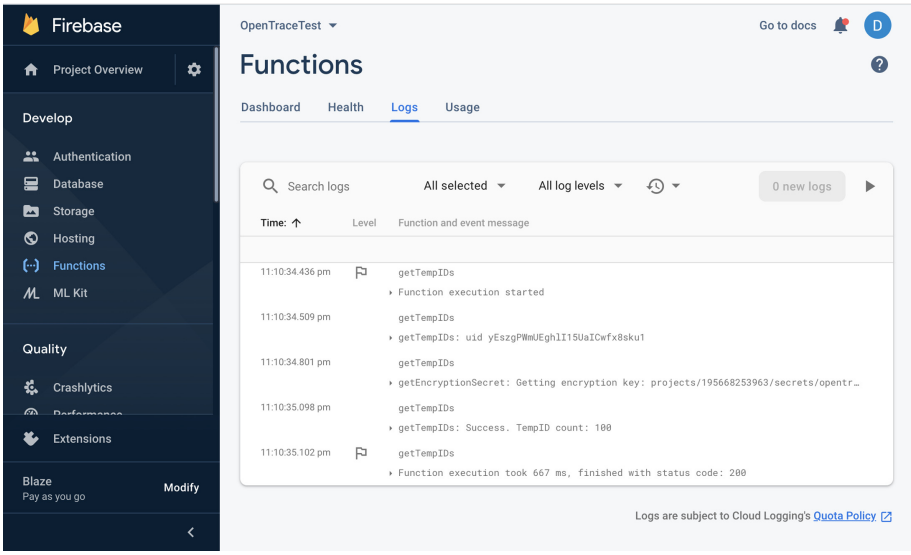

Fig. 3. Example of Firebase Functions logging visible to the operator of the OpenTrace app. Observe that there is fine-grained logging of individual function calls per user (the uid value in these logs is a unique identifier linked to a users phone number). The tempIDs function is, for example, regularly called by the OpenTrace app to refresh the set of tempIDs available for a mobile handset to advertise on Bluetooth.

The app is instrumented to record a variety of user events and log these to the backend server using Firebase Analytics.

The Firebase privacy documentation [4] outlines some of the information that is exchanged with Google during operation of the API. This privacy documentation does not state what is logged by Firebase Storage but notes that Firebase Authentication logs user phone numbers and IP addresses. Also that Firebase Analytics makes use of a number of identifiers including: (i) a user-resettable Mobile ad ID to "allow developers and marketers to track activity for advertising purposes. They're also used to enhance serving and targeting capabilities." [14], (ii) an Android ID which is "a 64-bit number (expressed as a hexadecimal string), unique to each combination of app-signing key, user, and device" [7], (iii) a InstanceID that "provides a unique identifier for each app instance" and "Once an Instance ID is generated, the library periodically sends information about the application and the device where it's running to the Firebase backend." [12] and (iv) an Analytics App Instance ID that is "used to compute user metrics throughout Analytics" [11]. The Firebase Analytics documentation [11] states that "As long as you use the Firebase SDK, you don't need to write any additional code to collect a number of user properties automatically", including Age, Gender, Interests, Language, Country plus a variety of device information. It also states that "Analytics derives location data from users' IP addresses".

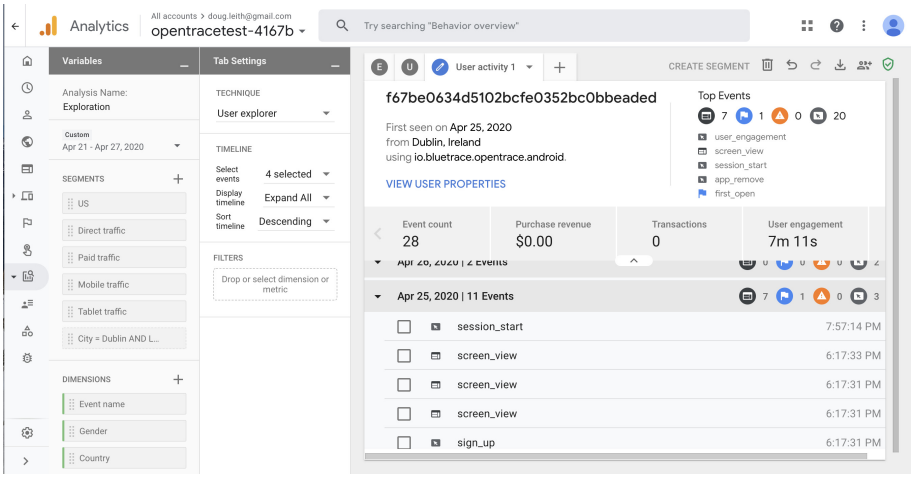

Fig. 4. Example of Firebase Analytics data visible to the operator of the OpenTrace app. Observe that data is available on events occurring per individual device.

Figure 4 shows an example of the data made available to the operator of the OpenTrace app by Firebase Analytics. It can be seen that per device event data is available showing for example when OpenTrace is started on the device, when it is viewed etc.

The data collected by Google during operation of its Firebase services need not be stored in the same country as the user of an app is located. The Firebase

privacy documentation [4] states that "Unless a service or feature offers data location selection, Firebase may process and store your data anywhere Google or its agents maintain facilities". It also states "A few Firebase services are only run from US data centers. As a result, these services process data exclusively in the United States" and it appears that these services include Firebase Authentication, which OpenTrace uses to stores user phone numbers.

It is important to note that only a filtered version of this data collected by Google is made available to users of its backend Firebase services. The Firebase Analytics documentation states that "Thresholds are applied to prevent anyone viewing a report from inferring the demographics, interests, or location of individual users" [11].

5 Cryptography

The tempIds used in OpenTrace are transmitted in Bluetooth beacons and are an encrypted form of a Firebase user identifier that can be linked to the user phone number using the Firebase Authentication service, together with two timestamps indicating the time interval during which the tempID is valid (to mitigate replay attacks).

In OpenTrace encryption is based on a single long-term symmetric secret that is stored in Google Cloud's Secret Manager service. The encryption is reversible so that the user identifier and timestamps can be recovered from an observed tempID given knowledge of this secret. When a person is detected to be infected with Covid-19 the tempIDs in beacons observed by the app on their phone can therefore be decrypted by the health authority to obtain the phone numbers of people who have been in proximity to the infected person. However, this setup also means that if a data breach occurs and the secret is disclosed then any recordings of tempIDs observed in Bluetooth beacons can also be decrypted by third parties. Alternative designs that ensure that only the tempIds associated with a user testing positive for Covid-19 could be decrypted would seem more desirable. It is also important to add provision for key updates and other key management, which is currently absent in OpenTrace (Table 1).

6 Measurements of Data Transmitted by OpenTrace App

6.1 Data Sent on Initial Startup

Upon installation and first startup the OpenTrace app makes a number of network connections. Note that there has been no interaction with the user beyond startup of the app, in particular no user consent to sharing of information.

The app initialises Firebase upon first startup, which generates the following POST request (standard/uninteresting parameters/headers are omitted):

```
POST https://firebaseinstallations.googleapis.com/v1/projects/opentracetest-4167b/
installations
Parameters:
   key=AIzaSyAB...
```

Table 1. Summary of network connections made by OpenTrace in response to user interactions.

Startup of app
• firebaseinstallations.googleapis.com (sends fid)
• settings.crashlytics.com
• app-measurement.com (sends app_instance_id, Google Advertising Id)
Entering phone number
• www.googleapis.com (sends phone number)
Entering 6-digit code
• www.googleapis.com (sends 6-digit code, receives)
• europe-west1-opentracetest-4167b.cloudfunctions (fetches tempIDs)
Agreeing to app permissions (bluetooth, location, battery)
Agreeing to app storage permission
• europe-west1-opentracetest-4167b.cloudfunctions (fetches tempIDs)
While idle
• app-measurement.com (app_instance_id, Google Advertising Id)

```
Headers:
   X-Android-Cert: 2AE4BED9E4F0...
Request body:
   {"appId": "1:195668253963:android:0e1d8...", <...>
    "fid": "f4vnM2vqSLuOgcpB8FbDd_",<...> }
```

The value of the parameter "key" is hardwired into the app to allow it to access Firebase, and so is the same for all instances of the app. Similarly, the X-Android-Cert header is the SHA1 hash of the app and the "appId" is a Google app identifier and both are the same for all instances of the app. The "key" parameter and X-Android-Cert appear in many of the network requests made by the app but to save space are omitted from now on. The "fid" value appears to be the Firebase Instance ID, discussed above. This uniquely identifies the current instance of the app but is observed to change for a fresh install of the app (i.e. deleting the app and then reinstalling). The response to this request echoes the fid value and includes two tokens which appear to be used to identify the current session.

Next, OpenTrace tries to fetch settings for Crashlytics:

```
GET https://settings.crashlytics.com/spi/v2/platforms/android/apps/io.bluetrace.
opentrace/settings
Parameters:
   instance: da2618e19123d5...
   display_version: 1.0.41-debug-b39f5...
   icon_hash: 8229e07efb...
Headers:
```

```
X-CRASHLYTICS-DEVELOPER-TOKEN: 470fa2b4ae...
X-CRASHLYTICS-API-KEY: 129fa51a5bbee6...
X-CRASHLYTICS-INSTALLATION-ID: e8854e81...
```

The "instance" parameter differs from the fid, its value is observed to changes upon a clean install of the app. Similarly, the X-CRASHLYTICS-INSTALLATION-ID value changes upon a clean install. These two values appear to be used to identify the instance of Crashlytics (and so the app). The "display_version" parameter is the OpenTrace VERSION_NAME value from BuildConfig.java in the app source code and is the same for all copies of the app. Its not clear what the X-CRASHLYTICS-DEVELOPER-TOKEN, X-CRASHLYTICS-API-KEY and icon_hash values are, but they are observed to stay unchanged upon fresh install of the app and so do not seem to be tied to the app instance. The response to the request to settings.crashlytics.com is "403 Forbidden" since Crashlytics has not been configured in the Firebase server backend.

OpenTrace now makes its first call to Firebase Analytics:

```
GET https://app-measurement.com/config/app/1%3A195668253963%3Aandroid%3
A0e1d84bec59ca7e66e160e
 Parameters:
   app_instance_id: f67be0634d5102bcfe0352bc0bbeaded

 POST https://app-measurement.com/a
  <...>
 \x02_o\x12\x04auto\x07\x03_et\x18\x01\x12\x02_e\x18\xd0\xa4\xdb\x92\x9b. \x00\x1a\
 x14\x08\xd0\xa4\xdb\x92\x9b.\x12\x04_fot \x80\x82\xfc\x93\x9b.\x1a\x0e\x08\xd0\xa4
 \xdb\x92\x9b.\x12\x03_fi \x01\x1a\x0f\x08\xee\x9a\xdc\x92\x9b.\x12\x04_lte \x01\
 x1a\x0e\x08\xef\x9a\xdc\x92\x9b.\x12\x03_se \x01 \xec\x9a\xdc\x92\x9b.(\xd0\xa4\
 xdb\x92\x9b.0\xd0\xa4\xdb\x92\x9b.B\x07androidJ\x019R\x07Pixel 2Z\x05en-us‘<j\
 x0emanual_installr\x16io.bluetrace.opentrace\x82\x01\x191.0.41-debug-b39f57f-F4D3\
 x88\x01\xa0\xac\x01\x90\x01\xf9\x8a\x01\x9a\x01$1d2635f5-2af7-4fb3-86e...\xa0\x01\
 x00\xaa\x01 f67be0634d5102bcfe0352bc0bbeaded\xb0\x01\xda\x8f\xd6\xd8\xe4\xe5\xa5\
 xf4\x8e\x01\xb8\x01\x01\xca\x01-1:195668253963:android:0e1d84bec59ca7e66e160e\xe0\
 x01\x01\xf2\x01\x16f4vnM2vqSLuOgcpB8FbDd_\xf8\x01)\x98\x02\x98\x9b\xbe\xa5\xfa\xf8\
 xe8\x02\xa0\x02\x00\xe8\x02\xb2\xeb\x86\x0b\xf0\x02\x00
```

The first request appears to be asking about configuration changes and the response is "304 Not Modified". The second request uploads information on events within the app (see below for further discussion). Both requests contain an app_instance_id value which acts to link them together (and also subsequent analytics requests). The second request also contains the Firebase fid value. In addition the second request contains the device Google Advertising Id (1d2635f5-2af7-4fb3-86e...) as reported by the Google/Ads section of the handset Settings app. Unless manually reset by the user this value persists indefinitely, including across fresh installs of OpenTrace, and so essentially acts as a strong identifier of the device and its user. The body of the second request contains a number of other values. Some identify the version of the app, and so are not sensitive, but the provenance of other values is not known to the authors.

6.2 Data Sent upon Phone Number Entry

After initial startup, navigation through the first two information screens generates no network connections. At the this screen the app asks the user to enter their phone number. Upon doing this and pressing the button to proceed the

following network connections are made. The first connection sends the phone number to Firebase:

```
POST https://www.googleapis.com/identitytoolkit/v3/relyingparty/
sendVerificationCode
Headers:
   X-Goog-Spatula: CjYKFmlvLmJsdWV0cmFjZS5vc...
Request Body:
   1: <phone number>
```

It is not clear how the X-Goog-Spatula value is generated but it seems to consist, at least in part, of base64-encoded information since base64-decoding yields "`6 ?io.bluetrace.opentrace? KuS+2eTwbmFXe/8epaO9wF8yVTE=`" followed by additional binary data.

Firebase now texts a 6-digit code to the phone number entered and the OpenTrace app asks the user to enter this code. Entering the code generates the following network connection:

```
POST https://www.googleapis.com/identitytoolkit/v3/relyingparty/verifyPhoneNumber
 Headers:
   X-Goog-Spatula: CjYKFmlvLmJsdWV0cmFjZS5vc...
Request body:
   1: AM5PThB...
   3: <6-digit code entered>
```

The X-Goog-Spatula header value is the same as for the previous request and so can be used to link them together (perhaps it's a form of short session id). The AM5PThB... value is from the response to the first request and presumably encodes the 6-digit value in a way that the server can decode so as to compare against the value entered by the user.

The response to a correct 6-digit code informs the app of the user id value `yEszgPWm...` used by Firebase (which is visible on the Firebase dashboard and directly linked to the users phone number), together with a number of other values including what seems to be a user identifier/authentication token that is used in the body of the following request:

```
POST https://www.googleapis.com/identitytoolkit/v3/relyingparty/getAccountInfo
 Headers:
   X-Goog-Spatula: CjYKFmlvLmJsdWV0cmFjZS5vc...
Request Body:
   1: eyJhbGciOiJSUzI1NiIsI...
```

The response contains the phone number previously entered by the user together with the user id value `yEszgPWm...` and some timestamps (presumably associated with account creation etc.).

At this point an account for the user has been successfully created/authenticated on Firebase and OpenTrace now uses the Firebase Functions service to request a batch of tempIDs:

```
POST https://europe-west1-opentracetest-4167b.cloudfunctions.net/getTempIDs
```

to which the response is 14KB of json:

```
{"result": {"refreshTime": 1587878233, "status": "SUCCESS",
    "tempIDs": [
        { "expiryTime": 1587835873,"startTime": 1587834973,
          "tempID": "RQVK+en..." },
        <...>
```

OpenTrace then makes a call to the getHandshakePin function hosted by Firebase Functions:

```
POST https://europe-west1-opentracetest-4167b.cloudfunctions.net/getHandshakePin
```

and the response contains the PIN that the health service operating the app needs to present to the user in order to confirm they should ask the app to upload the observed tempIDs to Firebase Storage.

6.3 Data Sent When Permissions Are Granted

After phone number entry and verification, the app asks the user for permission to use bluetooth, location and to disable battery optimisation for OpenTrace. These interactions do not generate any network connections. Finally OpenTrace asks for permission to access file storage on the handset. When this is granted OpenTrace makes a second call to https://europe-west1-opentracetest-4167b.cloudfunctions.net/getTempIDs and receives a further batch of tempIDs in response.

6.4 Data Sent When Sitting Idle at Main Screen

Once startup of OpenTrace is complete it sits idle at a main screen until the user, and it is in this mode of operation that it spends the bulk of its time. Roughly once an hour OpenTrace is observed to make a pair of connections to Firebase Analytics. This is consistent with Firebase Analytics documentation [10] which says that "analytics data is batched up and sent down when the client library sees that there's any local data that's an hour old." and "on Android devices with Play Services, this one hour timer is across all apps using Firebase Analytics".

The first connection of the pair:

```
GET https://app-measurement.com/config/app/1%3A195668253963%3Aandroid%3
A0e1d84bec59ca7e66e160e
Parameters:
   app_instance_id: f67be0634d5102bcfe0352bc0bbeaded
```

appears to be checking for updates, to which the response is 304 Not Modified.

The second connection is also to app-measurement.com and appears to send telemetry data logging user interactions with the app. When configured for verbose logging Firebase writes details of the uploaded data to the handset log, which can inspected over the USB connection to the handset using the "adb logcat" command. A typical entry log entry starts as follows:

```
protocol_version: 1
platform: android
gmp_version: 22048
uploading_gmp_version: 17785
dynamite_version: 0
config_version: 1587452740341144
gmp_app_id: 1:195668253963:android:0e1d84...
admob_app_id:
app_id: io.bluetrace.opentrace
app_version: 1.0.41-debug-b39f57f-F4D3
app_version_major: 41
firebase_instance_id: f4vnM2vqSLuOgcpB8FbDd_
```

```
dev_cert_hash: -8149097300642920486
app_store: manual_install
upload_timestamp_millis: 1587823341351
start_timestamp_millis: 1587823321657
end_timestamp_millis: 1587823321670
previous_bundle_start_timestamp_millis: 1587822575641
previous_bundle_end_timestamp_millis: 1587822932936
app_instance_id: f67be0634d5102bcfe0352bc0bbeaded
resettable_device_id: 1d2635f5-2af7-4fb3-86e...
device_id:
ds_id:
limited_ad_tracking: false
os_version: 9
device_model: Pixel 2
user_default_language: en-us
time_zone_offset_minutes: 60
bundle_sequential_index: 14
service_upload: true
health_monitor:
```

Inspection of the message transmitted over the network to Firebase shows a section of the following form:

```
\x07androidJ\x019R\x07Pixel 2Z\x05en-us‘<j\x0emanual_installr\x16io.bluetrace.
opentrace\x82\x01\x191.0.41-debug-b39f57f-F4D3\x88\x01\xa0\xac\x01\x90\x01\xf9\x8a
\x01\x9a\x01$1d2635f5-2af7-4fb3-86e...\xa0\x01\x00\xaa\x01
f67be0634d5102bcfe0352bc0bbeaded\xb0\x01\xda\x8f\xd6\xd8\xe4\xe5\xa5\xf4\x8e\x01\
xb8\x01\x03\xca\x01-1:195668253963:android:0e1d84...\xd0\x01\x89\xc7\xdb\x92\x9b.\
xe0\x01\x01\xf2\x01\x16f4vnM2vqSLuOgcpB8FbDd_\xf8\x01)\x98\x02\x98\x9b\xbe\xa5\xfa\
xf8\xe8\x02\xa0\x02\x00\xe8\x02\xb2\xeb\x86\x0b\xf0\x02\x00
```

within which many of the values from the log can be identified, including the firebase_instance_id, the app_instance_id, the resettable_device_id (the Google Advertising Id). Each handset log entry also includes a sequence of user_property values of the form:

```
user_property {
  set_timestamp_millis: 1587790361696
  name: first_open_time(_fot)
  string_value:
  int_value: 1587790800000
}
```

followed by a sequence of event entries of the form

```
event {
  name: user_engagement(_e)
  timestamp_millis: 1587823321657
  previous_timestamp_millis: 1587822932919
  param {
    name: firebase_event_origin(_o)
    string_value: auto
  }
  <more param entries>
}
```

and these can also be identified within the body of the message sent by OpenTrace to Firebase Analytics.

6.5 Data Sent by TraceTogether (v1.0.33)

We repeated the above measurements using the closed-source TraceTogether app currently being used by the Singapore government. In summary, we observed similar behaviour to that seen with OpenTrace but with the following differences:

1. TraceTogether was not observed to download tempIDs from Firebase following initial startup. Presumably these are generated locally within the app, at least initially.
2. TraceTogether makes calls to asia-east2-govtech-tracer.cloudfunctions.net/getBroadcastMessage i.e. to a getBroadcastMessage function hosted on Firebase Functions but which is not present in OpenTrace.
3. TraceTogether makes calls to firebaseremoteconfig.googleapis.com and so presumably makes use of the Firebase Remote Config service (as already noted, there are hooks for this within OpenTrace, but these are not activated).

7 Summary and Conclusions

We have carried out an initial measurement study of the OpenTrace app and of a version of the related closed-source TraceTogether app deployed in Singapore. We find that the use of Google's Firebase Analytics service means the data sent by OpenTrace potentially allows the (IP-based) location of user handsets to be tracked by Google over time. We also find that OpenTrace stores user phone numbers in the Firebase Authentication service. This use of Firebase Authentication creates an obvious potential conflict of interest for Google whose primary business is advertising based on collection of user personal data. In addition, the Firebase Authentication service processes its data in US data centres even if users are located in other countries. Lastly, we note that OpenTrace relies on a single long-term secret key stored in a Google Cloud service and so is vulnerable to disclosure of this secret key. We plan to further investigate this and similar apps for privacy, security and efficacy in the coming weeks but would strongly recommend that anyone planning on using an app based on OpenTrace address these significant issues before deployment.

References

1. In Coronavirus Fight, China Gives Citizens a Color Code, With Red Flags, New York Times, 1 March 2020. https://www.nytimes.com/2020/03/01/business/china-coronavirus-surveillance.html
2. Apple and Google partner on COVID-19 contact tracing technology, 10 April 2020. https://www.apple.com/newsroom/2020/04/apple-and-google-partner-on-covid-19-contact-tracing-technology/
3. Coronavirus: Under surveillance and confined at home in Taiwan, BBC News, 24 March 2020. https://www.nytimes.com/2020/03/01/business/china-coronavirus-surveillance.html
4. Privacy and Security in Firebase, 27 November 2019. https://firebase.google.com/support/privacy

5. Coronavirus mobile apps are surging in popularity in South Korea, CNN, 28 February 2020. https://edition.cnn.com/2020/02/28/tech/korea-coronavirus-tracking-apps/index.html
6. BlueTrace: A privacy-preserving protocol for community-driven contact tracing across borders, 9 April 2020. https://bluetrace.io/static/bluetrace_whitepaper-938063656596c104632def383eb33b3c.pdf
7. Android Reference Guide: Android Id. https://developer.android.com/reference/android/provider/Settings.Secure.html#ANDROID_ID. Accessed 26 Apr 2020
8. CoronaMadrid Covid-19 App. https://www.coronamadrid.com/. Accessed 26 Apr 2020
9. Decentralised Privacy-Preserving Proximity Tracing (DP-3T) Demo App. https://github.com/DP-3T/dp3t-app-android. Accessed 26 Apr 2020
10. Firebase Blog: How Long Does it Take for My Firebase Analytics Data to Show Up? https://firebase.googleblog.com/2016/11/how-long-does-it-take-for-my-firebase-analytics-data-to-show-up.html. Accessed 26 Apr 2020
11. Firebase Help: Automatically collected user properties. https://support.google.com/firebase/answer/6317486. Accessed 26 Apr 2020
12. Firebase Reference Guide: FirebaseInstanceId. https://firebase.google.com/docs/reference/android/com/google/firebase/iid/FirebaseInstanceId. Accessed 26 Apr 2020
13. Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/. Accessed 26 Apr 2020
14. Google Ad Manager Help: About mobile advertising IDs. https://support.google.com/admanager/answer/6274238. Accessed 26 Apr 2020
15. OpenTrace Source Code. https://github.com/OpenTrace-community. Accessed 26 Apr 2020
16. TraceTogether App Website. https://www.tracetogether.gov.sg/. Accessed 26 Apr 2020
17. The Mandatory Registration of Prepaid SIM Card Users, GSMA White Paper, November 2013. https://www.gsma.com/publicpolicy/wp-content/uploads/2013/11/GSMA_White-Paper_Mandatory-Registration-of-Prepaid-SIM-Users_32pgWEBv3.pdf
18. Cortesi, A., Hils, M., Kriechbaumer, T., Contributors: mitmproxy: A free and open source interactive HTTPS proxy (v5.01) (2020). https://mitmproxy.org/
19. Machanavajjhala, A., Kifer, D., Gehrke, J., Venkitasubramaniam, M.: L-diversity: privacy beyond k-anonymity. ACM Trans. Knowl. Disc. Data (TKDD) **1**(1), 3-es (2007)
20. Golle, P., Partridge, K.: On the anonymity of home/work location pairs. In: Tokuda, H., Beigl, M., Friday, A., Brush, A.J.B., Tobe, Y. (eds.) Pervasive 2009. LNCS, vol. 5538, pp. 390–397. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01516-8_26
21. Srivatsa, M., Hicks, M.: Deanonymizing mobility traces: using social network as a side-channel. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 628–637 (2012)
22. Sweeney, L.: k-anonymity: A model for protecting privacy. Int. J. Uncertain. Fuzz. Knowl. Based Syst. **10**(05), 557–570 (2002)