Mobile Handset Privacy: Measuring The Data iOS and Android Send to Apple And Google

Douglas J. Leith
School of Computer Science & Statistics,
Trinity College Dublin, Ireland
25th March, 2021

Abstract—**We investigate what data iOS on an iPhone shares with Apple and what data Google Android on a Pixel phone shares with Google. We find that even when minimally configured and the handset is idle both iOS and Google Android share data with Apple/Google on average every 4.5 mins. The phone IMEI, hardware serial number, SIM serial number and IMSI, handset phone number etc are shared with Apple and Google. Both iOS and Google Android transmit telemetry, despite the user explicitly opting out of this. When a SIM is inserted both iOS and Google Android send details to Apple/Google. iOS sends the MAC addresses of nearby devices, e.g. other handsets and the home gateway, to Apple together with their GPS location. Users have no opt out from this and currently there are few, if any, realistic options for preventing this data sharing.**

I. Introduction

In this paper we investigate the data that mobile handset operating systems share with the mobile OS developer in particular what data iOS on an iPhone shares with Apple and what data Google Android on a Pixel phone shares with Google. While the privacy of mobile handsets has been much studied, most of this work has focussed on measurement of the app tracking/advertising ecosystem and much less attention has been paid to the data sharing by the handset operating system with the mobile OS developer. Handset operating systems do not operate in a standalone fashion but rather operate in conjunction with backend infrastucture. For example, handset operating systems check for updates to protect users from exploits and malware, to faciltate running of field trials (e.g. to test new features before full rollout), to provide telemetry and so on. Hence, while people are using an iPhone the iOS operating system shares data with Apple and when using a Pixel the operating system shares data with Google, and this is part of normal operation.

We define experiments that can be applied uniformly to the handsets studied (so allowing direct comparisons) and that generate repoducible behaviour. Both Apple and Google provide services that can be, and almost always are, used in conjunction with their handsets, e.g. search (Siri, OkGoogle), cloud storage (iCloud, Google Drive), maps/location services (Apple Maps, Google Maps), photo storage/analytics (Apple Photo, Google Photos). Here we try to keep these two aspects separate and to focus on the handset operating system in itself, separate from optional services such as these. We assume a privacy-conscious but busy/non-technical user, who when asked does not select options that share data with Apple and Google but otherwise leaves handset settings at their default value.

In these tests we evaluate the data shared: (i) on first startup following a factory reset, (ii) when a SIM is inserted/removed, (iii) when a handset lies idle, (iv) when the settings screen is viewed, (v) when location is enabled/disabled, (vi) when the user logs in to the pre-installed app store. We note that these tests can be partly automated and used for handset operating system privacy benchmarking that tracks changes in behaviour over time as new software versions are released.

Table I summarises the main data that the handsets send to Apple and Google. This data is sent even when a user is not logged in (indeed even if they have never logged in). In addition to the data listed in this table, iOS shares with Apple the handset Bluetooth UniqueChipID, the Secure Element ID (associated with the Secure Element used for Apple Pay and contactless payment) and the Wifi MAC addresses of nearby devices e.g. of other devices in a household of the home gateway. When the handset location setting is enabled these MAC addresses are also tagged with the GPS location. Note that it takes only one device to tag the home gateway MAC address with its GPS location and thereafter the location of all other devices reporting that MAC address to Apple is revealed. Also note that sharing of these Wifi MAC addresses allows linking of devices using the same network, e.g. in the same household, office, shop, cafe, and so the construction of a social graph over time and place.

Both iOS and Google Android transmit telemetry, despite the user explicitly opting out of this[1]. However, Google collects a notably larger volume of handset data than Apple. During the first 10 minutes of startup the Pixel handset sends around 1MB of data is sent to Google compared with the iPhone sending around 42KB of data to Apple. When the handsets are sitting idle the Pixel sends roughly 1MB of data to Google every 12 hours compared with the iPhone sending 52KB to

Email: doug.leith@tcd.ie.

[1] On iOS the Settings-Privacy-Analytics&Improvements option is set to off and on Google Android the Settings-Google-Usage&Diagnostics option is also set to off. We note that at the bottom of the Google text beside the "Usage&Diagnostics" option it says "Turning off this feature doesn't affect your device's ability to send the information needed for essential services such as system updates and security". Our data shows that the "essential" data collection is extensive, and likely at odds with reasonable user expectations.

	IMEI	Hardware serial number	SIM serial number	Phone number	Device IDs	Location	Tele-metry	Cookies	Local IP Address	Device Wifi MAC address	Nearby Wifi MAC addresses
Apple iOS	✓	✓	✓	✓	UDID, Ad ID	✓	✓	✓	✓	✗	✓
Google Android	✓	✓	✓	✓	Android ID, RDID/Ad ID, Droidguard key	✗	✓	✓	✗	✓	✗

TABLE I
SUMMARY OF HANDSET DATA SHARED WITH APPLE AND GOOGLE WHEN USER IS NOT LOGGED IN.

Apple i.e., Google collects around 20 times more handset data than Apple. In 2020 it is estimated that in the US there are 113M iPhone users[2] and 129M Android users[3]. Assuming all of the Android users have Google Play Services enabled then scaling up our measurements suggests that in the US alone Apple collects around 5.8GB of handset data every 12 hours while Google collects around 1.3TB of handset data. When the handset is idle the average time between iOS connections to Apple is 264 seconds, while Google Android connects to Google on average every 255 seconds i.e. both operating systems connect to their back-end servers on average every 4.5 minutes even when the handset is not being used.

With both iOS and Google Android inserting a SIM into the handset generates connections that share the SIM details with Apple/Google. Simply browsing the handset settings screen generates multiple network connections to Apple/Google.

A number of the pre-installed apps/services are also observed to make network connections, despite never having been opened or used. In particular, on iOS these include Siri, Safari and iCloud and on Google Android these include the Youtube app, Chrome, Google Docs, Safetyhub, Google Messaging, the Clock and the Google Searchbar.

The collection of so much data by Apple and Google raises at least two major concerns. Firstly, this device data can be fairly readily linked to other data sources, e.g. once a user logs in (as they must to use the pre-installed app store) then this device data gets linked to their personal details (name, email, credit card etc) and so potentially to other devices owned the user, shopping purchases, web browsing history and so on. This is not a hypothetical concern since both Apple and Google operate payment services, supply popular web browsers and benefit commercially from advertising. Secondly, every time a handset connects with a back-end server it necessarily reveals the handset IP address, which is a rough proxy for location. The high frequency of network connections made by both iOS and Google Android (on average every 4.5 minutes) therefore potentially allow tracking by Apple and Google of device location over time.

With regard to mitigations, of course users also have the option of choosing to use handsets running mobile OSs other than iOS and Google Android, e.g. /e/OS Android[4]. But if they choose to use an iPhone then they appear to have no options to prevent the data sharing that we observe, i.e. they are not able to opt out. If they choose to use a Pixel phone then it is possible to startup the handset with the network connection disabled (so preventing data sharing), then to disable the various Google components (especially Google Play Services, Google Play store and the Youtube app) before enabling a network connection. In our tests this prevented the vast majority of the data sharing with Google, although of course it means that apps must be installed via an alternative store and cannot depend upon Google Play Services (we note that many popular apps are observed to complain if Google Play Services is disabled). However, further testing across a wider range of handsets and configurations is needed to confirm the viabillity of this potential mitigation. When Google Play Services and/or the Google Play store are used then this mitigation is not feasible and the data sharing with Google that we observe then appears to be unavoidable.

A. *Ethical Disclosure*

The mobile OS's studied here are deployed and in active use. Measurements of Google Play Services backend traffic were previously disclosed in [5], but the present study is broader in scope. We informed Apple and Google of our findings and delayed publication to allow them to respond. To date Apple have responded only with silence (we sent three emails to Apple's Director of User Privacy, who declined even to acknowlege receipt of an email, and also posted an information request at the Apple Privacy Enquiries contact page at apple.com/privacy/contact but have had no response). Google responded with a number of comments and clarifications, which we have incorporated into this report. They also say that they intend to publish public documentation on the telemetry data that they collect. A key consideration is what mitigations are possible, and on what time scale can they be deployed. It seems likely that any changes to Apple iOS or Google Android, even if they were agreed upon, will take a considerable time to deploy and keeping handset users in the dark for a long open-ended period seems incorrect.

II. RELATED WORK

The privacy and security of mobile handsets has been the subject of a substantial literature, e.g. see [8], [9] and references therein. However, there has been little work reporting on the traffic between handset operating systems and their associated backend servers. Probably closest to the present work is the recent analysis of the data that web browsers share with their backend servers [4] and of the data shared

[2] https://www.statista.com/statistics/236550/percentage-of-us-population-that-own-a-iphone-smartphone/

[3] https://www.statista.com/statistics/232786/forecast-of-andrioid-users-in-the-us/

[4] https://e.foundation.

by Google Play Services [5]. The latter is motivated by Covid contact tracing apps based on the Google-Apple Exposure Notification (GAEN) system, which on Android require that Google Play Services be enabled. The present work is broader in scope, but also motivated in part by this since the data shared by Apple iPhones running Covid contact tracing apps remains largely unknown. The measurements that we report here indicate that on an iPhone running a covid contact tracing app the data collection by Apple iOS is remarkably similar to that by Google Play Services on Android phones and users appear to have no option to disable this data collection by iOS.

To the best of our knowledge there has been no previous systematic work reporting measurements of the content of messages sent between iOS and its associated backend servers.

III. 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 [11], [6].

Issues arise, however, when data can be tied to a specific user, especially over extended durations and old/new device pairs. There are at least two main ways that this can occur. Firstly, when a user logs in, as they must to use the pre-installed app store, then this device data gets linked to their personal details (name, email, credit card etc). Secondly, every time a handset connects with a back-end server it necessarily reveals the handset IP address which acts as a rough proxy for user location via existing geoIP services. Many studies have shown that location data linked over time can be used to de-anonymise, e.g. see [7], [10] and later studies. 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 [10]. Pertinent factors here are (i) the presence of identifiers within transmitted messages that allow them to be linked together and (ii) the frequency with which messages are sent e.g. observing an IP address/proxy location once a day has much less potential to be revealing than observing one every few minutes.

Once device data is associated to a specific user it can then potentially be linked to other data held by Apple and Google, or by third parties. This might include other devices owned the user, shopping purchases, web browsing history and so on, and such data linkage can quickly lead to privacy breaches. This is not a hypothetical concern since both Apple and Google operate payment services and supply popular web browsers.

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 each operating system directly send to its backend servers and (ii) Does the data that each operating system transmits to backend servers potentially allow tracking of the IP address of the app instance over time.

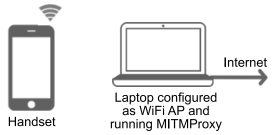


Fig. 1. Measurement setup. The mobile handset is configured to access the internet using a WiFi access point hosted on a laptop, use of cellular/mobile data is disabled. The laptop also has a wired internet connection. When an app on the handset starts a new network connection the laptop pretends to 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.

IV. Measurement Setup

A. Viewing Content Of Encrypted Network 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 [3] as a proxy and adjust the firewall settings to redirect all WiFi HTTP/HTTPS traffic to mitmdump so that the proxying is transparent to the handset. In brief, when a process running on the handset starts a new network 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 Figure 1.

The immediate difficulty encountered when using this setup is that handset system processes typically carry out checks on the authenticity of server certificates received when starting a new connection and abort the connection when these checks fail. To circumvent these checks we root/jailbreak each handset and configure it as follows:

1) Apple iOS: Cydia substrate is installed on a jailbroken iPhone and a custom substrate script is used to carry out bypass SSL certificate pinning within handset processes. On launch of a process this script is run and modifies the implementations of the SSL_se_custom_verify and SSL_get_psk_identity methods within the /usr/liblibboringssl.dylib library to bypass SSL certificate checks. This script is invoked on launch of all processes that make use of the com.apple.AuthKit, com.apple.UIKit and com.apple.aps.framework frameworks and also the following processes: com.apple.softwareupdated, com.apple.AssetCacheLocatorService, com.apple.imfoundation.IMRemoteURLConnectionAgent, com.apple.mobileactivationd, com.apple.itunescloudd, com.apple.identityservicesd, com.apple.akd, com.apple.itunesstored. The mitmproxy CA cert is also installed on the handset as a trusted certificate.

2) Google Android: On Android it is sufficient to install the mitmproxy CA cert as a trusted certificate in order to pass the SSL certificate checks carried out by Google Play Services and other system apps. However, unlike with iOS installing a trusted CA cert on Android requires rooting the phone. In Android 10 the system disk partition on which trusted certs are stored is read-only and security measures prevent it being mounted as read-write. Fortunately, folders within the system disk partition can be overriden by creating a new mount point corresponding to the folder, and in this way the mitmdump CA cert can be added to the /system/etc/security/cacerts folder.

B. Additional Material: Connection Data

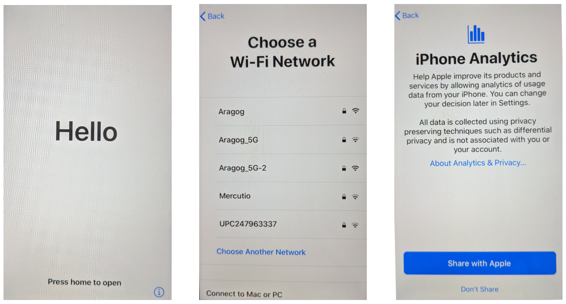
The content of connections is summarised and annotated in the additional material available anonymously at https://www.dropbox.com/s/qaazwyaj2ihj4qa/apple_google_additional_material.pdf.

C. Hardware and Software Used

Mobile handsets: Google Pixel 2 running Android 10 (build QP1A.190711.019 with Google Play Services ver. 20.45.16 and Google Play ver. 23.0.11-21) rooted using Magisk v20.4 and Magisk Manager v7.5.1 and running Frida Server v12.5.2, Apple iPhone 8[5] running iOS 13.6.1 (17G80) and jailbroken using Checkra1n 0.10.2 and running Cydia 0.9. 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`. Note that (i) at the firewall we blocked UDP traffic on port 443 so as to force any QUIC traffic to fall back to using TCP since we have no tools for decrypting QUIC, (ii) iOS uses port 5223 for Apple Push Notifications but we did not inspect this traffic, (iii) similarly Google Cloud Messaging uses port 5228 and (iv) both handsets use NTP and DNS. 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). On iOS this is used to install the Cydia substrate script for bypassing SSL pinning within the system processes and on Android a root adb shell is used to install the mitmproxy CA cert on the handset as a trusted cert.

D. Device Settings

1) Apple iOS: Following a factory reset it is not possible to proceed with startup without first connecting the handset to a network, see Figure 2(b) (there is no option to skip/continue).

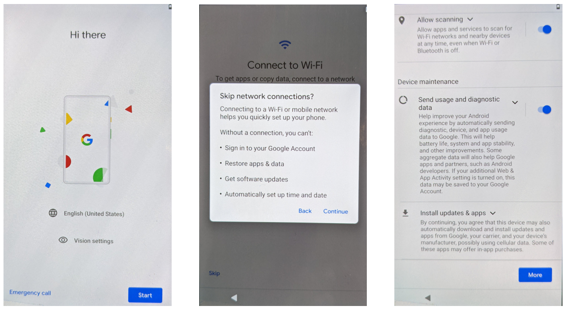


Help Apple improve its products and services by allowing analytics of usage data from your iPhone. You can change your decision later in Settings.

All data is collected using privacy preserving techniques such as differential privacy and is not associated with you or your account.

(a) (b) (c)

Fig. 2. Selected startup screens observed following iPhone factory reset.



To get apps or copy data, connect to a network

Skip network connections?

Connecting to a Wi-Fi or mobile network helps you quickly set up your phone.

Without a connection, you can't:

- Sign in to your Google Account
- Restore apps & data
- Get software updates
- Automatically set up time and date

Allow apps and services to scan for Wi-Fi networks and nearby devices at any time, even when Wi-Fi or Bluetooth is off.

Help improve your Android experience by automatically sending diagnostic, device, and app usage data to Google. This will help battery life, system and app stability, and other improvements. Some aggregate data will also help Google apps and partners, such as Android developers. If your additional Web & App Activity setting is turned on, this data may be saved to your Google Account.

By continuing, you agree that this device may also automatically download and install updates and apps from Google, your carrier, and your device's manufacturer, possibly using cellular data. Some of these apps may offer in-app purchases.

(a) (b) (c)

Fig. 3. Selected startup screens observed following Google Pixel 2 factory reset.

Connecting the handset to a WiFi network and proceeding, the user is presented with a number of option screens. Since our focus is on a privacy-conscious user we did not select any of the options that share data with Apple (we note that the opt-in options were always placed first and prominently highlighted while the opt-out option was de-emphasised, e.g. see Figure 2(c) for a typical example). Specifically, (i) on the “Apps & Data” screen we selected the “Don’t Transfer Apps & Data” option, (ii) on the “Keep Your iPhone Up to Date” screen we selected “Install Updates Manually”, (iii) on the “Location Services” screen we selected “Disable Location Services”, (iv) on the “Siri” and “Screen Time” screens we selected “Set Up Later in Settings”, (v) on the “iPhone Analytics” screen we selected “Don’t Share” (see Figure 2(c)). We did not log in to an Apple user account during the startup process, although the option was given to do this.

2) Google Android: Unlike with iOS, on Android following a factory reset it is possible to proceed with startup with and without a network connection. We collected data for both choices. Similarly to iOS, during startup the user is presented with a number of option screens and once again we did not select any of the options that share data with Google (we note that all of the option toggle switches default to the opt-in choice, and so it is necessary for the user to actively select to opt-out, e.g. see Figure 3(c) for a typical example). Specifically, we deselected the (i) “Free up space” option, (ii) “Use location” option, (iii) “Allow scanning” option and (iv) the “Send usage and diagnostic data” option, see Figure 3(c). Note that there is no option to deselect automatic updates, see the text shown at the bottom of the screen in Figure 3(c). We

[5] We were constrained to use a pre-A12 iPhone as in later models a soft bootloader is used which Apple have patched to prevent jailbreaking. Similarly, we were prevented from using iOS 14 on the handset since no iOS 14 jailbreaks were available when we carried out our measurements. Recently, it has become possible to jailbreak iOS 14 on rather old hardware (iPhone 6’s and earlier) but we leave collecting iOS14 to future work.

did not log in to an Google user account during the startup process

E. Test Design

We seek to define simple experiments that can be applied uniformly to the handsets studied (so allowing direct comparisons) and that generate repoducible behaviour. Both Apple and Google provide services that can be used in conjunction with their handsets, e.g. search (Siri, OkGoogle), cloud storage (iCloud, Google Drive), maps/location services (Apple Maps, Google Maps), photo storage/analytics (Apple Photo, Google Photos). Here we try to keep these two aspects separate and to focus on the handset as a device in itself, separate from optional services such as these. We also assume a privacy-conscious but busy/non-technical user, who when asked does not select options that share data with Apple and Google but otherwise leaves handset settings at their default values[6].

One important caveat is that while both Apple and Google provide an app store (Apple App Store, Google Play store), on iOS handsets the Apple App Store is the only way to install public apps whereas on Android handsets use of the Google Play store is optional, at least in principle, and other app stores can be used plus users have the option to directly install apps via the adb shell. Since confining users to pre-installed system apps is overly restrictive, the Apple App Store is not really an optional service on iPhones and we therefore include it in our tests. Note that use of the Apple App Store requires a user to log in to an Apple account, and so disclose their email address and other personal details. Because of this, for comparison we also consider the Google Play store (which also requires log in to a Google account and disclosure of a user's email address).

A second caveat is that on iPhones the pre-installed Apple Settings app must be used to configure device settings (e.g. to enable/disable location), and similarly on Android the Google Settings app must be used. Since these settings apps are not optional for handset users we also include them in our tests.

With these considerations in mind, for each handset we carry out the following experiments:

1) Start the handset following a factory reset (mimicking a user receiving a new phone), recording the network activity.

2) Remove and re-insert SIM, recording the network activity.

3) Following startup, leave the handset untouched for several days (with power cable connected) and record the network activity. This allows us to measure the connections made when the handset is sitting idle. This test is repeated with the user is logged in and logged out and with location enabled/disabled.

4) Open the pre-installed app store and log in to a user account, recording the network activity. Then log out and close the app store app.

5) Open the settings app (Apple/Google Settings) and view every option but leave the settings unchanged, recording the network activity. Then close the app.

6) Open the settings app and enable location, then disable. Record the network activity.

F. 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. Where possible we try to find more information on the nature of observed values from public documents, e.g. Apple and Google documentation, as well as by comparing them against known software and device identifiers e.g. the hardware serial number and IMEI of the handset. However, we note that there is little public documentation of the internal APIs between handset processes and their back-end servers and public privacy policy documents also tend to lack the necessary level of detail. We also contacted both Apple and Google for clarification of identifiers and data sent.

V. Startup Following Factory Reset of Handset

A. Apple iOS

Upon first startup, the iOS handset initially makes a series of connections to the sa.apple.com/grandslam endpoint. In the first connection the handset sends its Unique Device Identifier (UDID) via the X-Mme-Device-Id header value (this value persists across a factory reset). In the next connection the handset sends its hardware serial number together with the UDID, acting to link the two:

```
POST https://gsa.apple.com/grandslam/MidService/
startMachineProvisioning
Headers
  X-Apple-Client-App-Name: Setup
  X-Apple-I-SRL-NO: C8PVCB1HJC67       //Handset hardware
serial number
  X-MMe-Client-Info: <iPhone10,4> <iPhone OS;13.6.1;17G80>
<com.apple.akd/1.0 (com.apple.akd/1.0)>
  X-Mme-Device-Id: 7c2694081d97b...12dc5bc5   //UUID
  User-Agent: akd/1.0 CFNetwork/1128.0.1 Darwin/19.6.0
```

Later during the startup process the local IP address of the handset (i.e. not of the gateway, but of the handset itself) is sent in a POST request to /lcdn-locator.apple.com:

```
POST https://lcdn-locator.apple.com/lcdn/locate
Headers
  User-Agent: AssetCacheLocatorService/111 CFNetwork
/1128.0.1 Darwin/19.6.0
POST body
{"locator-tag":"#eefc633e","local-addresses":["
192.168.2.6"],"ranked-results":true,"locator-software":[{"
build":"17G80","type":"system","name":"iPhone OS","version
":"13.6.1"},{"id":"com.apple.AssetCacheLocatorService","
executable":"AssetCacheLocatorService",<...>
```

A connection to https://humb.apple.com/humbug/baa sends a base64 encoded payload that decodes to XML from com.apple.bluetoothd which contains the handset hardware serial number and a UniqueChipID value:

[6] There is also an important practical dimension to this assumption. Namely, each handset has a wide variety of settings that can be adjusted by a user and the settings on each handset are generally not directly comparable. Exploring all combinations of settings between a pair of handsets is therefore impractical. A further reason is that the subset of settings that a user is explicitly asked to select between (typically during first startup of the handset) reflects the design choices of the handset developer, presumably arrived at after careful consideration and weighing of alternatives. Note that use of non-standard option settings may also expose the handset to fingerprinting.

```
<...><key>AppID</key>
      <string>com.apple.bluetoothd</string>
<...><key>SerialNumber</key>
      <string>C8PVCB1HJC67</string> //Handset hardware
serial number
      <key>UniqueChipID</key>
      <integer>2577607982549050</integer><...>
```

The UniqueChipID value is not the handset ECID (Exclusive Chip Identification) value[7], the unique serial number of the handset system-on-a-chip (SoC) hardware which is also sometimes referred to as the "Unique Chip ID".

A connection to urlsmp-device-content.apple.com/static/region/v2/config.json shares the handset SEID (Secure Element ID) with Apple via header x-apple-seid:

```
GET https://smp-device-content.apple.com/static/region/v2/
config.json
Headers
  X-Apple-I-MD-RINFO: 17106176
  X-Apple-I-MD-M: 5ekJNohUcJUP1EFYFrRt...myO+0J7ljk6EljMp
  X-Apple-I-MD: AAAABQAAA...SjbTAAAAAw==
  User-Agent: Setup/1.0 CFNetwork/1128.0.1 Darwin/19.6.0
  x-apple-soc-type: t8015
  X-Apple-Web-Service-Session: 486081CF-EDD9-4B65-B5D3-0628
E298C7D7
  x-apple-seid:
046E4BABBA428001720717223094256 6E22AD196DF33D24C
```

The SEID identifies the secure element embedded in the NFC chip used by Apple Pay for contactless payment.

Despite selecting the 'Don't Share" option on the "iPhone Analytics" screen during the startup process, telemetry data is sent to xp.apple.com/report/2/psr_ota.

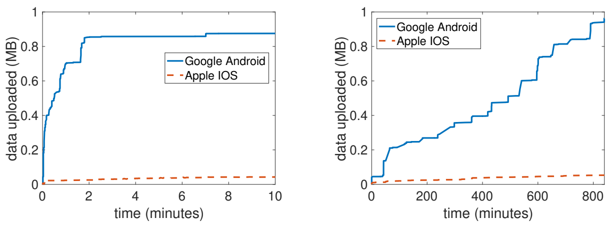
During the startup process 2.6GB of data is downloaded to the handset, a substantial quantity that appeared to consist mainly of software updates to the pre-installed apps. We observed roughly 42KB of data sent to Apple servers (via URL parameters, headers and POST data).

B. Google Android

Upon startup the first connection that the Android handset makes that sends data is to Google Analytics endpoint app-measurement.com:

```
GET https://app-measurement.com/config/app/1%3A286455739530
%3Aandroid%3A4a942425ed36c2fa?app_instance_id=
93a09622...1ee8f0e52&platform=android&gmp_version=17786
```

The app instance id that is sent is linked to the device RDID (Resettable Device Identifier or so-called Ad ID, used for measurement and ads[8]) in a later call to app-measurement.com. The next connection is made by the DroidGuard process (used for device attestation, part of Google's SafetyNet service), which send the device hardware serial number (which persists across a factory reset) to www.googleapis.com/androidantiabuse. The device RDID is now sent to www.gstatic.com and app-measurement.com. Early in the startup process a call is made to youtubei.googleapis.com/deviceregistration that sends a rawDeviceId value and in due course a call to the android.clients.google.com/checkin endpoint is made:



(a) Startup

(b) Idle

Fig. 4. Volume of data uploaded to Google and Apple servers (a) during first 10 minutes of startup after factory reset and (b) when handset lies idle.

```
POST https://android.clients.google.com/checkin
POST body decoded as protobuf:
<...>
7: 85879...846810 // Google loggingId
9: "e60d...158" // Wifi Mac Address
10: "357...984248" // IMEI
11: ""
12: "Europe/Dublin"
14: 3
15: "bfMkwy...c2WT62otR8JkI=" //SHA-1 of OTA certs
16: "HT7A...4090" //Handset hardware serial number
<...>
24: "CgZmMZ-F5fTSEEAA...MFUwYaZqw" //Droidguard device key
<...>
```

This shares the handset Wifi MAC address, its hardware serial number and IMEI, effectively linking these three persistent device identifiers together. A subsequent call to android.clients.google.com/checkin further links these values to the Google AndroidId (a persistent device identifier that requires a factory reset to change) and a variety of security tokens[9]. The Droidguard device key is a large, opaque binary message: the contents are intentionally obfuscated by Google and it remains unclear whether it contains device/user identifiers[10].

Cookies are sent in a number of calls, starting with one to fonts.gstatic.com and then to play.googleapis.com (this call includes the AndroidId and so links that with the cookie).

Despite deselecting the "Send usage and diagnostic data" option during the startup process, a substantial quantity (approximately 1.2MB) of telemetry/logging data is sent by the handset to play.googleapis.com/log/batch and play.googleapis.com/play/log. The handset also sends 1.1MB of device data to www.googleapis.com/experimentsandconfigs and 181KB to android.clients.google.com/checkin. In total, around 3.6MB of data is sent to Google servers (via URL parameters, headers and POST data), see Figure 4(a), and 952MB of data are received. That is, almost two orders of magnitude more data is uploaded by Android (3.6MB) than by iOS (42KB) during startup, while Android downloads around a third (952MB) of the data of iOS (2.6GB).

[7] By connecting the device to a macbook laptop then itunes reports the ECID as 9285220B9893A, which when converted to decimal is 2577607991855418. It can be seen that this is similar to the UniqueChipID, but not the same.

[8] See https://developers.google.com/ads-data-hub/guides/rdid-matching.

[9] There is a Google help page [1] for the /checkin endpoint with partial information on the data sent and the microG project [2] has also partially reverse-engineered the data format used by this endpoint (there is no documentation, but microG is open source). Our measurements are consistent with both of those.

[10] https://github.com/microg/GmsCore/issues/1139

VI. Connections Made When Handset is Idle

A. Apple iOS

When the iPhone is left idle, roughly every 2-3 days it sends data to gsas.apple.com/grandslam:

```
POST https://gsas.apple.com/grandslam/GsService2/postdata
Headers
  User-Agent: akd/1.0 CFNetwork/1128.0.1 Darwin/19.6.0
  X-Apple-I-UrlSwitch-Info: MDAxMDI5LTA...GF0YQ== //
x-apple-adsid base64 encoded
  X-Apple-HB-Token: MDAxMDI5LTA1LTk...5XN3dnPT0= //
x-apple-adsid base64 encoded
  X-Mme-Device-Id: 7c2694081d...71412dc5bc5 //UDID
  X-Apple-I-MD-RINFO: 17106176
  X-Apple-I-SRL-NO: C8PVCB1HJC67 //Handset hardware serial
number
  X-Apple-I-MD-M: 5ekJNohU...YjL2Z //Anisette machineID[11]
  X-Apple-I-MD: AAAABQAA...AAAAw==
POST body
<...>
            <key>iccid</key>
            <string>8935311180135555145</string> // SIM
Integrated Circuit Card Identifier
            <key>imei</key>
            <string>356765081821496</string> //Handset IMEI
<...>    <key>number</key>
                        <string>+35389...97590</string> //
Handset phone number
<...>  <key>pn</key>
            <string>+35389...97590</string>
<...>  <key>ptkn</key>
            <string>7534B939D....6E2750CB0FD0</string>
<...>  <key>sn</key>
            <string>C8PVCB1HJC67</string> //Handset hardware
serial number
<...>
```

It can be seen that this message sends (and links together) many handset identifiers including: the handset hardware serial number, the handset UDID, the IMEI, the SIM serial number, the handset phone number, the Apple advertising ID plus the X-Apple-I-MD-M security token/anisette machine identifier.

In addition, the handset makes a number of unexpected connections:

1) Although the user is not logged in to an Apple account (and so the Apple App Store cannot be used) periodic connections are made to init.itunes.apple.com and bag.itunes.apple.com that send a cookie that can act as a device identifier.

2) Similarly, since the user is logged out icloud is unused yet connections are made to icloud services that send device identifiers, including the handset UDID.

3) Although Siri is not enabled on the handset, connections are made to server smoot.apple.com by the parsecd process associated with Siri. When a URL is typed in Safari, corresponding telemetry logging the URL is sent to smoot.apple.com. Again, this occurs despite the fact that Apple telemetry is disabled in the device settings.

4) Although use of location is disabled, the locationd and geod processes associated with location services in the handset periodically make network connections. The locationd process downloads files that likely relate to GPS chipset settings, with no unique device identifiers sent. However, the geod process uploads binary messages to gsp85-ssl.ls.apple.com:

```
POST https://gsp57-ssl-locus.ls.apple.com/dispatcher.arpc
Headers
  User-Agent: geod/1 CFNetwork/1128.0.1 Darwin/19.6.0
POST body
\x00\x01\x00\x08en-IE_IE\x00\x0ecom.apple.geod\x00\x...f
f8:4:2e:c:1c:28\x10\xa...&\n\x0f8c:4:ff:13:2:9e\x10\xb...\
x1170:4d:7b:95:14:c0\x10\...n\x1170:4d:7b:95:14:c8\x10\xc0
...\x10f2:18:98:92:17:5\x10...
```

While it is not clear what information is contained in this binary message, it can be seen to contain the MAC addresses of nearby devices sharing the same WiFi network as the handset e.g. f2:18:98:92:17:5 is the WiFi MAC address of a nearby laptop, 70:4d:7b:95:14:c0 the MAC address of the WiFi access point.

5) Despite selecting the ‘Don’t Share” option on the “iPhone Analytics” screen during the startup process, telemetry is sent to xp.apple.com.

In addition, connections made by the adprivacyd process, which appears related to managing advertising settings, also send a cookie and transmit an opaque binary message.

The Safari browser makes periodic connections to the Google Safe Browsing anti-phishing service and connections are made to mesu.apple.com that appear to be checking for updates. However, no unique device identifiers appear to be sent in these connections [4].

When the handset is idle the average time between iOS connections to Apple servers is observed to be 264 seconds i.e. less than 5 minutes.

1) Inserting SIM Into Handset: When a SIM is inserted the handset sends SIM identifiers to albert.apple.com:

```
POST https://albert.apple.com/deviceservices/activity/
phoneNumberSimNotification
Headers
  User-Agent: CommCenter/7581 CFNetwork/1128.0.1 Darwin
/19.6.0
POST body
<...> <key>InternationalMobileEquipmentIdentity</key>
      <string>356765081821496</string> // IMEI
      <key>InternationalMobileSubscriberIdentity</key>
      <string>2721101</string>
      <key>PhoneNumber</key>
      <string>089...97590</string> // Phone number
      <key>SerialNumber</key>
      <string>C8PVCB1HJC67</string> // Handset hardware
serial number
      <key>UniqueDeviceID</key>
      <string>7c2694081d97b76...2dc5bc5</string> //Handset
UDID
<...>
```

2) Enabling Location: When location is enabled in the handset settings, additional handset network connections are made. In particular:

```
POST https://gsp10-ssl.apple.com/hcy/pbcwloc
Headers
  User-Agent: locationd/2394.0.33 CFNetwork/1128.0.1 Darwin
/19.6.0
POST body decoded as protobuf:
<...>
 1: "f2:18:98:92:17:5" // Wifi MAC address of nearby device
 2: 11
 3: 18446744073709551584
 4 {
  1: 0x404aa..c4edd17953 // hex encoded handset latitude
  2: 0xc0193..5ed5618f3 // hex encoded handset longitude
 <...>
```

[11] https://www.duffy.app/blog/icloud-auth-headers

The POST body not only contains a list of MAC addresses of devices sharing the same WiFi network as the handset but a pair of hex values that when converted to doubles give the latitude and longitude of the the handset (accurate to within 10m of the true position).

In addition, connections to api-glb-dub.smoot.apple.com send a X-Apple-FuzzedLatLong header with the approximate device location (latitude and longitude) .

B. Google Android

When the Google Pixel 2 is left idle, roughly every 6 hours it makes a connection to android.googleapis.com/checkin that sends many device identifiers:

```
POST https://android.googleapis.com/checkin
Headers
  Cookie: NID=204=sa8sIUm5eJ9...NabihZ3RNI
POST body decoded as protobuf:
2: 3876027569814251330 //AndroidId
<...>
  6: "27205" //Mobile operator
  7: "27211" //SIM operator
  8: "WIFI::"
 <...>
  16 {
     1: "27211" //SIM operator
    2: "Tesco Mobile" //Mobile carrier
<...>
    6: "272110103800000" //SIM IMSI, uniquely identifies
caller on cellular network
    7: "0AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF" //Mobile
Group ID Level 1
    8: "\025\345" //SHA-256 hash of SIM IMSI
<...>
9: "e60d4b46d158" //Wifi MAC address
10: "357537080984248" //IMEI
11: "" //When user is logged in this reports the user email
address
12: "Europe/Dublin"
13: 0x41559e6d59911873  //Security token
14: 3
15: "bfMkwynjHzXGBPc2WT62otR8JkI="
16: "HT7AC1A04090" //Handset hardware serial number
<...>
24: "CgYqtj3OocES-2UKBoGpQIpsRtIQQA...Sy6P2voE9Sz"
//Droidguard device
key
<...>
```

In many ways this is akin to the iOS connection to gsas.apple.com/grandslam/GsService2/postdata discussed above. The set of identifiers shared is slightly different, but the connection acts to link to together multiple long-lived device identifiers including the handset hardware serial number and IMEI, the SIM IMSI/serial number. In addition to these the android.googleapis.com/checkin connection shares the AndroidId (a persistent device identifier that requires a factory reset to change), the user email address (when the user is logged in). the handset Wifi MAC address, the opaque Droidguard binary device key and also tags these with a cookie.

The handset also connects regularly to Google's SafetyNet device attestation service at www.googleapis.com and to what seems to be Google's A/B testing infrastructure at www.googleapis.com/experimentsandconfigs. The SafetyNet connections send the hardware serial number, RDID device identifier (associated with advertising/measurement) and the Droidguard device key while the A/B testing connections send a cookie and authentication token together with device details.

CARRIER_SERVICES,ANDROID_DIALER,ONEGOOGLE_MOBILE,GOOGLE_NOW_LAUNCHER,DRIVE,COPRESENCE_NO_IDS,AUTOFILL_WITH_GOOGLE,SCOOBY_EVENTS,SCOOBY_EVENT_LOG,BEACON_GCORE,NETREC,BRELLA,GOOGLE_HELP,PHOTOS,CALENDAR,CALENDAR_UNIFIED_SYNC,BUSINESS_VOICE,IDENTITY_FRONTEND,GMS_CORE_PEOPLE,LATIN_IME,DL_FONTS,CAR,ICING,ACTIVITY_RECOGNITION,ANDROID_CONTACTS,ANDROID_GROWTH,ANDROID_GSA,CLIENT_LOGGING_PROD,GOOGLETTS,CAST_SENDER_SDK,ANDROID_VERIFY_APPS,ANDROID_DIALER,ANDROID_BACKUP,ANDROID_MESSAGING,ANDROID_OTA,ANDROID_GMAIL,ANDROID_SNET_GCORE,GAL_PROVIDER,GLAS,TACHYON_LOG_REQUEST,CLEARCUT_FUNNEL,CLEARCUT_LOG_LOSS,DIALER_ANDROID_PRIMES,CARRIER_SERVICES_ANDROID_PRIMES,TURBO_ANDROID_PRIMES,PHOTOS_ANDROID_PRIMES,ANDROID_MESSAGING_PRIMES,GOOGLETTS_ANDROID_PRIMES,SETTINGS_INTELLIGENCE_ANDROID_PRIMES,ANDROID_GSA_ANDROID_PRIMES,SAFETYHUB_ANDROID_PRIMES,WIFI_ASSISTANT_PRIMES,DRIVE_ANDROID_PRIMES,GMAIL_ANDROID_PRIMES,STREAMZ_ANDROID_GROWTH,STREAMZ_ANDROID_GSA,STREAMZ_ONEGOOGLE_ANDROID,STREAMZ_HERREVAD,STREAMZ_CALENDAR,STREAMZ_PHOTOS_ANDROID,STREAMZ_ANDROID_AUTH_ACCOUNT,STREAMZ_GELLER,STREAMZ_NGA,BUGLE_COUNTERS,PSEUDONYMOUS_ID_COUNTERS,GMAIL_COUNTERS,WESTWORLD_COUNTERS,GOOGLE_KEYBOARD_COUNTERS,ANDROID_CONTACTS_COUNTERS,WALLPAPER_PICKER_COUNTERS,PLATFORM_STATS_COUNTERS

TABLE II
LOGGING SOURCES OBSERVED IN PLAY.GOOGLEAPIS.COM/LOG/BATCH TELEMETRY.

Despite deselecting the "Send usage and diagnostic data" option during the startup process the handset sends a substantial volume of telemetry/logging data to Google servers, see Figure 4(b). This occurs mainly to two endpoints, namely play.googleapis.com/log/batch and play.googleapis.com/play/log. The first is associated with Google Play Services and the second with the Google Play store app. Data is sent every 10-20 minutes, and sometimes more frequently. Logging/telemetry data sent to play.googleapis.com/log/batch is tagged with device AndroidId and an authentication token but the content is largely opaque (binary protobufs with complex structure). The messages sent to play.googleapis.com/log/batch appear to be aggregated from multiple logging sources, see Table II for a list of their names. The CARRIER_SERVICES and ANDROID_DIALER logging sources are observed to send details of mobile operator and phone number, amongst other things[12].

The messages sent to play.googleapis.com/play/log are tagged with the AndroidId and RDID persistent device identifiers.

Several pre-installed system apps make regular network connections that share device identifiers and details:

1) The Nexus launcher searchbar connects to www.google.com/complete/search using a cookie and events related to the app are logged by Google Analytics app-measurement.com.
2) The Clock app connects to Google Analytics ssl.google-analytics.com/batch.
3) The SafetyHub app periodically connects with Google's Firebase service at android.clients.google.com, sending the device AndroidId and the app FirebaseId
4) The Youtube app makes connections to youtubei.googleapis.com/youtubei/v1/account and

[12] Google have since told us that on foot of our recent GAEN measurement study [5] version V52I and later of the CARRIER_SERVICES log source no longer transmit the handset phone number.

youtubei.googleapis.com/youtubei/v1/log_event, both send a device identifier and an authentication token. The Youtube app (or a process on its behalf) also makes connections to www.googleadservices.com, sending the RDID device identifier used for advertsing. In addition, the Youtube app makes periodic probe connections to i.ytimg.com/generate_204 and youtubei.googleapis.com/generate_204.

5) The Chrome browser app makes periodic connections to the Google Safe Browsing anti-phishing service safebrowsing.googleapis.com and the Chrome update service pdate.googleapis.com. It also connects to accounts.google.com. No identifiers are sent.
6) The Google Docs and Messaging apps (or a process on its behalf) connect to growth-pa.googleapis.com/google.internal.identity.growth.v1.GrowthApiService/GetPromos, sending device details but no unique identifiers.
7) Connections are made to mobilenetworkscoring-pa.googleapis.com/v1/GetWifiQuality that may include a device identifier (via the X-Client-Data header). The purpose of this connection is unclear.

When the handset is idle the average time between Android connections to Google servers is observed to be 255 seconds i.e. similar to the 264 seconds observed for iOS.

1) Inserting SIM Into Handset: When a SIM is inserted into the handset a connection is made to android.clients.google.com/fdfe/uploadDynamicConfig that sends the SIM IMSI (which uniquely identifies a caller on the cellular network) and links this to the AndoidId:

```
POST https://android.clients.google.com/fdfe/
uploadDynamicConfig
Headers
  user-agent: Android-Finsky/22.8.42-21<...>
  x-dfe-device-id: 35ca6a89e6662742 // hex-encoded
AndroidId
  x-dfe-device-config-token: CisaKQoT...4MTQyMzM1
  x-dfe-device-checkin-consistency-token:
ABFEt1VpT4...Cnef2U7bDsS2p
  x-dfe-phenotype: H4sIAAAAA...9cD-bQEAAA
  x-dfe-encoded-targets: CAESGLOVg...oE5ALQ+64G
POST body decoded as protobuf:
<...>
    1: 272110103800000 // SIM IMSI
    2: "Tesco Mobile" // Mobile carrier
    3: "0AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF" // Mobile
Group ID Level 1
    4: "tescomobile.liffeytelecom.com" // Mobile carrier
APN
<...>
 4: "dHVKO5LRPfOs71S8ynLFp9:..qXOTpK9e" '// Firebase id of
Google Play store app
<...>
```

In addition, connections are made to android.googleapis.com/checkin and /play.googleapis.com/log/batch. The connection to android.googleapis.com/checkin shares the handset IMEI and SIM IMSI plus mobile carrier details with Google and links these to the AndroidId, handset hardware serial number, Wifi MAC address and a cookie. The connection to /play.googleapis.com/log/batch

```
POST https://play.googleapis.com/log/batch
Headers
  x-server-token: CAESKQDyi0h8ELN...KGowA
  user-agent: com.google.android.gms/204516037 <...>
  cookie: NID=204=VBQLUjKJOc3FW...X-lC0Ro // Cookie
POST body decoded as protobuf:
<...>
    1: 3876027569814251330 // AndroidId
 <...>
  5: "\324\020\010...(48.0.335972766-carrierservices\_V48E\
_RC01\032\'\010\340\203\273 \022 6.8.076 (Ent\_RC11.phone\
_dynamic)2C\n\003272\022\00211\032\014Tesco
Mobile\"(0AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF(\000\3...\
n\r+35389...7590r/\010..."\230\001\001"
  6: "CARRIER\_SERVICES"
  7: "204=VBQLUjKJOc3FWJoecvEe...X-lC0Ro" // Cookie
<other telemetry>
```

The +35389...7590 value is the handset phone number, the mobile carrier details are also sent.

2) Enabling Location: When the user is logged out and location is enabled no additional handset network connections are observed.

VII. Connections When Interacting With Settings App

A. Apple iOS

When the Settings app is opened and the various options viewed (but not changed), this action generates multiple network connections:

1) A connection is made to idiagnostics.apple.com that sends the handset hardware serial number
2) Telemetry is sent to xp.apple.com
3) The adprivacyd process makes connections to cf.iadsdk.apple.com/adserver/2.6/config, iadsdk.apple.com/adserver/2.6/optout/optout_optin and bag.itunes.apple.com. The connection to bag.itunes.apple.com sends a cookie.
4) The geod process makes connections to gspe35-ssl.ls.apple.com/geo_manifest/dynamic/config, gsp-ssl.ls.apple.com/ab.arpc and gsp64-ssl.ls.apple.com/hvr/v3/use. The latter two connections transmit binary messages which are largely opaque but which can be seen to contain SIM mobile carrier details, amongst other things.
5) The gamed process makes connections to init.gc.apple.com, static.gc.apple.com, profile.gc.apple.com. The latter two connections send authentication tokens that are linked to the device.
6) The Preferences agent makes connections to init.itunes.apple.com and play.itunes.apple.com which send authentication tokens that are linked to the device.

B. Google Android

When the Settings app is opened and a user navigates amongst the various options the following network connections are observed:

1) The helprtc process makes connections to firebaseinstallations.googleapis.com and android.clients.google.com. These send the Firebase Id and the device Android Id.

2) A connection is made to pagead2.googlesyndication.com that appears to send identifiers
3) Telemetry is sent to www.google.com. This is tagged with the device Android Id, the phone IMEI and includes mobile carrier details as well as information on the radio signal strength, battery level, volume settings, number of handset reboots, whether the phone is rooted.

VIII. Summary

We investigate what data iOS on an iPhone shares with Apple and what data Google Android on a Pixel phone shares with Google. We find that even when minimally configured and the handset is idle both iOS and Google Android share data with Apple/Google on average every 4.5 mins. The phone IMEI, hardware serial number, SIM serial number and IMSI, handset phone number etc are shared with Apple and Google. Both iOS and Google Android transmit telemetry, despite the user explicitly opting out of this. When a SIM is inserted both iOS and Google Android send details to Apple/Google. iOS sends the MAC addresses of nearby devices, e.g. other handsets and the home gateway, to Apple together with their GPS location. Currently there are few, if any, realistic options for preventing this data sharing.

References

[1] Learn about the Android Device Configuration Service, Google Help Pages (Accessed 5 August 2020), https://support.google.com/android/answer/9021432?hl=en

[2] microG Project (Accessed 5 August 2020), https://microg.org/

[3] Cortesi, A., Hils, M., Kriechbaumer, T., contributors: mitmproxy: A free and open source interactive HTTPS proxy (v5.01) (2020), https://mitmproxy.org/

[4] Leith, D.J.: Web Browser Privacy: What Do Browsers Say When They Phone Home? IEEE Access (2021). https://doi.org/https://doi.org/10.1109/ACCESS.2021.3065243

[5] Leith, D.J., Farrell, S.: Contact Tracing App Privacy: What Data Is Shared By Europe's GAEN Contact Tracing Apps. In: Proc IEEE INFOCOM (2021)

[6] Machanavajjhala, A., Kifer, D., Gehrke, J., Venkitasubramaniam, M.: l-diversity: Privacy beyond k-anonymity. ACM Transactions on Knowledge Discovery from Data (TKDD) **1**(1), 3–es (2007)

[7] P., G., K, P.: On the Anonymity of Home/Work Location Pairs. In: Pervasive Computing (2009)

[8] Razaghpanah, A., Nithyanand, R., Vallina-Rodriguez, N., Sundaresan, S.: Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In: Proc NDSS (2018). https://doi.org/https://doi.org//10.14722/ndss.2018.23353

[9] Reardon, J., Feal, Á., Wijesekera, P., On, A.E.B., Vallina-Rodriguez, N., Egelman, S.: 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 603–620. USENIX Association, Santa Clara, CA (Aug 2019), https://www.usenix.org/conference/usenixsecurity19/presentation/reardon

[10] 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)

[11] Sweeney, L.: k-anonymity: A model for protecting privacy. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems **10**(05), 557–570 (2002)