Persistent vs Service IDs in Android: Session Fingerprinting from Apps

Efthimios Alepis(✉) and Constantinos Patsakis

Department of Informatics, University of Piraeus,
80, Karaoli & Dimitriou, 18534 Piraeus, Greece
talepis@unipi.gr

Abstract. Android has conquered the mobile market, reaching a market share above 85%. The post Lollipop versions have introduced radical changes in the platform, significantly improving the provided security and privacy of the users. Nonetheless, the platform offers several features that can be exploited to fingerprint users. Of specific interest are the fingerprinting capabilities which do not request any dangerous permission from the user, therefore they can be silently shipped with any application without the user being able to trace them, let alone blocking them. Having Android AOSP as our baseline we discuss various such methods and their applicability.

1 Introduction

Mobile devices, especially smartphones have become an indispensable part of our daily lives, as a big part of our communications and daily activities is processed and monitored by them. One of the main reasons for their wide adoption is that they have a plethora of embedded sensors that allow them to understand their context and adapt accordingly. For instance, through luminosity and proximity sensors as well as accelerometers, mobile phones may adapt the UI to fit better to user expectations. Moreover, thanks to GPS, mobile devices are location aware enabling them to render content according to the spacial restrictions significantly improving the user recommendations.

Data mining and data profiling can be used in order to collect valuable information about a particular user or group of users, in order to generate a profile [12], which can be further used by companies to gain profit. As stated in [14], this kind of information, namely user profiling, is valuable also for advertisers who want to target ads to their users and in return, advertisers may pay more money to their hosting applications' developers. Building user profiles requires, as the authors state, sensitive privileges in terms of permissions, such as Internet access, location, or even retrieving installed applications in a user's device [14]. To this end, we may infer that collecting and successfully fusing user data from more than one service can create even better and more complete user profiles, which will consequently translate in higher monetization. Looking back in 2009, it was quite clear that:

> "Once an individual has been assigned a unique index number, it is possible to accurately retrieve data across numerous databases and build a picture of that individual's life that was not authorised in the original valid consent for data collection" [19].

The above has been realised by tech giants. For instance quoting a statement from Google's current privacy policy [10]:

> "We may combine personal information from one service with information, including personal information, from other Google services - for example to make it easier to share things with people you know. Depending on your account settings, your activity on other sites and apps may be associated with your personal information in order to improve Google's services and the ads delivered by Google"

In order to "enable" data fusion from different sources and services, one could argue that unique identifiers should be either implicitly or explicitly present. In particular, a value describing a quantity of some valuable variable might be useless if it is not accompanied by a unique identifier that would allow us to track its source. Contrariwise, identifiers coming from different services that are matched, may act as a "bridge" between these services to combine their corresponding datasets and integrate them.

During the last decade relevant surveys have revealed that the majority of both iOS and Android apps were transmitting the phone's unique ID and the user's location to advertisers. These findings are confirmed by "The Haystack Project" [11] which revealed that nearly 70% of all Android apps leak personal data to third-party services such as analytics services and ad networks [20].

All this wealth of information apart from the benign usage for the user benefit has been a constant target by companies who wish to monetize it, mainly through targeted advertisement. The recent advances in big data and data mining have enabled the extraction of information from theoretically diverse data, leading to the revealing of a lot of sensitive data through data fusion. To this end, many fingerprinting techniques have been introduced in order to link data flows to specific individuals. Apparently, since Android in currently the prevailing mobile platform, most companies are targeting it with their apps, under the freemium model, harvesting user data to monetize them. There is even a common saying in the privacy community suggesting that "If you're not paying for the product, you are the product". To this end, If users are not paying for an app, they are usually selling their profiles (with or without their knowledge/consent) to an ad network, which will use their unique identifiers to track and target them.

In view of the above and targeting at improving the OSes privacy, the new coming Android O, makes a number of privacy-related changes to the platform, many of which are related to how unique identifiers are handled by the system [9], and in particular aiming to help provide user control over the use of identifiers [2]. One of the most important improvements concern "limiting the use of device-scoped identifiers that are not resettable".

Clearly, during app environment of Android hardware identifiers can greatly facilitate companies' attempt to deanonymise users. Therefore, Google has been gradually introducing specific measures to restrict them. In fact, Google decided to introduce further restrictions in one more identifier; not hardware based, namely Android_ID. While this attempt might seem noble, in this work we show that these restrictions do not actually serve the purpose, while apps may be deprived of many persistent identifiers, ephemeral IDs can actually serve their purposes in the attempt to deanonymize their users and fuse their data.

1.1 Main Contributions

The main contribution of this work is to study user fingerprinting from mobile apps, which correspondingly and as already discussed can lead to user profiling. In this regard we assume that apps and software services which profile users, want to correlate the information that each one of them has collected to fine tune their profiles. Conceptually, in order to provide proof for our claims, we explore all the available communication channels that mobile apps could utilize in Android AOSP in order to identify that specific profiles are installed in the same device. Furthermore, we suggest that the existing underlying mechanism is able to function without using unique hardware identifiers, nor dangerous permissions which could alert the user, or demand further user interaction. While many of the communication mechanisms are apparent, e.g. inter-process communication, there is a wide misconception that the upcoming changes in Android O will eradicate many such issues. Therefore, initially we analyse each possible communication channel and ways it can be used to transfer the needed information. Moreover, by providing statistical evidence we discuss when these changes are expected to be noticed by the average user. Finally, despite the touted changes in Android ID, we detail new methods that can provide permanent cross-app IDs that can be collected even if an app is uninstalled.

1.2 Organisation of This Work

The rest of this work is organized as follows. In the next section we present the related work. Section 3 provides the problem setting and our basic assumptions. Then, Sect. 4 presents all the available communication channels. In Sect. 5 illustrates possible temporary and ephemeral identifiers that can be used to link users between applications using Android AOSP as our reference point. Finally, the article concludes with some discussion about our research findings and providing statistics regarding the adoption timeline of the expected anonymization mechanisms of Android O.

2 Related Work

Unique identifiers have been used for a long time and facilitate many tasks in modern database systems as they allow us to perform record linkage between different entities and extract the necessary information and thus knowledge from the corresponding database tables. The most typical example of a unique identifier is the Social Security Number, which allows us to distinguish two people from each other. However, in the digital era, unique identifiers can be considered hardware identifiers like the MAC address of the network card, or a set of properties such as browser fingerprints which consist among others of the browser version, OS, fonts, and browser plugins.

In the Android ecosystem there is a plethora of unique identifiers which have so far been extensively exploited by advertisement companies to track users and their interests as ad libraries have become more and more greedy and rogue [3,18] while apps may deliberately leak information to the ads [4,21] harnessing arbitrary amounts of users' sensitive information directly or indirectly [5]. A key role in this procedure is the use of unique identifiers [17]. Acknowledging this situation, Google initially introduced some recommendation guidelines for the proper use of unique identifiers in Android [1]. Then, Google gradually started requesting more permissions from the apps to allow them access to these identifiers. For instance, a typical unique identifier for mobile phones are IMEI and IMSI, however, after Marshmallow, the user has to grant the dangerous `READ_PHONE_STATE` permission to an app to access them. While many users may ignore app permissions [8], for many others it works as an obstacle, forcing many companies to comply with the rule.

Despite the ads, apps may collaborate in order to perform malicious acts which independently would not be allowed to perform. Orthacker et al. [15] study this problem from the aspect of permissions. In this regard, the malicious apps which are installed in the victim's device may result in "possessing" and correspondingly using dangerous permissions that other normal apps do not. The concept is that the user would not allow camera and microphone permission to a single app. However, since the permissions are requested by two apps which are seemingly independent, the permissions are "spread" so the user grants them, yet an adversary controls both of them getting access to the desired resource. Contrary to Orthacker et al. we do not aim to resources, but access to information that the user would not share to one specific app to prevent his profiling.

In Nougat, the current stable version of Android, Google prohibited unprivileged access to even more hardware identifiers, such as the MAC address of the WiFi card, by restricting access to `/proc`. While the latter measure creates many issues with applications targeting towards security and privacy services as Google has not provided any permission so far to access this information, undoubtedly, it leaves little space to adversaries to exploit.

3 Temporary and Ephemeral Identifiers

3.1 Problem Setting

While the aforementioned issues have led to the introduction of many changes to Android, improving the security and privacy of the OS, in terms of user fingerprinting, from the side of apps, we argue that little has been achieved.

Certainly, direct access and/or unprivileged access to hardware identifiers has been removed, therefore, permanent or long term identifiers are not going to be available in the coming version of Android, nonetheless, this is not what the advertisement companies are actually trying to do. Undoubtedly, such access facilitates the correlation process, nonetheless, it is a great misconception to consider that a unique device identifier from a device is all that two apps need to correlate user information. More specifically, owning a unique identifier, however not being able to communicate it to others cannot be considered a threat. Similarly, having access to a communication channel, yet failing to uniquely identify the transmitted data results in data loss. Randomly generated identifiers, locally stored in apps, as it will be shown, offer a solution, however also suffer from lack of persistency. In this work we present methods which bypass these obstacles and result in identifying users and also allow communication of this information to other parties.

3.2 Basic Assumptions and Desiderata

In what follows we assume that the user has installed at least two applications in his device. In the same sense, this approach can be generalized in software services that communicate and/or handle a number of mobile apps. Our reference is Android AOSP as it provides all the baseline security and privacy methods therefore all derivative versions may have glitches which are vendor specific, perhaps apart of CopperheadOS[1] which is a hardened version of Android. Moreover, in order to highlight the magnitude of the presented privacy issue, we further assume that the two aforementioned applications did not request any permission from the user during installation, nor during runtime. We also assume that these apps do not belong to the same developer, nonetheless, the developers have decided to cooperate in exchanging user data to create a more fine-grained profile of their users. Finally, we assume that even in the cases where the user has authenticated himself to each app, for each of them he uses completely different credentials e.g. the username is different in both apps. One can easily deduce that by "relaxing" these assumptions, our work becomes much easier.

Apparently, one way to achieve their goal, the app developers only need to determine that the two apps are running in the same device and exchange the corresponding IDs. Inarguably, the two apps do not "care" whether the user has bought a new device and installed both of them there, since their goal is to extend the user profile that each one of them has created by fusing all the available information. This profile spans throughout a session, therefore, their goal is to anonymize a session, regardless of its span. If they manage to exchange the user IDs, then the developers may use them to request the needed information from each other. Note that due to the nature of Android Package Manager class, all apps are aware of which apps are installed in the system without requesting any permission. Therefore, the challenge lies in the exchange of the user IDs through a communication channel.

[1] https://copperhead.co/android/.

For the scope of clarity, in what follows we omit the use of e.g. encryption and digital signatures to hide the content of exchanged messages or to verify their source and authenticity. Moreover, we study each method independently considering how one could exchange the needed information, even if the previous ones did not exist.

Finally, it is apparent that even if the user does not authenticate to the app, hence there is no directly linked user ID, the app may create a random ID and use it as the session ID. Since this is linked to the session and can be stored by the app in its storage space, this ID can also serve as the user ID for the lifespan of the app. However, as it will be discussed in the next sections, a random ID can also serve in the cases of local communication between apps, while for remote communication further measures should be taken.

4 Exchanging Unique Identifiers

In the following paragraphs, we discuss methods that allow apps to correlate user information without using unique hardware identifiers. More importantly, all the methods which are described do not require any permission; let alone dangerous ones, as they depend on inherent Android mechanisms and structures, so no user interaction is required. The basic overview of the proposed methods is illustrated in Fig. 1, while code snippets that provide these functionalities can be found in the Appendix and illustrate not only how easy they can be applied, but also that many of them could be realised through reflection to avoid static code analysis.

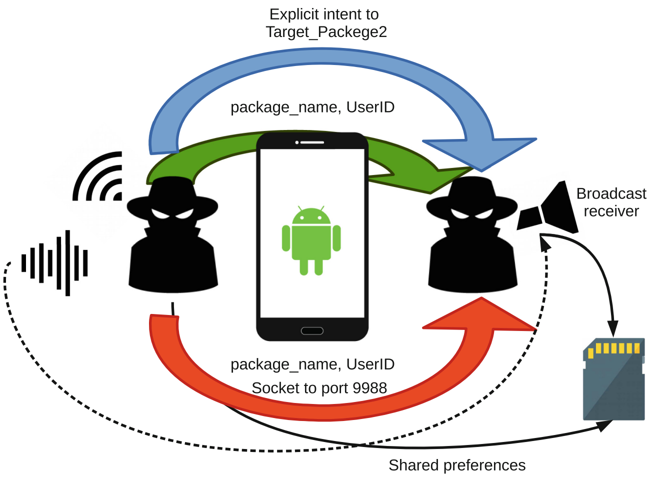


Fig. 1. Proposed fingerprinting methods

4.1 Sockets

In Android each application corresponds to a different user and is executed in an isolated VM. Therefore, apps cannot directly access the data of each other. The same applies for their temporary files, which actually are stored in the protected installation directory of each app under `/data`. Since they cannot directly share a file using the filesystem, an obvious way to exchange some information between two apps is by opening a socket. Socket programming is one of the most fundamental primitives in every Unix-like system, and despite many changes, Android is still one of them. While sockets are very common in Android, many security issues have been raised [6,13]. For our scenario the case is rather straight forward, as one of the apps needs to open a socket in a predefined port and await for connections which will transmit a message of the form: $(package\_name, ID_1)$. Clearly, to make these methods more stealth and secure, port knocking could also be considered along with encryption to counter man-in-the-middle attacks. The response will contain the ID of the other app, allowing both developers to request the desired data from each other.

4.2 Android IPC

As already discussed, Android apps belong to different users. Moreover, apps and system services run in separate processes. This restriction actually improves the security, stability, and memory management of the system. For instance, since each app runs as a different process of another user, should the system regard it as unnecessary or functioning improperly, it can easily "kill" it without jeopardizing further dependencies, allocating immediately the freed resources to the system.

To facilitate Inter-process communication (IPC), Android uses the binder framework (Binder) which exposes simple to use APIs. Due to its critical role, the security of Binder has been studied, revealing major security issues [16]. Some of the most generic Android mechanisms, such as Intents, Services, Content Providers, Messenger, and also common system services like Telephony and Notifications, utilize IPC infrastructure provided by the Binder framework.

One of the most profound ways to use Binder for our work is intents. More precisely, since we assume that each one of the two apps knows that the other is installed, the use of explicit intents is apparent. A similar approach can also be implemented through the use of Broadcast Receivers who are associated with intents due to the use of the Binder. Broadcast Receivers allow applications to register for system or application events. Therefore, once a registered event happens, the corresponding receivers are notified and a result can be transmitted to them. In this regard, the two cooperating apps may agree upon an app event, so that they can register to each other and exchange the required information.

Bound services provide another straightforward solution, as they represent a "server" component in a client-server interface. Bound services allow components, such as activities of other apps to bind to a service, send requests, receive responses, and perform IPC. A typical bound service may be utilized in order to serve another application component, running in the background. In the same sense, the messenger interface provides another well-defined IPC infrastructure that enables mutual authentication of the endpoints, if required.

Finally, a more Linux-based approach would be to use shared memory. This operation however is quite similar to the Binder-based approach, that provides a "wrapper" for the more complicated remote procedure calls (RPC) mechanisms.

4.3 Shared Preferences

Modern applications are not static and in order to adapt according to the user preferences, they need a registry to keep track of them and cater for future changes. In Android, to keep the preferences of each app isolated from the others, this registry is held in the private folder of each app in the form of an XML file. More precisely, in a file named `/data/data/package_name/shared_prefs/filename.xml`. These files however can be tagged as `MODE_WORLD_READABLE` and/or `MODE_WORLD_WRITEABLE` allowing other apps to read and write data, if they are aware of where they are stored. Apparently, the two cooperating apps can use this mechanism in conjunction with encryption to exchange the corresponding user IDs. Clearly, the exact same mechanism could be used to exchange information with a temporary file stored in the SD card, however, for the latter a dangerous permission is required.

4.4 Clipboard

Clipboard is one of the most widely used features in GUIs as it enables users to seamlessly copy information from one app to another. This is rather important in Android due to the size constraints of the device it usually operates, where typing is not as easy as in common desktop computers. Clearly, adding some information in a public readable and writable channel such as the clipboard, implies several risks which can be easily exploited [7]. In the case of Android, apps do not have to request any permission to access the clipboard, but additionally they can subscribe to receive clipboard change events allowing them collect the shared information, as well as append their data. Clearly, using the proper format and encryption, two cooperating apps can easily exchange the needed information using the clipboard.

4.5 Internet

Utilizing the Internet for communication is probably one of the most obvious solutions. Applications having harvested user data, aim foremost to transmit them to a remote service for further processing. To this end, the "Internet" permission is requited, however, as a "normal" permission (from Marshmallow and above), this requires no specific user action, nor can it be withdrawn. Interestingly, since the last Android versions, this permission is found as the most "used" one in all the available applications. Nevertheless, having a number of apps accessing the Internet does not necessarily imply that these apps are able to exchange information about a specific user. The main reason for this is because the aforementioned "techniques" reside inside a mobile device, they represent local communication channels. On the contrary, Internet access is not "local" and involves a chaotic number of possible endpoint combinations. Even in the case where all apps point to a specific web server, there is always the challenge of determining which of the available apps reside in the same device. For this reason, in order for two or more apps to establish a communication channel between them in order to exchange user specific data, unique device or user identifiers should be present.

4.6 Sensors

While dangerous permissions require user's consent, normal permissions are automatically granted and cannot be revoked. Theoretically, these permissions do not imply any security and privacy threat for the user, nonetheless, they can be used to deduce other sensitive information. For instance, the acceleration sensor does not request any permission to be used, however, it can be used as a covert channel to slowly receive information, if combined with the corresponding vibration pattern. In our use case, we assume that a trusted third party issues a request to all apps named pkg_1 to wait to receive a vibrating signal which matches a specific pattern. The pattern is triggered by pkg_2 which turns on vibration (through a normal permission) and encodes in the form of e.g. Morse code the aforementioned pattern. Should one installation of pkg_1 detect this pattern, then the apps can exchange the corresponding user ID. Similarly, instead of the vibration/accelerometer pair, one could use light/luminosity combination or try to correlate the sensed information at a given timeframe. While this task could be achieved, the clustering effort implies a lot of communication from each device making the method less practical. Alternatively, after exchanging a unique identifier through sensors, the cooperating apps could utilize the Internet communication, as already discussed, in order to communicate.

5 Identifiers

This problem of accurately identifying whether two or more mobile applications reside in the same device may be resolved by accurately matching the available identifiers of the apps. However, as already discussed, it becomes apparent that these identifiers cannot be randomly generated by the apps, since there are cases where they would never match (e.g. two random IDs from two different apps). The required identifiers should become available by a more "generic" entity, that is the user in question. To this end, the following subsections describe possible ways of deanonymizing users, without using hardware identifiers.

5.1 Procfs Information

Exploiting the concept of a trusted third party which orchestrates the exchange of the collected information, apps can use other "public" information which is shared in Android AOSP. A typical example is the uptime which indicates how many milliseconds the device is running since last boot. This information can be retrieved by the corresponding API call, or through the `/proc/uptime` file. While the apps may not collect this information simultaneously, so the clusters may contain many possible pairs, this can be overpassed by reading both the uptime and the current-time timestamps and making the required subtraction. This operation can also be easily improved by acquiring additional public information such as battery status. Interestingly, other unique, session specific, identifiers, can be found in `procfs` many of which may even be vendor specific. An attempt to map the variations of Android can be found in Android Census[2]. Some of the most profound and wide spread world readable files that can be used for deanonymization in `/proc` are listed below. In each case we provide an overview of the contents and how it can be used to allow to apps that they simultaneously operate in the same device, creating a session ID.

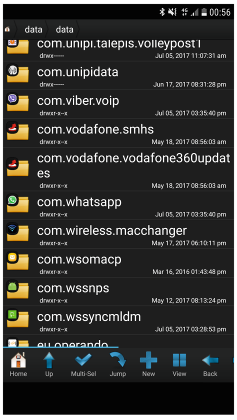
- `boot_stat`: Contains statistics about the boot process. Due to the randomization of the process IDs and the randomness of running times of each process, it is highly impossible for two devices to have the same statistics, even if the vendor is the same.
- `diskstats`: This file, as the name suggests, contains information about the disc usage. Again, two devices are not expected to have the same statistics. However, since these statistics are subject to time constraints, if the two cooperating applications do not take the snapshots simultaneously, minor changes may appear.
- `interupts`: The file contains information about the interrupts in use and how many times the processor has been interrupted. Small variations of the contents may appear from the timing of the snapshots.
- `meminfo`: Similar to `diskstats`, but for memory usage.
- `pagetypeinfo`: Keeps track of free memory distributions in terms of page order. As in `diskstats`, minor variations may appear due to snapshot timing.
- `stat`: Here several statistics about kernel activity are recorded. Similar to the case of `diskstats`.
- `usblog`: This file keeps track of USB usage and can be used for fingerprinting due to the stored timestamps.
- `vmstat`: This file keeps virtual memory statistics from the kernel, hence minor variations are expected.
- `zoneinfo`: This file provides details about the memory management of each zone, so minor variations are expected due to timing of the snapshots.

[2] https://census.tsyrklevich.net/.

5.2 Application Metadata

As already discussed, the Android's Package Manager class is capable of reporting all the installed apps to anyone requesting this information without requesting any permission. Moreover, apps can subscribe to the system event of app installation to be notified when other apps are installed and update the list of installed apps accordingly. Theoretically, this information can be used as a device fingerprint since this information is expected to differentiate two users. More interestingly though, while the `/data/pkg_name` is by default private and cannot be accessed by other apps, the exact creation date of each folder, as well as the folder's size can be retrieved for any app folder providing the necessary identifying information. Figure 2 illustrates the creation dates of a number of directories in Android regarding application installations.

Moreover, even if the set comprising of all the applications' metadata changes during time (e.g. new app installations and/or app uninstallations), its subsets can be still used as unique identifiers. A number of installed applications within a mobile device, accompanied with their installation date and their folder size can be considered a unique identifier. Furthermore, considering the fact that users have a "tendency" to use specific apps in each device they own, and even more, they most probably install the apps they have already purchased from importing their profile in Google Play Services in each new mobile device, we may safely assume that the apps' package names, especially the paid ones, within a mobile device may further uniquely identify users.



```
00:56
data  data
com.unipi.talepis.volleypost1
drwx------              Jul 05, 2017 11:07:31 am
com.unipidata
drwx------              Jun 17, 2017 08:31:28 pm
com.viber.voip
drwxr-x--x              Jul 05, 2017 03:35:40 pm
com.vodafone.smhs
drwxr-x--x              May 18, 2017 08:56:03 am
com.vodafone.vodafone360updat
es
drwxr-x--x              May 18, 2017 08:56:03 am
com.whatsapp
drwxr-x--x              Jul 05, 2017 03:35:40 pm
com.wireless.macchanger
drwxr-x--x              May 17, 2017 06:10:11 pm
com.wsomacp
drwxr-x--x              Mar 16, 2016 01:43:48 pm
com.wssnps
drwxr-x--x              May 12, 2017 08:13:24 pm
com.wssyncmldm
drwxr-x--x              Jul 05, 2017 03:28:53 pm
eu.operando
```

Fig. 2. Installed packages and the creation date of the folders

6 Discussion

The upcoming version of Android, dubbed "O" at the moment of writing, introduces a significant change for one of the identifiers that Google was promoting, namely Android_ID, a 64-bit value. Up to Nougat, Android_ID was scoped per user, but since most devices had one user, this ID was actually a unique identifier for the device. Notably, this identifier was generated on first boot, or user creation, and was expected to change only once the user wiped his device. However, in "O" this changes radically, as Android_ID is now scoped per-app. More precisely, the Android_ID is computed based on the package name, the signature, the user, and the device, theoretically deterring apps from correlating user information.

However, in Android "O" Google decided to add even more restrictions than Nougat. Therefore, the `android.os.Build.SERIAL` which returns the hardware serial also requires the dangerous `PHONE` permission. Moreover, the `ro.runtime.firstboot` property is no longer available as well as other identifiers such as `persist.service.bdroid.bdaddr` and `Settings.Secure.bluetooth_address` related to the Bluetooth MAC address and a camera hardware identifier for some HTC devices `htc.camera.sensor.front_SN`. Finally apps can get the MAC address of the WiFi card only if they are granted the `LOCAL_MAC_ADDRESS` permission.

The above changes significantly remove many capabilities of apps in collecting device specific unique identifiers. Table 1 illustrates these changes, however, many forms of transferring the needed data, especially the ones proposed by the authors, still remain.

Table 1. Applicability of fingerprinting methods.

	Marshmallow	Nougat	O
Sockets	X	X	X
Binder-based methods	X	X	X
Broadcast receivers	X	X	X
Clipboard	X	X	X
Shared preferences	X	X	
Android ID	X	X	
Procfs information	X	X	
Sensors	X	X	X
Application metadata	X	X	X

Since many of the aforementioned changes will be introduced to Android O, one major question is when the user is expected to experience them. To answer this question one needs to consider the Android diversity. Apart from the different vendor flavors that Android comes, the system is highly fragmented, see

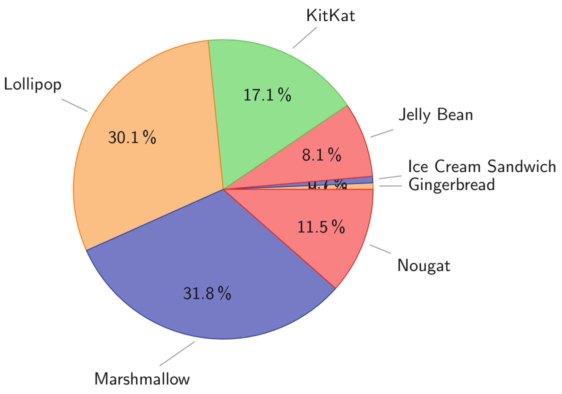


Fig. 3. Android vessions market share as of time writing Source: https://developer.android.com/about/dashboards/index.html.

Fig. 3. To determine how fast developers redesign their apps to provide features that new versions of Android offer, we used data from Tacyt[3] a platform from ElevenPaths which downloads and analyses each application's versions from Google Play and others. The reported results in Fig. 4 are per version and illustrate which API level each version is targeting. In blue and red the reference

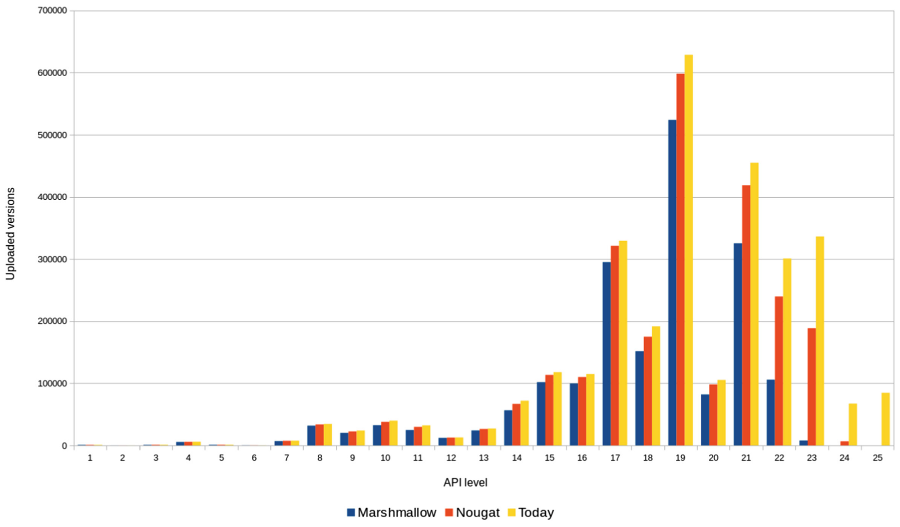


Fig. 4. Target API level per release date (Marshmallow, Nougat) and till today.

[3] https://tacyt.elevenpaths.com.

dates are the release dates of Marshmallow and Nougat respectively, while yellow represents the versions to date. While the retrieved information refers to versions and not apps, the reader can easily monitor the resulting trends. It can be observed that even after 16 months since the introduction of Nougat; the current stable version, not only few users have switched to it, but even fewer apps have integrated these features. While the most targeted API is 19, new developers' targeting is towards Marshmallow (API level 23) which dates back to October 5, 2015. Following this trend, one may speculate that the transition of apps to Android O is expected to take about two years, therefore the mechanisms discussed in this work not only currently affect millions of users, but are expected to stand for the years to come.

Finally, it is worthy to note that from the aforementioned fingerprinting capabilities, app metadata still constitute a novel unique identifier which remains even if an app is uninstalled. In this regard, app metadata can serve not only as an alternative unique ID to Android ID, but also as an alternative to Advertiser ID, working on all Android versions up to O. In fact, while Advertiser ID is user-resettable, app metadata are persistent and can only be erased after factory reset. Therefore, Table 2 summarizes the persistence of each identifier, whether it can be reset, and whether a dangerous permission is required.

Table 2. Unique identifiers and their persistence.

Unique identifier	Post O era	Persistent	User-resettable	Dangerous Permission
Android ID	X			X
MAC	X	X		X
Advertising ID	X	X	X	?
Build SERIAL	X	X		X
App metadata	X	X		
IMEI	X	X		X
IMSI	X	X		X
IP addresses	X			X
GSF android_ID	X	X		X
Contact profile	X	X	X	X

Acknowledgments. This work was supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the *OPERANDO* project (Grant Agreement no. 653704). The authors would like to thank *ElevenPaths* for their valuable feedback and granting them access to Tacyt.

Appendix

Sample Code

Category	**Sender**	**Receiver**	**Required Permissions**

Explicit Intents	intent.putExtra("msg","Some Data");
	startActivity(intent);
		String s = getIntent().getStringExtra("msg");	None

Intents Returning Results	intent.putExtra("package", "Package1");
	intent.putExtra("ID", "123456789");
	startActivityForResult(intent,request_code);"
		intent.putExtra("msgResult","Some data");
		setResult(RESULT_OK,intent);
		finish();"
			No

Local Sockets	ls.connect(new LocalSocketAddress(
	        SOCKET_ADDRESS));
	String msg = "Some Data";
	ls.getOutputStream().write(msg.getBytes());
	ls.getOutputStream().close();"
		LocalSocket ls = server.accept();
		InputStream input = ls.getInputStream();
		int readbytes = input.read();"
			No

Remote Sockets	Socket socket = serverSocket.accept();
	OutputStream outputStream;
	outputStream = socket.getOutputStream();
	String msg = "Some Data";
	PrintStream ps = new PrintStream(outputStream);
	ps.print(msg);
	ps.close();"
		InputStream input = socket.getInputStream();
		int readBytes = input.read();
		socket.close();"
			Internet

Bound Services	bindService(intent, mConnection, Context. BIND_AUTO_CREATE);"	public IBinder onBind(Intent intent) {return mBinder;}"	No

Broadcast Receivers	intent.setAction(SOME_ACTION);
	intent.putExtra("msg","Some Data");
	sendBroadcast(intent);"
		String s = arg1.getExtras().getString("
		        msg");
		}"
			No

App Local Storage	Editor edit = prefs.edit();
	edit.putString("msg", "Some Data");
	edit.commit();"
		SharedPreferences prefs = context.
		        getSharedPreferences("SP", Context.
		        MODE_WORLD_READABLE);
		String s= prefs.getString("msg", "No Data found
		        ");"
			No

Device Storage	String msg = "Some Data";
	fos.write(msg.getBytes());
	fos.close();"
		DataInputStream dis = new DataInputStream(fis);
		BufferedReader br = new BufferedReader(new
		        InputStreamReader(dis));
		String s = br.readLine();
		in.close();"
			Read/Write Storage

ClipBoard	clipboardManager = (ClipboardManager)
	        getSystemService(CLIPBOARD_SERVICE);
	ClipData clipData;
	clipData = ClipData.newPlainText("msg", "Some
	        Data");
	clipboardManager.setPrimaryClip(clipData);"
		clipboardManager = (ClipboardManager)
		        getSystemService(CLIPBOARD_SERVICE);
		ClipData clipData = clipboardManager.
		        getPrimaryClip();
		ClipData.Item item = clipData.getItemAt(0);
		String s = item.getText().toString();"
			No

References

1. Android Developers: Best practices for unique identifiers. https://developer.android.com/training/articles/user-data-ids.html. Accessed 5 July 2017
2. Android Developers Blog: Changes to device identifiers in Android O. https://android-developers.googleblog.com/2017/04/changes-to-device-identifiers-in.html. Accessed 24 July 2017
3. Book, T., Pridgen, A., Wallach, D.S.: Longitudinal analysis of android ad library permissions. arXiv preprint arXiv:1303.0857 (2013)
4. Book, T., Wallach, D.S.: A case of collusion: a study of the interface between ad libraries and their apps. In: Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, pp. 79–86. ACM (2013)

5. Demetriou, S., Merrill, W., Yang, W., Zhang, A., Gunter, C.A.: Free for all! assessing user data exposure to advertising libraries on android. In: NDSS (2016)
6. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: Why eve and mallory love android: an analysis of android SSL (in)security. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 50–61. ACM (2012)
7. Fahl, S., Harbach, M., Oltrogge, M., Muders, T., Smith, M.: Hey, you, get off of my clipboard. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 144–161. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_12
8. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: user attention, comprehension, and behavior. In: Proceedings of the Eighth Symposium on Usable Privacy and Security, p. 3. ACM (2012)
9. Google: Android o behavior changes. https://developer.android.com/preview/behavior-changes.html. Accessed 24 July 2017
10. Google: Google privacy policy. https://www.google.com/intl/en/policies/privacy/. Accessed 24 July 2017
11. Haystack: The haystack project. https://haystack.mobi/. Accessed 24 July 2017
12. Hildebrandt, M., Gutwirth, S. (eds.): Profiling the European Citizen, Cross-Disciplinary Perspectives. Springer, Dordrecht (2008). https://doi.org/10.1007/978-1-4020-6914-7
13. Jia, Y.J., Chen, Q.A., Lin, Y., Kong, C., Mao, Z.M.: Open doors for bob and mallory: open port usage in android apps and security implications. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 190–203. IEEE (2017)
14. Kohno, T. (ed.): Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, 8–10 August 2012. USENIX Association (2012). https://www.usenix.org/publications/proceedings/?f[0]=im_group_audience%3A334
15. Orthacker, C., Teufl, P., Kraxberger, S., Lackner, G., Gissing, M., Marsalek, A., Leibetseder, J., Prevenhueber, O.: Android security permissions – can we trust them? In: Prasad, R., Farkas, K., Schmidt, A.U., Lioy, A., Russello, G., Luccio, F.L. (eds.) MobiSec 2011. LNICST, vol. 94, pp. 40–51. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30244-2_4
16. Peles, O., Hay, R.: One class to rule them all 0-day deserialization vulnerabilities in android. In: Proceedings of the 9th USENIX Conference on Offensive Technologies, p. 5. USENIX Association (2015)
17. Son, S., Kim, D., Shmatikov, V.: What mobile ads know about mobile users. In: NDSS (2016)
18. Stevens, R., Gibler, C., Crussell, J., Erickson, J., Chen, H.: Investigating user privacy in android ad libraries. In: Proceedings of the 2012 Workshop on Mobile Security Technologies (MoST) (2012)
19. The Guardian: Morality of mining for data in a world where nothing is sacred (2009). https://www.theguardian.com/uk/2009/feb/25/database-state-ippr-paper
20. Vallina-Rodriguez, N., Sundaresan, S., Razaghpanah, A., Nithyanand, R., Allman, M., Kreibich, C., Gill, P.: Tracking the trackers: towards understanding the mobile advertising and tracking ecosystem. CoRR abs/1609.07190 (2016). http://arxiv.org/abs/1609.07190
21. Vanrykel, E., Acar, G., Herrmann, M., Diaz, C.: Leaky birds: exploiting mobile application traffic for surveillance. In: Grossklags, J., Preneel, B. (eds.) FC 2016. LNCS, vol. 9603, pp. 367–384. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54970-4_22