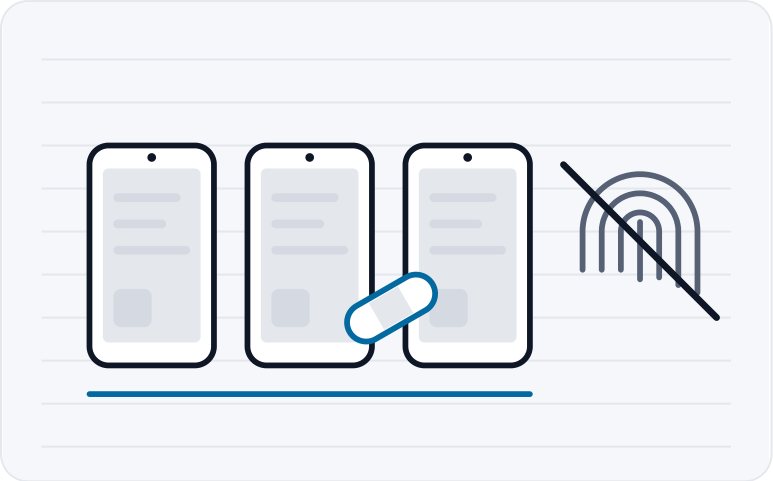
A-SIT Technology

E-Government Innovation Center

1 December 2023 Analysis · Mobile Security · Privacy & Crypto · 14 min read

Reducing the Fingerprintability of Android Smartphones

Gerald Palfinger



Authors	Gerald Palfinger
Version	2023-11-30
Source	https://technology2.a-sit.at/en/publications/reducing-the-fingerprintability-of-android-smartphones/
Note	Machine translation.

Key findings

- Fingerprintable sources differ between devices running Android 14 but stay stable on each individual device.
- Patches that align return values with a reference device are generated automatically from the list.
- The patches are applied to individual apps; no change to the Android runtime or the device is needed.
- The resulting patch package contains 512 patches for 162 classes.
- Limits: the locale can affect what an app displays; correctly implemented remote attestation defeats the approach.

This report introduces a method which makes it possible to reduce the fingerprintability of Android smartphones. This involves changing information sources in the programming interface so that they are reporting the same information across different devices. Initially, a search was made for information sources that differ between different devices and thus potentially allow a fingerprint to be created in order to subsequently identify a device across different applications. With the help of the detected information sources, patches are automatically generated which adapt the returned values of the information sources against a reference device in order to reduce the fingerprintability. These patches can be applied directly to applications and therefore do not require any changes to the device itself.

1. Introduction

Smartphone applications have become indispensable tools that offer a wide range of functions and services. They serve not only for entertainment, but also enable communication in everyday working life and access to a variety of applications that make daily life easier. To make these applications possible, there is an extensive interface that can be used to program applications. In addition to access to device functions, this interface also provides access to various information sources. While directly identifiable information has either been removed or requires permission from the user, a fingerprint can be created by combining many information sources. The literature has shown that there is a large number of information sources providing such information on both Android and iOS [1] [2].

To improve the privacy of users, the distinguishing features between devices should be reduced, as this makes it harder to create a unique fingerprint. This is the goal of this project. Changing the programming interface directly on the system, however, would require a far-reaching intervention in the smartphone's system, which for security reasons is not possible without unlocking the bootloader and additionally rooting the device. The approach chosen was therefore to patch potentially fingerprinting applications directly. This also has the advantage that possible effects remain limited to the patched applications. The following approach was chosen for creating the patches. First, information sources that differ between the examined devices were searched for on different devices. These potentially provide information that may be suitable for creating a fingerprint. These sources are collected so that, in the next step, patches can be developed that adapt the information across the different devices. An application was developed for this purpose which creates the patches automatically from the collected information sources. The patches are applied to the applications using the patch routine from [3].

2. Background

It was shown in [2] that, similar to the browser, a unique fingerprint of a device can also be created on Android by combining information. To this end, the authors manually searched for information sources that could make it possible to fingerprint a device. The documentation of the programming interface was chosen as the starting point. Likewise, starting from the settings available on the device, ways were sought to retrieve this information from an application. Finally, it was shown which information sources can be used to create a fingerprint. In addition, applications on Android were examined for their fingerprinting activities. It was shown in [4] that tracking and advertising libraries in particular retrieve a large number of information sources that can be used to create a fingerprint.

With Android 10 [5], access for normal applications to many unique identifiers that were still accessible in earlier versions was removed. These included, for example, the device ID, various MAC addresses and the IMEI of the device. System applications, that is, those preinstalled on the smartphone, still have access to these identifiers through privileged permissions. For example, the WLAN service still requires access to the MAC address of the WLAN interface, while the telephony service has access to various information required for mobile communications, such as the IMEI. In [6], it was investigated whether these services can be exploited to disclose this privileged information. It was found that, even under Android 10, one method still returned the unique identifier ICCID. Since fixing it via a security update caused problems, this error was only fixed with Android 11. While no further problems were identified in Android itself, a large number of ways to circumvent the protections were found on devices from other manufacturers. Through device-specific customisations, a large number of identifiers could still be accessed.

In [7], it was systematically investigated which information sources can be used to create a fingerprint. For this purpose, a framework was created that automatically retrieves information sources in the programming interface and compares them between different devices.

Information sources that deliver different values are assumed to be fingerprintable. A large number of information sources suitable for creating a fingerprint were found. These included methods, fields and also content providers, all of which are accessible via the programming interface.

It has also been shown that creating a fingerprint is possible on iOS [1]. The authors selected 29 information sources from the programming interface. These were then collected on different devices with the help of an application in order to create a fingerprint of these devices. In the experiments, the authors showed that by collecting the information sources again, devices can be recognised with high probability based on the previously created fingerprint.

In [8], an analysis environment was created that aims to detect previously unknown libraries which collect data for fingerprinting iOS devices. For this purpose, several known fingerprinting libraries were analysed. The information sources used by these libraries were collected and listed in the paper. As it turned out, many libraries only need information from a small number of information sources to create a fingerprint. The behaviour of these libraries therefore differs from browser fingerprinting, in which a large amount of information is often retrieved. In a further experiment, an attempt was made to detect fingerprinting activities in real applications. For this purpose, a Frida analysis environment on a jailbroken iPhone is used. It was assumed that the values for fingerprinting are read out one after another within a short time window. If enough potentially fingerprintable values are read within one time window, the application is therefore assumed to be creating a fingerprint. With this procedure, further information sources used for fingerprinting could be identified. These include, for example, various accessibility settings or other settings, as well as device and operating system properties.

To obtain a comprehensive overview of fingerprintable information sources, an analysis framework that automatically detects such information sources was also created for iOS [9]. Similar to Android, methods are called here and their return values collected, and properties of class objects are read. It was found that hundreds of methods and objects can potentially be used to create a fingerprint.

In [10], an extension for browsers was presented that is intended to reduce the fingerprintability of Firefox and Chrome. To this end, the programming interface of the browser is restricted. In particular, methods and properties are restricted or modified that make it possible to determine information about the device used,

the browser instance, the user, and the environment. Likewise, sensor outputs of the device and the location of the user can be simulated.

3. Implementation

To reduce the fingerprintability of the information sources of the Android programming interface, it must first be determined which information sources are potentially fingerprintable. For this purpose, various devices were compared using an automated tool, and the differing information sources were recorded in a list. Based on this list, patches were generated that change the return values of the information sources. These patches were generated automatically using a purpose-built tool. Finally, the created patches must be applied to individual applications. These three steps are discussed in more detail in the following paragraphs.

3.1. Collection of the Information Sources

To implement the method, a list of fingerprintable information sources and the unified return value is needed first. The framework from [7] is used to detect these. To obtain a list that is as up to date as possible, testing is done on devices running the current Android 14. Both methods of the programming interface are called and data is retrieved from content providers. The return values of the methods and the retrieved data are stored. To remove temporary values that change after an application or device restart, the collection is run twice per device. The collected and cleaned data of the devices are then compared with each other. Data that differ between the devices but are stable on the device itself are marked as fingerprintable.

The information sources found are compiled in a list. In addition to the exact signature of the method or the name of the content provider, this contains the values obtained on the test devices. This list serves as the basis for the automatic creation of the patches. An example of elements of this list is shown in Listing 1. The list contains three examples each of elements from content providers (starting with `//`) and return values of methods (starting with `public`). These are the selected notification sound, the autofill service used (which allows conclusions about the password manager used), enabled input methods, as well as the size of icons, the country and the mobile network operator used.

```
   //settings/system/notification_sound$notification_sound :::[(
   'p6p',
   ['[content://media/internal/audio/media/68?title=Iapetus&can
   onical=1, false]']), ('p4',
   ['[content://media/internal/audio/media/171?title=Pixie%20Du
   st&canonical=1, false]'])]
   //settings/secure$autofill_service :::[('p6p',
   ['[com.kunzisoft.keepass.libre/com.kunzisoft.keepass.autofil
   l.KeeAutofillService, true]']), ('p4',
   ['[com.google.android.gms/.autofill.service.AutofillService,
   false]'])]
   //settings/secure$enabled_input_methods :::[('p6p',
   ['[com.android.inputmethod.latin/.LatinIME;
1337596075;774684257:com.kunzisoft.keepass.libre/com.kunziso
   ft.keepass.magikeyboard.MagikeyboardService:com.simplemobile
   tools.keyboard/.services.SimpleKeyboardIME:org.kde.kdeconnec
   t_tp/org.kde.kdeconnect.Plugins.RemoteKeyboardPlugin.RemoteK
   eyboardService:org.fcitx.fcitx5.android/.input.FcitxInputMet
   hodService, true]']), ('p4',
   ['[com.google.android.inputmethod.latin/com.android.inputmet
   hod.latin.LatinIME:com.google.android.tts/com.google.android
   .apps.speech.tts.googletts.settings.asr.voiceime.VoiceInputM
   ethodService, true]'])]
   public int
   android.app.ActivityManager.getLauncherLargeIconSize() :::[('
   p6p', ['168']), ('p4', ['126'])]
   public java.lang.String
   android.icu.util.ULocale.getDisplayName() :::[('p6p',
   ['Deutsch (Österreich)']), ('p4', ['English (United
   States)'])]
   public java.lang.String
   android.telephony.TelephonyManager.getNetworkOperatorName ():
   ::[('p6p', ['educom']), ('p4', ['T-Mobile'])]
```

Listing 1: Examples of differences between the examined devices, including the contained example values, shown in list form.

3.2. Patch Creation

A program was developed to create the patches automatically. The list of information sources shown in the previous section, which contains the values obtained on the devices, is used as the starting point. If the program recognises a method by the `public` attribute, the information required to change this method is extracted from it. This includes in particular the type of the return value, the name of the class, the method name, and the data types of the method's parameters. In addition, it is detected whether the method is static or an instance method. For the method `getDisplayName()` from Listing 1, `java.lang.String` was extracted as the type of the return value. The class was recognised as `android.icu.util.ULocale` and

the method name as `getDisplayName()`. Since there are no parameters, no information about any parameters is extracted in this case. The method under consideration is an instance method, as no `static` attribute appears in the method signature.

The information extracted by the program is then used to create the actual patch. This automatically created patch is shown for the method `getDisplayName()` in Listing 2. Since this is an instance method, the instance object is additionally passed to the patch method as a parameter. In the example shown, it has the name `thiz` . In addition to the modified method, the original method is also called from it. This is done to trigger any desired changes to the instance object or effects on the program flow.

```
  @PatchClass("android.icu.util.ULocale")
  public class AndroidIcuUtilULocalePatches {

  @PatchInstanceMethod public static java.lang.String
getDisplayName(android.icu.util.ULocale thiz) {
OriginalMethods.android_icu_util_ULocale.getDisplayName(); return "US"; } }
```

Listing 2: An example of an automatically created patch. It modifies the method `getDisplayName()` of the class `android.icu.util.ULocale` .

To modify the information available via content providers, the methods for retrieving content providers are adapted. As a result, adapting the content providers also requires no deep intervention in the system services that provide the respective content providers. Content providers are identified by their content URI starting with `content://`. The entry `autofill_service` from Listing 1 is, for example, accessible via the content provider named `settings/secure`. The associated URI would therefore be `content://settings/secure` . To change the values of this content provider, the retrieval options in the programming interface are adapted. Content providers can in particular be retrieved via a so-called `ContentResolver`. The content URI is passed to the method `query(Uri,…)` of the `ContentResolver`. This method is overridden by a patch. The patched method reacts to content URIs that have been identified as fingerprintable. To do so, it checks the passed URI against a list of content providers identified as fingerprintable. If a patched application requests access to such a content URI, the application is returned a reference to a patched content provider instead of the reference to the fingerprintable content provider. In this case, the original content provider can be completely replaced by a patched one. This strategy is applied, for example, to content providers that provide access to system settings, since these can generally always be influenced by the users. It is equally possible, however, to modify only the access to individual values identified as fingerprintable.

3.3. Patch Application

The patches created in the previous step are then combined into a patch package. This can then be applied to individual applications. Because the changes are made to the program package, no change to the Android runtime environment, and thus no change to the device itself, is necessary. In implementing the countermeasures, no programming interfaces are changed whose use the user must explicitly grant by

allowing a permission. The created patch package contains a total of 512 patches for 162 classes, which are applied to the application to be patched. After the patches have been applied, the patched application can be installed on a device.

3.4. Discussion

To reduce fingerprintability, the essential approach is to imitate a reference device as faithfully and consistently as possible without restricting the usability of applications. Some of the changes can, however, affect the applications used. For example, unifying the locale can affect the language and currency symbols displayed in the application. To work around this, individual patches can be disabled for affected applications.

To apply changes to applications, they must be repackaged and therefore also re-signed. Since the original key used for the application is reserved for the application developer, this new signature must be made with a different, locally generated key. This can cause problems with some applications. This is especially the case when Android's Key Attestation is used to verify the integrity of the application. While local signature checks can be removed by the patching framework used, this is technically not possible with a correctly implemented remote attestation. In this case, the framework cannot be used successfully. According to [3], however, such a check is used in only a very small number of applications.

4. Conclusion

This report presented a framework that can reduce the fingerprintability of the programming interface of Android smartphones. To this end, potentially fingerprintable information sources were first collected by comparing different smartphones running Android 14. A reference device was chosen from these devices, to which other devices were to be adapted. For this purpose, patches were created automatically that adapt the return values of methods of the programming interface to a chosen reference device. These patches can be applied to individual program packages, so that the rest of the system is not affected by the changes. Some of the changes can, however, affect the patched application. For example, the language may deviate from the system settings or certain functions may not be available.

References

[1] A. Kurtz, H. Gascon, T. Becker, K. Rieck und F. Freiling, „Fingerprinting Mobile Devices Using Personalized Configurations," in Privacy Enhancing Technologies, 2016.

[2] W. Wu, J. Wu, Y. Wang, Z. Ling und M. Yang, „Efficient Fingerprinting-Based Android Device Identification With Zero-Permission Identifiers," IEEE Access, pp. 8073-8083, 2016.

[3] F. Draschbacher, „A2P2 - An Android Application Patching Pipeline Based On Generic Changesets," in International Conference on Availability, Reliability and Security, Benevento, Italy, 2023.

[4] C. F. Torres und H. Jonker, „Investigating Fingerprinters and Fingerprinting-Alike Behaviour of Android Applications," in European Symposium on Research in Computer Security, 2018.

[5] „Android Developers," [Online]. Available: https://developer.android.com/about/versions/10/privacy/changes.

[6] M. H. Meng, Q. Zhang, G. Xia, Y. Zheng, Y. Zhang, G. Bai, Z. Liu, S. G. Teo und J. S. Dong, „Post-GDPR Threat Hunting on Android Phones: Dissecting OS-level Safeguards of User-unresettable Identifiers," in NDSS, 2023.

[7] G. Palfinger und B. Prünster, „AndroPRINT: analysing the fingerprintability of the Android API," in International Conference on Availability, Reliability and Security, Virtual, 2020.

[8] K. Heid, V. Andrae und J. Heider, „Towards detecting device fingerprinting on iOS with API function hooking," in European Interdisciplinary Cybersecurity Conference, Stavanger, Norway, 2023.

[9] G. Palfinger, „OCScraper: Automated Analysis of the Fingerprintability of the iOS API," in SECRYPT, Rome, 2023.

[10] L. Polčák, M. Saloň, G. Maone, R. Hranický und M. McMahon, „JShelter: Give Me My Browser Back," in 20th International Conference on Security and Cryptography, SECRYPT 2023, Rome, 2023.