Android Data Storage Locations and What App Developers Do with It from a Security and Privacy Perspective

Kris Heid[1], Tobias Tefke[1,2], Jens Heider[1] and Ralf C. Staudemeyer[2]
[1]Fraunhofer SIT, Rheinstr. 75, D-64295 Darmstadt, Germany
[2]Schmalkalden University of Applied Sciences, Blechhammer, D-98574 Schmalkalden, Germany

Keywords: Security and Privacy, Android, Software and Application Security, Data Security, Personal Data Leakage, File System Security, Dynamic Analysis.

Abstract: Many Android apps handle and store sensible data on the smartphone, such as for example passwords, API keys or messages. This information must of course be protected and thus more and more protected storage options and storage isolation techniques were implemented in recent Android version. This results in good security and privacy mechanisms provided to Android developers. However, the question is how well these measures are implemented in todays apps. In this publication, we are presenting an automated dynamic analysis environment which we use to analyze the top 1000 Android apps. Filesystem API accesses of these apps are evaluated and judged how well Android's protected storage locations are leveraged or abused.

1 INTRODUCTION

Mobile operating systems become more and more widespread. People rather own a smartphone than a personal computer. Many tasks, where previously a desktop computer was necessary, can now be done with smartphones. Thus, smartphones are integrated into our private as well as business life. Due to the omnipresence, mobile operating systems are prone to be attacked by hackers and malicious apps or leveraged for data collection by advertisement companies. Mobile users must be protected against such behavior. To provide protection, one has to look at the OWASP Mobile Top 10[1]. Especially the first two elements: *M1: Improper Platform Usage* and *M2: Insecure Data Storage*. They show that data storage at improper locations are major issues on mobile operating systems.

Apps often manage login credentials, API keys, store (user entered) personal information such as health data and private messages, credit card/payment information or search terms. A user expects that such information is securely stored. The user expects for example login credentials to just be accessible to the respective app which stored them. However, there also exists data which is desirable shared among apps, such as for example photos.

The Android developers guide[2] gives advice where which data should be stored securely. As of today, Android provides more than a hand full of different app exclusive and publicly shared data storage locations with different methods to access each location. Nevertheless, it is in the developer's responsibility to comply with these norms. As later on shown by related work, complying to stated norms is still a major issue to app developers.

Sensitive data, stored at shared locations, might leak to (partly) malicious apps. Over the past years, Android tackled this problem by successively restricting app's permissions to the file system. Firstly, permissions where introduced to which the user had to comply at install time. Later on, these became fine granular by requesting permissions to resources during run time, when needed. Lastly, scoped storage was introduced with Android 10, allowing setting app read/write permissions to specific folders. Especially enforcing scoped storage for apps with an Android 11 *target SDK* is a big step towards protecting sensitive information.

However, there is still room for app misbehavior since often app users (without technical background) don't realize the impact of their decisions to allow or deny permissions. Also, app developers often support very old Android versions and need to adapt their code to more than a hand full of Android versions, which is tricky and possibly a big cause of non-compliance to Android developers guide.

We have dynamically tested the top 1000 apps from Google Play (former Playstore) to get an idea

[1]https://owasp.org/www-project-mobile-top-10/
[2]https://developer.android.com/training/data-storage

Heid, K., Tefke, T., Heider, J. and Staudemeyer, R.
Android Data Storage Locations and What App Developers Do with It from a Security and Privacy Perspective.
DOI: 10.5220/0010838200003120
In *Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022)*, pages 378-387
ISBN: 978-989-758-553-1; ISSN: 2184-4356
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

of how good or bad the current apps comply with secure storage locations of their data. In our approach, we stimulate the app for ten minutes and track the used file system related Android-API functions. Afterwards, we analyze the API calls and evaluate the written files on the file system for violations of storage locations postulated in Android developer guides.

The remainder of this paper is structured as follows: First, we have a look at related work, their findings and used approaches. Afterwards, a brief introduction into the Android storage locations is given. In the next step, the experimental setup to carry out the dynamic tests is described. Lastly, findings of the dynamic tests are presented and a conclusion is drawn.

2 RELATED WORK

We categorize related work into two categories as other work relates to ours in different aspects. Firstly, we review other work researching app's insecure data storage. Afterwards, we also highlight analysis tools focusing on finding insecure data storage.

2.1 Research Findings

Previous work confirmed that storing sensitive information on external storage is problematic. (Schmeelk and Tao, 2020) evaluated insecure data storage in health care apps. The authors analyzed more than 200 health care apps in order to find occurrences of insecure data storage. All apps collect personally identifiable information. Their research shows that approximately half of the investigated healthcare apps potentially store sensitive data insecurely.

Furthermore, 70 out of 200 apps wrote data to shared storage, but only nine of them included cryptographic libraries. The authors assumed these libraries are being used for encrypting sensitive information before storing it on shared storage. This indicates that more than 60 of the 200 apps have likely been vulnerable to insecure data storage.

(Liu et al., 2014) criticise that the Android developer documentation advises that apps should store private app data in a directory on the external storage which exposes sensitive data to other apps.

The authors analysed 1648 apps and concluded that about a quarter of them leak private data. They selected 30 apps considered to potentially leak data and proved that 27 apps of them stored data insecurely.

Insecure data storage of apps has been further confirmed by (Gisdakis et al.,). In their research, they also showed that additional information can easily and accurately be derived from leaked data. (Du et al., 2018) could further prove that apps store sensitive data insecurely. They also evaluated whether apps perform input validation when rereading stored information. Here, their study showed that from almost 14,000 apps more than 30% did not perform input validation.

2.2 Related App Analysis Tools

App analysis tools have also been developed by other researchers. (Yang and Yang, 2012) presented a tool called LeakMiner, making use of static taint analysis. They identified 145 out of 1750 apps leaking information. However, the author's definition of sensitive data is very strict and findings are not further classified.

Work of (Friedman and Sainz, 2016) focuses on file system usage and tracking the file type. The authors point out that they did not analyze file content. Their paper aims to support the design of file systems and backup solutions.

(Bläsing et al., 2010), (Burguera et al., 2011) and (Bierma et al., 2014) use dynamic analysis methods in order to detect malware in apps. (Bläsing et al., 2010) also used static analysis methods. However, they monitor system calls instead of intercepting Java calls. The collected syscalls are being used to distinguish between normal and malicious apps. (Burguera et al., 2011) improved the approach of (Bläsing et al., 2010) by using crowd sourcing. In the approach of (Bierma et al., 2014), disk images are being compared after the app has been driven, file streams are not being monitored.

2.3 Differentiation to Related Work

The studies presented in Section 2.1 either use static analysis techniques or rely on attack models in which an adversary copies the public storage directory of an application and performs the analysis later on. In our work we use dynamic analysis techniques in order to find occurrences of insecure data storage. In most cases this leads to a higher degree of accuracy in detecting potential vulnerabilities in comparison to other approaches. Furthermore, it is exactly logged which function call is responsible for the leakage of sensitive information. This makes it easy to find the line of code from which an insecure data storage action originates. Moreover, in comparison to some presented approaches, our experimental setup does not require access to the source code of the evaluated app. Also, the introduction of new storage concepts such as scoped storage, the Media Store API and Storage Access Framework require an evaluation how

well these concepts are adapted, which related work doesn't cover.

2.4 Contribution

The first, but not the central contribution of this publication is a tool and a method to execute large scale dynamic app tests. The tool is able to stimulate an app (= UI interaction + system broadcasts) and observe the app's Android API interaction. It also automatically analyzes observed Android API interactions for privacy and security violations. We analyzed file system API interactions with the 1000 most popular apps on Google Play in this publication. The dynamic analysis tool is capable of detecting different kinds of data storage and evaluate occurrences of insecure data storage.

We see this publication's main contribution in the presented results of the dynamic analysis runs. We contribute insights on current app's implementation details on file system and different storage class interactions. We evaluate the usage of scoped storage in current apps and access to system files and executables. Additionally, app's sensitive data storage locations revealed during the dynamic analysis are evaluated from the security and privacy perspective.

3 THE ANDROID FILE SYSTEM

The Android operating system provides app developers with different ways to store data. Data can be stored in different locations with varying security and privacy levels. Therefore, app developers can select a storage location based on the sensitiveness of the information to be stored. The following subsections cover the different storage locations and their intended containable data types and use case.

3.1 Internal Storage

Each app is being provided with its own exclusive storage directory on the internal storage. The internal storage directories of the installed apps are segregated from each other. Access is controlled by the kernel, which makes it impossible to access other app's exclusive storage (without root). Furthermore, since Android 10 the internal storage directories can be encrypted[3].

With these security features, the internal app exclusive storage provides protection against lost device and malicious app scenarios. However, the size of the internal storage is in most cases rather small in comparison to external storage. Therefore, the Android developer documentation proposes to use external storage for storing larger data volumes[3].

3.2 External Storage

Android provides the option to store data on external storage. Formerly, this was the SD card, but since newer phones mostly don't use SD cards anymore and the term external storage was established. Each app with the permission to access external storage can read and write to it[3]. External storage can be used as shared storage to exchange data between apps[3]. However, external storage should not be used to store sensitive data in a shared directory.

Since Android 10, it is possible to use app exclusive directories on external storage, which are not accessible to other apps if scoped storage is enabled. Here, apps do not need special permissions in order to access their files. Those directories are removed if the app is uninstalled[3].

To further improve the security, Android 10 introduces scoped storage. Scoped storage improves the privacy of user's files by giving apps the permission to specifically read or write certain files or folders and not all external storage[4]. If apps target API level 30 (Android 11) or newer, scoped storage is mandatory. However, apps targeting API level 29 (Android 10) are still able to opt out of scoped storage.

Media files like pictures, music and videos have dedicated folders where they should be written, shared and automatically detected. These locations are accessible through the MediaStore API and out of the scoped storage's influence area[5]. Respectively, downloads and documents have their dedicated folders on the external storage, accessible through the Storage Access Framework. Moreover, large data sets, like machine learning data, can be exchanged between apps[6] using the BlobStoreManager[7]. Here, it is also possible to select apps with access such files.

[3]https://developer.android.com/training/data-storage/app-specific (acc. on 10/02/2021)

[4]https://source.android.com/devices/storage/scoped (acc. on 09/02/2021)

[5]https://developer.android.com/training/data-storage/shared/media (acc. on 09/02/2021)

[6]https://developer.android.com/training/data-storage/shared/datasets (acc. on 09/02/2021)

[7]https://developer.android.com/reference/android/app/blob/BlobStoreManager (acc. on 10/02/2021)

3.3 Shared Preferences

Shared Preferences make it possible to store data as key-value pairs. Those are being stored in plain text files in internal storage[8]. Until Android 7, data stored in Shared Preferences could either only be accessible to the corresponding app or to all apps[9]. Shared Preferences for later Android versions are always app-specific, they can be also be used for storing sensitive information.

3.4 Databases

Apps can use SQLite databases located on internal storage[8] to store structured data, which can also be encrypted. Therefore, they can also be used to store sensitive information.

There are also options to use databases which store the data on remote servers. A popular example is Firebase[8]. However, evaluating data stored remotely is beyond the scope of this work.

3.5 Android Keystore

In order to store cryptograhic keys securely, Android provides a so-called *Keystore*[10]. The *Keystore* can be stored as a file and the app developer can decide on its location.

In order to access the *Keystore*, a secret is required[11]. As long as the secret for the *Keystore* is not exposed, the keys can be considered safe. Keys inside the *Keystore* can not be extracted even if the kernel has been compromised (Mayrhofer et al., 2019).

However, the hardware must still be trustworthy, which can hardly be guaranteed (Irvine and Levitt, 2007). To address this, phones can implement Strongbox, in which the *Keystore* is being realized in tamper resistant hardware. Then, the keys can still not be read if cold boot attacks happen or hardware bugs are being exploited, unless an attacker gains access to the low-level communication interfaces (Mayrhofer et al., 2019).

[8]https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05d-testing-data-storage (acc. on 09/02/2021)

[9]https://developer.android.com/training/data-storage/shared-preferences (acc. on 09/02/2021)

[10]https://developer.android.com/training/articles/keystore (acc. on 09/02/2021)

[11]https://developer.android.com/reference/java/security/KeyStore (acc. on 09/02/2021)

3.6 Cloud Storage

It is also possible to store data in cloud storage. In this case transfer of sensitive data should be encrypted. Furthermore, server certificates should be verified and only trusted if the server returns the correct certificate in order to prevent man-in-the-middle-attacks[12]. However, evaluating cloud storage is beyond the scope of this work.

4 EXPERIMENTAL SETUP

In order to automatically and dynamically analyze several apps we used different components as depicted in Figure 1. An *App Stimulation* module to interact with the app on different levels, a *Behavior Monitor* module to log the app's interaction with system APIs in the background and a *Control Logic* module which coordinates all components and collects all logging for evaluation. As test set we are using the list of top 1000 apps in Google Play according to 42Matters[13] (as of 28th Aug 2021). The apps run on a Pixel 4a with Android 11 during the dynamic analysis.

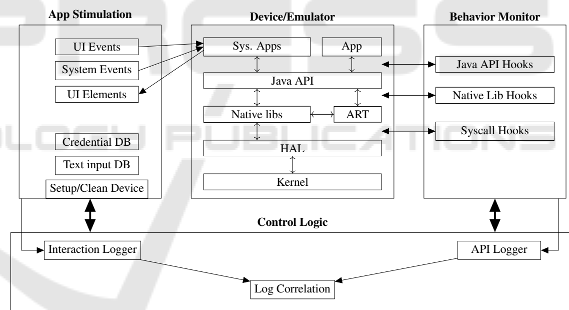


Figure 1: Tool Composition for a Dynamic Analyzer.

4.1 Control Logic

In order to produce reliable and consistent test results, we are setting up the environment in the following steps:

1. Analyzing of the *AndroidManifest.xml* included in every APK file is done. The (main) activities, receivable system events/broadcasts and used permissions are collected.
2. Lay out bait: Create predefined contacts, SMS messages, call history log and calendar entries.

[12]https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05g-testing-network-communication (acc. on 09/02/2021)

[13]https://42matters.com

The data in these elements is crafted to be unique and possibly recognizable when used by the app.

3. Install the app under test via ADB.
4. Grant all permissions to the app via ADB.
5. Start the *Behavior Monitor* module to hook all relevant APIs interacting with the file system. (see Section 4.3 for details)
6. Start app stimulation: This will interact with the UI for 10 minutes and afterwards send all broadcasts registered by the app. (see Section 4.2 for details)
7. Collect all API interaction from the *Behavior Monitor*.
8. Analyze API interaction logs and copy written files from the test device to the local (desktop-PC) folder for a detailed analysis.
9. Collect entered text into app's text fields and interacted elements from the *App Stimulation* module.
10. Analyze collected data and print an incident when data was written to an improper storage location. Further descriptions on the different analyses are provided in the respective evaluation in Section 5.
11. Restart with the next app.

4.2 App Stimulation

The following describes the applied app stimulation concepts, even though this step is essential, it is not the main contribution of this publication and thus only briefly described.

The easiest option to stimulate UI is Android Studio's Exerciser Monkey[14] which does fire completely random UI events. However, we wanted to be able to overcome login activities and provide context aware text inputs, which Exerciser Monkey is not capable of. Thus, we chose Appium[15] as our UI stimulation basis. It's a framework to interact with UI elements and has much useful features already included. We use Appium to read the currently shown UI elements and evaluate the context. If for example text input elements with the labels: username, password and a login button (or variations) are found, appropriate input data will be provided. A more detailed interaction process is described in the following:

1. Start app under test.
2. Collect UI elements (incl. content description, text, ID ...)
3. Check if current UI package name equals the app's package name, which indicates that the shown UI belongs to the app under test. If not, press back button up to three times or restart the app and go to step 2.
4. Evaluate logcat messages about an app crash. If app crashed at startup, go to step 1. If app crashed repeatedly, stop interaction.
5. Collect registered UI handlers. There exist UI handler for: login forms, login with Google, addresses, date picker, advertisement ....
6. Each handler receives the list of UI elements and evaluates labels and IDs. It calculates how many necessary and optional expected elements are found and gives a probability in percent how well it can handle the current screen.
7. Compare the probabilities of all handlers and take the one with the highest score. If no handler can drive the UI, randomly click an element.
8. Identify stuck at login: If login handler failed multiple times to log into the app and no other activity is reachable, stop interaction.
9. Go to step 1 if the elapsed time is less than 10 minutes.

We have two options available to overcome login masks. The easiest method is to use *Login with Google*, which some apps provide, since we're already logged in with a Google account during device preparation. We also use a local credentials database for some apps with manually created accounts to be used, since not all apps use *Login with Google*. However, the database currently only holds credentials for 20 apps. We did not yet have time to create more accounts. Also, many apps require a contract with the provider to login, such as for example apps from insurance companies. For such apps, one can not simple register only with a valid email address.

We use a text input database to provide context aware input into text fields. The text input fields content description, displayed text and ID are read and matched against a list of aliases for a specific text input. For example a list of aliases (start, from) provide an address to route from and the aliases (destination, to) provide a different address to route to. A navigation applications now usually provides two text input fields, where the content description, displayed text and/or ID (partially) match the given aliases and according input can be provided.

What we also consider essential for a thorough app analysis are broadcast (system event) support (currently not included in Appium). Thus we use specified broadcasts from the *AndroidManifest.xml* to

[14]https://developer.android.com/studio/test/monkey

[15]http://appium.io/

fire broadcasts via ADB to the application. This step should provide a higher coverage of the app's functionality.

4.3 Behavior Monitoring

We use Frida[16] to monitor usages of the Android API. Frida allows to dynamically hook the Java API, native libraries as well as syscalls. For now, we only concentrated on hooking the Java API and neglect the others. The main reasons are that syscalls fire at very high rates and hooking many syscalls might likely choke the app and demands extremely high performance on the observation interface. Native libraries are only sometimes used and not all Java API functionality use native libraries in the background. Also, some native interfaces sometimes miss out on information and context. As future work, we also want to evaluate the use of native libraries.

To collect information about Java API calls, we hooked the methods of the classes: `java.io.File`, `java.io.FileWriter`, `java.io.FileOutputStream`, `java.io.OutputStreamWriter`, `java.io.RandomAccessFile`, `java.io.FileChannel`, `java.io.OutputStream`, `java.io.Writer`, `java.nio.file.Files`, `android.content.ContentResolver`, `android.app.Context`, `android.app.SharedPreferences`, `android.database.sqlite.SQLiteDatabase`, `java.security.KeyStore`, `java.nio.file.Paths`, `android.app.Activity`

Each method hook collects the following information:

- Current function name and argument signature
- Argument and return values and in case of complex objects, their hash as given by `.hashCode()` inherit from `java.lang.Object`
- A timestamp.
- If the API call comes from the package under test.
- Additional fields are appended if the object contains additional information which might be relevant during evaluation. For example File objects' path (`oldFile.renameTo(file)`) can be made visible by returning the arguments decoded value `file.toString()`. Otherwise, complex objects would only return `java.io.File@<hashCode>` and might hide important information, even though references to previously logged elements with the same hash code can be made.

[16]https://frida.re

Additionally, all function calls that will delete files contain code to copy the respective file to a temporary location before deletion, for later analysis. The used Frida scripts can be reviewed on Github[17] to get a better idea of the hooked API calls.

After app stimulation finished, the collected API call list is handed to the *Control Logic* module running on the desktop PC which will later evaluate the collected API calls.

5 EVALUATION

After having run the automated app stimulation on 1000 apps for 10 minutes plus about 5 minutes aftermath each, we have analyzed the logged file operations for critical, discouraged or unusual usage. The following sections summarize our key observations.

5.1 Scoped Storage

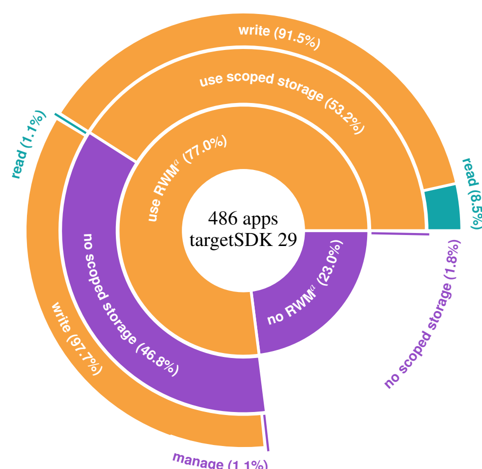
We see scoped storage (see Section 3.2) as a huge privacy benefit, since users can give per file/folder access to apps. Scoped storage was introduced with Android 10 (SDK 29) and enforced in Android 11 (SDK 30). Apps with *targetSDK* 29 have the option to opt out of scoped storage. From *targetSDK* 30 on this is no longer possible for apps.

A first aspect where we wanted to retrieve insights was the external storage usage with *targetSDK* 29 in relation to the possibility to opt out of scoped storage.

Thus, we statically analyzed permissions set in *AndroidManifest.xml* file of all apps. Like shown in Figure 2, 77% of the 486 apps with *targetSDK* 29 contain permissions to access external storage. About 46.8% of these apps actively opt out of using scoped storage. Presumably, many apps still stick with their legacy app folder on the external storage root and the new concept requires extensive special handling to support old Android versions.

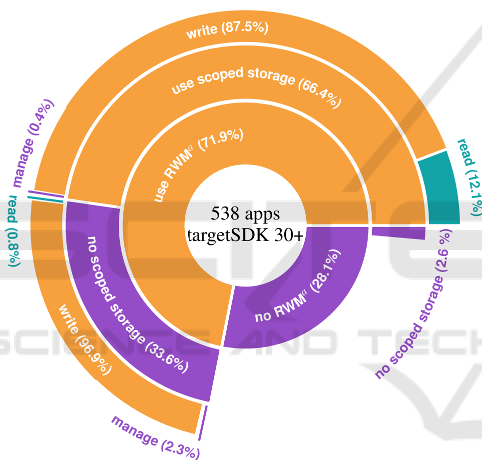
To our surprise, 97.7% of these apps contain permission to write external storage and not only read it. At first, we assumed that the app developers declared more permissions than they actually use. To check this assumption, we investigated how many apps exclusively read or write to the public external storage during the dynamic analysis. Once again, we were surprised to actually see that 6% of all apps accessing external storage were apps with only read access and respectively 94% apps with write access. This matches with the numbers in Figure 2.

[17]https://github.com/root-intruder/frida-android-filesystem-hooks



[a] RWM = apps (not) using read, write, manage external storage permission.

Figure 2: Used permissions and scoped storage among Android apps targeting SDK 29.



[a] RWM = apps (not) using read, write, manage external storage permission.

Figure 3: Used permissions and scoped storage among Android apps targeting SDK $\geq$ 30.

Looking at *targetSDK* 30 and above apps in Figure 3, there are many apps still opting out of scoped storage, even though this property is ignored in Android 11. We assume that this is either a leftover of the migration to SDK 30 or the developers want to have easy storage access on Android 10, since especially activating the manage external storage option is rightly not as easy as one click.

Interestingly, a few apps without permissions to access external storage also actively opt out, which makes no sense. Either, the developers are unaware of the effects of this configuration or it was copy and pasted from other apps.

To investigate the reasons behind the common configuration of opting out of scoped storage by apps with *targetSDK* 30, we searched for apps with such a configuration, where our analysis detected usage of scoped storage. This can mainly be detected by observing Intents started by an app. The Intents: OPEN_DOCUMENT_TREE, OPEN_DOCUMENT and CREATE indicate the use of scoped storage. Out of all 1000 analyzed apps, we only found two apps using such Intents during the dynamic analysis. Unfortunately, both apps had *targetSDK* 29. Thus, they were not usable for a behavior comparison in between target SDKs, since *targetSDK* 29 apps are still allowed to opt out of scoped storage on Android 11.

The fact that only two apps use respective scoped storage Intents sounds like the used UI interaction was not thorough enough to come to every corner of the UI. At some part, this might be true and is most likely due to the fact that many apps hide most functionality behind login masks which the used approach can only partly overcome. At the other side of the coin, there is also a positive explanation: Scoped storage is only rarely used in apps. Apps can access media locations (pictures, videos) and write documents and files to the `Download` or `Documents` folder without using scoped storage and the latter even without additional permissions. Also, the availability of app exclusive external storage since Android 9 drives developers away from using app folders on shared external storage. These two options seem to cover use-cases of many apps which is also supported by our observations during analysis. During our dynamic analysis, 11.3% of the apps with write permissions used app exclusive external storage and only 3.4% wrote files to common media storage locations (DCIM, Download, Documents...). The new storage access methods and our observations lead to the conclusion, that the scoped storage's goal of adding privacy to app and user data has been well achieved and using scoped storage is barely necessary with Android's data storage concepts.

Nevertheless, we would like to see fewer apps opting out of using scoped storage and embrace privacy with this new concept.

5.2 Access to System Files

In the next step we are analyzing the used system files. We define system files, as files located on the filesystem root (/) except for: the apps private directory (/data/user*/0/*packageName*), the app installation directory (/data/app/~~*randomHex==*/*packageName**) and the external storage (/storage/emulated/*). We carefully went through the list and like to present some peculiarities that we have discovered.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

At the top of the charts are `/proc/meminfo` giving information about used and free memory as well as webview related files. 22% of all apps use these folders. 13% of all apps use `/system/etc/security/cacerts/*` folders containing certificates used for establishing encrypted network connections, such as `https`. 11% of all apps use files related to GooglePlayServices (com.google.android.gms). This comes as no surprise, due to the large number of apps using Webviews and Google ads in Webviews nowadays.

2% of all analyzed apps are searching for super user binaries. We discovered, that the used paths even though they are partially quite uncommon for the `su` executable are frequently searched by different apps. To our surprise, most apps didn't noticeably cut functionality or refused to start. Also, rarely qemu files are searched possibly to detect if the app runs in an emulator.

Some unexpected findings come after a lot of other unspectacular common Linux files. In total 11 (1%) of all apps read the CPU min/max frequency and states. We can understand this behavior for the video chat apps in the list, to adapt for example the encoding based on available processing resources. However, the ringtone and wallpaper download app, Pinterest or a barcode scanner most certainly don't need such information.

What's also interesting, is the fact that a supermarket app iterates over all installed fonts in `/system/fonts`. This behavior is well known to be used for device fingerprinting.

An all-in-one app containing a system cleaner, anti-virus, notification cleaner, network analyzer and phone booster searches for temperature sensors at ten different locations and for all installed apps. Such a placebo app would be a definitive candidate for a deeper manual inspection. A different file cleaner app searches at more than 20 different possible locations for external storage mount points to search for files to clean. This is an excellent example for reasons to enforce scoped storage.

5.3 Shared External Storage Abuse

In this section, we want to take a closer look at the files stored at the external storage (SD-card), especially locations which are publicly accessible to other apps. App exclusive external storage[18] like (/storage/emulated/0/Android/*) is out of focus. We are looking at files written and read and if the written files are actually written to the correct folders as stated in the developer notes[19]. Files that should only be readable by the respective app should either go to /storage/emulated/0/Android/*/*packageName*/ folder [18] or the internal app exclusive storage. Also, files meant to be publicly available should go to respective folders. For example photos from a camera app should go to the DCIM folder on the external storage. Audio files should go to the Music, Ringtones or Podcasts folder. Each app is allowed to generate a sub-folder for their files in this respective folder. We consider these conventions a good practice, keeping the external storage clean and private files private.

Table 1: Apps writing to external storage root (/storage/emulated/).

package (version)	target SDK	to app excl. int.[a]	to app excl. ext.[b] [18]	sensitive info[c]	to shared media folders[d]
com.alphainventor.filemanager (2100271)	29				✘
com.sec.android.easyMover (372305100)	30		✘	✘	
org.telegram.messenger (24063)	29		✘	✘	
com.alibaba.intl.android.apps.pos… (74201)	29	✘			
mobi.charmer.fotocollage (318)	29	✘			✘
com.magicvfacemakeup.beauty.makeup…(8)	28	✘			
org.readera (1521)	29		✘		✘
com.quvideo.xiaoying (6810092)	29		✘		
com.intsig.camscanner (61501)	29		✘	✘	✘
com.palabs.fonty (30)	23		✘		
com.binance.dev (23500)	29	✘		✘	
com.playit.videoplayer (20508047)	29				✘
com.huion.inkpaint (70)	30	✘			
com.adsk.sketchbook (453889)	29	✘	✘	✘	✘
com.yy.freemusic.global (402108310)	29	✘			✘
com.wondershare.filmorago (646)	29				✘
com.videoeditorpro.android (3205061)	29	✘		✘	
com.simplescan.scanner (151)	29		✘	✘	✘
com.hitrolab.audioeditor (5075)	29		✘	✘	✘
com.xinhuamm.xinhuanews (193)	29		✘	✘	
com.invitation.maker.birthday.card (88)	29		✘		
Sum		8	12	9	10

[a,b] if the data could be moved to internal/external app exclusive storage
[c] if written data contains (potentially) sensitive information
[d] data should go to shared media folders (DCIM, Music…)

We have analyzed file write paths of all 709 apps which have the permission to write files to `/storage/emulated`. During dynamic analysis, writes to the external storage appeared on 7% of all apps with external storage write permission. Within these apps, the apps in Table 1 wrote files or folders directly to the external storage's root folder. We carefully went through the files that these apps wrote and analyzed if the files could also be moved to the app's exclusive storage on the internal storage or the external storage. Thereby, we analyzed the size and type of the written files, since the Android Developer guide states to write larger files on the app's exclusive folder on the external storage, whereas smaller (text) files can go to the internal storage. For 18 apps,

[18] exclusive only with scoped storage

[19] https://developer.android.com/training/data-storage

we could definitely tell that these files could very well life in the app exclusive folders on internal or external storage. Mostly those files were cache and log files or additionally downloaded content used inside the app. We identified 9 out of the 21 apps which had sensitive information written to public external storage folders. These files either contained content where the user has interest to keep them private to one app or also log files usable to spy on the apps internal status and configuration. In some cases one could even potentially change app behavior through a modification in these files. In the last column of Table 1 we identified 10 apps which should use common storage locations (DCIM, Music... folders) for their generated data to avoid clutter of external storage.

At a closer look at the files, we identified four apps possibly revealing private data to other apps with access to external storage. Telegram uses a folder on the public external storage to store all media (pictures/videos) files from chats. We think, that (depending on the chats) such private data should be hidden from other apps. The ”easyMover” app temporarily transfers all settings, contacts, pictures etc. to the external storage’s root to later transfer them to an OTG connected device. In our opinion, with all this collected data, there is a huge potential for unauthorized apps to gather information. To make this process secure, a safe file encryption is required, which we could not verify with the used Android-API hooks.

Last but not least, we also identified four apps in Table 1 which store media on the external storage root instead of the respective media folders. For example, pictures should be stored in the external storage’s DCIM folder. From a security or privacy perspective, this is non-critical, it only clutters the external storage structure and respective files are harder to be found.

We identified only one app writing to the external storage’s root folder, which means, that the intentions behind scoped storage work well and the external storage might become clutter free with a higher saturation of *targetSDK* 30 apps.

On the positive side, we found 86 apps (8% of all analyzed apps) using app exclusive external storage and 26 apps (2% of all analyzed apps or 4% of the apps with write permission) writing files to respective media storage locations (DCIM, Music...) with best practice.

In conclusion, we only saw interaction with external storage at 22% of the apps which would have the necessary permissions. We have two different assumptions: Firstly, apps might hold more permissions than necessary. Secondly, our mostly random app UI exploration didn’t trigger all functionality hidden deep inside the menus and many apps (60%) required logins, beyond our provided ”Login with Google” option. Thus, we think it’s better to compare the findings of well behaving apps versus the ones not complying to standards. With this perspective, about 16% of all recorded external storage accesses were non-standard compliant with some of these with privacy risks.

Table 2: Apps writing sensitive data.

package (version)	stored data type	severity	CWE
com.avira.android (7.8.1)	email	LOW	312
de.mwwebwork.benzinpreisblitz (1.25.4)	hometown	LOW	312
de.nebenan.app (1.5.144)	hometown, street	LOW	312
com.jvstudios.gpstracker (2.21)	street	LOW	312
com.tophatch.concepts (2021.09.4)	street	LOW	312
com.masterappstudio.qrcodereader (1.3.8)	phone number	LOW	312
net.easypark.android (15.27.0)	phone number	LOW	312
com.microsoft.translator (4.0.504i)	navigation address	LOW	312
com.instabridge.android (20.2.2)	navigation address	LOW	312
de.aboutyou.mobile.app (6.34.0)	navigation address	LOW	312

6 SENSITIVE INFORMATION STORAGE

In this section we evaluate which information is stored by the app. As described in Section 4.1 step 2, we create predefined contacts, call logs, calendar entries etc. which can later on be identified if used inside the app. Also, entered information into the text fields of the UI is predefined. For example, there exists a predefined address to be entered into address search fields or an email address, a phone number and many more which will be entered into identified fields described in Section 4.2. In this section, we analyze what an app does with provided data. Thus, we’d like to for example identify apps reading contacts and store them on shared external storage. Thus, the analysis program went over all Android API calls storing files and searched function arguments as well as the resulting finished written file contents. Written SQL databases have been dumped to a text file for analysis. The provided data is also searched as plain text base64 and hex encoding and as UTF-8 and ISO_8859_1 character encoding. Whenever, predefined data has been found, an incident report with a CWE is created. Different severity levels are assigned, depending on the found information and where it was stored by the app. For example, contacts written to public external storage have a high severity, whereas contacts stored at the app’s exclusive internal storage have low severity.

After running the analysis, we were glad to find no severe incidents, as shown in Table 2. All incidents were *CWE-312: Cleartext Storage of Sensitive Information* with a low severity. The respective apps stored different data, which was entered during UI interac-

tion, inside the app's exclusive internal storage. This storage location is restricted to the respective app, but it is not necessarily encrypted. In general, we don't claim that the app developers did something wrong or could do better. However, our future goal is to publish which types of sensitive data apps store, such that privacy concerned users can choose between equivalent apps. For some users, the information stored on their device is relevant, which could be revealed and used against them through a malware infection or a forensic analysis.

7 CONCLUSION & FUTURE WORK

In this work, we presented a dynamic app analysis environment, which is capable of stimulating the app (UI input & system broadcasts) and observe the app's file system API interactions. The presented results of our concept's implemented dynamic analysis of 1000 apps shows the benefit compared to manual investigations and will be used for further large scale dynamic app analysis.

The presented concept successfully managed to reveal non-standard conform behavior as well as privacy and security critical behavior in the analyzed apps. Thus, the concept is able to automatically judge apps based on their interaction with storage locations.

In summary, the evaluation showed that enforced scoped storage together with app exclusive and media specific folders on external storage really increases the privacy and security and thus is necessary. The analysis results for accessed system files revealed apps accessing resources one would not typically expect. Even though, our automated analysis doesn't yet collect enough information for a deeper insight, it very well delivers suspicious interactions as starting points for a deeper manual inspection. The written files analysis revealed that still many apps store app related files in shared folders which harm user's privacy and might even be usable to (maliciously) influence the app's behavior.

We are certain that many more interesting statistics and peculiarities could be revealed from the collected API interactions. Due to time and page limitations, we must postpone these to future work. Furthermore, our goal is to apply our observation capabilities to other Android APIs. Due to necessary logins in many apps, we see a huge potential by supporting more login capabilities besides *Login with Google*. This step will increase the UI interaction coverage in many apps, an aspect which must definitely be evaluated in future work.

ACKNOWLEDGMENTS

The project underlying this report was funded by the German Federal Ministry of Education and Research under grant number 16SV8520. The author is responsible for the content of this publication.

REFERENCES

Bierma, M., Gustafson, E., Erickson, J., Fritz, D., and Choe, Y. R. (2014). Andlantis: Large-scale android dynamic analysis. *CoRR*, abs/1410.7751.

Bläsing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., and Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 55–62.

Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. (2011). Crowdroid: Behavior-based malware detection system for android. Association for Computing Machinery.

Du, S., Zhu, P., Hua, J., Qian, Z., Zhang, Z., Chen, X., and Zhong, S. (2018). An Empirical Analysis of Hazardous Uses of Android Shared Storage. *IEEE Transactions on Dependable and Secure Computing*.

Friedman, R. and Sainz, D. (2016). File system usage in android mobile phones. Association for Computing Machinery.

Gisdakis, S., Giannetsos, T., and Papadimitratos, P. Android Privacy C(R)Ache: Reading Your External Storage and Sensors for Fun and Profit. In *Proc. of the 1st ACM Workshop on Privacy-Aware Mobile Computing*, PAMCO '16, page 1–10. Association for Computing Machinery.

Irvine, C. E. and Levitt, K. (2007). Trusted Hardware: Can It Be Trustworthy? In *In Proceedings of the 44th Annual Design Automation Conference, DAC '07, ACM*.

Liu, X., Diao, W., Zhou, Z., Li, Z., and Zhang, K. (2014). An Empirical Study on Android for Saving Non-shared Data on Public Storage. *CoRR*.

Mayrhofer, R., Stoep, J. V., Brubaker, C., and Kralevich, N. (2019). The Android Platform Security Model. *CoRR*, abs/1904.05572.

Schmeelk, S. and Tao, L. (2020). Mobile Software Assurance Informed through Knowledge Graph Construction: The OWASP Threat of Insecure Data Storage. *Journal of Computer Science Research*.

Yang, Z. and Yang, M. (2012). Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*, pages 101–104.

SCIENCE AND TECHNOLOGY PUBLICATIONS