Examining the Privacy Vulnerability Level of Android Applications

Georgia M. Kapitsaki[a] and Modestos Ioannou
Department of Computer Science, University of Cyprus, Aglantzia, Cyprus

Keywords: Mobile Applications, Privacy, Privacy Vulnerabilities, Android Permissions.

Abstract: Mobile applications are often granted access to various data available on the mobile device. Android applications provide the notion of permissions to let the developers define the data their applications require to function properly. However, through accessing these data, applications may gain direct or indirect access to sensitive user data. In this paper, we address the detection of privacy vulnerabilities in mobile applications in Android via an analysis that is based mainly on the use of Android permissions. Different aspects of the application are analyzed in order to draw conclusions offering an aggregated view of permission analysis in the form of a penalty score, a feature that is missing in previous approaches that analyze permission use in Android. Our work is supported by a web application prototype of *App Privacy Analyzer* that allows users to upload an application and view the respective analysis results comparing them with other applications uploaded in previous uses of the system. This approach can be useful for security and privacy analysts and developers that wish to examine the privacy vulnerability level of their Android applications, but also for end users with technical expertise. We have used the tool for the analysis of 800 Android applications and are discussing the results the observed permission use.

1 INTRODUCTION

The Android OS uses the notion of permissions in order to let developers define the data their application requires to function properly. Android permissions have been investigated in many previous works in order to recommend appropriate applications that consider user's security requirements (Zhu et al., 2014) or analyze code use in terms of permissions (Wang et al., 2015a).

Although a plethora of previous works along with official documents on using permissions exist, permission misuse is still often observed in Android applications. Tools that assist developers, and security and privacy analysts in evaluating the level of privacy protection in the Android applications they are developing or using is a required step towards understanding the implications of privacy protection in mobile applications and can assist in preventing relevant misuse. The introduction of the EU General Data Protection Regulation (GDPR) has rendered the need of introducing more elaborate analysis techniques even more important, as end users need to be well informed on the collection of their personal data (Tăbuşcă et al., 2018).

[a] https://orcid.org/0000-0003-3742-7123

Having as motivation the above, in this work, we introduce a process for the analysis of privacy vulnerabilities in Android applications with a focus on permissions. The process is based on the analysis of the Application Package Kit (APK) file of the application. Although similar tools are available (e.g. Exodus Privacy[1]), we integrate different mechanisms in one approach that also assigns an aggregated penalty score in each application, allowing a comparative view in respect to other relevant applications that are available in the system. Based on this approach, we have analyzed 800 Android applications. This way we are drawing some conclusions on the current permission use in Android applications that are discussed in this work. The resulting tool, *App Privacy Analyzer*, is available as a prototype web application and also offers statistics on privacy vulnerabilities considering the applications that have been analyzed so far.

The contribution of this work is threefold:

1. We introduce an aggregated score calculation for privacy vulnerabilities based on a combination of different approaches of code analysis.
2. We offer an online web application that is open for use, providing access to its source code.

[1] https://reports.exodus-privacy.eu.org/trackers/

Kapitsaki, G. and Ioannou, M.
Examining the Privacy Vulnerability Level of Android Applications.
DOI: 10.5220/0007955100340045
In *Proceedings of the 15th International Conference on Web Information Systems and Technologies (WEBIST 2019)*, pages 34-45
ISBN: 978-989-758-386-5

3. We perform an analysis on existing applications to see where they stand in respect to this scoring maintaining a repository with these data.

We argue that our approach offers a useful mechanism for security and privacy analysts and developers to assess the vulnerability level of their applications, whereas it can also be used by end users that have some technical knowledge in order to understand the privacy level of applications they are using or intend to use.

The rest of the paper is structured as follows. Section 2 provides background information by describing the use of Android permissions. The introduced approach is presented in detail in section 3. Section 4 gives some implementation details, and Section 5 introduces the results of the application analysis performed by *App Privacy Analyzer* discussing the main conclusions drawn. Threats to validity are briefly discussed in Section 6. Section 7 is dedicated to the presentation of previous works in the area and finally, Section 8 concludes the paper outlining directions of future work.

2 ANDROID PERMISSIONS

Android has moved from ask-on-install (AOI) model to ask-on-first-use (AOFU) model regarding permissions providing more freedom to users. Applications of Android version 5.1.1 or lower ask the user, if she accepts all permissions associated with an application in order to be able to use it. Otherwise, the installation of the application cannot proceed. More recent versions (since Android version 6.0) inform the user about the need to access a specific resource after application installation and upon execution. If the user accepts, the preference is saved for subsequent uses by the same application. Otherwise, the application will ask the same question the next time the resource needs to be accessed. If the user however, selects the *Never ask again* option, the question will not appear again, but the user will not be able to use the application functionality related with the permission. Android uses the following permission categories:

- *Normal*: refers to permissions requesting access to resources that do not affect the system or other applications. Since Android 6.0, access to normal permissions is provided by default. The user of the application is not asked about them, but she can view them before installing the application.
- *Signature*: these permissions are provided to the application on its installation, only if the certificate of the application requesting the permission matches the certificate of the application that defines this permission.
- *SignatureOrSystem*: similar to signature permissions, but used by vendors that distribute devices with applications embedded in the Android operating system.
- *Dangerous*: these permissions are considered unsafe, since the resources they access may affect user privacy or device data, and may even affect other installed applications. GPS and camera access belong to this category.
- *Custom*: these permissions are defined by the application developer and refer to application resources and not system resources.

Custom permissions are not used further in our work. Since a large number of permissions exist, Android places each permission within a group inside the main permission category (e.g. WRITE_CALENDAR and READ_CALENDAR are placed in the *CALENDAR* group). Requests for dangerous permissions are based on the permission category. For instance, if the application asks permission for READ_CALENDAR and it then requires also WRITE_CALENDAR, permission will be granted without asking the user again, since the two belong to the same group. This is not the case however, for the other permission categories.

Android permissions have disadvantages that have been studied in the past. For instance, some permission categories are too vague to allow users to make meaningful decisions (Felt et al., 2011b), and many applications declare permissions they are not actually using in their code (Felt et al., 2011a). Moreover, permissions along with sandboxing are not adequate measures to prevent attacks in Android applications. Privilege escalation attacks are possible due to transitive permission usage that permits non-privileged applications to invoke higher-privileged applications that do not have sufficient protection mechanisms in their interfaces (Davi et al., 2010). A previous work has identified whether applications are over privileged or follow the least privilege principle based on a combination of runtime information and static analysis (Geneiatakis et al., 2015). Permission usage relates though to the application functionality and these aspects, i.e. sensitive information transmission and application functions, should be viewed together, as addressed in the DroidJust tool (Chen and Zhu, 2015).

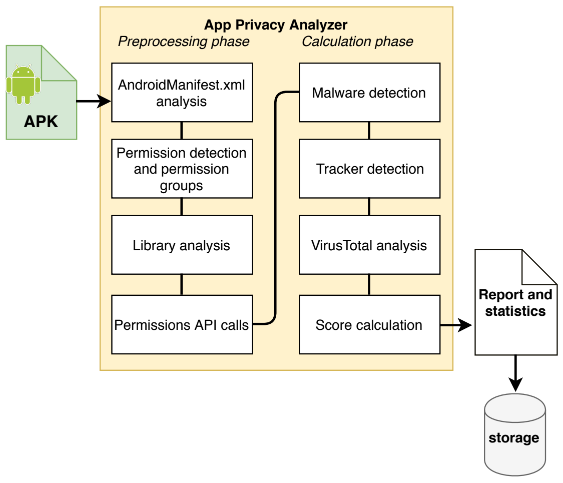


Figure 1: Steps of Android App Privacy Analyzer.

3 APPLICATION ANALYSIS PROCESS

3.1 Preprocessing Phase

The steps of the analysis performed by *App Privacy Analyzer* in order to infer the privacy vulnerability level of Android applications are depicted in Figure 1. Initially, the APK is decompiled and the AndroidManifest.xml file is detected. The following attributes are extracted with some being privacy-relevant and other being generic and used only for information purposes:

- *label*: the application name.
- *package*: corresponds to an application unique identifier, referred to as Google Play ID. It usually corresponds to the project's directory structure (e.g. com.example.android.app.diary).
- *versionName* and *versionCode*: version of the application (name and number respectively).
- *minSdkVersion* and *targetSdkVersion*: version of the lowest possible API and the targeted API respectively.
- *allowBackup*: determines whether application data can be backed up and restored (true/false value). Although this functionality can be useful, it also brings some security implications for application data.
- *debuggable*: indicates whether the application can be debugged during execution on user's device (true/false value). It is also considered a vulnerability, as it can expose data, and should be examined during penetration testing.

During the manifest analysis, the hash of the application based on SHA-256 (that is still considered secure) is also calculated (Gilbert and Handschuh, 2003). This is required in order to be able to uniquely identify applications, even when application repackaging has been performed, and avoid performing the analysis on an application that has already been processed.

In the next step (*Permission detection and permission groups*) we focus on permissions the application uses. Declared permissions can be extracted from AndroidManifest.xml, whereas used permissions are detected based on source code analysis. *App Privacy Analyzer* matches the declared permissions of the manifest file with whether they are actually used in the application source code. The following permission groups are formed as a result:

- Declared: permissions that are declared in AndroidManifest.xml;
- Declared and used: permissions that are declared in the manifest and are actually used in the application;
- Not declared but used: permissions identified in the source code but not in the manifest;
- Declared but not used: permissions that are declared in the manifest file but are not present in the source code.

In the subsequent step of *Library analysis*, *App Privacy Analyzer* extracts from the source code third party libraries used by the application. Based on this, we also find the permissions that are declared in the third party libraries used by the application, in order to detect permissions used via these libraries indirectly, as they are also important for the user. The following data are collected for each detected library: library name, match ratio, package name, library permissions list, popularity, library type (e.g. ads) and library website.

Used permissions (from the *declared and used* and the *not declared but used* groups) are matched with the method they are used in, pointing to the call that is related with a permission (*Permissions API calls* step). This is performed via static code analysis. We then detect the specific places at the source code, where this permission call is made, in order to be able to indicate this information to the user (path to source code file and method name).

3.2 Calculation Phase

After the preprocessing phase, we calculate the vulnerability level of Android applications using a combination of techniques. Initially, a machine learn-

Table 1: Comparison of classification algorithms for malware detection.

Algorithm	Accuracy (%)
Naive Bayes	90.201
Bagging	91.709
k-Nearest Neighbor (kNN)	93.216
Stochastic Gradient Descent (SGD)	92.965
Locally Weighted Learning (LWL)	91.709

ing approach to malware detection that relies on permissions is adopted (Lopez and Cadavid, 2016). A dataset[2] of 398 Android applications was used to train a classifier that characterizes an application as malware or non-malware. We used 10-fold cross validation to compare different classification algorithms and k-Nearest Neighbor was adopted, as it demonstrated the best accuracy, although it is only slightly better than Stochastic Gradient Descent (Table 1). We are using this technique only partially in order to determine whether an application can be regarded as malware, as we observed that permission-based only analysis is not sufficient to characterize an application as malware.

Tracker use in the application is also examined via tracker libraries (*Tracker detection* step). Identified libraries are matched against the tracker list provided by *Exodus Privacy*. Tracker detection is indicated to the user but it is not further considered in the calculation of the vulnerability level. We finally perform an external additional malware analysis via *VirusTotal*[3].

Based on the analysis of the previous steps, we provide a consolidated view of privacy vulnerabilities of the application. We calculate an aggregated score, that allows the comparison of applications. We use the following process for the score calculation. A penalty score is assigned to the application for each characteristic that is considered relevant to a privacy vulnerability. The scoring used for this purpose is shown in Table 2 and has been created from empirical observations and by experimenting with different scoring, but can be adapted in the future. For permissions, the penalty score indicated is applied for each permission independently (e.g. if an application has three declared and used dangerous permissions, the respective penalty score is 30). We also add a low penalty score to applications that declare permissions they do not use. Applications are penalized more, when they are using permissions they do not declare with a different penalty score for each permission category (Dangerous, Signature/SignatureOrSystem, Normal).

[2]https://www.kaggle.com/xwolf12/datasetandroidpermissions

[3]https://www.virustotal.com

We explain below the rationale of choosing each property that should be penalized:

- *Dangerous, Signature/SystemOrSignature Permission Declared but not Used*: Although some applications may keep these permissions as declared for future needs, this leads to misinforming the user and should be avoided. The penalty score applied is very low.
- *Dangerous, Signature/SystemOrSignature Permission Declared and Used*: many applications require permissions of these categories for their functionality. However, their use should be kept to the minimum level, and for this reason the applications are penalized.
- *Dangerous, Signature/SystemOrSignature, Normal Permission not Declared but Used*: in the case of dangerous permissions, this is a severe vulnerability case, as the user is unaware of the use of the permission. A high penalty score is assigned.
- *APK is Debuggable*: debuggable Android applications pose a severe risk, as it allows sensitive device data to be extracted. As this may even contain data for banking transactions, the penalty score applied is high.
- *APK Allows Backup*: application backup data can be easily modified[4], and for this reason a penalty score is applied. However, the score is lower than in the case of debuggable property, as it is less critical.
- *APK Classified as Dangerous*: it is based on the machine learning analysis. A high penalty score is provided, as the classification is based on the existence of permissions in applications. However, less emphasis is put in this analysis in comparison to VirusTotal, as we encountered many false positives (e.g. Facebook, Messenger, Instagram applications were characterized as malware).
- *APK Regarded Malware by VirusTotal*: VirusTotal is currently considered the most popular website for multi-antivirus scanning, aggregating antivirus products and online scan engines. The penalty score is high due to the significance of characterizing an application as malware in this tool and is assigned, when at least 30% of the VirusTotal engines detect the application as malware. However, if an application is characterized as malware by VirusTotal, but does not contain a large number of permissions (e.g. dangerous permissions), it is not indicated as a high risk application (based on the scores described later in the section).

[4]https://securitygrind.com/exploiting-android-backup/

The maximum penalty score is 100. Although the total of some applications may surpass this score (e.g. when there are seven not declared but used dangerous permissions), this was used as an upper limit so that the application is not penalized more.

Based on the total penalty score of an application, we place each application in one of the following risk levels. The penalty score is not adequate to characterize an application as risky, but we argue that it provides a measure for security and privacy analysts to understand where their applications stand and decide whether additional examination is required, whereas it can assist end users providing them comparative information on an application they are using or are considering to use:

- *no risk* (score between 0 and 24) corresponding to applications that have at most two dangerous permissions or a larger number of permissions of other categories;
- *low risk* corresponding to a low number of permissions (score 25-49);
- *average risk* (score 50-79) including applications that are usually characterized as malware and use an average number of permissions;
- *high risk* for applications that require further investigation (score 80-100).

4 IMPLEMENTATION

The *App Privacy Analyzer* tool has been implemented as a web application using the Spring[5] and Vaadin[6] frameworks, employing Java and web technologies, whereas Python has been used for application analysis purposes, since many of the existing tools we have relied on are implemented in Python. In order to extract information from AndroidManifest.xml file, we have used the *AndroGuard*[7] reverse engineering tool. For the quantitative permission analysis from the source code we rely on *PermissionChecker* that forms part of *RiskInDroid* (Risk Index for Android) tool and may be used for research purposes (Merlo and Georgiu, 2017). RiskInDroid uses machine learning techniques to calculate a numeric risk value for Android applications between 0 and 100. For finding third party libraries used in the applications, including tracker libraries, we use *LibRadar*[8] (Ma et al., 2016). For the permission-related methods, we are employing *Pscout* (Permission Scout) that performs static analysis in order to extract a permission specification using light-weight call-graph analysis to Android API (Au et al., 2012). Pscout lists the permissions that every Android API call requires and provides these permissions used in a CSV (Comma-Separated Values) file. In our work, it is used to detect permission method calls, in order to identify where a permission call is performed. We have employed the permission datasets provided by Pscout for Android APIs 9, 10, 14-19 and 21-22, in order to cover method changes between these versions, but the appropriate version is used for each analyzed application.

Application analysis results are stored in a dedicated database, so that existing APKs can be directly retrieved without analyzing the same application again. Applications are uniquely identified using the hash created in the preprocessing phase. A prototype implementation of *App Privacy Analyzer* is available and its source code is available in the respective repository on GitHub[9]. For the machine learning algorithms, we used the corresponding implementations in WEKA (Waikato Environment for Knowledge Analysis) (Hall et al., 2009).

Portions of the screenshots of the online application are shown in Figure 2 and Figure 3 with the main data of an example application and its aggregated score, and the detected permissions in used libraries and permission calls, respectively. Data collected from the application for information purposes, e.g. *label*, *minSdkVersion* are also shown to the user. Regarding VirusTotal, the users can visit the respective analysis in the link provided (visible in Figure 2), whereas in the example provided there is the indication that 43 engines were detected by VirusTotal in this application.

5 RESULTS AND DISCUSSION

There is no ground truth for Android permission use in applications and for this reason it is not feasible to compare the results of our approach to a baseline, in terms of penalty scores used. As aforementioned, we have experimented with different values for the penalty scores before establishing the current form. Regarding the independent steps, we are using external tools to guide our analysis and we are relying on their accuracy. However, we do provide a comparative view of applications analyzed by *App Privacy Analyzer*, and we compare our results with VirusTotal. We collected a number of Android applications

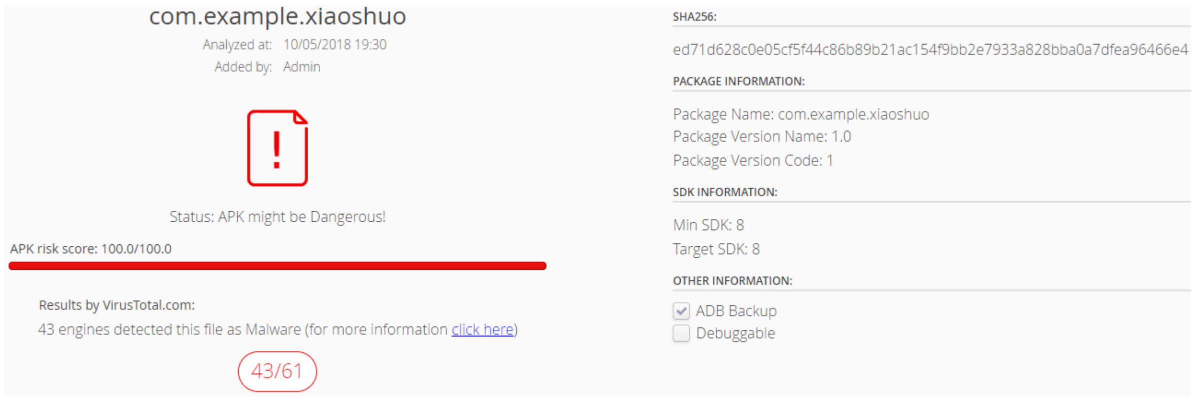
[5]https://spring.io/

[6]https://vaadin.com/

[7]https://github.com/androguard/androguard

[8]https://github.com/pkumza/LibRadar

[9]https://github.com/CS-UCY-SEIT-lab/AndroidAppPrivacyAnalyzer



com.example.xiaoshuo

Analyzed at: 10/05/2018 19:30

Added by: Admin

Status: APK might be Dangerous!

APK risk score: 100.0/100.0

Results by VirusTotal.com:

43 engines detected this file as Malware (for more information click here)

43/61

SHA256:

ed71d628c0e05cf5f44c86b89b21ac154f9bb2e7933a828bba0a7dfea96466e4

PACKAGE INFORMATION:

Package Name: com.example.xiaoshuo

Package Version Name: 1.0

Package Version Code: 1

SDK INFORMATION:

Min SDK: 8

Target SDK: 8

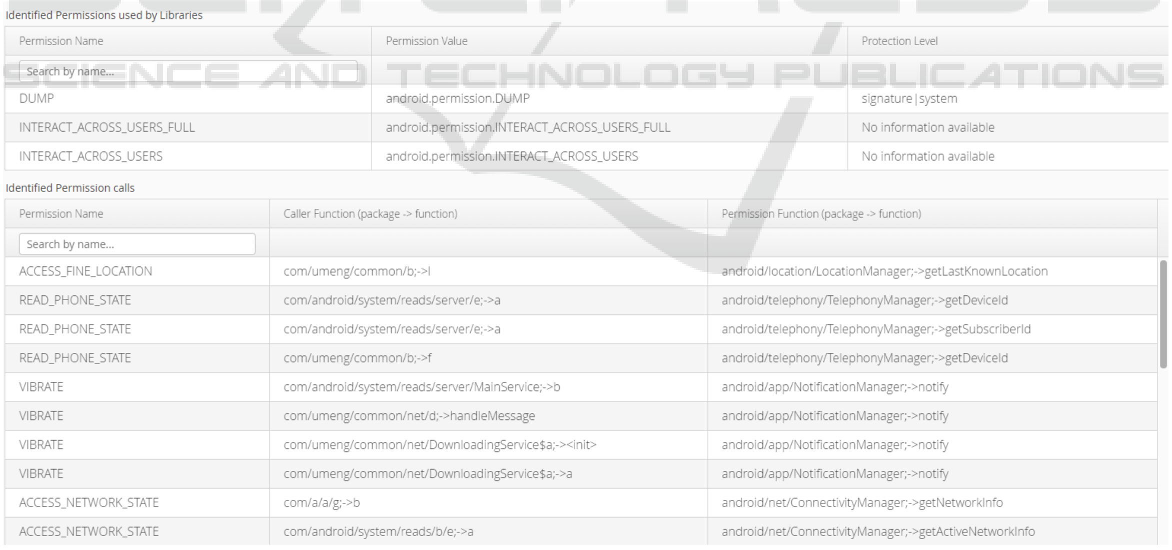
OTHER INFORMATION:

- [✓] ADB Backup
- [ ] Debuggable

Figure 2: Basic data of analyzed application.

Table 2: Penalty scores for application properties.

Property	Score
Dangerous permission declared but not used	3
Signature/SystemOrSignature declared but not used	1.5
Dangerous permission declared and used	10
Signature/SystemOrSignature permission declared and used	5
Dangerous permission not declared but used	15
Signature/SystemOrSignature permission not declared but used	3.75
Normal permission not declared but used	7.5
APK is debuggable	8
APK allows backup	3
APK classified as dangerous	15
APK regarded malware by VirusTotal	20



Identified Permissions used by Libraries

Permission Name	Permission Value	Protection Level
Search by name...		
DUMP	android.permission.DUMP	signature | system
INTERACT_ACROSS_USERS_FULL	android.permission.INTERACT_ACROSS_USERS_FULL	No information available
INTERACT_ACROSS_USERS	android.permission.INTERACT_ACROSS_USERS	No information available

Identified Permission calls

Permission Name	Caller Function (package -> function)	Permission Function (package -> function)
Search by name...		
ACCESS_FINE_LOCATION	com/umeng/common/b;->l	android/location/LocationManager;->getLastKnownLocation
READ_PHONE_STATE	com/android/system/reads/server/e;->a	android/telephony/TelephonyManager;->getDeviceId
READ_PHONE_STATE	com/android/system/reads/server/e;->a	android/telephony/TelephonyManager;->getSubscriberId
READ_PHONE_STATE	com/umeng/common/b;->f	android/telephony/TelephonyManager;->getDeviceId
VIBRATE	com/android/system/reads/server/MainService;->b	android/app/NotificationManager;->notify
VIBRATE	com/umeng/common/net/d;->handleMessage	android/app/NotificationManager;->notify
VIBRATE	com/umeng/common/net/DownloadingService$a;-><init>	android/app/NotificationManager;->notify
VIBRATE	com/umeng/common/net/DownloadingService$a;->a	android/app/NotificationManager;->notify
ACCESS_NETWORK_STATE	com/a/a/g;->b	android/net/ConnectivityManager;->getNetworkInfo
ACCESS_NETWORK_STATE	com/android/system/reads/b/e;->a	android/net/ConnectivityManager;->getActiveNetworkInfo

Figure 3: Permissions calls locations in analyzed application.

from the F-Droid[10] catalogue of Android applications and from APKPure[11] with a total of 800 Android applications, since both F-Droid and APKPure provide access to the APK file. These applications differ from the dataset of malware[12] applications that we have used in the malware detection step. The applications belong to different categories, range from minSdk version 1 to 24 and targetSdk 1 to 28 with most belonging to minSdk version 16 (19.1%) and

[10]https://f-droid.org

[11]https://apkpure.com/

[12]https://github.com/ashishb/android-malware

targetSdk version 25 (19.9%). The dataset is available on GitHub[9].

Table 3 shows the 10 dangerous permissions most usually employed in the analyzed Android applications. The last column shows the positions of the permission in the list of the 40 most risky permissions according to (Wang et al., 2014). The three numbers correspond accordingly to the position of the permission in the list of most risky permissions considering three different machine learning techniques for calculation: Correlation Coefficient (CorrCoef), ranking with mutual information and t-test. Permissions with no numbering do not appear in the lists. This may be attributed to the fact that Android permission categories have been changed, since this previous publication. Table 4 shows the top 10 found permissions regardless of the category. The most usual dangerous permission, i.e. WRITE_EXTERNAL_STORAGE, is in the tenth position. Normal permissions that were considered dangerous in previous Android versions appear in the list, e.g. INTERNET.

Regarding applications that use permissions they do not declare, this was observed in 9.6% of cases. Most permissions are declared but not used (36.67%) and 18.3% are declared and used, which is a relatively low percentage. A large number of permissions comes from libraries used (35.44%) indicating that caution is needed when using third party libraries, as it may not be feasible to control the permissions they are using. Among the applications, allowBackup is set in 84.75%, posing a threat for most applications, but only 1.5% are indicated as debuggable indicating that this is a vulnerability identified by most developers. The average penalty score is 44.69 with 39.625% applications belonging to the no risk range, 19.875% to the low risk and 3.75% to the average risk range.

Our tool characterized 187 of applications as vulnerable. This corresponds to 23.375% of applications being assigned to the high risk level. Among them, 14.5% reach a penalty score of 95 and above. We manually analyzed some of the applications characterized as risky. As many applications have a large number of functionalities, the number of permissions they use increases. For example, many applications request storage access, camera access (mostly for QR scan purposes) and location access. We are not considering the number of functionalities the application offers in our analysis, as we rely on the APK files, but future research work could perform further source code analysis or combine the information for the application description in Google Play to draw conclusions about the complexity of functionality offered by the application and its connection with permission use. Via manual analysis we also detected applications that requested permissions, such as access to make calls or send SMS, with no relevant functionality to explain these requests and therefore, their characterization as vulnerable is justified. Their results can be considered close to the results of the *AVC UnDroid*[13] tool, where 31.6% of applications that have been submitted for analysis are characterized as malicious.

The applications that are characterized as most vulnerable (aggregated penalty score above 95) are usually linked with the following dangerous permissions:

- ACCESS_COARSE_LOCATION
- ACCESS_FINE_LOCATION
- WRITE_EXTERNAL_STORAGE
- READ_PHONE_STATE
- and READ_EXTERNAL_STORAGE

Different trackers have also been found in these applications, e.g. Google Analytics, Google Ads, Amazon Advertisement. Note that the trackers list we are using consists of 152 trackers. The analysis results provide a good indication for vulnerabilities, but further examination is required in order to draw more concrete conclusions for each application.

We also investigated whether the number of dangerous permissions found in an application is related to whether the application uses permissions that it does not declare. We ran a two independent samples t-test. The difference was statistically significant. We observed that applications with at least one not declared but used permission (regardless of the permission group, i.e. declared, declared but not used, not declared but used, permission coming from a library) have a larger number of dangerous permissions ($t = -6.708, p = 0.000$). The descriptive statistics are displayed in Table 5. This result justifies the use of these parameters in the penalty score calculation.

Finally, we compared our results with VirusTotal. We compared the penalty score with the number of engines of VirusTotal that identified the application as malware. We found that 47.1% of applications identified as malware by at least one VirusTotal engine were placed in the high risk category (in 121 applications at least 1 engine detected the application as malware). There were many cases with a high penalty score that were not identified by VirusTotal: 72.5% of applications in the high risk category were not detected by any VirusTotal engine. This difference is expected, as the aim of VirusTotal is malware detection. May applications are privacy risky, but do not contain viruses.

[13]http://undroid.av-comparatives.info

Table 3: Frequency of top dangerous permissions.

Permission	**Frequency**	**Position in** (Wang et al., 2014)
WRITE_EXTERNAL_STORAGE	63	19, 13, 19
ACCESS_COARSE_LOCATION	36.75	-
READ_PHONE_STATE	36.125	7, 4, 3
ACCESS_FINE_LOCATION	32.875	39, 38, 39
READ_EXTERNAL_STORAGE	31.5	8, 5, 5
CAMERA	17.75	31, 22, 31
GET_ACCOUNTS	16.875	-
READ_CONTACTS	11.875	17, 17, 17
RECORD_AUDIO	9.625	-
RECEIVE_SMS	9.125	2, 2, 10

Table 4: Frequency of top permissions.

Permission	**Frequency (%)**
INTERNET	88.5
DUMP	81.375
INTERACT_ACROSS_USERS_FULL	81
INTERACT_ACROSS_USERS	81
WAKE_LOCK	78.5
VIBRATE	68.875
BACKUP	65.875
ACCESS_NETWORK_STATE	63.625
MANAGE_APP_TOKENS	63.375
WRITE_EXTERNAL_STORAGE	63

Table 5: Group statistics for t-test comparing not declared but used with dangerous permissions.

Permission	**N**	**Mean number of dangerous permissions**	**Std. deviation**
$\geq$1 not declared but used	522	6.77	5.903
none not declared but used	277	3.53	7.479

6 THREATS TO VALIDITY

In relevance to *external validity*, referring to the extent we can generalize our findings, we present results based on the analyzed applications that form a relatively small dataset, but we do not expect large deviations for a larger number of applications (Yin, 2013). A further limitation is that we are considering the current handling of permissions by Android. If this changes in the future, our approach will have to be adapted.

Regarding *internal validity*, we rely in some steps of our approach on existing source code analysis tools and assume that the results they provide are accurate. For instance, we use LibRadar to detect tracker libraries, but if they are not adequately detected, our approach is also affected.

Construct validity, referring to whether a test measures the intended construct, may be affected by the values chosen for the penalty scores for the aggregated score calculation. In order to minimize this effect, we have adapted the penalty scores based on observations from our initial experiments and empirical analysis in previous works. We do not expect to be affected by *conclusions validity*, as the analysis performed employs simple statistics.

7 RELATED WORK

Many previous works have focused on malware of Android applications, but are not further analyzed as they have a different target, although some of them make use of permissions (Lindorfer et al., 2015; Yang et al., 2015; Idrees et al., 2017). Regarding permission analysis, one work examines the risks associated with Android permissions, both on the level of individual permissions and on the level of a group of collaborative permissions (Wang et al., 2014). This way permissions are ranked with respect to their risk with READ_SMS and RECEIVE_SMS obtaining the highest risk scores.

Suggestions as to how a permission system should be designed are made in (Rosen et al., 2013). Disadvantages found are that permissions are not fine-grained enough, that there should be a differentiation between actions performed by users and actions performed in the background, and that the differences between Android application code and third-party code should be examined. The authors also create a knowledge base of privacy-relevant API calls and use it to produce application behavior profiles. Although it would have been useful to utilize this knowledge base also in the framework of our work, it seems that it is no longer available. Improvements on how permissions are handled in terms of user interactions have been investigated in a usability approach (Micinski et al., 2017). The authors have measured how user interactions and sensitive resource use are related in existing applications and have then performed an online survey to see how users view interactions with mobile applications in terms of connecting user actions in an application with access to sensitive resources

In order to assist permission management from the user's perspective, in (Wijesekera et al., 2017) the authors propose an approach that uses context to adapt user decisions for access to resources. Initially, the authors use a field study to analyze the contextual aspects of user privacy decisions to regulate access to sensitive resources. At a later stage, privacy decisions are made by the proposed system without user's intervention. Another approach finds the balance between using ads in mobile applications and free use (Leontiadis et al., 2012). On the one hand, the user is interested in maintaining her privacy and, on the other hand, developers would like to maximize their advertisement revenue through user profiling. The authors propose a market-aware privacy protection framework based on a feedback loop that adjusts privacy protection level on mobile phones in response to advertisement generated revenue.

Another group of previous works focus either on tracking information flow or providing permission controls at finer levels of granularity. In (Zimmeck et al., 2017), the authors introduce a system that checks data practices of Android applications against privacy requirements using their privacy policies and the legislation. They have found that in a large number of applications there are potential inconsistencies between what the applications states to do and what the code of the application does.

Works that are closer to our approach perform static analysis that can be performed on mobile applications for various purposes, as described in a previous survey (Li et al., 2017). Information flows that may violate privacy relying on the application's Dalvik bytecode are examined in the theoretical approach of a security type system in (Mann and Starostin, 2012). The type system uses a privacy policy that defines method and field signatures as input. Methods in the application are then examined to see whether they respect the specified signatures. Dynamic analysis is used in (Wang et al., 2015a) in conjunction with static analysis to determine the precise set of permissions that an application needs to run correctly in relevance to the declared permissions. In this approach, permission usage information is initially extracted from API invocations, and then a dynamic testing technique monitors the runtime behavior of the application.

Other approaches focus on inferring the permissions an API needs. The ApMiner approach is able to learn from the usage of APIs and permissions in other applications published in a marketplace (Karim et al., 2016). Using this information, it can help the development of new applications by recommending the permissions to be added in the application taking into account the APIs it uses. This work is further enhanced in (Bao et al., 2016), where the authors introduce an approach based on collaborative filtering. This approach is based on the assumption that applications that share similar features, based on the APIs they use, share also similar permissions.

Going one step further, the work in (Wang et al., 2015b) employs text mining techniques to infer the purpose of permission use. It is based on decompiled code to extract application-specific (e.g. calls to APIs, use of Intent, Content Provider) and text-based (e.g. words extracted from package names, class names and other sources) features, and has been used in the case of contacts and location permissions, i.e. READ_CONTACT_LIST and ACCESS_FINE_LOCATION.

VetDroid is based on dynamic analysis for identifying sensitive behaviors of Android applications (Zhang et al., 2014). It observes how applications use permissions to access system resources and how this information is then used in the application. It has been used to facilitate malware analysis. RiskMon shares some similarities with our approach, since it assigns a risk score to each application (Jing et al., 2015). This is performed on every access attempt on sensitive information and at the end a cumulative score is calculated for each application. Based on this, it performs automated permission revocation if needed. An extension of the Android OS is provided as a proof-of-concept implementation of RiskMon.

Some online tools offer users the possibility to analyze Android applications. Exodus Privacy[1] detects ads, tracking, analytics, whereas it also reports suspicious network traffic. Imported trackers are detected based on a tracker list. Concerning permissions, Exodus analyzes AndroidManifest.xml and reports indicated permissions. DNS, UDP, HTTP and HTTPS traffic is supported as detected with dynamic application analysis using a simulated Android device. AVC UnDroid[13] analyzes the APK of uploaded applications and uses different tools: Buster Sandbox Analyzer[14] (evaluates malware suspicions based on process behavior), ssdeep[15] (computes context triggered piecewise hashes or fuzzy hashes and detects potentially risky applications) (Kornblum, 2006), APKTool[16] (for performing reverse engineering on the APK) and security engines. There is an image indica-

[14]http://bsa.isoftware.nl

[15]https://ssdeep-project.github.io/ssdeep

[16]https://ibotpeaches.github.io/Apktool

Table 6: Online privacy analysis tools comparison.

Property	**Exodus Privacy**	**AVC UnDroid**	**App Privacy Analyzer**
tracker support	✓	✗	✓
traffic analysis support	✓	✗	✗
declared permissions detection	✓	✓	✓
used permissions detection	✗	✓	✓
permission level indication	✓	✓	✓
permission call location indication	✗	✗	✓
actions/intents indication	✗	✓	via VirusTotal
malware detection	✗	partial	✓
adware SDK detection	✗	✓	✗
statistics	most frequent trackers	submissions per country clean vs. malicious APKs analyzed APKs number analyzed APKs release dates ad-supported APKs malware families adware SDKs	most frequent permissions most frequent dangerous permissions analyzed APKs number declared vs. used permissions uses per permission declared vs. used per permission average penalty score
tools used	tracker list	Buster Sandbox Analyzer, ssdeep, APKTool, security engines	AndroGuard, PermissionChecker, LibRadar, Pscout, VirusTotal
input	Google Play URL	APK file	APK file

tion of how dangerous an application is, but it is not indicated how this is calculated. Malware detection information does not appear in the analysis report.

In regard to previous works, we employ different mechanisms in order to offer a consolidated view of the application. We introduce an aggregated vulnerability score based on the independent examinations providing a summary of the analysis report. To the best of our knowledge, this is the first work that introduces this construct that allows users to compare the application with other analyzed applications. Table 6 summarizes the comparison of our approach with Exodus Privacy and AVC UnDroid that are the closest existing approaches. In comparison to these tools, we provide more statistics concerning permission use, but do not offer adware information, since it is not directly related to privacy, unless adware is classified as spyware. The Exodus tool detects trackers based on the detection of code signatures in the application (that refer to tracker packages) and their matching to predefined tracker lists. Instead, we use LibRadar to detect tracker libraries, but utilize also the tracker list of Exodus. We cover the main capabilities offered by AVC UnDroid, but use AndroGuard instead of APKTool for reverse engineering purposes. Permission call location indication is also a feature that is missing from existing approaches, but it is useful for easily identifying the permission call inside the application source code, especially for large applications.

8 CONCLUSIONS

In this paper, we have presented our approach on analyzing Android applications for privacy vulnerabilities. The approach employs different steps, wherein an aggregated score based on application characteristics as detected in these steps is calculated for each application. We described the results of using the tool on 800 Android applications. The approach can be a useful tool for privacy and security analysts and developers in identifying issues that need further examination and for users that can get an overview of issues in applications they are using or intend to use.

As future work, we intend to study how the use of common libraries affects permission analysis for a given application, as a previous work has identified that this may lead to inaccurate results, if analysis mechanisms do not consider library code that can be considered noise for the application (Li et al., 2016). We will examine the consideration of more characteristics in the calculation of the aggregated penalty score, considering also the compliance of the Android application with EU GDPR. We also intend to put the user in the loop of privacy handling providing more user-friendly ways for presenting to the users information on the privacy level of applications focusing on users with limited or non technical expertise, and users of younger ages. Finally, we intend to add more steps in our analysis. Dynamic code analysis tools can be utilized for this purpose, such as the Mobile Security Framework (MobSF[17]), Dynamic Executable Code Analysis Framework (DECAF) (Henderson et al., 2014) and DroidBox[18].

[17]https://github.com/MobSF/Mobile-Security-Framework-MobSF

[18]https://github.com/pjlantz/droidbox

ACKNOWLEDGMENT

This work was partially funded by the European Community CEF-TC-2018-1 - Safer Internet (grant agreement number INEA/CEF/ICT/A2018/1614254) CYberSafety II (http://www.cybersafety.cy/) project.

REFERENCES

Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. (2012). Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM.

Bao, L., Lo, D., Xia, X., and Li, S. (2016). What permissions should this android app request? In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 36–41.

Chen, X. and Zhu, S. (2015). Droidjust: Automated functionality-aware privacy leakage analysis for android applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, pages 5:1–5:12, New York, NY, USA. ACM.

Davi, L., Dmitrienko, A., Sadeghi, A.-R., and Winandy, M. (2010). Privilege escalation attacks on android. In *international conference on Information security*, pages 346–360. Springer.

Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. (2011a). Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA. ACM.

Felt, A. P., Greenwood, K., and Wagner, D. (2011b). The effectiveness of application permissions. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*, WebApps'11, pages 7–7, Berkeley, CA, USA. USENIX Association.

Geneiatakis, D., Fovino, I. N., Kounelis, I., and Stirparo, P. (2015). A permission verification approach for android mobile applications. *Comput. Secur.*, 49(C):192–205.

Gilbert, H. and Handschuh, H. (2003). Security analysis of sha-256 and sisters. In *International workshop on selected areas in cryptography*, pages 175–193. Springer.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18.

Henderson, A., Prakash, A., Yan, L. K., Hu, X., Wang, X., Zhou, R., and Yin, H. (2014). Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 248–258, New York, NY, USA. ACM.

Idrees, F., Rajarajan, M., Conti, M., Chen, T. M., and Rahulamathavan, Y. (2017). Pindroid: A novel android malware detection system using ensemble learning methods. *Computers & Security*, 68:36 – 46.

Jing, Y., Ahn, G. J., Zhao, Z., and Hu, H. (2015). Towards automated risk assessment and mitigation of mobile applications. *IEEE Transactions on Dependable and Secure Computing*, 12(5):571–584.

Karim, M. Y., Kagdi, H., and Penta, M. D. (2016). Mining android apps to recommend permissions. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 427–437.

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97.

Leontiadis, I., Efstratiou, C., Picone, M., and Mascolo, C. (2012). Don't kill my ads!: Balancing privacy in an ad-supported mobile application market. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems &#38; Applications*, HotMobile '12, pages 2:1–2:6, New York, NY, USA. ACM.

Li, L., Bissyandé, T. F., Klein, J., and Traon, Y. L. (2016). An investigation into the use of common libraries in android apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 403–414.

Li, L., Bissyand, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Traon, L. (2017). Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67 – 95.

Lindorfer, M., Neugschwandtner, M., and Platzer, C. (2015). Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 422–433. IEEE.

Lopez, C. C. U. and Cadavid, A. N. (2016). Machine learning classifiers for android malware analysis. In *Communications and Computing (COLCOM), 2016 IEEE Colombian Conference on*, pages 1–6. IEEE.

Ma, Z., Wang, H., Guo, Y., and Chen, X. (2016). Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 653–656. ACM.

Mann, C. and Starostin, A. (2012). A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th annual ACM symposium on applied computing*, pages 1457–1462. ACM.

Merlo, A. and Georgiu, G. C. (2017). Riskindroid: Machine learning-based risk analysis on android. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 538–552. Springer.

Micinski, K., Votipka, D., Stevens, R., Kofinas, N., Mazurek, M. L., and Foster, J. S. (2017). User interactions and permission use on android. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 362–373, New York, NY, USA. ACM.

Rosen, S., Qian, Z., and Mao, Z. M. (2013). Appprofiler: A flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 221–232, New York, NY, USA. ACM.

Tăbuşcă, A., Tăbuşcă, S.-M., Garais, G. E., and Enăceanu, A. (2018). Mobile apps and gdpr issues. *Journal of Information Systems & Operations Management*, 12(1).

Wang, H., Guo, Y., Tang, Z., Bai, G., and Chen, X. (2015a). Reevaluating android permission gaps with static and dynamic analysis. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6.

Wang, H., Hong, J., and Guo, Y. (2015b). Using text mining to infer the purpose of permission use in mobile apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 1107–1118, New York, NY, USA. ACM.

Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., and Zhang, X. (2014). Exploring permission-induced risk in android applications for malicious application detection. *IEEE Transactions on Information Forensics and Security*, 9(11):1869–1882.

Wijesekera, P., Baokar, A., Tsai, L., Reardon, J., Egelman, S., Wagner, D., and Beznosov, K. (2017). The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 1077–1093. IEEE.

Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., and Enck, W. (2015). Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 303–313.

Yin, R. K. (2013). *Case study research: Design and methods*. Sage publications.

Zhang, Y., Yang, M., Yang, Z., Gu, G., Ning, P., and Zang, B. (2014). Permission use analysis for vetting undesirable behaviors in android apps. *IEEE Transactions on Information Forensics and Security*, 9(11):1828–1842.

Zhu, H., Xiong, H., Ge, Y., and Chen, E. (2014). Mobile app recommendations with security and privacy awareness. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 951–960, New York, NY, USA. ACM.

Zimmeck, S., Wang, Z., Zou, L., Iyengar, R., Liu, B., Schaub, F., Wilson, S., Sadeh, N., Bellovin, S. M., and Reidenberg, J. (2017). Automated analysis of privacy requirements for mobile apps. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, volume 2017.