Revisiting Static Analysis of Android Malware

François Gagnon
Cybersecurity Research Lab - Cégep Sainte-Foy, Québec, Canada
frgagnon@cegep-ste-foy.qc.ca

Frédéric Massicotte
CCIRC - Canada's Cyber Incident Response Centre, Ottawa, Canada

Abstract

The mobile malware threat is fought by both static and dynamic analysis, two complementary approaches in need of constant sharpening. In this paper, static analysis is revisited to update and deepen knowledge about Android malware, correlate malicious samples through common artifacts, and further understand malware developers' modus operandi. By looking at more than 200,000 malware samples, our study revealed interesting new insights such as: the presence of duplicated permissions in the manifest, the variation of the certificate validity period between malware and benign applications, the pertinence of looking at each sample's certificate file name, and the presence of Android applications nested inside other applications (APKs inside APKs). We also seek to revisit previous findings from related work on Android static analysis in order to confirm or refute them. In some cases, our findings are significantly different from previous work (e.g., diversity of certificates used to sign malware). Therefore, since the Android malware landscape is evolving, we conclude that our overall knowledge must be kept up-to-date.

1 Introduction

Static analysis of Android malware is the subject of numerous efforts that we classify into two categories. The first category uses either machine learning (e.g., [10]) or signature-like rules ([16]) in order to automatically detect Android malware from the information obtained by "training". The second approach provides human actionable intel to better understand the different features that can be leveraged to study malware samples (e.g., [3]). In this paper, the term static analysis is used in the broad sense of analyzing an Android application without executing it; it is not limited to the source code per se and also include text files and resources forming the application.

The approaches in the automated detection category are interesting because they scale to a large body of malware samples; however, they are susceptible to improper training and decay. Improper training may come from a lack of representative samples and by neglecting to use a historically consistent training dataset (as mentioned in [2]). Decay means knowledge must be revisited occasionally to account for the evolution of malware.

The approaches in the second category provide essential knowledge to properly engineer automated solutions and keep them up-to-date (thus countering decay) and are of particular interest to cyber incident responders.

This paper falls in the second category, providing human actionable intel. The data extraction from Android applications was performed automatically through static analysis techniques. Then, the resulting database has been explored manually (and semi-automatically) by an analyst to extract knowledge. The main contributions are:

- Providing an extensive study of various features (including more features than previous studies [1] - [16], see Section 5) that are meaningful in static analysis of Android malware.
- Revisiting conclusions of previous studies to see if they still hold true with a larger and more recent dataset of malware samples.
- Providing insights for cyber incident responders on what to look for when analyzing Android applications.

The rest of the paper is structured as follows. Section 2 provides some background regarding Android applications and their static analysis, while Section 3 presents the dataset and the tools used to extract static information. A presentation of the results follows in Section 4. Lastly, Section 5 discusses related work and Section 6 provides concluding remarks plus directions toward future work.

2 Background

Android applications are distributed through an APK file (Android Application Package), an archive following the zip format which can easily be decompressed and exam-

ined. Inside the archive lies a multitude of information related to the application, its authors, and its production process (i.e., packaging). By inspecting the contents and attributes of various files inside an APK, we can extract relevant information about the application that can help security analysts and researchers perform their tasks.

AndroidManifest.xml: APKs contain a manifest file, which provides information to the Android operating system on how to handle this application. A detailed analysis of several fields in the manifest was performed since it contains interesting pieces of information (some mandatory and some optional). (see Section 4.1).

META-INF: The `META-INF` folder contains information regarding the certificate used to package the application. Android requires each application to be signed by its developer before being deployed. A deep analysis of the certifcate was performed as it can provide a very good indication on the author of an application (results are presented in Section 4.2).

classes.dex: DEX files contain the executable code of the application (Dalvik EXecutable). Android applications are written in Java, compiled into bytecode, and assembled into DEX. We did not perform a deep analysis of DEX files; however, some specific elements were studied in that area (see Section 4.4)

others: Although it might contain interesting information, we did not explore the `resources.arsc` file, nor the `asset`, `res`, and `lib` directories.

From the large number of sources available in an APK, only key elements where selected for detailed analysis. Although the whole picture was not completely covered, a wider analysis was performed compared to similar work previously done (see section 5).

3 Dataset & Experiment

For this project, about 220,000 Android applications were statically analyzed through an automated process.[1] Of those, 208,221 are believed to be malware samples, while 10,007 are believed to be benign (a.k.a. legitimate) applications. We say "believe" because providing such a label on several recent samples is a hard and ultimately inaccurate task. The dataset is bound to have mislabelled samples. However, since our objective is to understand the big picture of the Android malware landscape and not to provide an automated way to detect new malicious samples, some mislabelling will not affect the conclusions. Various ground truth labelling approaches have been used by previous studies in the literature. For instance, [13] considers a sample malicious if it is tagged as such by at least 4 antiviruses from virustotal[2], whereas [3] uses a threshold of 22. In our cases, we consider a sample malicious if it was received by CCIRC from the commercial malware feed of a security vendor. Benign samples are those downloaded randomly from the Canadian Google Play store during a five day period starting on 2016-05-24. Table 1 provides the result of consulting virustotal for the samples in our dataset. Our dataset is also very recent with 91% of the malicious samples (and 80% of the legitimate ones) from either 2015 or 2016 (see Section 4.3).

Table 1: Dataset Analysis on virustotal

Nb Detection on virustotal	Nb Samples in Benign Dataset	Nb Samples in Malware Dataset
0	8,301 (83%)	144 (0.1%)
$[1,4[$	1,229 (12.3%)	3,603 (1.7%)
$[4,22[$	457 (4.6%)	54,703 (26.3%)
$[22,...$	20 (0.2%)	147,794 (71.7%)
N/A	0	1,977 (0.9%)

Note that the dataset used has not been pre-filtered and could be considered imbalanced through the inclusion of several instances from the same family. However, due to the nature (meta-information) of the features considered in this study, polymorphic variants of the same sample should, if done carefully to avoid easy correlation between the two variants, exhibit different values for those features. For instance, there is no reason for two variants to be signed by the same certificate, nor should the variants share the same application version number. Moreover, since the features considered here are not related to the (malicious) behaviour of the samples but more to the packaging process, having similar features does not necessarily imply variants of the same family. It means they have the same origin (packaging process). Although finding large clusters of samples sharing the same features indicates a possibly biased dataset, it also clearly illustrates the possibility of easily detecting basic clusters; which was one of the original goals of this experiment. Nonetheless, the reader should keep in mind that the dataset has not been pre-filtered when interpreting the results.

For the analysis, existing third party tools are leveraged to perform the experiment and automatically extract information from an APK. *ApkParser*[3] is used to decode the manifest. Based on the unzipped archive, the list of files inside the APK is produced; in particular, the last modification date of each file is kept as part of the archive. The X509 certificate is accessible inside the META-INF folder of the unzipped apk; *openSSL*[4] is used to decode

[1] More details on the dataset, including the list of SHA1, are available at `https://goo.gl/8qZrZn`

[2] `www.virustotal.com`

[3] `https://github.com/clearthesky/APK-parser`

[4] `openssl pkcs7 -inform DER -in fileName -noout -print_certs -text -out fileName.txt`

each certificate. For each DEX file in an APK, *dex2jar*[5] is used to decompile the code and retrieve the original bytecode files. Once an application has been processed, the various pieces of information extracted are inserted into a database from which statistics and comparisons can be done.

4 Results

This section presents the results from the experiment. They are grouped based on the source/nature of the information used - manifest is discussed in Section 4.1, certificates in Section 4.2, creation date in 4.3, and file listing in 4.4.

4.1 Manifest

The manifest contains information fields about the application, so it is a natural data source for static analysis. Free form fields (with no or little restrictions) contain interesting details to identify the (re)packaging process for malware samples.

4.1.1 Manifest.appPackage

Package names (`appPackage`) for benign applications are almost always unique (9,996 distinct on 10,006 manifests for a ratio of 0.999), while malicous applications reuse that field abundantly (106,574 distinct ones on 205,008 manifest for a ratio of 0.51). Note that according to Android's documentation, the `appPackage` must be unique for each application, and update of an application must retain the original `appPackage`.

Further analysis of malware reveals that 33,729 samples use "com.yongrun.app.sxmn" (figure 1 shows grouping of samples by the `appPackage`) and 7,223 have the prefix "com.ym.refpackage.jxyq" in their `appPackage`.

4.1.2 Manifest.appName

The name given to the main activity of the application (`appName`) is considered here. 42,751 (20% of malware) samples do not specify an `appName` at all, and only 26,444 distinct `appNames` (ratio of 0.16) are observed; therefore, malware reuse `appName` even more than `appPackage`. 5,994 (60%) benign apps do not specify an `appNames` with 3,005 distinct values observed (a ratio of 0.75).

Some values are used as `appName` prefix for several samples: 20 use com.amazon and 1,364 use com.qihoo.util.StubApp.

[5] `https://github.com/pxb1988/dex2jar`

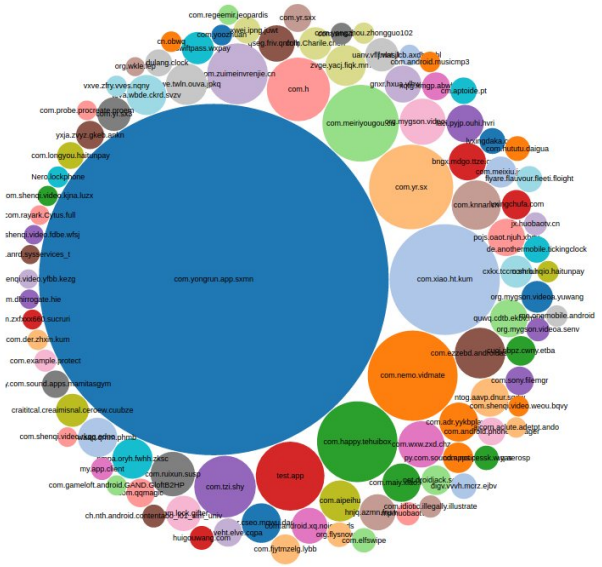


Figure 1: Grouping of Malware Samples by Package Names (showing 100 most popular)

4.1.3 Manifest.appLabel

The `appLabel` is the label given to the main activity of the application. This label is shown to the user in the list of installed applications. Only 904 malware and 40 benign samples do not provide a value for `appLabel`. There are 62,717 distinct `appLabels` for malware (ratio of 0.30), whereas legitimate apps have 9,826 distinct values (ratio of 0.99). Nearly half of the malware samples (91,588) have an appLabel containing Chinese characters.

4.1.4 Manifest.appVersionCode

Each update of a given app must be tagged with a new, higher, numeric value for `appVersionCode`. Malware have 9,757 distinct `appVersionCodes` (ratio of 0.048), while benign apps have 1,624 values (ratio of 0.162). 7,234 malware use the value "2,147,483,647" for `appVersionCode` which was the largest allowed value by the Google Play store at some point in the past.[6] However, a high `appVersionCode` (>1,000,000) is not indicative of a malware (7% of legitimate samples vs 8% of malware), nor is a low (<10) value (47% vs 54%). However, there exists much more malware with the value "1" (42%) than there are benign (14%). While [8] claimed that malware tend to have lower app versions than benign apps, our data indicates this is not the case, except when considering only `appVersionCode` of exactly "1". Note that [8] used `appVersionName` (see Section 4.1.5),

[6] At the moment of writing it is 2,100,000,000 (`https://developer.android.com/studio/publish/versioning.html`), however it was higher in the past (see discussion at `https://goo.gl/7Dd5tB`)

which is a free form text field with no semantic restrictions ("1.0" does not necessarily precede "3.4") instead of `appVersionCode`. This could distort the meaning of their claim.

From another angle, having the same `appVersionCode` as another app could be a hint that both apps are from the same place (especially for outlier values). For instance, three groups of malware samples have an `appVersionCode` matching a numeric date pattern (20160112, 20160411, and 20160804). Deeper investigation showed that, for all the samples in those groups, the `creationDate` falls within a few days (maximum of three days difference) of the date encoded in the `appVersionCode` field.

4.1.5 Manifest.appVersionName

For malware, there are 11,107 distinct `appVersionName` (ratio of 0.053). The most popular value is "1.0" with 71,941 samples (34.55%), far ahead of the second most popular "7.8.0" with 9,337 samples. 8,318 samples have a unique `appVersionName`.

For benign applications, there are 2,179 distinct`appVersionName` (ratio of 0.218). The most popular is "1" with 1,565 samples (15.64%), far ahead of the second most popular "1.1" with 509, and then "1.2" with 333.

4.1.6 Manifest.minSDKVersion

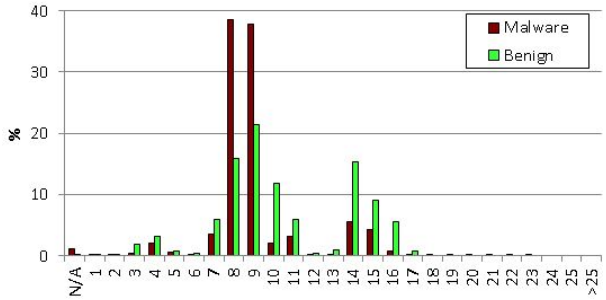
`MinSDKVersion` specifies the minimum API level that must be supported by an Android device for the application to be installed.

Figure 2a shows the distribution of `minSDKVersion` for malware and benign samples. 1.09% of malware samples omit this value (N/A), in comparison to 0.24% of benign apps. Finally, malware samples clearly gravitate towards "8" and "9" for `minSDKVersion` (74%), whereas benign apps require five `minSDKVersion` values ("8", "9", "10", "14", "15") to obtain the same 74% coverage.

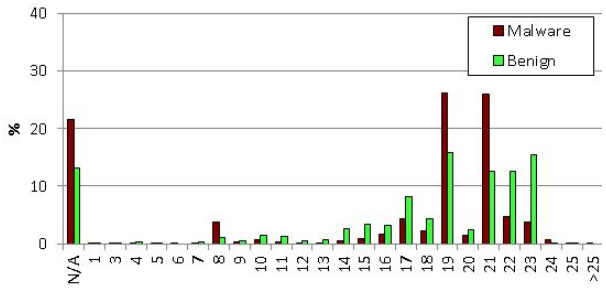
4.1.7 Manifest.targetSDKVersion

The `targetSDKVersion` specifies up to which API level the application has been tested to work without compatibility translation. Figure 2b provides graphical visualization for distribution of `targetSDKVersion`.

Out of the whole dataset, 22% of malware samples do not specify a `targetSDKVersion` (which defaults to `minSDKVersion`) and this drops to 13% for benign apps. 50% of malware samples use either "19" or "21" as `targetSDKVersion`, while benign apps require five values ("17", "19", "21", "22", and "23") to cover 50%. Even though the actual maximum acceptable value is "25", 25 malware samples use higher, invalid values (high



(a) `MinSDKVersions`



(b) `TargetSDKVersions`

Figure 2: SDK versions

values seen are "26", "27", "30", "41" and even "999"); a mistake not observed among benign samples.

4.1.8 Manifest.requestedPermissions

We observed an average of 21.54 `requestedPermissions` by malware samples (7.83 for benign). This confirms the observations of DroidMat [12] that malware generally request more permissions.

In 2012, [15] presented a top 20 of the most `requestedPermissions` by malware for the malgenome dataset. Two years later, [10] confirmed this top 20 with a very similar dataset. Our top 20 removes 7 permissions (35%) from [10, 15] (ACCESS_FINE_LOCATION, READ_CONTACTS, WRITE_SMS, CALL_PHONE, WRITE_CONTACTS, WRITE_APN_SETTINGS, and RESTART_PACKAGE), replacing them with READ_EXTERNAL_STORAGE, WRITE_SETTINGS, MOUNT_UNMOUNT_FILESYSTEMS, INSTALL_SHORTCUT, CHANGE_NETWORK_STATE, SYSTEM_ALERT_WINDOW, AND GET_TASKS. This indicates an evolution in the malware ecosystem.

During the analysis, we noticed that several malware (65% vs only 7.6% for legitimate apps) request the same permission twice. We conjecture this `duplicatedPermission` anomaly to be indicative of an automated repackaging process where permissions required by the malicious code are added automatically to the manifest even if they are already present.

4.2 Certificate

Each Android application must be signed using the developer's private key. APKs are thus bundled with an X509 certificate containing the developer's public key for signature validation. Such certificates are a rich source of information regarding the origin of malicious samples. The vast majority of samples, both malicious and benign, come with a self-signed certificate (99.89% and 99.92% respectively). Hence, this is not an interesting indicator. Worse yet, it opens the door wide for abusive automated generation of a large number of certificates.

4.2.1 Certificate.Serial

Some earlier work [9] used the certificate serial number (`serialNumber`) to identify identical certificates. However, this approach does not properly capture the equality relationship. Indeed, in our malware dataset, some `serialNumbers` are shared among different certificates. For instance, `serialNumber` "1204248826" corresponds to three distinct certificates, while "1829006857", "1427696956", "2055325431", and "740422034" are seen each in two distinct certificates. The `signature` field (Section 4.2.2) of a certificate, based on a cryptographic hash function, is more appropriate to represent certificate equality .

In our malware dataset, 5.3% are signed with the debug certificate (serial "10623618503190643167" [9]), while 0.3% are signed with the platform certificate (serial "12941516320735154170" [15]). No legitimate application is signed by these certificates (they would be rejected by the Google Play store).

4.2.2 Certificate.Signature

Using the `signature` to uniquely identify certificates, coverage statistics can be computed. 23,108 certificates are needed to cover 50% of the malware dataset, a ratio of 0.11 (this number is 3,504 for benign apps, ratio of 0.35). This is drastically different from the results of [9] where only 5 certificates are needed to cover 50% of their dataset (ratio of 0.001). Our hypothesis is that the small size and lack of diversity of the dataset used in [9] led to unrepresentative numbers.

Overall, 126,024 distinct certificates were seen for malware giving a ratio of 0.605 (8,496 for legitimate with a ratio of 0.85), whereas [9] has a ratio of 0.137 (622 distinct certificates for 4,554 malware); which is again a drastic difference.

So, when two samples share the same certificate (same `signature`), they are likely to originate from the same developer which provides easy association of samples. In the malware dataset, one certificate is used to sign 10,379 samples, eight are used to sign more than 1,000 samples each, and 59 are used to sign more than 100 samples each.

However, having distinct certificate `signatures` (or `serialNumbers` for that matter) does not automatically mean a different origin. In our dataset, we found nine samples having distinct `signatures` but the same `publicKey` (hence same private key). It is probabilistically unlikely that key pairs generated independently would yield the same value. Our hypothesis is that the malware developer choose to regenerate new certificates but reused the same key.

4.2.3 Certificate.Subject

We see only 62,093 distinct `subjects` for the malware dataset, even though we had 126,024 distinct certificates (distinct `signatures`). Note that for benign apps, the groupings by `signatures` vs `subjects` are very similar (8,496 `signatures` and 8,159 `subjects`).

While some subject values are too generic to draw any strong conclusions regarding the potential similar origin of two samples (e.g., 2,000 use the string "Unknown" for the fields C, ST, L, O, OU, and CN), others seems randomly generated and thus very specific (e.g., 3,324 use the string "llpfihhdydkfffovoaxbl1bm" for those fields).

4.2.4 Certificate.FileName

For the malware and legitimate datasets, more than 97% of certificate files are ".RSA" while the rest are ".DSA". For the malware dataset, we see 17,905 distinct certificate `fileNames` (ratio of 0.086) and only two `fileNames` (CERT.RSA and ZZW.RSA) are needed to cover 50% of the samples. For benign apps, 926 distinct `fileNames` are observed (the ratio of 0.093 is very similar to the one for malware) and just one `fileName` (CERT.RSA) is enough to cover half the samples (actually 80%).

Furthermore, 2,249 malware samples have a `fileName` following the pattern "8 digits dot RSA", which could represent a date (yyyymmdd). Of those, 1,594 samples have a `fileName` value that exactly matches the app `creationDate` (see Section 4.3); this may indicate an automated generation process.

4.2.5 Certificate.ValidityDates

X509 certificates include a "not valid before" and a "not valid after" date (`certStartDate` and `certEndDate` respectively). For HTTPS, these dates must be verified before accepting a certificate. This is not the case for Android as one can manually install an APK signed by a not-yet-valid or expired certificate. However, to be accepted on the Google Play store, an APK must[7] have a certifi-

[7] `https://developer.android.com/studio/publish/app-signing.html`

cate expiring after 2033-10-22 and a `certStartDate` in the past. For both datasets (malware and benign), a few erroneous `certStartDates` are observed (before 1980 or, for one malware, after 2080).

Interesting peculiarities can be observed from the `validityPeriods` (delta between start and end dates). The minimum value for `validityPeriod` is 18 years for legitimate applications, whereas it is only one day for malware (recommendation[7] is at least 25 years). 13,568 malware samples (6.6%) have a `validityPeriod` of less than one year. We observe that, in general, certificates for legitimate apps are valid longer than malware (average of 188.5 vs 10 years); hence, a very short `validityPeriod` is indicative of malware.

We formulate the following hypothesis to explain the observations surrounding `validityPeriod`: malware packaged through a fully automated environment uses command line tools (e.g., *openssl*) instead of *Android Studio* to generate a certificate for each malware version. Since in *Android Studio* the validity period is specified in years, while it is in days for *openssl*, this leads to shorter `validityPeriod` for malware. Indeed, about 70% of the benign dataset has a `validityPeriod` in years (the number of days in the `validityPeriod` can be divided exactly by 365), whereas only about 30% of the malware dataset have this property.

4.3 Creation Date

The `creationDate` of an APK can be obtained through the "last modification date" of a file inside the archive. For this research, we relied on the `AndroidManifest.xml` file for creation date as in the normal packaging process, the manifest file is (re)encoded every time the application is packaged. A more robust approach would be to take the latest date among the "last modification date" for the manifest and dex files[8]. Deeper investigation regarding abnormal `creationDates` (e.g., prior to 2008) yielded two significant clusters: 1979 (between 1979-11-29 16:00 and 1979-11-30 5:00, all precisely on the hour) and 1980 (1980-01-01 between 1:00 and 13:00, again all straight on the hour). Our hypothesis is that these dates represent reset dates (epoch time) on specific given system (e.g., MS-DOS.[9]). In general, `creationDate` outliers should be investigated for possible similarities.

In [3], Allix conjectured that malware were created in a Monday-Friday work-like cycle by analyzing `creationDates`. Our results confirm Allix's finding as we observe less activity (both for malware and legitimate applications) on the week-end.

[8] In the malware dataset, less then 6,000 apks have a dex file modified after the manifest file.

[9] `http://www.computerhope.com/jargon/e/epoch.htm`

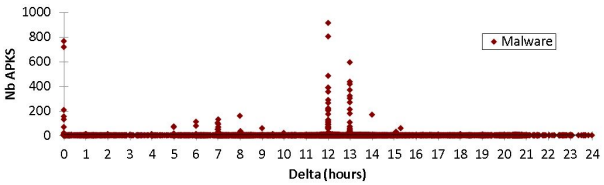


Figure 3: Delta Between Creation Date and Certificate `certStartDate` for Malware Samples

Another interesting observation is related to `creationDelta`, the delta between the application `creationDate` and the `certStartDate` of the certificate. Figure 3 shows the number of samples having a specific `delta`, with second-level precision, for malware. The figure shows three big peaks and seven smaller ones. The leftmost peak (at X = 0 hour) is understandable as several apps have the same (or very similar) `creationDate` and `certStartDate` (since both the APK and the certificate are automatically generated upon packaging). Closer investigation of the other peaks indicates that they are all precisely on the hour, the most popular being 12 and 13 hours. Other peaks are at 5, 6, 7, 8, 9, 14, and 15 hours. 4,430 have a `delta` of exactly (to the second) 12 hours while the number jumps to 27,736 when including a range of ± 10 seconds around the 12 hours `delta`. Our hypothesis is that this reflects an automated packaging environment which relies on online services (across different time zones), where the time zones would explain the "hourly" peaks.

4.4 File List

Investigation of the list of files inside an APK yields interesting peculiarities. First, some APKs include another APK file inside their archive (`embeddedAPK`). This happens more frequently for malware samples (13,211 occurrences) than for benign (97 samples). The `embeddedAPK` finding is noteworthy and requires further investigation.

Second, the average number of files `nbFiles` inside APKs is 246.5 and 606 for malware and benign apps respectively. A possible explanation here is that legitimate applications are packaged with more resources files (e.g., GUI elements for different screen resolutions) while malicious ones tend to neglect that aspect.

Third, a quick targeted investigation revealed 263 samples in the malware dataset (none in benign dataset) showing an obvious presence of the droidjack RAT: those samples have source code in a package named droidjack.

Finally, the malware dataset has 8,250 samples containing all the same DEX file (with md5 cfdba92d344b57fecabadab26296f84c).

5 Related Work

Permissions have been a focal point of Android malware static analysis. For instance, [6, 10, 15], focus entirely on that, while others [5, 7, 12, 13, 16, 8, 9] study permissions among a few other things. [11] even proposes a review and classification of these permission-based approaches.

We took a different approach and explored several features that can easily be extracted with static analysis. We studied more than 25 features, whereas previous work have been usually limited to five or less. Instead of focusing on the detection of malware through static analysis, we aim to enhance our understanding of the malware creation process (e.g., packaging) and to find ways to connect different samples (malicious or not) together by using artifacts present in their packaging structure. Below is a comparison of our work with some existing publications that are closely related.

[8] is limited to four elements of the manifest file inside the APK: the `requestedPermissions`, the `appVersionCode`, and the registered receivers and filters. In our study, we explored other elements of the manifest and several elements outside the manifest. Moreover, while their dataset was limited to less than 650 apps (including both malware and legitimate apps), our study contains nearly 220,000.

[5] aims at creating fingerprints to identify malware and classify them into families by using static analysis. They use `requestedPermissions` from the manifest as well as source code (from DEX files) and libraries to do so. They tested their implementation on roughly 1,000 samples, most of which are from the malgenome dataset. The limited number and the absence of recent samples (malgenome dates back from 2012) make it uncertain that the results still apply for today's malware (i.e., susceptible to decay). Our work is complementary as the features we studied are mutually exclusive to theirs (with the exception of `requestedPermissions` where we added the innovative twist of duplicated permissions).

[14] proposes DroidMOSS, a tool to detect repackaged Android apps through static analysis. Their main focus is source code inspection, but they also analyze two other supporting features: certificate `subject` and manifest `appPackage`. Their dataset includes 22,000 benign applications.

The work by Allix et al. ([1], [2], and [4]) uses machine learning to detect malware through static analysis. Their approach relies on source code analysis (extracting the control flow graph and the opcode sequences). They used over 50,000 applications for their tests, including the malgenome dataset. [4] is one of the rare studies on static analysis to use another source of malware, namely samples from virustotal. Moreover, [4] also mentions certificate `signatures` (two APKs signed by the same certificate) as a side element (not part of their proposed solution). Our work relies on other static features and has been performed on a larger, more recent dataset.

[3] is closely related to our work as they focus more on forensics (i.e., identifying artifacts that could be used by incident responders) than on malware detection. They analyzed a large dataset of 594,000 APKs (325,214 from the Google Play store) and provided some very interesting intel. However, they looked mainly at two elements inside an APK: the APK `creationDate` and the certificate `signature` (to identify certificates used to sign multiple samples). They also indicate that one could inspect the certificate `subject` but did not provide an analysis.

The AndroTracker tool [9], extracts intents and `requestedPermissions` from the manifest; permissions from the source code (through an analysis of API calls); commands also from the source code (looking for a list of keywords); and certificate `serialNumber` from the certificate file. Their dataset contained 55,000 APKs (4500 malware and 51,000 legitimate). The discussion regarding the problem of using certificate `serialNumber` (see Section 4.2.1) and the different numbers regarding certificate coverage (see Section 4.2.2) provide updates on their results.

[16] presents DroidRanger, a permission-based footprinting tool to automatically detect malicious apps in markets. They focus mainly on `requestedPermissions` and receivers with discussions about API calls extracted from source code. Their approach is rule-based and they build their rules through manual analysis of malicious samples. They used a dataset of 20 malware to build their rules and then tested their solution on about 200,000 apps from markets. Our work has a larger scope both in terms of features examined and malware samples studied, but follows the same forensic approach.

6 Conclusion

In this paper, we looked at a very broad set of over 25 features that can be extracted from an Android application through static analysis, 15 of which are very interesting for forensics. Our experiment was performed on a large dataset containing more than 200,000 recent malware samples and 10,000 benign applications.

Looking at statistics from these features (comparing malware with benign apps), we found that many features behave differently for malware and are therefore interesting sources of information for eventual classification of samples. The presence of duplicate permissions in the manifest file is an interesting indicator (see Section 4.1.8) in addition to the validity period of the certificate, which seems to differ significantly in malware (see Section 4.2.5). We are not aware of previous work studying

these specific aspects.

We revisited some statistics found in the literature; sometimes confirming the findings, other times contradicting them. This supports our initial idea that general knowledge about Android static analysis must be revisited occasionally to account for the evolution of malware samples, benign applications, and the overall mobile ecosystem (e.g., Android API). For instance, Section 4.2.2 shows a clear evolution from previous work regarding the distribution of certificate signatures.

6.1 Future Work

Although we took a broad approach by considering a larger set of features, a lot of elements inside an APK remain to be explored in future work (e.g., source code). Moreover, several hypotheses remain to be confirmed or better understood. In particular: Are duplicated permissions really a sign of repackaging an existing application? What is the meaning of embedded APKs? Why do so many samples have a delta of exactly 12 hours between the sample creation time and the certificate `certStartDate`?

Finally, we are currently exploring the forensic capabilities of the gathered data. For instance, detailed analysis of the samples with `appPackage` prefix "com.ym.refpackage.jxyq" (mentioned in Section 4.1.1) indicated that: `AppVersionCode` and `appVersionName` are distinct for each of the 7,223 samples and so is the `appName` which appears randomly generated. The samples' `creationDates` are all between 2016-08-03 and 2016-12-11, and all have distinct certificate `signatures`. Most samples use the `fileName` "TMP.RSA" to store the certificate (only 926 other samples use this particular `fileName`). Those not using that particular `fileName` use five random uppercase letters for the `fileName`. From here, we can observe malware creators refining their automated packaging environment. Indeed, all the samples using "TMP.RSA" were packaged before 2016-10-31, whereas those using five random letters were packaged after 2016-11-01. Moreover, all the early samples also use the same certificate `subject` (C=xx, ST=xx, L=xx, O=xx, OU=xx, CN=shuany), while the more recent samples use a random certificate `subject` (C=ab, ST=cd, L=df, O=gh, OU=ij, CN=klmnopq) where letters a-q are randomly generated.

Acknowledgments

We are grateful to Canada's Cyber Incident Response Centre for providing data, help, and access to their infrastructure for experimentation.

References

[1] ALLIX, K., BISSYANDÉ, T. F., JEROME, Q., JACQUES, K., RADU, S., AND LE TRAON, Y. Large-scale machine learning-based malware detection: confronting the "10-fold cross validation" scheme with reality. In *Proceedings of the 4th ACM conference on Data and application security and privacy* (2014), pp. 163–166.

[2] ALLIX, K., F. BISSYANDÉ, T., KLEIN, J., AND LE TRAON, Y. Machine learning-based malware detection for Android applications: History matters! *Technical Report - University of Luxembourg*, 10993-17251 (2014).

[3] ALLIX, K., JEROME, Q., BISSYANDÉ, T. F., JACQUES, K., RADU, S., AND LE TRAON, Y. A forensic analysis of Android malware - how is malware written and how it could be detected. In *IEEE 38th Annual International Computers, Software and Applications Conference* (2014), pp. 384–393.

[4] ALLIX, K., JEROME, Q., RADU, S., AND ENGEL, T. Using opcode-sequences to detect malicious Android aplications. In *IEEE International Conference on Communications* (2014).

[5] BILLAH KARBAB, E., DEBBABI, M., AND DJEDIGA, M. Fingerprinting Android packaging: Generating DNAs for malware detection. In *16th Annual USA Digital Forensics Research Conference* (2016), pp. 533–545.

[6] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On lightwieght mobile phone application certification. In *Proceedings of the 16th ACM conference on CCS* (2009), pp. 235–245.

[7] FAHL, S., HARBACH, M., MUDERS, T., SMITH, M., BAUMGARTNER, L., FREISLEBEN, B., AND SMITH, M. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CSS'12)* (2012), pp. 50–61.

[8] FELDMAN, S., STADTHER, D., AND WANG, B. Manilyzer: Automated Android malware detection through manifest analysis. In *IEEE 11th International Conference on Mobile AdHow and Sensor Systems* (2014).

[9] KANG, H. J., JANG, J.-W., MOHAISEN, A., AND KIM, H. K. AndroTracker: Creator information based Android malware classification system. In *Proceedings of the 15th International Workshop on Information Security Application* (2014).

[10] ROVELLI, P., AND VIGFUSSON, Y. PMDS: Permission-based malware detection system. In *Proceedings of the 10th ICISS International Conference* (2014), pp. 338–357.

[11] TCHAKOUNTÉ, F. Permission-based malware detection mechanisms on Android: analysis and perspectives. *JOURNAL OF COMPUTER SCIENCE 1*, 2 (2014).

[12] WU, D.-J., MAO, C.-H., WEI, T.-E., LEE, H.-M., AND WU, K.-P. Droidmat: Android malware detection through manifest and API calls tracing. In *Proceedings of the Seventh Asia Joint Conference on Information Security* (2012), pp. 62–69.

[13] XU, W., ZHANG, F., AND ZHU, S. Permlyzer: Analyzing permission usage in Android applications. In *Proceedings of the IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)* (2013), pp. 400–410.

[14] ZHOU, W., SHOU, Y., JIANG, X., AND NING, P. Detecting repackaged smartphone application in third-party Android marketplaces. In *ACE CODASPY'12* (2012), pp. 317–326.

[15] ZHOU, Y., AND JIANG, X. Dissecting Android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (2012).

[16] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 2012 NDSS Symposium* (2012).