Dissecting APKs from Google Play: Trends, Insights and Security Implications

Pedro Jesús Ruiz Jiménez*†, Jordan Samhi*‡, Tegawendé F. Bissyandé*§, Jacques Klein*¶
*SnT, University of Luxembourg, Luxembourg
†pedro.ruiz@uni.lu, ‡jordan.samhi@uni.lu, §tegawende.bissyande@uni.lu, ¶jacques.klein@uni.lu

***Abstract*—Researchers generally look for specific files within Android application packages (APKs) during their analysis, focusing on common files such as Dalvik bytecode or the Android manifest. However, Android apps are complex archive files containing various types of files. Failing to account for all files during analyses can compromise end-user security, and despite the wealth of existing techniques to analyze Android apps, only a few studies explore the diversity of files within apps.**

To bridge this gap, we propose the first large-scale empirical study that dissects the content of Android apps from Google Play. In our study, we explore the different file types and their usage trends. We enhance our analysis by exploring compressed files and the files they contain. We finally investigate to which extent developers use disguised files, i.e., files whose extension is conventionally associated with a file type different than its own (e.g., a Dalvik dex file with the extension ".png"), and study if they are a hint of maliciousness. Our results show that: ❶ Android apps comprise diverse file types, with over 15 000 distinct file extensions and more than 1000 unique file types found in our dataset containing over 400 000 APKs; and ❷ we found many cases where developers use a wrong relation between the file type and its extension to load malicious code at runtime.

I. Introduction

Android apps are not only made of bytecode and XML files. They are complex archive files, which, beyond the bytecode and XML files, contain files of various types such as pictures, binaries, text files, code in other languages, scripts, compressed files, etc. Some APKs even come with files that are split into several parts or with code hidden in non-obvious file types, such as pictures [1], [2], [3]. While there might be various reasons for app developers to make such implementation decisions, they can hinder maintenance and even analyses. For example, failing to account for all files leads to incomplete analyses by state-of-the art tools [4], [5], [6], [7], [8], [9], [10], which in turn can threaten security.

The research literature contains various works aiming at extracting information from Android apps. For instance, in [11], [12], [13], [14], the authors have studied features extracted from the market where the app was published (e.g., their popularity, their description, etc.). In [15], [16], [17], [18], [19], the authors rather extract features directly from apps to find markers of maliciousness. However, the extent to which existing Android app analysis techniques overlook or exclude various types of files and the proportion of these overlooked files that should be considered critical is unknown. Even more surprising, the composition of Android apps, i.e., what are the common and uncommon files contained in a given app, is unknown. To bridge this knowledge gap, we propose a large-scale empirical study that aims at dissecting the content of apps from Google Play to understand them. We go beyond surface-level analysis by exploring insights that were previously unexplored. Our empirical study investigates the composition of apps and inconsistencies between files and their types, which must be considered during analysis since they are signs of maliciousness.

Our study is carried out in three main steps, toward providing a comprehensive view of the composition of APKs:

1) First, we provide a large-scale study of the composition of apps and collect data on the files inside APKs.
2) Second, we present the first study focusing on files embedded within APKs, specifically examining files contained within compressed archives inside APKs.
3) Lastly, we study files whose type and extension are inconsistent and measure their impact on maliciousness in APKs.

Some of the most significant insights of our study are: ❶ a large diversity of file types within Android apps with over 1000 unique file types; ❷ an even larger diversity of file extensions within Android apps with over 15 000 unique file extensions; ❸ the extended usage of custom file extensions by developers found in over 10% of Android apps; ❹ a non-negligible number of APKs are embedded with other APKs and almost 10% of APKs contain compressed files; ❺ the knowledge that some developers use inoffensive looking extensions to hide the true nature of their files; and ❻ we identified two indicators of maliciousness: files containing code with unconventional extension (e.g., native code in a ".png" file), and an APK embedded within another APK.

Our investigations serve *researchers* by uncovering new knowledge about app composition and trends in app development practices. The findings will provide researchers with a foundation for future studies in the domain of Android app analysis. Overall, we contribute to the research community by *proposing the first study that dissects Android apps to understand their composition at a large scale.*

The main findings of our study indicate that analysis of Android apps (automated or manual) *must* not be restricted to analyzing DEX and Android Manifest files. Indeed, our study shows that Android apps are made of a wealth of different files, including code, that *must* be accounted for during analysis.

We release our artifacts, as well as additional data at:
https://github.com/Trustworthy-Software/DissectingAPKs

II. Background

The type of a file is often characterized by its file extension. For instance, the file "foo.txt" is assumed to be a text file (because of the extension ".txt"). However, a file extension can be manipulated arbitrarily by developers. To get more reliable results related to the types of files, one can rely on the *python-magic* interface, which depends on the *libmagic* file type identification Python library [20]. Given a file as input, the command `magic.detect_from_content()` returns the MIME type, the magic type, and the encoding of the file.

MIME type [21], also referred to as Multipurpose Internet Mail Extensions, serves to indicate the format and characteristics of files. These MIME types are composed of a type and subtype, separated by a slash (e.g., "image/png"). The type indicates the broad file category, such as image or audio, while the subtype specifies the exact file type.

Magic type refers to a file type given by the Python magic library [20]. Concretely, this library retrieves the file *magic number* [22], which is a value embedded in files (often at the beginning). This magic number is then used to identify the type of the file. This is a line from the "etc/magic" file, used by the magic library to identify files:

`0`	`short`	`0x5AD4`	`DOS executable`
offset	type	magic number	magic type

Encoding [23] defines how the file characters are represented. It determines the characters we can read or write from the file.

Additionally, we introduce two novel definitions that allow us to summarize the unique characteristics met by some of the files found in our dataset. These particular files will be studied in depth during the last step of our study.

Discrepant files represent files whose extension is not related to any file type (e.g., an image file using the extension ".mypng"). Developers might modify the extension to customize it. While not purposefully malicious, this behavior can lead to files being overlooked by analyzers.

Disguised files are files whose extension is conventionally associated with a file type different than its own (e.g., a Dalvik dex file with the extension ".png"). A malicious developer could modify the extension to make a file look harmless and hide it within an app. If the file is missed during the static analysis, it can introduce unknown code on any device downloading the app.

III. Composition of Android apps

Android apps are complex archive files containing a huge diversity of files (e.g., XML files, code files, binaries, media files, etc.). This diversity has never been studied in the literature. This section bridges this gap by proposing the first large-scale empirical study on the composition of Android apps.

A. Empirical Setup

First, we detail the process for building our dataset, depicted in Figure 1. In the following, we give the details for each step.

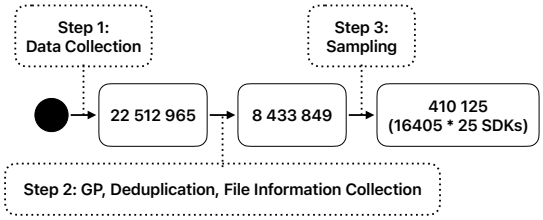

Fig. 1: Dataset Evolution (GP: Google Play).

Step 1: Data Collection. Our empirical study relies on the AndroZoo repository [24], a dataset of Android apps central to our research. We have considered the entire dataset as of October 2023, i.e., 22 512 965 APK files (the standard file format to distribute Android apps). Note that an app, identified through its unique package name (e.g., com.example.MyApp), may be associated with several APK files, reflecting the existence of multiple versions of the app within AndroZoo. To disambiguate, we will refer specifically to APK files in this paper.

Since one of our goals is to study trends in app composition, it is essential to establish the time frame in which the apps were developed. Note that we did not rely on the "dex_date" metadata provided by AndroZoo since it is unreliable. For instance, in October 2023: 8 679 512 apps have their dex date set to 1980, 6 780 599 have it set to 1981, etc. Therefore, we decided to complement the metadata provided by AndroZoo by adding a new field that we named *sdk*. The idea is that since the SDKs are linked to Android versions, we get a sufficient and reliable approximation of when the app has been released. We retrieved the *sdk* information of each APK by inspecting its "AndroidManifest.xml" file. More specifically, we consider the field *targetSdkVersion*, which is present in 93.2% of APKs. In cases where the *targetSdkVersion* was null, we used the *platformBuildVersionCode* field since they represent the same value. Eventually, we collected SDK information for 94% of the apps. The remaining 6% have no information regarding the SDK version.

For each APK, in addition to the *sdk* field, we collected the following information from the metadata: *SHA-256 hash, VirusTotal score, package name, app version, APK size, Dalvik bytecode size, markets, SDK, and permissions.*

Step 2: Google Play, Deduplication and File Information Collection. First, we only retain APKs from the Google Play Store. The dataset is left with 19 614 016 APKs after applying this filter. Second, we deduplicate apps, i.e., we only retain the latest version of a given app (i.e., among APK files with identical package names, we only keep the most recent APK). The dataset is left with 8 433 849 APKs after applying this filter.

For each of the 8 433 849 APKs in our dataset, we augmented their metadata with additional information. Specifically, for each file inside any APK associated to an app, we extracted the following information: ❶ the file encoding; ❷ the MIME type; ❸ the magic type; ❹ the file extension; and

❺ whether the file is compressed or within a compressed file.

Note that inside APK files (an APK file is just an archive file containing multiple files), one can find other archive/compressed files. We took that into account in our analysis and for every compressed file detected in an APK, the file is decompressed, and its content is considered in our study. To identify compressed files without relying on their extensions, we attempted to decompress all files using the *Zipfile* library [25] and flagged those that were decompressed successfully as compressed.

Step 3: Sampling. Our initial dataset, from AndroZoo, has a very long tail distribution (timewise). To reduce biases, we devise a sampling procedure that ensures that we consider representative APKs across the Android timeline: we aim for the same number of APKs per relevant SDK version.

Our sampling procedure was the following: ❶ In our dataset, the SDK that contains the most APKs is SDK 30, with 1 152 062 APKs. A representative sample (with 99% confidence level and 1% margin of error) for 1 152 062, is 16 405. Therefore, we decided to collect at least 16 405 APKs from each SDK to achieve a representative and distributed sample. ❷ *Sample process*: As seen in Figure 2, SDKs from version 8 to 33 (except SDK version 9) are above this threshold (i.e., 16 405 APKs). We then randomly selected 16 405 APKs per SDK version whose number of APKs is above the threshold (25 SDK versions). As a result, an evenly distributed dataset of 410 125 APKs was generated over 25 SDK versions (i.e., 16 405 APKs x 25 SDKs = 410 125 APKs). These 25 SDKs cover a large timeline from 2010 to 2023. Our evolution study leverages SDK versions to associate with the timeline of Android APKs.

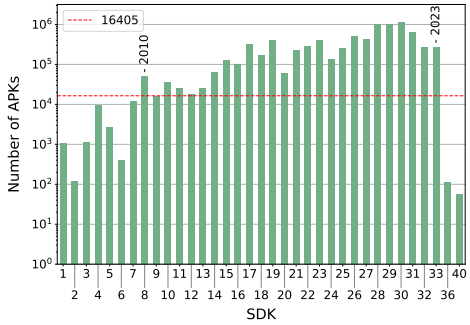

Fig. 2: Top 35 SDKs.

B. Research Questions

Section III answers the following research questions:

RQ1: What is the distribution of files in terms of numbers and types in APKs?

RQ2: How did the distribution of file types and sizes within APKs evolve over time?

C. Empirical Findings

This section presents an analysis of various aspects related to the composition and properties of all APKs in our dataset.

***RQ1:** APK Number of Files and their Diversity:*

Overall Statistics on Files. Overall, we extracted 271 155 026 files from the 410 125 APKs of our dataset (661 files on average per APK and a median of 485 files). Table I presents statistics on the types of the files. Overall, nine different encoding types were found. However, the dataset contains many more MIME types (234), Magic types (1099), and more than 15 000 different extensions (15 349).

TABLE I: Number of Unique File Type Identifiers.

Field	Encoding	MIME type	Magic type	Extension
Count	9	234	1099	15 349

Table II shows the number of files per encoding in our dataset. The binary encoding is, by far, the most common encoding in our dataset with more than 240 million files. Additionally, the presence value indicates the percentage of APKs from our dataset which contain at least a file with said encoding, finding binary files in every single app from our dataset of 410 125 APKs. Tables III and IV show the top 20 most common MIME types and Magic types, respectively, while Table V shows the top 40 most used extensions in our dataset. First, these tables show that the most prevalent file type in our dataset is PNG files (i.e., pictures). Second, we can see that though the second most prevalent file type is XML from the Magic type and extension, the second most used MIME type is "application/octet-stream". This is explained since most XML files in Android apps are compiled XML files (e.g., the AndroidManifest.xml file which is present in every single app). These tables also show that ".dex" files are not the only code files. For instance, there is a larger number of ".js" and ".class" files. We also find a large number (1 124 050) of ".so" extensions, which represent binary files. Eventually, note that we count 587 062 "Dalvik Dex files", indicating that there is not only one single dex file in Android apps. **These numbers show ① the high diversity of types of files in APKs, and ② code in APKs is not restricted to DEX files.** Our investigations demonstrate the need for more comprehensive static analyses that account for *all* code files present in apps.

RQ1 answer: We find that: ❶ APKs tend to be composed of a large number of files, with over 600 files per APK on average; and ❷ there is a large diversity of file types in APKs, with over 15 000 unique file extensions in our dataset associated with over 1000 different magic types.

TABLE II: Encodings.

Encoding	**Count**	**Presence**	**Encoding**	**Count**	**Presence**
binary	241 972 123	100%	utf-16le	34 473	0.3%
us-ascii	24 354 131	99%	utf-16be	6187	0.05%
utf-8	4 636 896	35%	ebcdic	2296	0.2%
iso-8859-1	104 384	6%	utf-32le	6	0.001%
unknown-8bit	44 530	1%			
Total Count	271 155 026				

TABLE III: Top 20 MIME Types (app: application).

MIME type	Count	Presence	MIME type	Count	Presence
image/png	119 838 955	99%	image/x-tga	1 296 018	52%
app/octet-stream	106 268 807	100%	app/x-sharedlib	1 132 056	29%
text/plain	18 905 627	99%	image/webp	1 019 330	2%
image/jpeg	4 601 195	46%	font/sfnt	896 739	33%
app/x-java-applet	3 166 796	8%	app/x-dosexec	497 721	6%
image/svg+xml	3 030 236	8%	image/gif	438 465	20%
text/html	2 576 405	33%	text/x-java	408 635	6%
app/json	1 945 881	25%	text/x-c	348 334	5%
text/xml	1 553 837	31%	audio/ogg	340 190	7%
audio/mpeg	1 367 471	14%	app/zip	194 845	7%

TABLE IV: Top 20 Magic Types.

Magic type	Count	Presence	Magic type	Count	Presence
PNG image	119 838 955	99%	JSON	1 945 881	25%
Android binary XML	85 454 543	99%	XML document	1 562 397	31%
data	15 813 819	100%	Targa image	1 296 011	52%
ASCII text	15 528 419	99%	RIFF	1 171 281	8%
JPEG image	4 601 195	46%	TrueType Font	896 739	33%
Java serialization	3 695 805	2%	ELF 32-bit shared	830 541	29%
compiled Java class	3 166 796	8%	Audio file ID3	768 541	11%
SVG image	3 019 743	8%	MPEG ADTS	626 404	8%
HTML document	2 576 394	33%	Dalvik dex file	587 062	99%
UTF-8 Unicode text	2 223 638	21%	SGML document	586 379	24%

TABLE V: Top 40 Extensions.

Extension	Count	Presence	Extension	Count	Presence
.png	120 035 855	99%	.dll	527 728	5%
.xml	88 079 179	99%	.kotlin_module	485 577	8%
none	11 717 336	44%	.gif	432 944	20%
.js	5 343 040	21%	.kotlin_builtins	431 258	15%
.jpg	4 283 785	44%	.mf	426 666	98%
.version	4 151 057	25%	.arsc	412 112	99%
.properties	3 656 582	49%	.sf	407 196	98%
.svg	3 037 053	8%	.rsa	399 061	96%
.class	3 036 890	1%	.xsb	342 235	0.03%
.kotlin_metadata	2 771 817	2%	.java	335 800	2%
.html	2 727 683	32%	.ogg	328 717	7%
.txt	1 965 585	30%	.map	212 431	4%
.json	1 633 817	24%	.mod	201 061	1%
.mp3	1 435 960	14%	.lua	190 052	0.2%
.so	1 124 050	28%	.resource	184 838	4%
.webp	1 024 007	2%	.bin	176 040	12%
.ttf	889 255	33%	.scss	173 336	1%
.res	787 994	0.5%	.zip	169 454	5%
.css	664 185	17%	.plist	159 721	1%
.dex	586 548	99%	.dat	158 829	6%

RQ2: *APK Size and Composition Evolution:*

Size Evolution. Figure 3 shows the size evolution of APK files across all the SDK versions from our dataset. The figure shows an upward trend, signifying bigger apps. This does not necessarily mean more code since apps are also made of different resources, such as pictures and videos. However, Figure 4 shows that there is also an upward trend in the size of DEX files (Dalvik bytecode) inside APKs, which means that developers use more and more code in their apps.

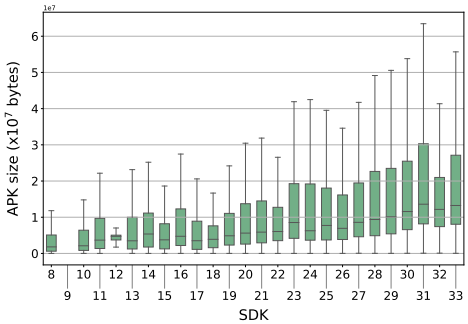

Fig. 3: APK Size Evolution.

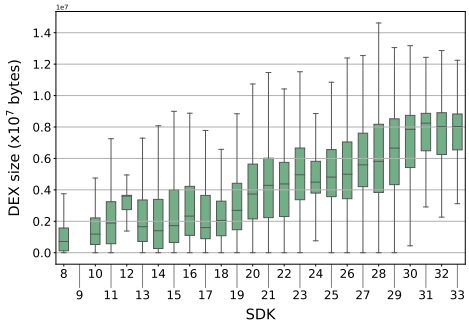

Fig. 4: Dex Size Evolution.

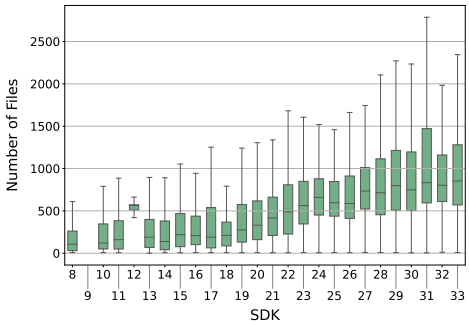

Fig. 5: Distribution of # Files per SDK.

File Distribution and Evolution. Figure 5 highlights the increasing number of files found within APKs for each SDK release. This upward trend could be linked to new techniques adopted in development, such as the usage of new and bigger libraries, as well as the increasing complexity of Android apps.

Figure 6 underscore the importance of the "binary" encoding in the composition of Android apps. This conclusion is expected given that ".dex" files, compiled ".xml" files, and ".so" files are "binary" files. Additionally, it is worth mentioning the sudden raise in the number of "us-ascii" files since SDK version 26.

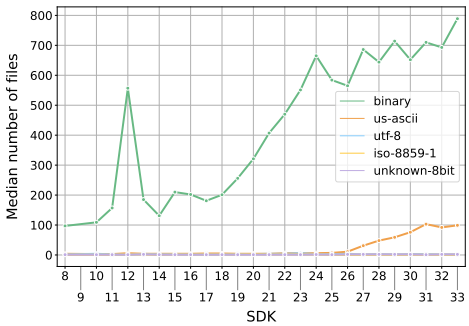

Fig. 6: Encoding Median Count Evolution.

Figure 7 shows the distribution of the number of files per APK for each MIME type, of which the median is not 0. Results indicate that a high usage of pictures in apps that can be attributed to the usage of images as icons, buttons,

and other graphical elements. The results also show that apps mainly comprise images and binary files (octet-stream files).

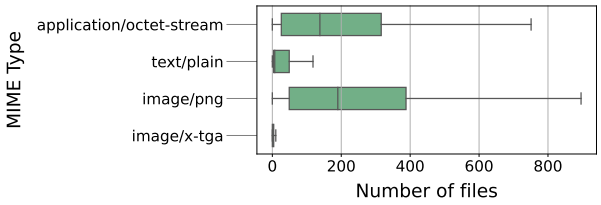

Fig. 7: MIME Type Count Distribution.

Figure 8 shows the evolution of the median number of files for the top 5 magic types in our dataset. We note: ❶ a steady rise of the number of Android binary XML; and ❷ a downward trend in the number of PNGs (for the recent SDK versions).

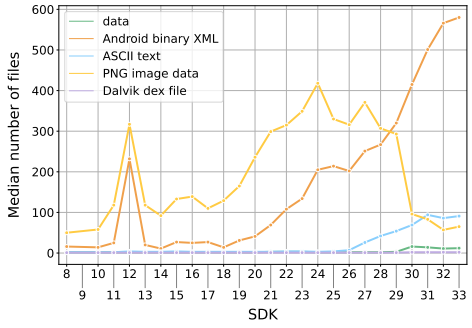

Fig. 8: Magic Type Median Evolution.

We computed a similar plot for the top 5 extensions in our dataset, but we omitted it for space reasons. Note that the results are similar to the results from Figure 8 (the plot can be found here [26]).

RQ2 answer: ❶ Over time, the size of APK files is growing steadily, as well as the number of files found within them; ❷ APKs are mostly composed of binary files (i.e., "code", images, etc.), whose numbers and sizes are also increasing over time; and ❸ The variety of file types evolves differently over time.

IV. Can you find even more files inside APKs?

Our empirical findings of the file types described in the previous section revealed the presence of files that consist of other embedded files. More specifically, these are compressed files (e.g., zip files) that encapsulate files within them. To ensure the comprehensiveness of our analysis, we must consider them in our study.

A. Empirical Setup

In this section, we rely on the dataset used in the previous section, i.e., 410 125 APKs distributed equally over 25 SDKs.

To identify a file as a compressed file, we rely on a simple heuristic: we define a compressed file as a file that can be successfully decompressed with the *Zipfile* library [25].

B. Research Questions

Section IV aims to answer the following RQ:

RQ3: To what extent do APKs contain compressed files?

C. Empirical Findings

Table VI shows statistics about compressed files found in our dataset. The first line reports the number of *Compressed files* we found, i.e., 223 370 in 35 462 apps. The second line reports the number of APKs that we found inside another APKs (an APK is a compressed file, a ZIP file more precisely). Finally, the last line reports the number of files that we found in the compressed files once the files are decompressed. The Count column is the number of these files we found in our dataset. The Presence column is the number of APKs in our dataset in which we found these files.

TABLE VI: Statistics about Compressed Files.

	Count	Presence
#Compressed files within APKs	223 370	8.65% (35 462)
#APKs within APKs	2889	0.37% (1509)
#Files within compressed files	6 415 488	8.58% (35 199)

Our empirical results reveal the following: ❶ roughly 9% (35 462) of the APKs of our dataset contain compressed files, highlighting the common adoption of compression techniques in Android apps; ❷ Over 6 million files have been newly discovered within the compressed files; and ❸ 1509 APKs contain another APK. This practice is especially concerning as it could provide a pathway for malicious actors to introduce unverified or harmful software onto users' devices.

To further investigate the first two findings, we computed Table VII and Table VIII. Table VII shows the top 20 most frequently used extensions of compressed files. It is not a big surprise that the ".zip" extension is the most used. It is, however, of particular interest that ".jar" and ".apk" are the second and sixth most common extensions. This presence can be attributed to their roles in Android apps. Indeed, both could be used to trigger code that is often not analyzed by existing static analyzers.

TABLE VII: Top 20 Compressed.

Extension	Count	Presence	Extension	Count	Presence
.zip	164 680	5% (22 832)	.mpkg	972	0.03% (108)
.jar	17 770	1% (4881)	.mcaddon	712	0.05% (199)
.xms	7806	0.001% (6)	.amr	542	0.1% (404)
.thmx	6283	0.1% (496)	.mcpack	506	0.03% (142)
.dll	3743	0.06% (256)	.mp3	498	0.07% (280)
.apk	2889	0.4% (1509)	.zw	432	0.008% (31)
.dat	2798	0.3% (1133)	.efa	415	0.0005% (2)
.so	2550	0.4% (1487)	.gtl	405	0.0007% (3)
.bn	1326	0.0005% (2)	.jet	397	0.04% (164)
.epub	1142	0.09% (389)	.cs	393	0.001% (4)

Table VIII shows the top 20 most used extensions of files found inside compressed files. Most of the files represent Java classes, pictures, and text data.

Since an APK can theoretically be extracted from another APK and installed on a given device, we decided to further investigate this mechanism since it could be used by malicious actors. To do so, we relied on VirusTotal [27], an online

TABLE VIII: Top 20 Within Compressed.

Extension	Count	Presence	Extension	Count	Presence
.class	3 008 136	1.00% (4087)	.lua	41 989	0.03% (114)
.png	1 460 133	5.23% (21 466)	.svg	27 308	0.16% (660)
.xml	234 998	1.51% (6207)	.sig	26 559	0.003% (13)
.json	195 666	0.56% (2309)	.dat	24 796	0.59% (2433)
.js	179 795	0.88% (3600)	.css	19 976	0.91% (3749)
.html	162 081	2.77% (11 374)	.mf	19 439	1.87% (7651)
none	136 966	2.13% (8749)	.java	18 724	0.10% (422)
.jpg	125 016	0.96% (3948)	.webp	18 130	0.01% (47)
.txt	115 201	1.08% (4444)	.ogg	17 542	0.09% (370)
.mp3	98 874	0.13% (539)	.diff	17 303	0.006% (25)

platform that analyzes files and URLs using more than 70 antiviruses. More specifically, we compute a VirusTotal score for each of the 1509 APK that contain another APK. This *VirusTotal score* is the number of antivirus from VirusTotal that flag the APK as malware. Note that different VirusTotal scores can be used as thresholds to flag an APK as malicious. The minimum threshold is 1, indicating that as soon as a single antivirus identifies the APK as malware, we will determine it as malicious. This approach can lead to many false positives since the conclusions from a single antivirus can be mistaken. In the literature, 5 or 10 antiviruses are usually used as thresholds, making the results more reliable (the more antiviruses flag an app, the more said app can be considered malicious).

Figure 9 reports the VirusTotal score related to the 1509 APKs containing APK(s). Results indicate that 69 APKs (4,57%)–out of the 1509 APKs that contain another APK(s)–are flagged by at least 10 antivirus from VirusTotal. In perspective to the number of APKs from our initial dataset that have a VirusTotal score higher or equal to 10, i.e., 68 221 out of 8 433 849 (0,8%), we can conclude that an APK inside another APK is a sign of maliciousness.

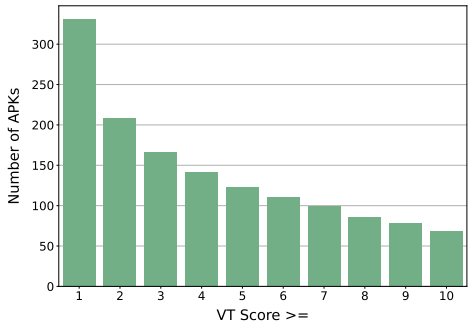

Fig. 9: VT Score of APKs Containing APK(s).

RQ3 answer: ❶ Almost 10% of the analyzed APKs contain compressed files; and ❷ APKs that contains another APK is a sign of maliciousness.

V. Disguised Files

Previously, we analyzed the composition of APKs in terms of quantity. In this section, we examine file consistency by verifying whether the extensions match their actual types (e.g., confirming if a .png file is a PNG image). Indeed, file extensions are widely used by analysts and static analyzers [28], [29] to identify files and include them in the analysis[1]. Nevertheless, these extensions are easily manipulable, potentially leading to incomplete analyses. These incomplete analyses might be overlooking potential threats hidden inside Android apps. Previous works [1], [2], [3] have indeed shown that some malware hides their malicious code in picture resources in Android apps. However, malicious code could be hidden in any file [30] within an app, and any file can be disguised as another file. This section presents the first large-scale empirical study to define and find within Android apps both discrepant files (i.e., files whose extension is not related to any file type, e.g., an image file using the extension ".mypng") and disguised files (i.e., files whose extension is related to a file type different than its own, e.g., a Dalvik dex file with the extension ".png"). Using these findings, we study the effect these files cause on an APK's maliciousness, their plausible security implications on Android devices, and whether this practice should be allowed during the development of Android apps.

A. Empirical Setup

In this section, we rely on two of the datasets defined in Section III, specifically on the distributed dataset (410 125 APKs) and the filtered dataset (8 433 849 APKs). The distributed dataset will allow us to draw conclusions from the trends followed by discrepant and disguised files. Meanwhile, the filtered dataset provides a larger number of APKs and files from Google Play to study.

To reveal discrepant and disguised files, one has to understand the relation between the extension, the MIME type, the encoding, and the magic type of a given file. To the best of our knowledge, no existing list maps these three pieces of information together (although there exist some non-comprehensive maps[2] which link some MIME types to their extension [31], [32]). Hence, we decided to create this mapping with our dataset. The final goal is to have a mapping informing us that, for instance, a file with the ".dex" extension will always be associated to the encoding "binary", the MIME type "application/octet-stream", and the magic type "Dalvik dex file". With this map, it is straightforward to detect inconsistencies between the file extension and the file type.

1) Mapping Construction: As a starting point for the mapping, we use the distributed dataset obtained in Section III. We define the notion of **relation**, which associates a triplet (encoding, MIME type, magic type) to an extension. Table IX presents an example of a relation.

TABLE IX: Structure of a Relation.

Type			Extension
Encoding	MIME Type	Magic Type	
`binary`	`application/octet-stream`	`Dalvik dex file`	`.dex`

[1]Soot initially relies on the method `getAllDexFilesInDirectory()` to locate all Dalvik dex files using the ".dex" extension

[2]Both maps are missing one of the key extensions of Android apps, i.e., ".dex", highlighting their limitations

Then, for each file of each APK, we compute the relation, i.e., we collect its MIME type, its extension, its encoding, and its Magic type, and eventually aggregate the results. We obtain the map illustrated in Figure 10. We can see for example, that the triplet *(binary|image/png| PNG image data)* is associated with 119 497 080 ".png" files, 64 788 ".jpg" files, etc. Overall, this triplet is associated with 1116 extensions (".bimg" being the less common). Overall, our map contains 1465 triplets, and 12 744 unique extensions. On average, a triplet is associated with 16 extensions (median equals 2). The maximum number of extensions for a given triplet is 3616.

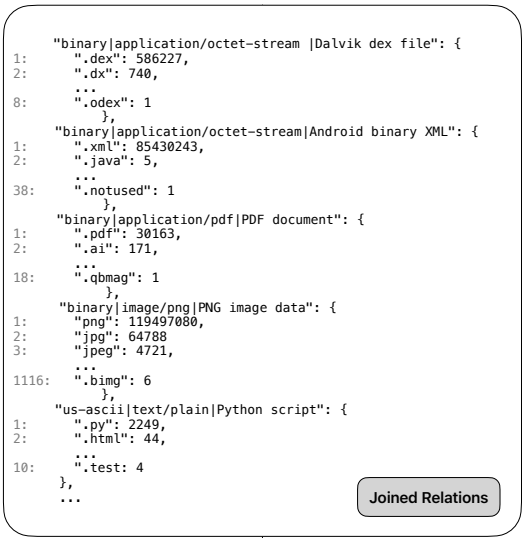

Fig. 10: An Extract of the Relation Map.

As seen in Figure 10, in many cases, like for "binary|image/png|PNG image data", the number of associated extensions is extensive (1116 in this case). Achieving accurate mapping where only the proper extensions are associated with its types would require a huge manual effort, which falls out of the scope of this study. As a result, we opted to manually select only a limited subset of keys, trying to cover the overall diversity of types, and the author manually filtered them[3]. This results in the map illustrated in Figure 11. This map has the same structure as the one from "Joined Relations" (i.e., Figure 10) but, with a more limited number of keys and associated values. Our filtered relations map contains 63 triplets and 100 unique extensions. On average, a triplet is associated with 2 extensions. The maximum number of extensions for a given triplet is 18.

Disclaimer: our goal is not to yield a comprehensive mapping (triplet, extension), which would require extensive work and is out of this paper's scope. Our goal is to propose a mapping that is "good enough" to draw valuable conclusions.

2) Flag Process: To flag a file f as unknown, non-disguised, discrepant, or disguised, we collect f's MIME type, encoding, magic type, and extension. Then, we build f's triplet (that we also called "type") and compare it with our previously built mapping. There are four possible outcomes: ❶ *Unknown*: if the type (i.e., the triplet) is not present in the map, we cannot label the relation as disguised or non-disguised due to the lack of information; ❷ *Non-disguised*: if we find the type in the map, and the extension matches any of the extensions linked to said type. ❸ *Discrepant*: if the type is present in the map, and the extension is not present in said map; and ❹ *Disguised*: if the type is present in the map, the extension does not match any of the extensions linked to said type, and the extension is present under a different type in said map. An example for every outcome can be seen in Figure 12.

[3]FileInfo complemented with Google searches were used during the manual check process

Fig. 11: Extract of the Mapping Relations after Manual Filtering.

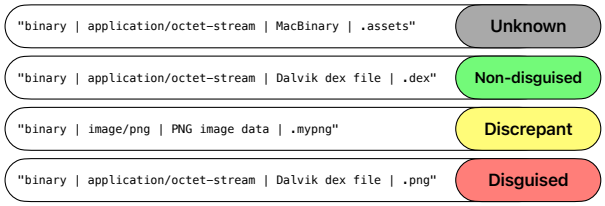

Fig. 12: Examples of Flagged Files.

Finally, we performed additional checks on the identified disguised files. In particular, we check if these files can pose security issues since their extension is inconsistent with their triplet type. To perform these checks, we rely on ❶ static analysis; ❷ dynamic analysis; ❸ ChatGPT; and ❹ manual analysis.

B. Research Questions

Section V aims to answer the following RQs:

RQ4: To what extent are discrepant and disguised files prevalent in APKs?

RQ5: To what extent do discrepant or disguised files hint at maliciousness?

C. Empirical Findings

This section presents an analysis of the flagged files found in our filtered dataset, containing 8 433 849 APKs (containing 6 842 234 526 files).

TABLE X: Statistics of the Files Flagged during our Study (VT: VirusTotal).

Flag	Count	Presence	Presence if VT Score	
			>= 5	>= 10
Total	6 842 234 526	8 433 849 (100%)	175 871 (2%)	68 221 (0.8%)
Unknown	1 286 118 871	8 430 112 (100%)	175 727 (2%)	68 154 (0.8%)
Non-disguised	5 553 004 470	8 430 112 (100%)	175 727 (2%)	68 154 (0.8%)
Discrepant	2 665 238	1 075 775 (13%)	14 935 (1%)	4935 (0.5%)
Disguised	445 947	48 835 (0.6%)	3108 (6%)	1994 (4%)
Disguised SO	2810	1638 (0.02%)	1185 (72%)	986 (60%)
Disguised DEX	99	99 (0.001%)	12 (12%)	6 (6%)

RQ4: *Discrepant and Disguised Files Frequency:*

Unknown Files. As seen in Table X, we encounter "Unknown" files in every APK within our filtered dataset. Indeed, on average, they represent almost 20% of all files found inside APKs. These files–not only–result from an incomplete mapping during our manual filtration step. We are purposefully ignoring the rest of the types by only keeping a handful of types with their corresponding extensions. The nature of these files is unknown in our study, so we cannot infer any intention from these files.

Non-Disguised Files. These files represent the files we are confident that are "Non-Disguised". Indeed, since the map was filtered manually, all associations between type and extension found in it have been reviewed. As expected, Table X shows that most files inside APKs are "Non-Disguised"; in fact, over 80% of all files have a correct relation between their type and extension, i.e., they are consistent.

Discrepant Files. Table X shows that "Discrepant" files, while not present in every APK, are still numerous. Also, as depicted by Figure 13, while in general, the number of "Discrepant" files have remained horizontal on early SDK versions –except for some peaks which are due to a limited number of APKs which contain a large number of discrepant files–, we can observe a clear and steep upward trend on the last SDKs. This trend could be given by developers using custom extensions during development. Nevertheless, this practice can lead to unsafe apps since static analyzers might overlook these files.

Disguised Files. As seen in Table X, the number of disguised files is limited, especially compared to the total number of files. Nevertheless, while they are few and only present in a small subset of APKs, they are worth investigating. Indeed, a "Disguised" file represent a file whose extension belongs within a different file type; therefore, the developer who modified said extension did it to hide the real nature of the file and to make it look like something else entirely different. While we do not know the reason why the developer intentionally changed the extension of the file, we hypothesize that some potential explanations are to either protect her code or to hide malicious behavior. We will check whether disguised files are related to maliciousness in the next RQ.

Figure 14 represents the number of "Disguised" files per SDK. We found "Disguised" files in all SDKs of our dataset. Additionally, we can observe a slight upward trend on the last SDKs, similar to the one for "Discrepant" files, although not as pronounced.

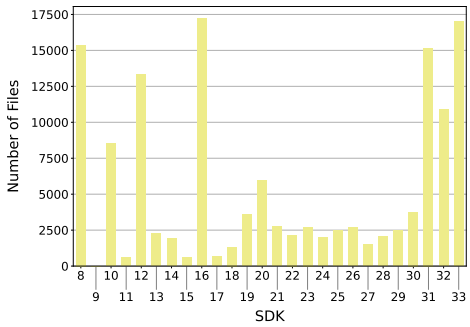

Fig. 13: Number of Discrepant Files Evolution.

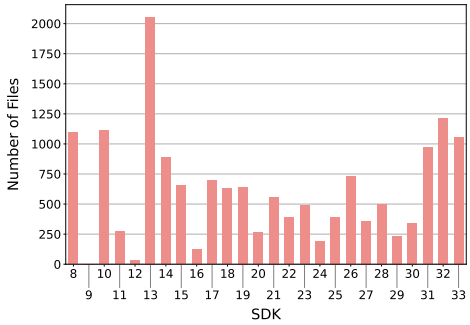

Fig. 14: Number of Disguised Files Evolution.

RQ4 answer: Our results show that: ❶ Over 10% of APKs of our dataset contain files with an extension unrelated to any file type, suggesting that such discrepancies are a fairly common practice; ❷ the number of discrepant files grew drastically over the last SDK versions; and ❸ in our dataset, we discovered that over 400 000 files have extensions that are not conventionally associated with their real file types.

RQ5: *Discrepant and Disguised Files Malicious Impact:*

Effect of Flagged Files on Maliciousness. The right part of Table X presents the VirusTotal score of the APKs containing the files. We can see that among all APKs analyzed, 68 221 (0,8%) have a VirusTotal score higher or equal to 10. This ratio is similar when an APK contains a discrepant file (0,5%). However, the percentage of APKs that are flagged by VirusTotal when containing disguised files is significantly higher (4%). Additionally, this percentage further increases when the disguised file has an extension ".dex" or ".so" to reach 6% and 60%, respectively. This result shows that disguised files are "a sign" of maliciousness. Especially for native (.so) and DEX code, which can be used by attackers to load malicious code surreptitiously.

Disguised DEX files[4]. Based on the results that we obtained from Table X, we focused on trying to further assess whether the presence of disguised DEX files is a sign of maliciousness.

[4]We define a disguised DEX file as a disguised file with the type "binary|application/octet-stream|Dalvik dex file" but associated with an extension different from ".dex" (e.g., a DEX file with a ".png" extension).

This would support our claim that, indeed, analysts and static analysis tools must be wary of disguised files as they might be hiding malicious intents. To determine the effect on maliciousness of said disguised DEX files we followed four separate approaches.

Initially, we focused on static and dynamic analysis to answer our RQ.

❶ *Static analysis*: Our idea was to rely on Soot [28] and FlowDroid [29], two static analyzers for Android apps, to detect data leaks. We first checked if both tools handle disguised DEX files. We found that they do take into account the disguised DEX files (according to their documentation and source code). However, we tried a small experiment: we used Soot and FlowDroid to compute the call graphs of the APKs with and without the disguised DEX files. The results indicate that these disguised DEX files do not have any impact on the call graph, i.e., not a single method from disguised DEX files was included in the call graph. We make the hypothesis that malicious developers rely on mechanisms that hinder static analysis tools to reach the disguised code, such as dynamic class loading, which Soot and FlowDroid do not handle.

❷ *Dynamic analysis*: given our previous hypothesis that disguised DEX files could be dynamically loaded at runtime, we executed the APKs and observed their runtime behavior. To do so, we relied on AndroLog [33], an Android instrumentation tool that adds log statements into app code (even in disguised files). Then, we exercised the APKs with Monkey [34] for 5 minutes. Our goal was to observe if disguised files code would trigger any malicious payload. Unfortunately, given the pseudo-random nature of Monkey, we could not observe any instance where the disguised code was triggered.

Since our two attempts, one with static analysis and one with dynamic analysis, were inconclusive, we decided to investigate further by relying on ChatGPT and manual analysis.

❸ *ChatGPT* [35]: we used jadx [36] to decompile the disguised DEX files and obtain their source code (i.e., Java files). We provided these Java files to ChatGPT to determine if any behavior within them is potentially malicious[5]. According to ChatGPT-4o, over 40% of the decompiled Java files are possibly malicious.

❹ *Manual analysis*: We used jadx to manually inspect the source code of the dex files. We searched for the disguised DEX file name in the APK source code to find if the APK dynamically loaded the disguised file. Given the tedious nature of the manual analysis, we only manually analyzed a subset of 5 APKs that contain disguised DEX files. After manual inspection, we found the pieces of code that dynamically load the disguised DEX files. Table XI shows an extract of our findings: the first column is the name of the disguised file found in the APK, the second column is the piece of code loading it dynamically. We can see in Table XI how the dynamic load is done with completely different methodologies for each "disguised" file, making it hard to identify as a unique pattern in Android source code.

We further inspected the 5 disguised DEX files (first column of Table XI) to manually look for malicious code within their Java files. We found suspicious pieces of code and decided to use VirusTotal [27] and ChatGPT [35] to confirm our suspicions. The VT score was 0 for 4 out of the 5 disguised DEX files. The remaining disguised DEX file had a VT score of 1. Therefore, VirusTotal did not find obvious signs of maliciousness inside these disguised DEX files. Nevertheless, according to ChatGPT, the pieces of code found in 4 disguised DEX files (the first 4 in Table XI) where potentially malicious given a series of properties found in the code, such as, dynamic URL fetching, dynamic content loading, enabling/disabling app components without consent, and manipulating Android framework components. The code from the last disguised DEX file was used to define a custom configuration during debugging tasks.

RQ5 answer: Our results show that: ❶ discrepant files have no effect on VirusTotal score; ❷ there is a link between the presence of disguised files in an APK and the probability of said APK being malicious; ❸ this link is even more visible for disguised files which should have a ".dex" or ".so" extension; and ❹ ChatGPT and manual analysis support these results by identifying malicious intent within disguised DEX files.

VI. Threats to Validity

Magic Library. The magic library used in our study might not return precise magic types for some files. Ambiguity in file identification could also affect the study's reliability. To overcome this threat, we used a recent and stable version of Magic library.

Manual Mapping. The mapping used to flag the different files found within the APKs was manually built. Therefore, there are some inherent limitations. A complete mapping would remove unknown files since all types should be mapped. These unknown files would become non-disguised, discrepant, or disguised. Some discrepant files might also become disguised if their extensions appear under newly included types. As mentioned in our paper, building a complete mapping is out of the scope of our research.

Compressed Files. The Zipfile library used to search for compressed files does not support *all* compression algorithms. Therefore, we might have missed compressed files in our study. This threat is mitigated since discovering more compressed files would exacerbate our results on the fact that static analysis should not only account for DEX and Android Manifest files.

VII. Related Work

This section provides an overview of previous work related to our study.

Static Analysis. Static analysis is key in ensuring the security of Android apps, permitting an in-depth examination of code

[5]The prompt given to ChatGPT was: "I will provide Java source code. You have to analyze it and answer a series of questions. First, you need to determine if the source code might be potentially malicious."

TABLE XI: Dynamic Load of Disguised DEX Files (Package Name of their APKs in Footnotes).

Disguised file	Dynamic load statement(s)	Trigger
yui.gif[a]	`Class loadClass = Sport.Companion.path1(getAssets().open("yui.gif"))...);`	onCreate()

back.png[b]	`private static String g = "back.png";`
	`f = new File(this.a.getDir("dex", 0), g);`
	`InputStream open = this.a.getApplicationContext().getAssets().open(g);`
	`FileOutputStream fileOutputStream = new FileOutputStream(f);`
	`while (true){`
	`  int read = open.read();`
	`  fileOutputStream.write(bArr, 0, read);`
	`}`
	`Class loadClass = new DexClassLoader(f.getAbsolutePath(), ...);`
		onResume()

ares.ttf[c]	`InputStream open = context.getAssets().open("ares.ttf");`
	`while (true){`
	`  int read = open.read();`
	`  byteArrayOutputStream.write(read);`
	`}`
	`ByteBuffer wrap = ByteBuffer.wrap(byteArrayOutputStream.toByteArray());`
	`instance = (DynamicInvoker) new InMemoryDexClassLoader(wrap, ...);`
		onPostResume()

libclasses.so[d]	`String dexpath = context.getApplicationInfo().nativeLibraryDir +`
	`  File.separator + "libclasses.so";`
	`File dexfile = new File(dexpath);`
	`File dest = new File(baseDir, "shell.dex");`
	`copyFileUsingFileStreams(dexfile, dest);`
	`DexFile result = DexFile.loadDex(dest.getAbsolutePath(), ...);`
		attachBaseContext()

uio.so[e]	`private static final String dex1 = "uio.so";`
	`File dex12 = new File(context.getDir("dex_support", 0) + dex1);`
	`DexFile dexFile = new DexFile(dex12);`
		attachBaseContext()

[a]com.christopherbarrett.wildmustanggallop [b]net.wallpapersmobile.foryouthebotsquadpuzzlebattles [c]fb.nckq.zoqg.xnqg.zkgq [d]com.taobao.live [e]com.happy_tp.android

without its execution. Many works have been designed and presented to account more than the Dalvik bytecode in Android apps. For instance, FlowDroid [29], which offers precise taint analysis, also reads side files such as resources and the AndroidManifest. Several works have been presented to also account for the binary code present in the "lib" folder in Android apps. For instance, DroidNative [37], JN-SAF [38], NDroid [39], and JuCify [40] account for native code to allow more comprehensive static analysis. Several works have also developed techniques to improve static analysis via the introduction of JavaScript code [41], [42]. More recently, ReuNify [43] was presented as a static analysis solution to account for the React Native framework.

Compared to these works, our study provides a comprehensive view of all types of files present in APKs. Static analyzers should extend their analysis by taking into account the various types of files our study highlights.

Android app comprehension. Android app comprehension is an extensive field with significant contributions from state-of-the-art research. Some of these papers focus on studying apps using the attributes obtained from the market [11], [12], [13], [14]. Meanwhile, many state-of-the-art papers [15], [16], [17], [18], [19] analyze the composition of Android apps while trying to find clear markers of maliciousness. Additionally, some papers [44], [45] study the structure of Android apps to gain insights into them.

Our paper introduces a novel perspective in the study of Android app composition by focusing on the attributes (e.g., type, extension, etc.) of the files found in Android apps.

Disguised files. The practice of disguising malicious files as seemingly harmless by deliberately modifying their extensions has been a subject of extensive study. Numerous state-of-the-art papers [46], [47], [48] shed light on the prevalence of these disguised files, especially as email attachments. Moreover, some papers offer innovative solutions to address the issue beyond email attachments. For instance, BrowserGuard [49] is designed to safeguard users from drive-by-downloads disguised files, providing an extra layer of protection in web security. Malicious actors also rely on steganography (a technique where additional data is concealed within the original one) to introduce malicious code through apparently inoffensive files [1], [2], [3], [30].

We introduce an innovative approach to identify disguised files in Android apps.

Compressed files. The paper by Yan et al. [50] delves into the method employed by malicious actors who utilize packers to evade antivirus detection. This technique, involving the concealment of malware within compressed files, has drawn the attention of several state-of-the-art papers [51], [52], [53], [54], which explore the challenges associated with this practice.

We investigate the prevalence of compressed files in Android apps.

VIII. Conclusion

We conducted an empirical study to dissect and understand the composition of Android apps. Our experimental results indicate that Android apps are made of a wealth of different types of files, that developers use many unconventional extensions for their files, and that APKs can contain compressed files, even including other APKs. Finally, our study shows that several file extensions are not consistent with their type, and this fact is an indicator of maliciousness.

IX. Acknowledgement

This research was funded in whole, or in part, by the Luxembourg National Research Fund (FNR), grant references REPROCESS Project (16344458) and UNLOCK Project (18154263).

REFERENCES

[1] G. Suarez-Tangil, J. E. Tapiador, and P. Peris-Lopez, "Stegomalware: Playing hide and seek with malicious components in smartphone apps," in *Information Security and Cryptology: 10th International Conference, Inscrypt 2014, Beijing, China, December 13-15, 2014, Revised Selected Papers 10*. Springer, 2015, pp. 496–515.

[2] P. Manev, "Hunting for malware masquerading as an image file, urlhttps://www.stamus-networks.com/blog/hunting-for-malware-masquerading-as-an-image-file," *Stamus Networks*, 2022. [Online]. Available: https://www.stamus-networks.com/blog/hunting-for-malware-masquerading-as-an-image-file

[3] K. Zanki, "Malware in images: When you can't see 'the whole picture', https://www.reversinglabs.com/blog/malware-in-images," *Reversing Labs*, 2021. [Online]. Available: https://www.reversinglabs.com/blog/malware-in-images

[4] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Trans. Priv. Secur.*, vol. 21, no. 3, apr 2018. [Online]. Available: https://doi.org/10.1145/3183575

[5] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein, "Raicc: Revealing atypical inter-component communication in android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 1398–1409. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00126

[6] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, vol. 15, 2015, p. 110.

[7] D. Wu, D. Gao, R. H. Deng, and C. Rocky K. C., "When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern android apps in backdroid," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 543–554.

[8] F. Pauck and H. Wehrheim, "Jicer: Simplifying cooperative android app analysis tasks," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 187–197.

[9] E. Blázquez and J. Tapiador, "Kunai: A static analysis framework for android apps," *SoftwareX*, vol. 22, p. 101370, 2023.

[10] J. Samhi, L. Li, T. F. Bissyande, and J. Klein, "Difuzer: Uncovering suspicious hidden sensitive operations in android apps," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 723–735. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3510135

[11] H. Wang, H. Li, and Y. Guo, "Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play," in *The World Wide Web Conference*, ser. WWW '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1988–1999. [Online]. Available: https://doi.org/10.1145/3308558.3313611

[12] H. Wang, Z. Liu, J. Liang, N. Vallina-Rodriguez, Y. Guo, L. Li, J. Tapiador, J. Cao, and G. Xu, "Beyond google play: A large-scale comparative study of chinese android app markets," in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 293–307. [Online]. Available: https://doi.org/10.1145/3278532.3278558

[13] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 221–233. [Online]. Available: https://doi.org/10.1145/2591971.2592003

[14] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 817–847, 2017.

[15] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[16] G. Shrivastava and P. Kumar, "Sensdroid: analysis for malicious activity risk of android application," *Multimedia Tools and Applications*, vol. 78, no. 24, pp. 35 713–35 731, 2019.

[17] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "Droidnative: Automating and optimizing detection of android native code malware variants," *computers & security*, vol. 65, pp. 230–246, 2017.

[18] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *2012 Seventh Asia joint conference on information security*. IEEE, 2012, pp. 62–69.

[19] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, 2013, pp. 45–54.

[20] A. Hupp. (2001) python-magic, https://pypi.org/project/python-magic. Python Package Index. [Online]. Available: https://pypi.org/project/python-magic/

[21] Mozilla. (2023) Mime types, https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types. Mozilla.

[22] IBM. (2018) What is a magic number?, https://www.ibm.com/support/pages/what-magic-number. IBM. [Online]. Available: https://www.ibm.com/support/pages/what-magic-number

[23] Microsoft. (2021) File encodings, https://learn.microsoft.com/en-us/dotnet/visual-basic/developing-apps/programming/drives-directories-files/file-encodings. Microsoft. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/visual-basic/developing-apps/programming/drives-directories-files/file-encodings

[24] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[25] "Zipfile library https://docs.python.org/3/library/zipfile.html," 2023, accessed November 2023.

[26] Anonymous, "Extension median count evolution," 2024. [Online]. Available: https://anonymous.4open.science/api/repo/SANER2025/file/Plots/Extension_Median_Count_Evolution.pdf

[27] J. Baker, "What is google's virustotal?, https://bestantivirus.com/blog/what-is-googles-virustotal.html," *Best Antivirus*, 2020. [Online]. Available: https://bestantivirus.com/blog/what-is-googles-virustotal.html

[28] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, p. 13.

[29] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, p. 259–269, jun 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594299

[30] D. Stevens, "Malicious pdf documents explained," *IEEE Security & Privacy*, vol. 9, no. 1, pp. 80–82, 2011.

[31] A. Fisher, "Github gist: Maps file extensions to mime types," 2015, accessed: October 2024. [Online]. Available: https://gist.github.com/adamfisher/16fe8c619ea389944d0f

[32] magic, "@magic/mime-types," 2023, accessed: October 2024. [Online]. Available: https://github.com/magic/mime-types/tree/master

[33] J. Samhi and A. Zeller, "Androlog: Android instrumentation and code coverage analysis," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 597â€“601. [Online]. Available: https://doi.org/10.1145/3663529.3663806

[34] Google, "Ui/application exerciser monkey," 2024, accessed August 2024. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey

[35] OpenAI, "Chatgpt: Gpt-4o conversational agent," 2024, accessed August 2024. [Online]. Available: https://chat.openai.com/

[36] J. Team, "Jadx: Dex to java decompiler," 2024, accessed August 2024. [Online]. Available: https://github.com/skylot/jadx

[37] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "Droidnative: Automating and optimizing detection of android native code malware variants," *computers & security*, vol. 65, pp. 230–246, 2017.

[38] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1137–1150.

[39] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan, "Ndroid: Toward tracking information flows across multiple android

contexts," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 814–828, 2018.

[40] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "Jucify: A step towards android code unification for enhanced static analysis," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 1232–1244. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3512766

[41] J. Bai, W. Wang, Y. Qin, S. Zhang, J. Wang, and Y. Pan, "Bridgetaint: a bi-directional dynamic taint tracking method for javascript bridges in android hybrid applications," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 677–692, 2018.

[42] S. Lee, J. Dolby, and S. Ryu, "Hybridroid: static analysis framework for android hybrid applications," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 250–261.

[43] Y. Liu, X. Chen, P. Liu, J. Grundy, C. Chen, and L. Li, "Reunify: A step towards whole program analysis for react native android apps," *arXiv preprint arXiv:2309.03524*, 2023.

[44] H. Cai and B. Ryder, "A longitudinal study of application structure and behaviors in android," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2934–2955, 2020.

[45] H. Cai and B. G. Ryder, "Droidfax: A toolkit for systematic characterization of android applications," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 643–647.

[46] I. Ghafir, V. Prenosil, M. Hammoudeh, F. J. Aparicio-Navarro, K. Rabie, and A. Jabban, "Disguised executable files in spear-phishing emails: Detecting the point of entry in advanced persistent threat," in *Proceedings of the 2nd International Conference on Future Networks and Distributed Systems*, ser. ICFNDS '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3231053.3231097

[47] N. Nissim, A. Cohen, C. Glezer, and Y. Elovici, "Detection of malicious pdf files and directions for enhancements: A state-of-the art survey," *Computers & Security*, vol. 48, pp. 246–266, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404814001606

[48] E. M. Rudd, R. Harang, and J. Saxe, "Meade: Towards a malicious email attachment detection engine," in *2018 IEEE International Symposium on Technologies for Homeland Security (HST)*. IEEE, 2018, pp. 1–7.

[49] F.-H. Hsu, C.-K. Tso, Y.-C. Yeh, W.-J. Wang, and L.-H. Chen, "Browserguard: A behavior-based solution to drive-by-download attacks," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 7, pp. 1461–1468, 2011.

[50] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," *IEEE Security & Privacy*, vol. 6, no. 5, pp. 65–69, 2008.

[51] P. Lagadec, "Opendocument and open xml security (openoffice. org and ms office 2007)," *Journal in Computer Virology*, vol. 4, no. 2, pp. 115–125, 2008.

[52] F. Daryabar, A. Dehghantanha, and H. G. Broujerdi, "Investigation of malware defence and detection techniques," *International Journal of Digital Information and Wireless Communications (IJDIWC)*, vol. 1, no. 3, pp. 645–650, 2011.

[53] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 40–45, 2007.

[54] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 2007, pp. 431–441.