Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting

Li Li[ξ], Daoyuan Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, David Lo, Lorenzo Cavallaro

Abstract—The Android packaging model offers ample opportunities for malware writers to piggyback malicious code in popular apps, which can then be easily spread to a large user base. Although recent research has produced approaches and tools to identify piggybacked apps, the literature lacks a comprehensive investigation into such phenomenon. We fill this gap by 1) systematically building a large set of piggybacked and benign apps pairs, which we release to the community, 2) empirically studying the characteristics of malicious piggybacked apps in comparison with their benign counterparts, and 3) providing insights on piggybacking processes. Among several findings providing insights analysis techniques should build upon to improve the overall detection and classification accuracy of piggybacked apps, we show that piggybacking operations not only concern app code, but also extensively manipulates app resource files, largely contradicting common beliefs. We also find that piggybacking is done with little sophistication, in many cases automatically, and often via library code.

1 Introduction

Android apps are distributed as software packages (i.e., APK files, which are actually archives in the ZIP format) that include developer bytecode, resource files and a Manifest which presents essential information about the app, such as permissions requested and the list of components, that the system must know about before it can run any of the app's code. Unfortunately, unlike for traditional software executables, Android app package elements can easily be modified by third parties [6]. Malware writers can thus build on top of popular apps to ensure a wide diffusion of their malicious code within the Android ecosystem. Indeed, it may be effective to simply unpack a benign, preferably popular, app and then *graft* some malicious code on it before repackaging it and distributing it for free. The resulting app, which thus piggybacks a malicious payload, is referred to as a *piggybacked* app.

Previous studies, have exposed statistics suggesting that malware is written at an industrial scale and that a given malicious component can be extensively reused in a bulk of malware [39], [50]. These findings support the assumption that most malware might be simply repackaged versions of official applications. Evidence of the widespread use of repackaging by malware writers is provided in MalGenome [50], a reference dataset in the Android security community, where 80% of the malicious samples are known to be built via repackaging other apps. A more recent analysis highlights the fact that even Google Play security checkers are challenged in detecting fake apps[1], providing further evidence that the problem is widespread.

In contrast with common repackaging, where the code of original apps may not be modified, piggybacking grafts additional code to inject an extra, often malicious, behaviour to original apps. A study of piggybacked apps, a specific subset of repackaged apps, can thus contribute in the research directions towards comprehending malware creation, distribution, etc. Indeed, piggybacked apps, because of the introduction of alien code, will present characteristics that analyzers can leverage to locate and investigate malicious payloads of Android malware.

To the best of our knowledge, state-of-the-art works have mainly focused on detecting repackaged apps rather than detecting piggybacked apps. Even so, the problem of detecting piggybacked apps is eventually related to the problem of detecting app clones and repackaged apps. The majority of state-of-the-art works, such as DroidMoss [48] and DNADroid [14], have focused on performing pairwise similarity comparisons of app code to detect repackaged apps. However, because millions of Android apps are now available in markets, pairwise comparison based approaches cannot scale. For example, considering the 2 million apps in Google Play, there would be $C^2_{2*10^6}$ candidate pairs for comparison[2].

To alleviate the scalability problem that exists in such approaches, PiggyApp [47] builds, for every app, a vector with normalized values of semantic features extracted from components implementing the app's primary functionality. Thus, instead of computing the similarity between apps based on their code, PiggyApp computes, as a proxy, the distance between their corresponding vectors. This approach however also remains impractical, since one would require

- [ξ] *The corresponding author.*
- *L. Li, D. Li, T. Bissyandé, J. Klein, and Y. Le Traon are with the Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg.*
  E-mail: li.li@uni.lu
- *D. Lo and L. Cavallaro are with Singapore Management University and Royal Holloway, University of London, respectively.*

Manuscript received XXX; revised XXX.

1. http://goo.gl/kAFjkQ – Posted on 24 February 2016
2. If we consider a computing platform with 10 cores each starting 10 threads to compare pairs of apps in parallel, it would still require several months to complete the analysis when optimistically assuming that each comparison would take about 1ms.

the dataset to contain exhaustively the piggybacked apps as well as their corresponding original apps. In practice however, many piggybacked apps are likely to be uploaded on different markets than where the original app can be found.

Overall, the aforementioned limitations could be overcome with approaches that leverage more knowledge on how piggybacked apps are built. Instead of a brute-force comparison, one could use semantic features that hint on probable piggybacking scenarios. Although piggybacked apps can be taken as repackaged apps, which have been well studied by literature works, the features extracted from repackaged apps may not be representative of piggybacking.

The goal of our work is to extensively investigate piggybacked apps for understanding how piggybacking is performed. To the best of our knowledge, this study is the first work attempting to provide a systematized knowledge on this topic to the community. We make the following contributions:

- We contribute to the research community efforts by building and sharing the first publicly available dataset [1] on piggybacked apps. This dataset, which was collected in a systematic way and includes for each piggybacked app its associated original app, can be used as a reference ground truth for assessing approaches. This dataset will be further systematically and regularly updated as new Android malware samples are collected in the AndroZoo project [2].
- We conduct a comprehensive study of piggybacked apps and report our findings on how piggybacked apps differentiate from benign apps, what actions are performed by piggybackers, what payloads are inserted, etc. Among several insights, we find that piggybacking is done with little sophistication, in many cases automatically, and often via library code.

The remainder of this paper is organized as follows. Section 2 presents a terminology related to piggybacked apps to which we will refer to in the remainder of this paper. Section 3 details how the benchmark dataset of piggybacked apps was collected. Section 4 describes dimensions of this study and all the findings. We discuss the implications of the findings and enumerate the threats to validity in Section 5. Section 6 enumerates related work and Section 7 concludes this paper.

2 Terminology

We now provide a terminology to which we will refer to in the remainder of this paper. There are several terms that have been leveraged in the literature to indicate actions involving app code reuse processes. Particularly, *cloning* is used to describe the process of constructing a program by reusing the functionality of other programs. In the Android ecosystem, it is straightforward to clone an app through *repackaging*. However, repackaging does not necessarily need to modify the bytecode of a given app. Indeed, one can repackage an app without changing anything but to switch the app owner. *Piggybacking* is defined in the literature as an activity where a given Android app is repackaged after manipulating the app content, e.g., to insert a malicious

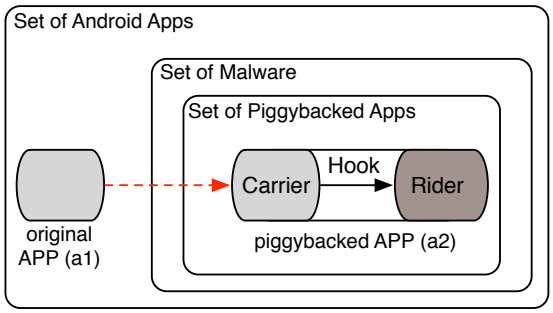

Fig. 1: Piggybacking terminology.

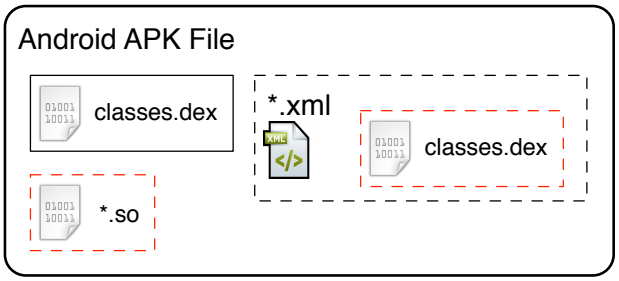

Fig. 2: An example of extra code of a given Android apps, where the extra code is highlighted with red dotted lines.

payload, an advertisement library, etc. Piggybacked apps thus constitute a specific subset of repackaged apps.

Fig. 1 illustrates the constituting parts of a piggybacked app. Such malware[3] are built by taking a given original app, referred to in the literature [47] as the **carrier**, and grafting to it a malicious payload, referred to as the **rider**. The malicious behaviour will be triggered thanks to the **hook** code that is inserted by the malware writer to connect his rider code to the carrier app code. The hook thus defines the point where carrier context is switched into the rider context in the execution flow.

Fig. 2 illustrates the simplified file structure of a given Android app. Besides the standard app code[4], compiled into `classes.dex`, Android apps can include additional code:

- Native compiled code in ELF format (`.so` files in Fig. 2);
- Some extra bytecode in another `dex` file hidden behind the format of a resource file such as XML.

In this work, we refer to any other executable code that is not in the standard `classes.dex` as *extra code* (i.e., all code present in dash rectangles in Fig. 2).

3 Dataset Collection

Research on Android security is challenged by the scarcity of datasets and benchmarks. Despite the abundance of studies and approaches on detection of piggybacked apps, access to the associated datasets is limited. Furthermore, while other studies focus on various kinds of repackaged apps,

3. Our definition of malware is related to the flags of AV products. Typically, (aggressive) adware are also included.

4. Throughout this manuscript, *app code* exclusively refers to artifacts produced by compiling code written in a programming language. App resource files, such as image and metadata files, are not considered as code.

ours targets exclusively piggybacked malware. Finally, related work apply their approach on random datasets and then manually verify the findings to compute performance. Conversely, we take a different approach and automate the collection of a ground truth that we share with the community [1].

3.1 Process

Our collection is based on AndroZoo [2], a large repository of millions of apps crawled over several months from several markets (including Google Play, appchina, anzhi), open source repositories (including F-Droid) and researcher datasets (such as the MalGenome dataset). We follow the process in Fig. 3 to collect the ground truth dataset of piggybacked apps.

First, we send all apps to VirusTotal[5] to collect their associated anti-virus scanning reports. Based on VirusTotal's results, we classify the set of apps to two subsets: one contains only benign apps while the other contains only malicious apps.

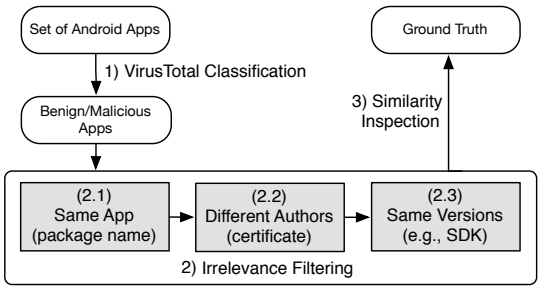

Fig. 3: The ground truth build process.

Second, we filter out irrelevant results, in an attempt to only focus on building piggybacking pairs. This step has further been divided into three sub-steps. In step (2.1), we filter the apps based on Application Package name recorded in the Manifest file of each app, which should identify uniquely the app[6]. Considering the identicality of package names, we were able to focus on a subset of about 540 thousands pairs $< app_g, app_m >$ of benign (app_g) and malicious[7] (app_m) apps sharing the same package name. This step yields our first set of candidate pairs. Before proceeding to the next step, we ensure that for each pair, the creation date of the benign app precede the one of the malicious app. In step (2.2), we do not consider cases where a developer may piggyback his own app to include new payload. Indeed, we consider piggybacking to be essentially a parasite activity, where one developer exploits the work of another to carry his malicious payload. Furthermore, developers may later "piggyback" their own app, e.g., to insert an advertising component to collect revenue, transforming these apps to adware (often classified as malware). We choose to clean the dataset from such apps. Thus, we discard all pairs where both apps are signed with the same developer certificate. This step brings the subset to about 70 thousands pairs. In step (2.3), we focus on cases where piggybackers do not modify version numbers and do not entirely re-implement functionality which would require new SDK versions. By considering pairs where both apps share the same version number and SDK requirements, we are able to compile a promising dataset of 1,497 app pairs where one app potentially piggybacks the other to include a malicious payload.

Finally, in order to build a final ground truth, we must validate the relevance of each pair as a piggybacking pair. To that end, we perform a *similarity analysis* where we expect that, given a pair of apps $< app_g, app_m >$, app_g's code is part of app_m and app_m includes new code to constitute the malicious payload.

3.2 Similarity analysis

Similarity analysis is essential to quantify and qualify the difference (or similarity) in the packages of two apps, notably for a pair of piggybacked apps. Given a pair of apps (app_1, app_2), we compute the following four metrics that have already been adopted by state-of-the-art tools such as Androguard [17], [29]:

- *identical* – a given method including both signature and implementation is exactly the same in both apps.
- *similar* – a given method has slightly changed (at the instruction level) between the two apps, i.e., methods with same signature but with different contents.
- *new* – a method has been added in the piggybacked app, i.e., methods exist in app_2 but not in app_1.
- *deleted* – a method has been deleted from the carrier code when including it in the piggybacked app, i.e., methods exist in app_1 but not in app_2.

Based on these metrics, we can now calculate the similarity score of pair (app_1, app_2) using Formula 1.

$$similarity = max\{\frac{identical}{total - new}, \frac{identical}{total - deleted}\} \quad (1)$$

where

$$total = identical + similar + deleted + new \quad (2)$$

3.2.1 Code to text representation

In order to enable a fast comparison for scaling to large datasets, our similarity analysis builds on a text representation that abstracts the implementation logic of each app method. Given two Android apps, our similarity analysis compares their included methods based on the sequence of statements that they implement. We first pre-define a mapping between statement types[8] and printable characters. For example, a static invocation statement could be mapped to the character *c*. Based on this mapping, every method can be represented at a high level as a short string. In the example of Fig. 4, the content of method *onCreate()* is represented by string *aabcdbe*.

In addition to the fast comparison scheme, applying code-to-text representation for similarity analysis brings in another advantage: it is resilient to simple obfuscation

5. http://virustotal.com, which hosts around 50 anti-virus products from providers such as Symantec, McAfee, Kaspersky.

6. Two apps with the same Application Package name cannot be installed on the same device. New versions of an app keep the package name, allowing updates instead of multiple installs.

7. In this study, we consider an app to be malware if at least one of the anti-virus products from VirusTotal has labeled it as such.

8. https://ssebuild.cased.de/nightly/soot/javadoc/index.html

scenarios that change the names of classes, methods and variables. As cons, the similarity measure is only an approximation. To ensure confidence in these approximations, we have experimentally validated on a small dataset of apps that we obtain are in line with the scores produced by the state-of-the-art Androguard tool [17].

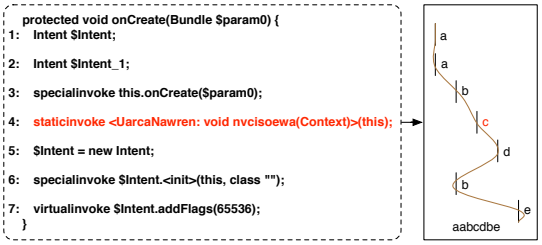

Fig. 4: Illustration of Code to Text transformation. Characters in the right box are representations of the type of statements.

3.2.2 Ground truth inference

Our similarity analysis allowed to consolidate our dataset by validating that apps in a piggybacking pair were indeed very similar in terms of code content; pairs with code similarity of less than a pre-defined 80% threshold are filtered out. Zhou et al. [49] have previously found, through empirical experiments, that 70% is already a good threshold for similarity-based repackaged/piggybacking app detection. In this work, we aim for a more constrained threshold.

We further drop cases of app pairs where no new code has been added (e.g., only resource files have been modified/added). The final set of ground truth is now composed of 950 pairs of apps.

3.2.3 Hook and rider identification

Methods that are found to be similar between a pair of app represent a sweet spot for characterising piggybacking. Given a method m present in two artifacts a_1 and a_2 but differing in their implementation, our similarity analysis takes one step further by performing a fine-grained analysis to localize the added/changed/deleted statements of m in a_2 w.r.t a_1. Such statements are considered as the hooks via which the malware writer connects his malicious code to the original app (a_1). Rider code is then inferred as the set of added methods and classes in the control-flow graph that are reachable from the hook statements.

3.2.4 Implementation

Our similarity analysis is implemented in Java on top of Soot [25], which is a framework for analyzing and transforming Java/Android apps. The code to text representation is conducted at the Jimple level [41], where Jimple is an intermediate representation of Soot. The transformation from Android Dalvik bytecode to Jimple is conducted by Dexpler [7], which now has been integrated into Soot. The main advantage of leveraging Jimple is that we are able to assign a printable character to every statement type because the number of statement types in Jimple is small.

3.3 Dataset Characterization

For each piggybacking pair in our dataset, we collect the metadata information (i.e., creation date, description, categorisation, download count) of the original app by looking in their Manifest file and by crawling description pages from specialized websites[9].

Temporal distribution of our dataset. As illustrated in Fig. 5, our dataset contains piggybacked apps from a period of 6 years. At the time of writing, Androzoo dataset and associated antivirus (AV) reports were limited to apps until mid-2014. We performed our study on the available apps, but committed to continuously update the dataset of piggybacking pairs as new samples get archived in Androzoo. (cf. Section 5.3 on the mitigation of this threat to validity).

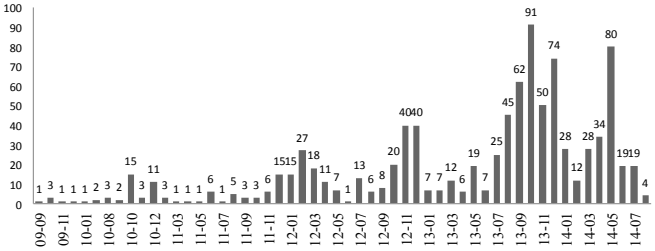

Fig. 5: Temporal Distribution of piggybacked apps in our dataset. (*app creation date* $\equiv$ *dex compilation timestamp)*

Variety of app categories. Statistics of app categories, illustrated in Fig. 6, show that Game apps are the most represented in our piggybacked datasets. A random sampling on our dataset of 2 million apps shows that Games also constitute the largest category of apps. Our dataset of piggybacking pairs further includes piggybacked apps from a variety of 22 categories such as Productivity, Tools, Entertainment, Personalization, Social Networking, Shopping, Sports, Weather, Transportation or News.

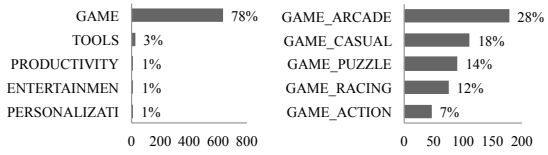

Fig. 6: Top 5 categories (left) and game sub-categories (right) of Android apps used to carry piggybacking payloads.

Distribution channels. Piggybacked apps from our dataset were found in several markets. While many were distributed on alternative markets such as anzhi (71%) and appChina (14%), some are actually spread via the official Google Play market (2.2%).

Popularity of apps. The distribution of download counts of original apps in piggybacking pairs and of a random sample of benign apps from Google Play, shows that mainly popular apps are leveraged by malware writers to carry their malicious payloads (cf. Fig. 7).

Variety of malicious behaviour. We have collected AV reports from around 50 AV engines provided by VirusTotal

9. Notably http://www.bestappsmarket.com

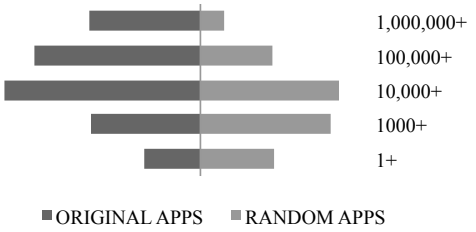

Fig. 7: Download counts of apps leveraged for piggybacking vs. apps randomly sampled in GooglePlay. (Sets are of same size)

for the piggybacked apps. As a proxy to the type/family of malware they are categorized in, we rely on AV labels. Our piggybacked apps were qualified with over 1000 distinct labels by the AVs. Even considering that each AV has its own naming system, this high number suggests a variety of malicious behaviour implemented in the piggybacked payloads.

Piggybacking delay. For each piggybacking pair, we compute the time difference to explore the time delay before an app is leveraged for piggybacking. Distribution [10] in Fig. 8 shows that piggybacking operation can be done on apps of any age, with a median delay of 33 days. On average however, malware writers wait half a year before using a "known" app to spread malicious payload.

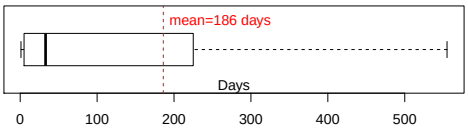

Fig. 8: App age when it is piggybacked (Median=33 days)

Piggybacking pair similarity. Fig. 9 plots the similarity analysis results of our collected dataset. Generally, a small amount of methods in carrier are modified in the piggybacked code and the remaining methods are kept identical. In most cases, piggybacked apps also introduce new methods while piggybacked apps remove methods from the original carrier code in only a few cases.

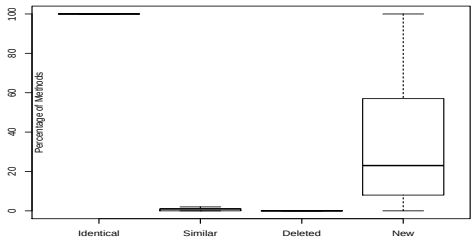

Fig. 9: Overview of the similarity analysis findings for the pairs of apps in our collected dataset.

10. This distribution is presented through a boxplot, which provides a convenient means to visually elucidate groups of numeric data. A boxplot is made up of five horizontal lines. As shown in Fig. 8, from left to right, the five lines (in black) are 1) the least value (MINIMUM), 2) the line where 25% of data values are below (LOWER QUARTILE), 3) the middle point of the data values (MEDIAN), 4) the line where 25% of the data values are above (UPPER QUARTILE), and 5) the greatest value (MAXIMUM).

4 Understanding Piggybacking

Our investigation into Android app piggybacking is carried out through a dissection of Android piggybacked apps. The dissection is conducted mainly in manual, with the support of python and shell scripts. In this section, we overview in Section 4.1 the various dimensions of dissection that we used to answer several relevant research questions for understanding the piggybacking process. Subsequently, we detail our study and the findings in Section 4.2.

4.1 Dimensions of dissection

Our study explores several aspects of Android app piggybacking. Overall we look into:

- ***Which*** *app elements are manipulated by piggybackers?*

An Android application package includes a *classes.dex* file that contains the main code implementing app functionality. Besides this code, the apk file includes icons and background image files, layout description XML files, as well as other resource files such as library Jar files and archives containing extra code.

Image and layout resource files are important for malware writers who must ensure, beyond advertised functionality, that the malicious app has the proper "look and feel". Launcher activity and component lists are also essential elements described in the Manifest file that could be manipulated during piggybacking to ensure that the app is transparently installed on user devices, and that the malicious code is triggered seamlessly. We investigate the extent of these modifications to answer the following research questions:

RQ-01 Are resource files as much concerned by piggybacking operations as app bytecode?
RQ-02 What types of resources are included in the malicious payload?
RQ-03 Are rider code samples redundant across piggybacked apps?
RQ-04 What changes are applied to the Manifest file during piggybacking?
RQ-05 What developer certificates can be found in piggybacked apps?

- ***How*** *app functionality and behaviour are impacted?*

Android apps are made up of four types of components: 1) Activity, which is used to represent the visible part of Android apps; 2) Service, which is dedicated to execute tasks in the background; 3) Broadcast Receiver, which waits for receiving user-specific or system events; and 4) Content Provider, which plays as a standard means for structural data access. These components may not be equally important for spreading malicious payloads.

App components use *Intents* as the primary means for exchanging information. When the Android OS resolves an intent which is not explicitly targeted to a specific component, it will look for all registered components that have *Intent filters* with actions matching the intent action. Indeed, Intent filters are used by apps to declare their capabilities. Piggybacked apps may declare new capabilities to trick users into triggering themselves the malicious behaviour.

Finally, every Android app must be granted the necessary permissions to access every sensitive resource or API

required for its functioning. If the malicious code interacts with more sensitive resources than the original app, new permissions must be requested in the Manifest file.

In the following research questions, we investigate in details the scenarios mentioned above.

RQ-06 Does the malicious payload require extra-permissions to execute?
RQ-07 Which types of components are more manipulated in piggybacking?
RQ-08 Is piggybacking always performed manually for every target popular app?
RQ-09 What kind of actions, encoded in *Intents*, are created by piggybacked rider code?
RQ-10 Are there new sensitive data leaks performed for the piggybacking needs?

- ***Where*** *malicious code is hooked into benign apps?*

To elude detection, malware writers must identify the sweet spots to graft malicious rider code to the benign carrier code. Programmatically, in Android, there are mainly two ways to ensure that rider code will be triggered during app execution. We refer to the two ways as **type**$_1$ and **type**$_2$ hooks. Type$_1$ hooks modify carrier code to insert specific method calls that connect to rider code, which does not need to modify the Manifest file. Conversely, Type$_2$ hooks come in as rider components (need to register in Manifest), which can be launched independently (e.g., via user actions such as clicking, or system events). It is thus important to investigate to what extent piggybackers place hooks that ensure that the rider code will be executed (e.g., with only type$_2$ hooks or both?).

Piggybackers may also attempt to elude static detection of malicious behaviour by dynamically loading parts of the rider code. Finally, rider code may deliver malicious behaviour from various families. We investigate these aspects through the following research questions:

RQ-11 Is the injected rider code complex?
RQ-12 Is the piggybacking rider code limited to the statically accessible code?
RQ-13 How tightly is the rider code integrated with the carrier code?
RQ-14 Are hook code samples specific to each piggybacked app?
RQ-15 What families of malware are spread via piggybacking?

4.2 Findings

In this section we report on the findings yielded by our investigation of the research questions outlined above. For each of the findings, named from **F1** to **F20**, we provide the take-home message before providing details on the analysis.

When a finding involves a comparison of a characteristic of piggybacked app w.r.t original apps, we have ensured that **the difference is statistically significant** by performing the Mann-Whitney-Wilcoxon (MWW) test. MWW is a non-parametric statistical hypothesis test for assessing the statistical significance of the difference between the distributions in two datasets [33]. We adopt this test as it does not assume any specific distribution, a suitable property for our experimental setting. In this paper, we consider a significance level at $\alpha = 0.001$[11].

F1► *The realisation of malicious behaviour is often accompanied by a manipulation (i.e., adding/removing/replacing) of app resource files.*

In our dataset collection, we only considered piggybacking cases where code has been modified to deviate from original behaviour implementation. Thus, we focus on investigating how other app elements are treated during piggybacking. As illustrated in Fig. 10, most (91%) piggybacked apps have added new resource files to the original apps, while only a few (6%) have left the resources files of the original apps untouched.

Fig. 10: Distribution of piggybacked apps that add/remove resource files to/from their original counterparts.

We first investigate the cases where the piggybacked app has removed resource files from the app carrying its malicious payload. Fig. 11 highlights that many image files are removed as well as some Java serialized objects. At a lesser extent, Plain text files, which may contain configuration information, and XML files, which may describe layouts, are also often removed/replaced.

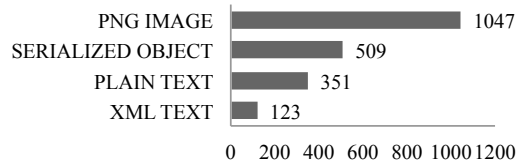

Fig. 11: Top types of resource files removed during piggybacking.

We further investigate the removal of resource files during piggybacking. A common case that we have found is that resource file removal actually corresponds to files renaming. For example, for the piggybacking pair (*A801CF*[12] to *59F8A1*) , six PNG files under the *drawable* directory have been slightly renamed. Besides file names, file extensions (e.g., from PNG to GIF) and parent directories (e.g., from *drawable-480dpi* to *drawable-xxhdpi*) can be changed during piggybacking. This finding suggests that piggybackers attempt to mitigate any possibility of being identified in basic approaches, e.g., through resource-based similarity analysis. We have computed for example the similarity scores in the piggybacking pair (*A801CF* to *59F8A1*), and found that the resource-based similarity score is 72.3% while the code-based similarity score reaches 99.9%.

11. When the null hypothesis is rejected, there is one chance in a thousand that this is due to a coincidence.

12. In this paper, we refer to apps using the last six letters of their SHA256 hash, to enable quick retrieval in Androzoo.

F2► *Piggybacking modifies app behaviour mostly by tampering with existing original app code.*

Although *classes.dex* remains the main site where app functionality is implemented, Android apps can carry extra code that may be invoked at runtime. We investigate the resource files added by piggybacked apps to check that they do not actually constitute the more subtle means used by malware writers to insert malicious payload. Fig. 12 shows that most of the added resources are media data (i.e., audio, video and image files). Extra DEX code have been added in only 8% of piggybacked apps. In 10% of cases, native compiled code in ELF format has been added.

These results suggest that sophisticated piggybacking which manipulates extra code is still limited. In most cases, malware writers simply modify the original app code.

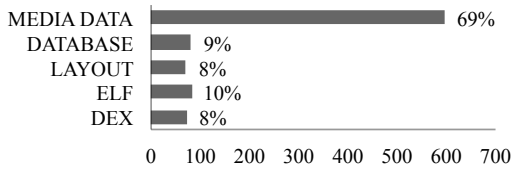

Fig. 12: Distribution of piggybacked apps according to the types of added resource files.

F3► *Piggybacked apps are potentially built in batches.*

In our effort towards understanding app piggybacking, we investigate the developers who sign the certificates of those apps. We found that some certificates have been used for several apps. For example, `RANGFEI.RSA` certificate key appears in 71 piggybacked apps of our dataset, suggesting that the owner of this signature is intensively producing batches of piggybacked apps.

We further consider the case of the 71 piggybacked apps signed with `RANGFEI.RSA` to investigate whether the developer injected similar malicious payloads in the apps. To that end, we first cluster the 71 apps through the Expectation-Maximization algorithm [34] using rider class names as features, yielding 8 clusters. We then investigate how different apps in one cluster are from apps in the others. To that end, we consider the set of labels and compute the Jaccard distance between sets of labels across clusters. We found that the clusters were highly distant (average distance > 0.7), suggesting different payloads in the associated piggybacked apps.

F4► *Piggybacking often requires new permissions to allow the realisation of malicious behaviour.*

For every piggybacking pair, we check in the Manifest files how requested permissions differ. We found that 812 (85%) piggybacked apps have requested new permissions that were not initially requested by their original counterparts. Fig. 13 enumerates the top 10 newly added permissions in our dataset of piggybacked apps. 6 out these 10 permissions are for allowing access to sensitive user/phone information. We note that almost half of the piggybacked apps require the `WAKE_LOCK` permission which enables it to keep device processor from sleeping or screen from dimming. This permission is mostly used by apps that must keep executing tasks even when the user is not actually using his device.

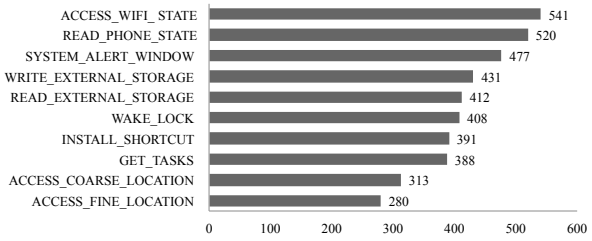

Fig. 13: Top 10 permissions that are newly added.

F5► *Some permissions appear to be more requested by piggybacked apps than non-piggybacked apps.*

From the newly added permissions of Fig. 13, we note that permission `SYSTEM_ALERT_WINDOW`, which is requested by 458 (i.e., $\simeq$50%) piggybacked apps, is only requested by 26 (i.e., 3%) original benign apps (as illustrated in Fig. 14). When considering the top 10 newly added permissions only, we note that they are requested, on average, by 65% piggybacked apps and only 26% benign original apps.

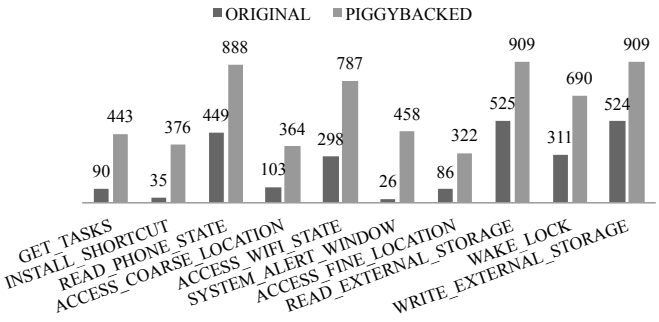

Fig. 14: Distribution of requests of the top 10 permissions across original and piggybacked apps.

Fig. 15 further shows that, excluding outliers, a given piggybacked app requests a minimum of 5, and median number of 7 (out of the 10) from the top permissions list, while original benign apps only request a maximum of 4.

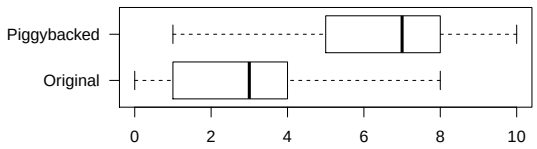

Fig. 15: Distribution of the number of top-10 permissions requested by original apps and piggybacked apps.

F6► *Piggybacking is probably largely automated.*

Since permissions are a key element in Android app programming, a malware writer must systematically ensure that the necessary permissions are granted to its app to enable the working of inserted malicious payloads. We investigate how permissions are added, and note that there seems to be a naive automated process for adding permissions. Indeed, we found that permissions requests are

often duplicated within the same Manifest file. For example, permission READ_PHONE_STATE, which is among the top newly added permissions by piggybacked apps, has been systematically duplicated in 58 apps. Often both permission requests appear consecutively in the Manifest file as illustrated in Listing 1 which is extracted from app *713414*, suggesting that the second permission was not manually added by a developer.

```
uses-perm:"android.permission.ACCESS_WIFI_STATE"
uses-perm:"android.permission.READ_PHONE_STATE"
uses-perm:"android.permission.READ_PHONE_STATE"
uses-perm:"android.permission.GET_ACCOUNTS"
```

Listing 1: Example of duplicated permission declaration.

In our dataset, we have found that there is at least one duplicate permission for 590 piggybacked apps, where 576 of them have a permission existing already in the original apps.

F7► *Piggybacked apps overly request permissions, while leveraging permissions requested by their original apps.*

As illustrated in Fig. 15, and discussed previously, piggybacked apps appear to "systematically" include a larger set of permissions than their original counterparts. We investigate the control-flow of piggybacked apps to check whether they reached the sensitive APIs that are protected by their requested permissions (based on PScout's results [4]). We found that 759 (or 80%) piggybacked apps have declared more permissions than necessary. This finding is inline with previous studies [8] on Android apps, including legitimate apps, which showed that app authors are usually unaware of what exact permissions are needed for the APIs leveraged in their apps. Similarly, our results suggest that piggybacked app writers are not aware of the permissions needed for the APIs accessed by their injected payloads.

Fig. 16 shows that half of the piggybacked apps have requested at least two more permissions that they do not leverage. At the same time, rider code actually use APIs protected by permissions originally requested by original apps.

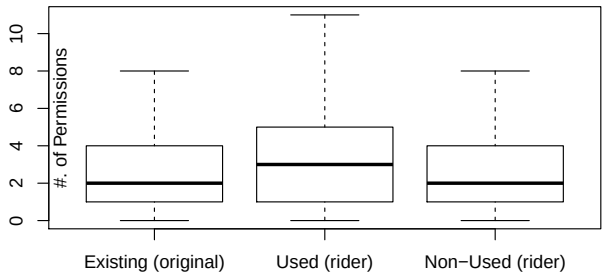

Fig. 16: Distribution of the number of permissions relating to the rider code.

F8► *Most piggybacked apps now include new user interfaces, implement new receivers and services, but do not add new database structures.*

Fig. 17 highlights the statistics of components added by piggybacking. 834 (or 88%) of apps have added Activity components during piggybacking: these represent new user interfaces that did not exist in the original apps. We manually check a random sample of those components and find that they are mostly for displaying advertizement pages. We checked all such apps and their associated original apps and found that they added new ad libraries to redirect the ad revenues to the accounts of piggybackers. Fig. 18 shows that half of the piggybacked apps even add several Activity components. Those components can represent different advertisement libraries. As an example, the piggybacking process that led to app *90D8A5* has brought in two ad libraries: *com.kuguo.ad* and *net.crazymedia.iad*. Although this phenomenon is not very common, it does suggest that piggybackers may take steps to maximize the opportunities of gaining benefits.

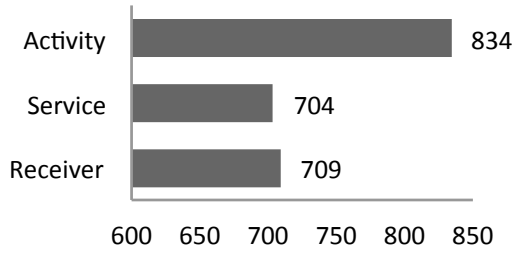

Fig. 17: # of piggybacked apps adding at least one new component.

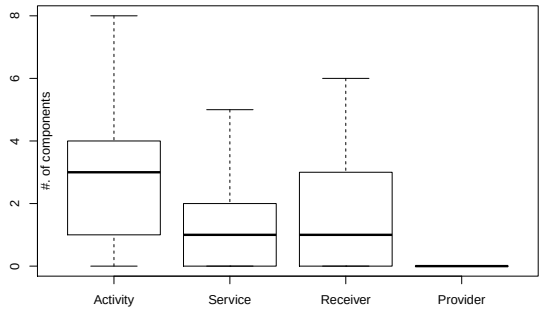

Fig. 18: Distribution of the number of components newly added.

Service and Broadcast Receiver components are also largely injected by piggybacking payload. While Broadcast Receivers register to device events (e.g., WIFI connection is active) so that the malicious code can be triggered, Service components run background tasks which can deplete device resources (cf. recent clicker trojans[13] found in GooglePlay). We found that, in most cases, when a *service* is injected, a *receiver* will be correspondingly injected. In this case, the receiver plays as a proxy for ensuring that the service will be launched. Indeed, it is much easier for trigger malicious behaviour via a receiver than to make a user start a service. This finding has also been reported at a smaller scale in our previous work [26].

Interestingly, we have found that some newly injected components are shipped with empty implementations. In Listing 2, component *com.idpack.IRE* is a newly injected class in app *39DE41*. In the manifest file, *com.idpack.IRE* has been

13. http://goo.gl/kAFjkQ

declared with a lot of capabilities (i.e., six actions), making it likely to be triggered. For example, it will be launched when a new app is installed (cf. line 7) or uninstalled (cf. line 8). However, as shown in line 2, the implementation of component *com.idpack.IRE* is surprisingly empty. When executing the component, it will reuse the implementation of its parent class named *com.is.p.Re*. Unfortunately, this indirection is often ignored in trivial static analysis approaches, leaving the real implementation of injected components unanalyzed. It would be interesting to investigate to what extent such tricks can impact the results of existing static analysis approaches. This, however, is out of the scope of this paper and therefore we consider it for future work.

```
1  //The implementation of component com.idpack.IRE
2  public class com.idpack.IRE extends com.is.p.Re { }
3
4  //The manifest declaration of component com.idpack.IRE
5  <receiver android:name="com.idpack.IRE">
6   <intent-filter>
7    action:"android.intent.action.PACKAGE_ADDED"
8    action:"android.intent.action.PACKAGE_REMOVED"
9    <data android:scheme="package" />
10  </intent-filter>
11  <intent-filter>
12   action:"android.net.conn.CONNECTIVITY_CHANGE"
13   action:"android.intent.action.USER_PRESENT"
14   action:"com.lseiei.downloadManager"
15   action:"com.cdib.b"
16  </intent-filter>
17 </receiver>
```

Listing 2: An example of empty component implementation (app *39DE41*).

Finally, we note that no piggybacked apps add new Content Provider components. This finding is inline with the intuition that malicious apps are not targeted at structuring data for sharing with other apps.

F9► *Piggybacking often consists in inserting a component that offers the same capabilities as an existing component in the original app.*

We noted that piggybacking may add a component with a given capability which was already declared for another component in the carrier app. This is likely typical of piggybacking since there is no need in a benign app to implement several components with the same capabilities (e.g., two PDF reader components in the same app). For example, in our ground truth dataset, we have found that in each of 551 (i.e., 58%) piggybacked apps, several components have the same declared capability. In contrast, 6% of the benign original apps included components with the same capability.

We then perform a manual investigation of the duplicated component capabilities and make two observations: 1) In most cases, duplicated component capabilities are associated with *Broadcast Receivers*. This finding is intuitively normal since receivers can be easily triggered (e.g., by system events) and consequently can readily lead to the execution of other (targeted) components. 2) All the duplicated component capabilities are likely to be newly injected. Because piggybackers intend to maximize their benefits, they will usually inject at the same time different modules. Those modules are independent from each other and each of them will attempt to maximize its possibility of being executed. As a result, each module declares a

```
1  receiver:com.sumase.nuatie.wulley.RsollyActivity
2   action:"android.intent.action.PACKAGE_ADDED"
3   action:"android.net.conn.CONNECTIVITY_CHANGE"
4   action:"android.intent.action.USER_PRESENT"
5  receiver:com.fuonw.suoetl.cuoll.TsenaActivity
6   action:"android.intent.action.PACKAGE_ADDED"
7   action:"android.net.conn.CONNECTIVITY_CHANGE"
8   action:"android.intent.action.BOOT_COMPLETED"
9  receiver:com.hunstun.tallsyen.blawe.RsekleeActivity
10  action:"android.intent.action.PACKAGE_ADDED"
11  action:"android.net.conn.CONNECTIVITY_CHANGE"
12  action:"android.intent.action.USER_PRESENT"
13 receiver:com.luotuy.mustn.VenrowoActivity
14  action:"android.intent.action.PACKAGE_ADDED"
15  action:"android.net.conn.CONNECTIVITY_CHANGE"
16  action:"android.intent.action.USER_PRESENT"
```

Listing 3: An example of duplicated component capabilities (app *4C35CC*).

receiver to listen to popular system events, further resulting in duplicated component capabilities. Listing 3 shows an example of duplicated component capabilities. All the four receivers (possibly from four modules as indicated by the name of receivers) are newly injected and all of them have declared the same capabilities.

Fig. 19 enumerates the top 10 duplicated capabilities that are leveraged by piggybacked apps. The three capabilities presented in Listing 3 are found to be the top duplicated capabilities by piggybackers. Actually, the corresponding system events, including 1) new app installed (*PACKAGE_ADDED*), 2) internet connection changed (*CONNECTIVITY_CHANGE*), and 3) phone unlocked (*USER_PRESENT*), are commonly-fired events in Android devices.

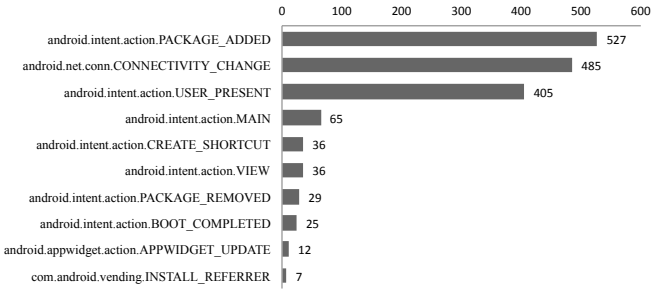

Fig. 19: Top 10 duplicated component capabilities (actions).

F10► *Piggybacked apps can simply trick users by changing the launcher component in the app, in order to trigger the execution of rider code.*

As previously explained, Android apps include in their Manifest file an Application package name that uniquely identifies the app. They also list in the Manifest file all important components, such as the `LAUNCHER` component with its class name. Generally Application package name and Launcher component name are identical or related identically. However, when a malware writer is subverting app users, she/he can replace the original Launcher with a component from his malicious payload. The app example from Listing 4 illustrates such a case where the app's package (`se.illusionlabs.labyrinth.full`, line 1) and launcher (`com.loading.MainFirstActivity`, line 10) differ.

```
1  <manifest package="se.illusionlabs.labyrinth.full">
2      activity:"se.illusionlabs.labyrinth.full.
3          StartUpActivity"
4       action:"android.intent.action.MAIN"
5  -    category:"android.intent.category.LAUNCHER"
6  +   activity:"com.loading.MainFirstActivity"
7  +    action:"android.intent.action.MAIN"
8  +    category:"android.intent.category.LAUNCHER"
9  +   receiver:"com.daoyoudao.ad.CAdR"
10 +    action:"android.intent.action.PACKAGE_ADDED"
11 +    <data android:scheme="package" />
12 +   service:"com.daoyoudao.dankeAd.DankeService"
13 +    action:"com.daoyoudao.dankeAd.DankeService"
14 </manifest>
```

Listing 4: Simplified view of the *manifest* file of *se.illusionlabs.labyrinth.full* (app's sha256 ends with `7EB789`).

We investigate our piggybacking pairs to identify cases where the piggybacked apps has changed their `LAUNCHER` component, comparing to the original benign app. Table 1 illustrates some examples on the original and updated `LAUNCHER`. These changes in the Manifest file are essential for piggybackers to ensure that their code is run. We found 73 cases in our dataset where the piggybacked app switched the original launcher component to one component that was added as part of its rider code.

TABLE 1: Illustrative examples of launcher changes.

Original Launcher	Updated Launcher
com.unity3d.player.UnityPlayerActivity	com.sorted.android.probe.ProbeMain
com.ubermind.ilightr.iLightrActivity	com.geinimi.custom.Ad3034_30340001
com.virgil.basketball.BasketBallActivity	cn.cmgame.billing.ui.GameOpenActivity
game.main.CatchThatKid_UIActivity	cn.dena.mobage.android.MobageActivity
jp.seec.escape.stalking.EscapeStalking	com.pujiahh.Main

F11▶ *Piggybacking is often characterized by a naming mismatch between existing and inserted components.*

Since Android apps are mostly developed in Java, different modules in the application package come in the form of Java packages. In this programming model, developer code is structured in a way that its own packages have related names with the Application package name (generally sharing the same hierarchical root, e.g., `com.example.*`). When an app is piggybacked, the inserted code comes as separated modules constituting the rider code with different package names. Thus, the proportions in components that share the application package names can also be indicative of piggybacking. Such diversity of package names can be seen in the example of Listing 4. The presented excerpt already contains three different packages. Since Android apps make extensive use of Google framework libraries, we systematically exclude those in our analysis to improve the chance of capturing the true diversity brought by piggybacking.

Fig. 20 illustrates that piggybacked apps present a higher diversity of packages names in comparison with their original benign counterparts.

F12▶ *Malicious piggybacked payload is generally connected to the benign carrier code via a single method call statement.*

Listing 5 illustrates a snippet showing an example of type$_1$ hook (cf. definition in Section 4.1), where the hook (line 4) is placed immediately at the beginning of the *onCreate* method to trigger the malicious payload when component *UnityPlayerProxyActivity* is activated. The fact

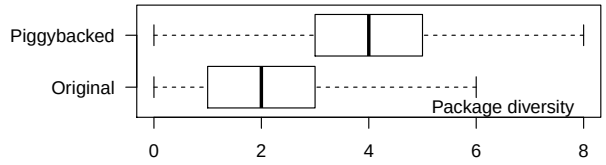

Fig. 20: Distribution of the number of different packages.

that a *static* method is leveraged somewhat suggests that piggybackers attempt to pay least effort to connect benign carrier code to rider code. The simplified implementation of *touydig.init()* is shown in lines 11-20. Generally, this method is used to initialize three modules which are simultaneously injected by piggybackers. All the three modules, including *umeng*, *kuguo*, and *crazymedia*, are injected to provide advertizement and consequently to redirect the revenues from the original developers to piggybackers.

```
1  //In class UnityPlayerProxyActivity
2  protected void onCreate(Bundle b) {
3    this.onCreate(b);
4  + com.gamegod.touydig.init(this);
5    $r2 = new String[2];
6    $r2[0] = "com...UnityPlayerActivity";
7    $r2[1] = "com...UnityPlayerNativeActivity";
8    UnityPlayerProxyActivity.copyPlayerPrefs(this, $r2);
9  }
10
11 class com.gamegod.touydig {
12 public static void init(Context $param0) {
13  //initialize umeng module
14  $String = $bundle.getString("UMENG_CHANNEL");
15  //initialize kuguo module
16  $KuguoAdsManager = KuguoAdsManager.getInstance();
17  $KuguoAdsManager.setCooId($param0,
        "ae63b5208de5422f9313d577b3d0aa41");
18  //initialize crazymedia module
19  net.crazymedia.iad.AdPushManager.init($param0, "8787",
        "wx1x6nkz277ruch5", false);
20 }}
```

Listing 5: Simplified view of the snippet of app *com.SkillpodMedia.farmpopfrenzy* (app's sha256 ends with `BF3978`). The '+' sign at Line 4 indicates the statement that was added to the original code.

In our dataset, 699 (74%) piggybacked apps use type$_1$ hooks to connect rider code to carrier code. Among those, 438 (i.e., 63% of type$_1$) hooks only include a single statement, creating a weak link between the call-graphs of rider and carrier code. Table 2 enumerates the most redundant hook statements used in piggybacked apps.

F13▶ *Piggybacking hooks are generally placed within library code rather than in core app code.*

We look into the carrier method implementations where hook statements are placed. Table 3 presents the top 5 methods where hook statements can be recurrently traced to. Interestingly, all five methods are actually part of well-known libraries that are used for benign functionality. We further found that these libraries actually propose utility functions for Game apps.

F14▶ *Injected payload is often reused across several piggybacked apps.*

We compute the pairwise similarity of rider code using the Jaccard distance at the class level, method level and API level. When the set of classes, methods or used APIs are the

TABLE 2: Top five type$_1$ hooks used by piggybacked apps.

Method	# piggybacked apps
$< com.basyatw.bcpawsen.DaywtanActivity : void\ Tawo(android.content.Context) >$	42
$< com.gamegod.touydig : void\ init(android.content.Context) >$	39
$< com.geseng.Dienghla : void\ init(android.content.Context) >$	34
$< com.gamegod.touydig : void\ destroy(android.content.Context) >$	24
$< com.tpzfw.yopwsn.Aervlrey : void\ Tdasen(android.content.Context) >$	15

TABLE 3: Top five methods in carrier code where hook statements are inserted.

Method	# piggybacked apps
$< com.unity3d.player.UnityPlayerProxyActivity : void\ onCreate(android.os.Bundle) >$	95
$< com.ansca.corona.CoronaActivity : void\ onCreate(android.os.Bundle) >$	29
$< org.andengine.ui.activity.BaseGameActivity : void\ onCreate(android.os.Bundle) >$	11
$< com.prime31.UnityPlayerProxyActivity : void\ onCreate(android.os.Bundle) >$	10
$< com.g5e.KDLauncherActivity : void\ onCreate(android.os.Bundle) >$	8

TABLE 4: Statistics on the pairwise similarity between rider code of piggybacked apps.

Jaccard Distance	Method-Level		Class-Level		Android-API-Level	
	# of Apps	# of Relevant Pairwise Combinations	# of Apps	# of Relevant Pairwise Combinations	# of Apps	# of Relevant Pairwise Combinations
$= 0$	599	8,890	605	9,078	791	11,482
≤ 0.05	806	10,178	794	10,814	914	18,662
≤ 0.1	821	10,384	813	11,222	933	25,788
≤ 0.2	845	10,718	833	11,836	967	42,660

TABLE 5: Top five Android APIs that are interested by rider code.

Method	Count
$< android.net.Uri : android.net.Uri\ parse(java.lang.String) >$	955
$< android.content.Context : java.lang.Object\ getSystemService(java.lang.String) >$	953
$< android.content.BroadcastReceiver : void\ < init > () >$	949
$< android.content.Context : android.content.pm.PackageManager\ getPackageManager() >$	941
$< android.app.Activity : void\ < init > () >$	939

same between two apps, the distance is zero. When there is no overlap this distance amounts to 1. Table 4 provides statistics which reveal that more than half piggybacked apps include a rider code which can be found in another piggybacked app. The Jaccard distance indeed amounts to 0 for 599, 605 and 791 piggybacked apps when comparing at the method, class and API levels respectively. Theoretically, There are in total $\binom{2}{\#ofapps}$ relevant combinations.

The similarity of comparison results at the class, method and API levels, suggest that when piggybackers reuse an existing rider code sample, they rarely make modifications. Finally, there are more pairs of rider code which are similar at the API level, suggesting that many piggybacking payloads often include code that perform similar tasks (e.g., for advertizements).

F15► *Piggybacking adds code which performs sensitive actions, often without referring to device users.*

Intents are special objects used by Android apps to exchange data across components. It allows app components to reuse functionalities implemented in other components (within or outside the app). Intent objects include a field for indicating the action that must be performed on the data exchanged (e.g., VIEW a PDF file, or DIAL a number). Fig. 21 enumerates the top actions that can be seen in all intents used by rider code across the piggybacked apps. In most cases, intents transfer data to be viewed, a necessary action for displaying advertizements. In many cases also, rider code installs shortcuts, performs dialing, shares the data outside the app, etc. We also note that two of the top actions (the *6th* and *8th*) are not system-defined. Instead they are defined in user code, and their presence in many piggybacked apps further confirms the reuse of payloads across apps.

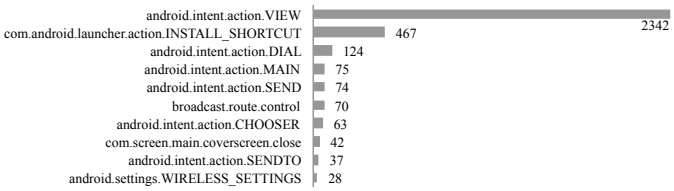

Fig. 21: Top 10 actions from Intents brought in rider code.

Intents are either implicit (the component which will realize the action will be selected by the user) or explicit (a component is directly designated, bypassing user's choice). Differences in distributions presented in Fig. 22 suggest that piggybacking rider code contain more explicit intents than implicit intents. In contrast, as shown in [27], benign apps actually attempt to use more implicit Intents (for showing Activities).

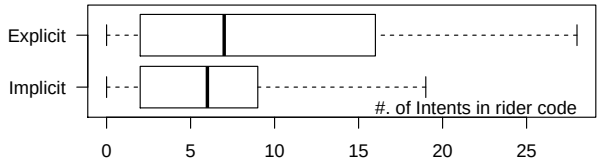

Fig. 22: Use of explicit vs. implicit Intents in rider code.

F16► *Piggybacking operations spread well-known malicious be-*

haviour types.

We investigate the labels provided by AV engines and extract the top appearing names to classify malware types. Table 6 provides statistics on the three outstanding types. Although most are flagged as Adware, we note that there are overlapping with Trojans and Spyware. This is inline with the recent analysis of a family of trojans by ESET, showing that they act as adware.

TABLE 6: Outstanding malware types reported by AV engines on the piggybacked apps.

App type	# of piggybacked apps	# of distinct labels
Adware	888	230
Trojan	495	451
Spyware	192	94

We further match the AV labels against 6 well-known malware family names reported in [50]. We found that our dataset contains samples belonging to each family as described in Table 7. For example, clustering the rider code of piggybacked apps in the GingerMaster[14] family based on class names and with EM algorithm yields 5 clusters, one of which contains over 80 samples, suggesting a high reuse of a malicious payload code.

TABLE 7: Number of piggybacked apps in known malware families.

ADRD	BaseBridge	Geinimi	GingerMaster	GoldDream	Pjapps
11	5	21	149	11	9

F17► *Piggybacked apps increasingly hide malicious actions via the use of reflection and dynamic class loading.*

The Android system provides DexClassLoader, a class loader that loads classes from .jar and .apk files. It can be used by malware writers to execute code not installed as part of their apps. We found 185 (19%) piggybacked apps whose riders dynamically load code. Such schemes are often used by malware writers to break the control-flow of app, and thus challenge static detection of suspicious behaviour [30], [31]. Fig. 23 provides the ratio of piggybacked apps using reflection and dynamic code loading. The ratio has substantially improved in recent years.

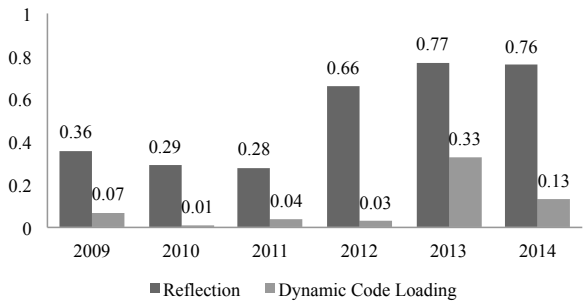

Fig. 23: Trend in the use of reflection and dynamic class loading by piggybacked apps.

F18► *Piggybacking code densifies the overall app's call graph, while rider code can even largely exceed in size the carrier code.*

14. These malware collect and send user info, including device ID and phone number, to a remote server.

We compute the proportion of piggybacked app code which is actually brought by rider code. Fig. 24 (left) highlights that, for most apps (71%), the rider code is smaller than the carrier code. However, for 29% of the piggybacked apps in our dataset, the rider code is bigger in terms of LOCs than the carrier code.

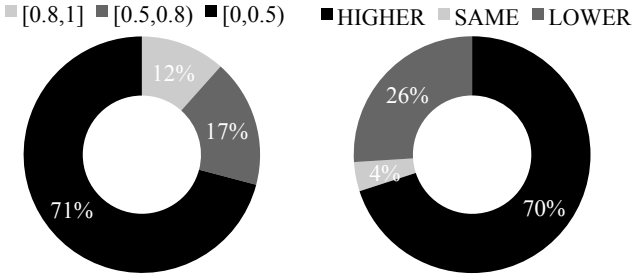

Fig. 24: Impact of Piggybacking on rider code proportion (left) and CG density (right).

We further build the call graphs of piggybacked apps and compare them to the call graphs of their corresponding original apps. Then we compute the density metric[15] using the number of real edges in the graph divided by the total number of possible edges. Distribution in Fig. 24 (right) highlights that in the large majority of cases, piggybacking code increases[16] the call graph density. This increase can be achieved by injecting complex rider code with higher density than the carrier code. Indeed, complex rider code is essential for piggybackers to make their code more difficult to understand by analysts. In other words, this finding is expected because the injected rider code will likely explore malicious scenarios that piggybackers wish to leave hidden to manual review and static analysis. On the other hand, in 1/4 cases the call graph density has lowered. We have sampled some piggybacked apps in such cases, and found that most use type$_2$ hooks (i.e., via events). Only a few piggybacked apps kept the same density metric values.

F19► *Piggybacked app writers are seldom authors of benign apps.*

We investigate the certificates used to sign the apps in our dataset. We collected 703 distinct certificates in original apps against only 194 distinct certificates in piggybacked apps. We only found 14 certificates where each have been used to sign both a piggybacked and another app in the set of originals. Looking into those certificates, we found that it is inline with the well-known copy/paste problem. As an example, we have found that a certificate issued to Android Debug, which should only be used to develop and test an app, is applied to released apps. Several of the overlapping certificates were actually due to this issue.

F20► *Piggybacking code brings more execution paths where sensitive data can be leaked.*

15. We leverage the Toolkit of GraphStream (http://graphstream-project.org) to compute the density metric. The computation is implemented in a complexity of O(1).

16. As for all comparisons, we remind the reader that we have checked that the difference was statistically significant.

Private data leaks in Android are currently a main focus in the research and practice community. We investigate the static behaviour of piggybacked apps w.r.t the data flows that they include using IccTA [27]. Fig. 25 show that piggybacked apps include, on median, 15 more flows from a sensitive source to a sensitive sink than original apps. We also found that these flows mostly represent data exchange across components. As an example, we have found 12 such leaks where Context information is collected, then transferred to another component which eventually forwards it outside the device via SMS.

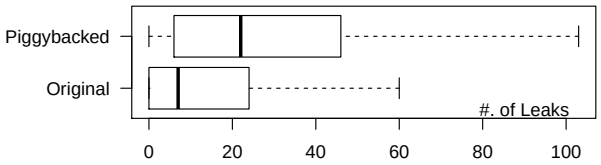

Fig. 25: Distributions of the number of leaks.

5 Discussion

We now explore three potential outcomes of our findings (cf. Section 5.1). Then, we discuss two problematic assumptions that have been commonly made in the Android community (Section 5.2) and the potential threats to validity of this study (cf. Section 5.3).

5.1 Potential outcomes

Understanding Android app piggybacking can help in pushing further a number of research directions:

1) Practical detection of piggybacked apps: With this work, we hope to follow on the steps of the MalGenome project and drive new research on the automated detection of piggybacked apps. Indeed, by enumerating high-level characteristics of Android malware samples, MalGenome has opened several directions in the research on malware detection, most of which have either focused on detecting specific malware types (e.g., malware leaking private data [27]), or are exploiting app features, such as permissions requested, in Machine Learning classification [3]. Similarly, we expect the devise of machine learning-based detection approaches which can **effectively model piggybacking** and thus avoid the need for impractical and unscalable pairwise comparisons across app markets. Indeed, if we consider piggybacked detection as a classification problem, we can address the main limitation in state-of-the-art works which all require the original app in order to search for potential piggybacked apps that use it as a carrier. Indeed, with a classification-based approach, original counterparts are only in the training phase. Once it is done, the classifier can be used in the wild to identify piggybacked apps where their original counterparts have not been seen before. In a similar research, researchers have already shown with Clonewise that it was possible to treat clone detection as a classification problem [11]. In preliminary experiments, using a feature set limited to basic features whose extraction was inexpensive (Permissions, Intents, rider API uses) we built classifiers that achieved acceptable performance in precision and recall (both over 70%) with 10-Fold cross validation on our dataset. More sophisticated and discriminative features can be explored from our findings. Indeed, as shown in our technical report [32], with a set of features including *new declared capabilities*, *duplicated permissions*, *diversity of packages*, etc., we are able to achieve a high performance of 97% for both precision and recall, suggesting the features built on our findings are promising for discriminating piggybacked apps from non-piggybacked apps.
2) Explainable malware detection: Current Machine Learning classifiers [9], [15], [35], [44] for Android malware detection are too generic to be relevant in the wild: features currently used in the literature, such as n-grams, permissions or system calls, allow to flag apps without providing any hint on which malicious actions are actually expected. A collection of piggybacked apps provides new opportunities for **retrieving a variety of malicious payloads** that can be investigated to infer fine-grained semantic features for malware detection.
3) Malicious code localisation: In their fight against malware, practitioners are regularly challenged by the need to localize malicious code within an app, e.g., in order to remove/block it, or to characterize its impact. Our work can be leveraged towards **understanding how malicious code are hooked in benign app code**, as well as which code features may potentially help statically discriminate malicious parts from benign parts. In our preliminary experiments [32], based on the identified behaviors of piggybacked apps, we are able to automatically localize the hook code with an accuracy@5 (the hook code is within the top 5 packages we ranked based on their probabilities of being hook code) of 83%, without knowing the original conterparts of the dissected piggybacked apps. More recently, Tian et al. [40] also propose an approach, namely *code heterogeneity analysis* (i.e., Examining Android apps for code regions that are unrelated in terms of data/control dependence), to distinguish different behaviors of the malicious component and the original app. As explicitly acknowledged by the authors, their prototype approach has some limitations, e.g., being unaware of implicit ICCs, does not work for complex code structures, etc. With the help of our comprehensive findings we present in this paper, their approach could be refined.

5.2 Validity of common assumptions

Based on our study, we are able to put in perspective an assumption used in the literature for reducing the search space of piggybacking pairs. With FSquaDRA [46], Zhauniarovich et al. have proposed to rely on similarity of apps' resource files to detect piggybacking. Finding F1 of our work however supports that resource files can be extensively manipulated during piggybacking.

We further plot in Fig. 26 the distribution of pairwise similarity for the piggybacking pairs of our dataset to highlight the significant difference between relying on resource-based similarity and code-based similarity.

Another common assumption in the literature is that library code is noisy for static analysis approaches who must

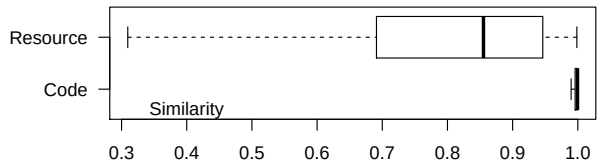

Fig. 26: Similarity distribution.

parse a large portion of code which is not part of app core functionality, leading to false positives. Thus, recent research works [5], [29] first heuristically prune known common libraries before applying their approaches on apps. We have shown however with our investigation that libraries, in particular popular libraries, are essential to malware writers who can use them to seamlessly insert hooks for executing malicious rider code.

5.3 Threats to validity

The main threat to validity of our study lies in the exhaustivity of our dataset. However, we have made our best effort to leverage the AndroZoo largest available research dataset of Android apps [2] to search for piggybacked apps. Our apps span a long timeline of 6 years, and we have checked that the piggybacked apps are classified by AVClass [36] as presenting 29 different labels (representing potential malware families).

Another threat to validity is that we did not include apps from the last year or so, due to delays in the collection of AV reports by the AndroZoo infrastructure. We are committed to continuously update the dataset of piggybacked apps within the AndroZoo project [2]. In that direction, we have already identified piggybacking pairs from AndroZoo where the samples of malware are created in 2016. We mitigate the threat to validity on the age of our dataset by checking and confirming that our findings remain valid for the most recent samples of piggybacked apps.

With regards to the restrictions used in the identification of piggybacking pairs, we would like to note for the readers that our focus in this work is not to exhaustively detect all piggybacking pairs. Instead, we aimed for collecting a sufficient number of obvious, thus accurate, piggybacking pairs in order to be able to dissect and understand piggybacking processes.

The original apps in our ground truth may not be the final original apps, i.e., they may also be repackaged versions of a previous app. To the best of our knowledge, there is no straightforward way to pinpoint whether a given original app from a piggybacking pair is the true original app. Therefore, it remains an interesting future work to the community.

AV labels from VirusTotal engines may not be perfect because of AV disagreements [24]. However, in this work, we only use them to get quick insights. Finally, by construction, our dataset of piggybacked apps has been built with a constraint on the maliciousness (in the sense of AV detection) of the piggybacked app by a different developer. Other "piggybacking" operations may be performed by the same developer with "benign" code.

Recently, there are many Android app packers (e.g., Bangcle, Ijiami, etc.) introduced for comprehensive protection of Android apps, attempting to prevent Android apps from being reverse engineered and repackaged/piggybacked. Intuitively, it becomes much harder to piggyback packed apps. Unfortunately, state-of-the-art approaches including AppSpear [43] and DexHunter [45] have already demonstrated promising results for extracting DEX code from packed apps, making it still possible to piggyback even packed apps.

6 Related Work

Dissection studies: In recent years there have been a number of studies dissecting Android malware [18], [19], [50]. Genome [50] provides an interesting overview on the landscape of Android malware based on samples collected from forums and via experiments. Our approach for piggybacked apps is however more systematic. Besides in Android community, dissection studies have also been widely performed in other communities. For example, Jakobsson et al. [22] have presented a comprehensive analysis on crimeware, attempting to understand current and emerging security threats including bot networks, click fraud, etc.

Repackaged/Cloned app detection: Although the scope of repackaged app detection is beyond simple code, researchers have proposed to rely on traditional code clone detection techniques to identify similar apps [13], [14], [16], [48]. With DNADroid [14] Crussell et al. presented a method based on program dependency graphs comparison. The authors later built on their DNADroid approach to build AnDarwin [13] an approach that uses multi-clustering to avoid the scalability issues induced by pairwise comparisons. DroidMOSS [48] leverages fuzzy hashing on applications' OpCodes to build a database of application fingerprints that can then be searched through pairwise similarity comparisons. Shahriar and Clincy [37] use frequencies of occurrence of various OpCodes to cluster Android malware into families. Finally, in [12], the authors use a geometry characteristic of dependency graphs to measure the similarity between methods in two apps, to then compute similarity score of two apps.

Instead of relying on code, other approaches build upon the similarity of app "metadata". For instance, Zhauniarovich et al. proposed FSquaDRA [46] to compute a measure of similarity on the apps' resource files. Similarly, ResDroid [38] uses application resources such as GUI description files to extract features that can then be clustered to detect similar apps. In their large-scale study, Viennot, Garcia, and Nieh [42] also used assets and resources to demonstrate the presence of large quantities of either rebranded or cloned applications in the official Google Play market.

Our work, although closely related to all aforementioned works, differs from them in three ways: First, these approaches detect repackaged apps while we focus on piggybacked apps. Although a piggybacked app is a repackaged app, the former poses a more serious threat and its analysis can offer more insights into malware. Second, most listed approaches perform similarity computations through pairwise comparisons [28]. Unfortunately such a process is computationally expensive and has challenged scalability. Third, these approaches depend on syntactic instruction sequences

(e.g., opcodes) or structural information (e.g., PDGs) to characterize apps. These characteristics are however well known to be easily faked (i.e., they do not resist well to evasion techniques). Instead, our detailed examination on piggybacked apps could provide hints to build semantic features of apps and hence could achieve better efficiency in detection of malicious apps. Semantic features are relevant as they allow to be resilient to most obfuscation techniques (e.g., method renaming). For example, the literature has shown that, input-output states of core methods [20] are semantic features which are more appropriate than the related syntactic features extracted from instructions in method definitions. In our study, findings **F6** and **F9** suggest that we can consider duplication of permissions and of capability declarations as semantic features characterizing piggybacking. The presence of sensitive data flows, as revealed by finding **F20**, is also semantically discriminating for identifying malicious behavior.

Piggybacked app search and Malware variants detection: Cesare and Xiang [10] have proposed to use similarity on Control Flow Graphs to detect variants of known malware. Hu, Chiueh, and Shin [21] described SMIT, a scalable approach relying on pruning function Call Graphs of x86 malware to reduce the cost of computing graph distances. SMIT leverages a Vantage Point Tree but for large scale malware indexing and queries. Similarly, BitShred [23] focuses on large-scale malware triage analysis by using feature hashing techniques to dramatically reduce the dimensions in the constructed malware feature space. After reduction, pair-wise comparison is still necessary to infer similar malware families.

PiggyApp [47] is the work that is most closely related to ours. The authors are indeed focused on piggybacked app detection. They improve over their previous work, namely DroidMoss, which was dealing with repackaged app detection. PiggyApp, similar to our approach, is based on the assumption that a piece of code added to an already existing app will be loosely coupled with rest of the application's code. Consequently, given an app, they build its program dependency graph, and assigns weights to the edges in accordance to the degree of relationship between the packages. Then using an agglomerative algorithm to cluster the packages, they select a primary module. To find piggybacked apps, they perform comparison between primary modules of apps. To escape the scalability problem with pair-wise comparisons, they rely on the Vantage Point Tree data structure to partition the metric space. Their approach differs from ours since they require the presence of the original to be able to detect its piggybacked apps.

7 Conclusion

We have investigated Android piggybacked apps to provide the research community with a comprehensive characterisation of piggybacking. We then build on our findings to put in perspective some common assumptions in the literature. We expect this study to initiate various research directions for practical and scalable piggybacked app detection, explainable malware detection, malicious code localization, and so on.

8 Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects AndroMap C13/IS/5921289 and Recommend C15/IS/10449467. This work was also partially supported by the Singapore's NRF Research Grant NRF2016NCR-NCR001-008 and by the UK EPSRC Research Grant EP/L022710/1.

References

[1] Shared data repository, Aug. 2015. https://github.com/serval-snt-uni-lu/Piggybacking.

[2] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *The 13th International Conference on Mining Software Repositories (MSR), Data Showcase Track*, pages 468–471. ACM, 2016.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[4] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[5] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *International Conference on Software Engineering (ICSE)*, 2015.

[6] Alexandre Bartel, Jacques Klein, Martin Monperrus, Kevin Allix, and Yves Le Traon. Improving privacy on android smartphones through in-vivo bytecode instrumentation. Technical report, arXiv:1208.4536, May 2012.

[7] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*, 2012.

[8] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *Software Engineering, IEEE Transactions on*, 40(6):617–632, 2014.

[9] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. A classifier of malicious android applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 607–614. IEEE, 2013.

[10] Silvio Cesare and Yang Xiang. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing - Volume 107*, AusPDC '10, pages 61–70, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.

[11] Silvio Cesare, Yang Xiang, and Jun Zhang. Clonewise: Detecting package-level clones using machine learning. In Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao, editors, *Security and Privacy in Communication Networks*, volume 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 197–215. Springer International Publishing, 2013.

[12] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 175–186, New York, NY, USA, 2014. ACM.

[13] J. Crussell, C. Gibler, and H. Chen. Andarwin: Scalable detection of android application clones based on semantics. *Mobile Computing, IEEE Transactions on*, PP(99):1–1, 2014.

[14] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 37–54. Springer Berlin Heidelberg, 2012.

[15] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 559–570, New York, NY, USA, 2013. ACM.

[16] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, pages 3:1–3:12, New York, NY, USA, 2014. ACM.

[17] Anthony Desnos. Android: Static analysis using similarity distance. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 5394–5403. IEEE, 2012.

[18] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[19] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[20] Quanlong Guan, Heqing Huang, Weiqi Luo, and Sencun Zhu. Semantics-based repackaging detection for mobile apps. In *International Symposium on Engineering Secure Software and Systems*, pages 89–105. Springer, 2016.

[21] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 611–620, New York, NY, USA, 2009. ACM.

[22] Markus Jakobsson and Zulfikar Ramzan. *Crimeware: understanding new attacks and defenses*. Addison-Wesley Professional, 2008.

[23] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.

[24] Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Brad Miller, Vaishaal Shankar, Rekha Bachwani, Anthony D. Joseph, and J. D. Tygar. Better malware ground truth: Techniques for weighting anti-virus vendor labels. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, AISec '15, pages 45–56, New York, NY, USA, 2015. ACM.

[25] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[26] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential Component Leaks in Android Apps: An Investigation into a new Feature Set for Malware Detection. In *The 2015 IEEE International Conference on Software Quality, Reliability & Security (QRS)*, 2015.

[27] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[28] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps. In *Technical Report*, 2016.

[29] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[30] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016.

[31] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *The 31st IEEE/ACM International Conference on Automated Software Engineering, Demo Track (ASE 2016)*, 2016.

[32] Li Li, Daoyuan Li, Tegawendé François D Assise Bissyande, David Lo, Jacques Klein, and Yves Le Traon. Ungrafting malicious code from piggybacked android apps. Technical report, SnT, 2016.

[33] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947.

[34] Tood K Moon. The expectation-maximization algorithm. *Signal processing magazine, IEEE*, 13(6):47–60, 1996.

[35] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *Intelligence and Security Informatics Conference (EISIC), 2012 European*, pages 141–147. IEEE, 2012.

[36] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[37] H. Shahriar and V. Clincy. Detection of repackaged android malware. In *Internet Technology and Secured Transactions (ICITST), 2014 9th International Conference for*, pages 349–354, Dec 2014.

[38] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 56–65, New York, NY, USA, 2014. ACM.

[39] Symantec. Internet security threat report. Volume 20, Symantec, April 2015.

[40] Ke Tian, Danfeng Daphne Yao, Barbara G Ryder, and Gang Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *Mobile Security Technologies (MoST)*, 2016.

[41] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. In *Sable Research Group, McGill University*, 1998.

[42] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. *SIGMETRICS Perform. Eval. Rev.*, 42(1):221–233, June 2014.

[43] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 359–381. Springer, 2015.

[44] S.Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A new android malware detection approach using bayesian classification. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 121–128, 2013.

[45] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: toward extracting hidden code from packed android applications. In *European Symposium on Research in Computer Security*, pages 293–311. Springer, 2015.

[46] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. Fsquadra: Fast detection of repackaged applications. In Vijay Atluri and Gnther Pernul, editors, *Data and Applications Security and Privacy XXVIII*, volume 8566 of *Lecture Notes in Computer Science*, pages 130–145. Springer Berlin Heidelberg, 2014.

[47] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 185–196, New York, NY, USA, 2013. ACM.

[48] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

[49] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM, 2012.

[50] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, May 2012.