Ungrafting Malicious Code from Piggy-backed Android Apps

Li Li	University of Luxembourg / SnT / SERVAL, Luxembourg
Daoyuan Li	University of Luxembourg / SnT / SERVAL, Luxembourg
Tegawendé F. Bissyandé	University of Luxembourg / SnT / SERVAL, Luxembourg
David Lo	Singapore Management University, Singapore
Jacques Klein	University of Luxembourg / SnT / SERVAL, Luxembourg
Yves Le Traon	University of Luxembourg / SnT / SERVAL, Luxembourg

20 April 2014

ISBN 978-2-87971-149-2 TR-SNT-2016-2

Ungrafting Malicious Code from Piggybacked Android Apps

Li Li
SnT, Univ. of Luxembourg
li.li@uni.lu

Daoyuan Li
SnT, Univ. of Luxembourg
daoyuan.li@uni.lu

Tegawendé F. Bissyandé
SnT, Univ. of Luxembourg
tegawende.bissyande@uni.lu

David Lo
Singapore Management Univ.
davidlo@smu.edu.sg

Jacques Klein
SnT, Univ. of Luxembourg
jacques.klein@uni.lu

Yves Le Traon
SnT, Univ. of Luxembourg
yves.letraon@uni.lu

ABSTRACT

To devise efficient approaches and tools for detecting malicious code in the Android ecosystem, researchers are increasingly required to have deep understanding of malware. There is thus a need to provide a framework for dissecting malware and localizing malicious program fragments within app code in order to build a comprehensive dataset of malicious samples. In this paper we address this need with an approach for listing malicious packages in an app based on code graph analysis. To that end we focus on piggybacked apps, which are benign apps repackaged with malicious payload. Our approach classifies each app independently from its potential clones based on machine learning, and detects piggybacked apps with a precision of about 97%. With the built classifier we were also able to find new piggybacked apps in market datasets, outside our ground truth. We also identify malicious packages with an accuracy@5 of 83% and an accuracy@1 of around 68%. We further demonstrate the importance of collecting malicious packages by using them to build a performant malware detection system.

1. INTRODUCTION

Malware is pervasive in the Android ecosystem. This is unfortunate since Android is the most widespread operating system in handheld devices and has increasing market shares in various home and office smart appliances. As we now heavily depend on mobile apps in various activities that pervade our modern life, security issues with Android web browsers, media players, games, social networking or productivity apps can have severe consequences. Yet, regularly, high profile security mishaps with the Android platform shine the spotlight on how easily malware writers can exploit a large attack surface, eluding all detection systems both at the app store level and at the device level.

Nonetheless, research and practice on malware detection have produced a substantial number of approaches and tools for addressing malware. The literature contains a large body of such works [1–4]. Unfortunately, the proliferation of malware [5] in stores and on user devices is a testimony that 1) state-of-the-art approaches have not matured enough to significantly address malware, and 2) malware writers are still able to react quickly to the capabilities of current detection techniques. Broadly, malware detection techniques either leverage malware signatures or they build machine learning (ML) classifiers based on static/dynamic features. On the one hand, it is rather tedious to manually build a (near) exhaustive database of malware signatures: new malware or modified malware are thus likely to slip through. On the other hand, ML classifiers 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 neither which malicious actions are actually detected, nor where they are located in the app.

The challenges in Android malware detection are mainly due to a lack of accurate understanding of what constitutes a malicious code. In 2012, Zhou and Jiang [6] have manually investigated 1260 malware samples to characterize 1) their installation process, i.e., which social engineering-based techniques (e.g., repackaging, update-attack, drive-by-attack) are used to slip them into users devices; 2) their activation process, i.e., which events (e.g., `SMS_RECEIVED`) are used to trigger the malicious behaviour; 3) the category of malicious payloads (e.g., privilege escalation or personal information stealing); and 4) how malware exploit the permission system. The produced dataset named MalGenome, have opened several directions in the research of malware detection, most of which have either focused on detecting specific malware types (e.g., malware leaking private data [7]), or are exploiting features such as permissions in ML classification [8]. The MalGenome dataset however has shown its limitations in hunting for malware: the dataset, which was built manually, has become obsolete as new malware families are now prevalent; and the characterization provided in the study is too high-level to allow the inference of meaningful structural or semantic features of malware.

1.1 The Goal of This Paper

The goal of our work is to build an approach for systematizing the dissection of Android malware and automating the collection of malicious code packages in Android apps. Previous studies, including our own, have exposed statistical facts which suggest that malware writing is performed at an "industrial" scale and that a given malicious piece of code can be extensively reused in a bulk of malware [5, 6]. Malware writers can indeed simply unpack a benign, preferably popular app, and then *graft* some malicious code on it before finally repackaging it. The resulting app which thus piggybacks a malicious payload is referred to as a *piggybacked* app. Our assumption that most malware are piggybacked of benign apps is confirmed with the MalGenome dataset where over 80% of the samples were built through repackaging.

In this work we focus on 1) collecting a curated set of piggybacked apps and their corresponding originals to constitute a ground truth; then 2) we use the ground truth to build a classifier for detecting other piggybacked apps; and finally 3) we identify the malicious payload from each piggybacked app.

1.2 Taxonomy of Piggybacked Malware

We now provide a taxonomy to which we will refer to in the remainder of this paper. Figure 1 illustrates the constituting parts of a piggybacked app. Such malware are built by taking a given original app, referred to in the literature [9] 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.

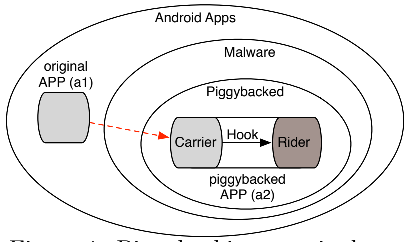

Figure 1: Piggybacking terminology.

1.3 Contributions

The contributions of this paper are as follows:

- We propose a ML approach for detecting piggybacked apps. To the best of our knowledge, we are the first to consider piggybacked app detection as a classification problem. The originality of our approach, compared to the literature, is that we analyze a single app independently from its potential clones. The state-of-the-art on cloned/repackaged/piggybacked app detection does comparisons between apps, either directly based on source code similarity, or based on the distance between extracted feature vectors. Such approaches thus require the presence of the carrier apps in the dataset to work. This requirement is however unrealistic, since the piggybacked app may be in a different market than the original app (e.g., Google Play vs AppChina).
- We propose an approach for localizing malicious code within piggybacked apps. Our approach yields a ranked list of most probable malicious packages. The collection of such malicious packages can be used by researchers and practitioners to build tools that output explainable results (i.e., when an app is flagged as a malware because it shares features from a particular malicious package, it is straightforward to indicate the relevant type/family of malware).
- We perform evaluation experiments which demonstrate that our identified features indeed allow to discriminate between piggybacked and non-piggybacked apps. Applying the classifier to apps in the wild, we were also able to find new piggybacked apps outside our ground truth dataset.
- Finally, we show the benefit of our work by building a malware detection tool with features extracted from the samples of identified malicious payloads.

2. GROUND TRUTH INFERENCE

Research on Android security, especially malware detection, is challenged by the scarcity of datasets and benchmarks. Despite the abundance of studies and approaches on detection of piggybacked apps, accessing the associated datasets is limited. Furthermore, while other studies focus on various kinds of repackaged apps, ours targets exclusively piggybacked malware. Finally, related work apply their approach on random datasets and then manually verify the findings to compute performance. We take a different approach and automate the collection of a ground truth that we share with the community [10]. The process that we used is depicted in Figure 2.

Our collection is based on a large repository of over 2 million apps crawled over several months and that was used for large-scale experiments on malware detection [11–13]. Apps in this repository were obtained from several markets (including Google Play, appchina, anzhi), open source repositories (including F-Droid) and researcher datasets (such as the MalGenome dataset). To identify malware, we sent all the apps to the anti-

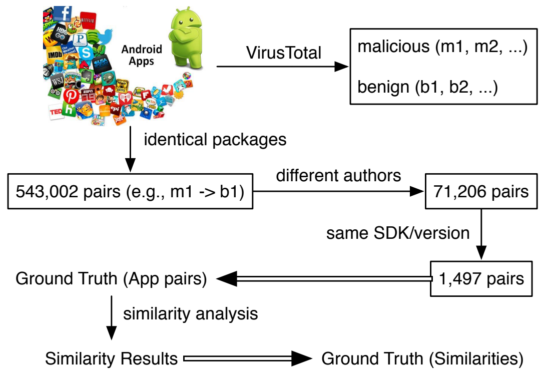

Figure 2: The ground truth build process.

virus products hosted by VirusTotal[1], and we collected the corresponding reports. We further filtered apps based on Application Package name. This information, which is recorded in the Manifest file of each app, is supposed to identify uniquely the app[2]. Considering the identicality of package names, we were able to focus on a subset of about 540 thousands pairs of apps, one benign and the other malicious, with the same app package name. In other words, this subset is composed of about $540,000$ pairs of apps $< app_g, app_m >$, for which app_g and app_m have the same package name, and app_g has been classified as a goodware and app_m as a malware[3] by anti-viruses from VirusTotal.

We do not consider cases where a developer may piggyback his own app to include new payload. Indeed, first we consider piggybacking to be essentially a parasite activity, where one developer exploits the work of another to carry his malicious payload. Second, developers piggyback their own app mainly to insert an advertising component to collect revenue transforming these apps to adware which are often classified as malware. As a result, to follow our definition of piggyback app and avoid the presence of too many adware, 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.

Finally, we assume that, in most cases, malware writers are not very sophisticated and they are using least effort to bypass some defences. For instance, they are not focused on modifying app details such as version numbers or reimplementing functionality which require new SDK versions. By considering pairs where both apps share the same version number and SDK requirements, we are able to compile a reliable ground truth with 1,497 pairs where one app piggybacks the other to include a malicious payload.

[1]http://virustotal.com hosts over 40 anti-virus products from providers such as Symantec, McAfee, Kaspersky.

[2]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.

[3]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.

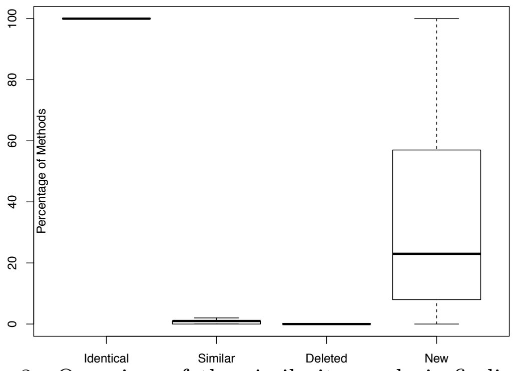

Figure 3: Overview of the similarity analysis findings for the pairs of apps in our collected dataset. In 610 pairs, the piggybacked app includes 100% of the carrier code. In the 863 apps where the carrier code is modified, on median only one method is concerned. In 256 pairs, a few methods have been deleted. Finally, in the large majority of pairs (1031), the piggybacked has new methods. The median value of the proportion of added methods is about 20%.

To validate the relevance of each pair in the ground truth, we further perform a *similarity analysis* to detect whether an original app indeed constitutes a piggybacked app, being responsible for carrying the malicious payload. 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. We leverage Androguard [14] to compute similarity metrics (*identical*, *similar*, *new* and *deleted*) at the method level between apps in each pair of the ground truth. The metrics are defined as follows:

- *identical*: a given method code is exactly the same in both apps.
- *similar*: a given method has slightly changed (at the instruction level) between the two apps.
- *new*: a method has been added in the piggybacked app.
- *deleted*: a method has been deleted from the carrier code when including it in the piggybacked app.

Figure 3 plots the similarity analysis results of our ground truth apps. 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 only in a few cases that piggybacked apps remove methods from the original carrier code.

This experiment also allowed to further consolidate the ground truth. Indeed, we have identified some cases of "piggybacked" apps where no new code was grafted to the carrier (e.g., only resource files have been modified/added). We exclude such apps from the ground truth. The final set of ground truth is now composed of 1031 pairs of apps.

Finally, we investigated the diversity of malicious rider code in the piggybacked apps from our ground truth. To that end we analyzed the new methods added by rider code. Out of the total 504,838 new methods in the set of piggybacked apps, 196,753 (i.e., 39% of new methods) can be found exactly identical in more than one piggybacked app. Moreover, we noted that 124 methods of rider code are shared by more than 100 apps.

3. APPROACH

We remind the reader that one primary objective of our work is to provide researchers and practitioners means to systematize the collection of malicious packages that are used frequently by malware writers. To that end, we propose to devise an approach for **automating the localization of malicious code snippets** which are used pervasively in malware distributed as **piggybacked apps**. We are thus interested in identifying malicious rider code as well as the hook code which triggers the malicious behaviour in rider code. To fulfil this objective we require:

- First, a scalable and realistic approach to the detection of piggybacked apps. Unfortunately, state-of-the-art approaches have limitations in scalability or in practicality (we will give more details in the following subsection).
- Second, reliable metrics to automatically identify malicious payload code within a detected piggybacked app. To the best of our knowledge, in the literature, there are no such works have addressed this before.

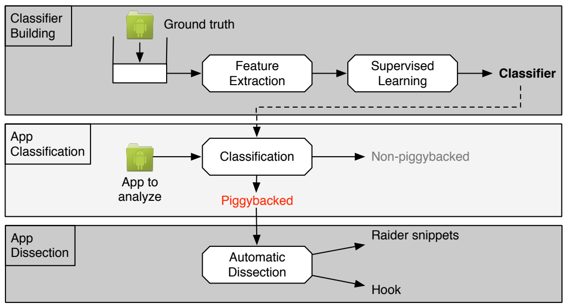

Figure 4: The main steps of our approach.

Figure 4 illustrates the main steps of our approach. We rely on the ground truth collected in Section 2 to build a machine learning classifier which will be applied for a given app to detect if this app is a piggybacked malware app or not. Then, if an app is flagged as piggybacked we statically analyze it to extract features on its constituting parts to distinguish the rider and hook code from the carrier code. The output of this step is a list of packages ranked with a probability score of maliciousness.

3.1 Piggybacked App Detection

We now provide details on the first step of our approach which consists in building a classifier for piggybacking detection. Contrary to previous work we have already collected a sizeable ground truth of piggybacked apps which can be immediately leveraged in our work. Before presenting the approach, we revisit current state-of-the-art approaches to list some limitations that motivated our use of Machine Learning techniques instead.

3.1.1 Limitations of the state-of-the-art

The problem of detecting piggybacked apps is eventually related to the problem of detecting app clones and repackaged apps. State-of-the-art approaches such as DroidMoss [15] or DNADroid [16] have focused on performing pairwise comparisons between apps to compute their degree of similarity. Obviously, such approaches, which are based on source code, cannot scale because of the combinatorial explosion of pairs to compare. For example, in the case of our dataset of 2 millions apps, there are $C^2_{2*10^6}$ candidate pairs for comparison[4].

With PiggyApp [9], the authors have improved this comparison approach by first extracting semantic features from the app components that they identified as implementing the primary functionality. They then build vectors using normalized values of the features. Thus, instead of computing the similarity between apps based on their code, they compute the distance of their associated vectors. This approach however remains impractical since one would require the dataset to contain exhaustively the piggybacked apps as well as their corresponding carrier apps. In practice however, many piggybacked apps are likely to be uploaded on different markets than where the original app can be found.

Conclusion. The main limitation in the state-of-the-art of piggybacked app detection is the requirement imposed by all approaches to have the original app in order to search for potential piggybacked apps which use it as a carrier. We claim however that because it is alien code that is grafted to an app to create a piggybacked app, there is a possibility to automate the identification of such apps separately from the original app, by considering directly how piggybacking is performed.

3.1.2 A Machine Learning-based approach

To get intuitions on how piggybacked apps are built, we consider samples from our ground truth and manually investigate how the piggybacked app differentiates from the original carrier app. Building on the characteristics of piggybacking that emerge, we propose a feature

[4] $C^2_{2*10^6} = 1.999999 * 10^{12}$. If we consider a computing platform with 10 cores each starting 10 threads to compare pairs of apps in parallel, we would still require several months to complete the analysis when optimistically assuming that each comparison would take about 1ms.

set for machine learning classification:

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

Listing 1: Simplified view of the *manifest* file of *se.illusionlabs.labyrinth.full* (app's sha256 ends with `7EB789`).

Component Capability declarations. In Android, *Intents* are the primary means for exchanging information between components. These objects contain fields such as the `Component name` to optionally indicate which app component to whom to deliver the object, some `data` (e.g., a phone number), or the `action`, which is a string that specifies the generic action to perform (such as *view* a contact, *pick* an image from the gallery, or *send* an SMS). 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 (e.g., a Messaging app will indicate being able to process intents with the `ACTION_SEND` action). Our manual investigation into piggybacked apps has revealed that they usually declare additional capabilities to the original apps to support the needs of the added malicious code. Usually, such new capabilities are used for the activation of the malicious behaviour. For example, piggybacked apps may add a new broadcast receiver with an intent-filter that declares its intention for listening to a specific event. Listing 1 illustrates an example of piggybacked app from our ground truth. In this Manifest file, components *CAdR* (line 12) is inserted and accompanied with the capability declaration for handling the `PACKAGE_ADDED` system event (line 13).

In our ground truth dataset, we have found that 813 (i.e., 78.9%) piggybacked apps added new capability declarations. Top intent-filter actions added by these apps include `PACKAGE_ADDED` (used in 696 piggybacked apps, but only in 66 original apps), and `CONNECTIVITY_CHANGE` (used in 681 piggybacked apps, but only in 36 original apps). Thus, some capability declarations are more likely to be found in piggybacked apps.

We have further 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 643 (i.e., 72.4%) piggybacked apps, several components have the same declared capability. In contrast, this duplication only happens in 100 (i.e., 9.7%) original apps. Thus, duplication of capability declarations (although for different components) may be indicative of piggybacking.

App Permission requests. Because Android is a privilege-separated operating system, where each application runs with a distinct system identity [17], every app must be granted the necessary permissions for its functioning. Thus, every sensitive resource or API functionality is protected by specific permissions (e.g., the Android permission `SEND_SMS` must be granted before an app can use the relevant API method to send a SMS message). Every app can thus declare in its Manifest file which permissions it requires given its use of the Android API, via the `<uses-permission>` tag. Malicious rider code in a piggybacked app often calls sensitive API methods or use sensitive resources that were not needed in the carrier app. For example, a game app can be piggybacked to send premium rate SMS. Thus, to allow the correct functioning of their malicious apps, piggybackers have to add new permission requests in the Manifest file.

The app example from Listing 1 illustrates this practice. Permission `GET_TASKS` has been added (line 3) along with the malicious code. Moreover, this example further supports the finding that piggybacking is done in an automated manner and with the least effort possible. Indeed, we note that a new entry requesting permission `WAKE_LOCK` (line 4) has been added in the piggybacked app, ignoring the fact that this permission had already been declared (line 2) in the carrier app. Within our ground truth, we have found that 87% (914) of piggybacked apps request new permissions. These new permission types also appear to be more requested by piggybacked apps than original apps. For example, permission `SYSTEM_ALERT_WINDOW` is additionally requested by 482 piggybacked apps, whereas this permission is only requested by 32 (i.e., 3%) original apps. Thus, some permissions requests may increase the probability for an app to be a piggybacked app. We have also identified 619 (i.e., 60%) piggybacked apps which duplicate permission requests. This happens more rarely in original apps: only in 91 (i.e., 8.8%) such apps. Thus, we consider that duplicating permission requests can be indicative of piggybacking as well.

Mismatch between Package and Launcher component names. As previously explained (cf. Section 2), 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. A mismatch between package name and the launcher component name can therefore be indicative of piggybacking. Again, the app example from Listing 1 illustrates such a case where the app's package (`se.illusionlabs.labyrinth.full`, line 1) and launcher (`com.loading.MainFirstActivity`, line 10) differ. In our ground truth, 1,045 piggybacked apps have a launcher component name that does not match with the app package name.

Package name diversity. 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 1. 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.

Sub-graph Density. Finally, we have computed the Package Dependency graph (PDG) of each piggybacked app in our study samples. We then noted that different modules (i.e., packages) can be located in a sub-graph with varying densities. Figure 5 illustrates the PDG of a piggybacked app, that is constituted of two disconnected sub-graphs whose densities are significantly different. One of the subgraph (the largest one in this case) represents the module constituting the rider code. We thus consider that comparing the variation between minimum, mean and maximum sub-graph's density can provide a hint on the probability for an app to be piggybacked.

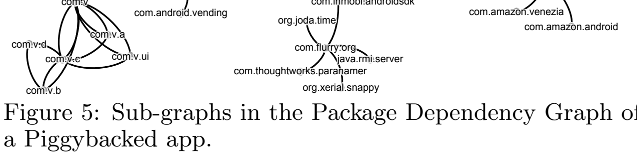

Figure 5: Sub-graphs in the Package Dependency Graph of a Piggybacked app.

Let E and V be the sets of edges and vertices of a PDG sub-graph sg. The density of sg is computed using formula (1).

$$density(sg) = \frac{|E|}{|V| * (|V| - 1)} \tag{1}$$

Summary of piggybacking detection approach: To design our machine-learning classifier, we build feature vectors with values inferred from the characteristics detailed above. Table 1 summarizes the feature set. The classifier is expected to infer detection rules based on the training dataset that we provide through the collected ground truth. For *duplicate capabilities*, *duplicate permissions* and *name mismatch* features, the possible values are 0 or 1. For the feature *new capability*, we analyze each app and for each of its declared capabilities we indicate in its feature vector the number of times this capability type was added by piggybacking in the training data. The same procedure is applied for *new permission* feature. For the *sub-graph density* feature, we add in the feature vector of each app, the minimum, maximum and average values of its PDG sub-graph densities. We then apply the ML algorithm which will then learn the correlation between feature vector values and piggybacking.

Table 1: Feature categories for ML classification.

Category	Feature	Values
App Info	new capability	occurrence frequency of capability in ground truth
	duplicate capabilities	boolean
	new permission	occurrence frequency of permission in ground truth
	duplicate permissions	boolean
Components	package diversity	Number of 'different package names'
	name mismatch	boolean
PDG	sub-graph density	max, min, mean

3.2 Identification of Malicious Riders and of Piggybacking Hooks

The ML classifier enables to produce a set of apps that are identified as piggybacked malware. These apps however require further analysis to *ungraft* the malicious rider code. To automate this approach, we consider the identification of malicious riders as a graph analysis problem.

Figure 6 illustrates the package dependency graph (PDG) of a piggybacked app[5]. The PDG is a directed graph which makes explicitly the dependency between packages. The values reported on the edges correspond to the number of times a call is made by code from a package A to a method in package B. These values are considered as the weights of the relationships between packages. In some cases however, this static weight may not reflect the relationship strength between packages since a unique call link between two packages can be used multiple times at runtime. To attenuate the importance of the weight we also consider a scenario where weights are simply ignored.

Identification of Hook/Rider Packages.

[5]Name: *rapscallion.sharq2*, the last six letters of its sha256 are *6486FE.*

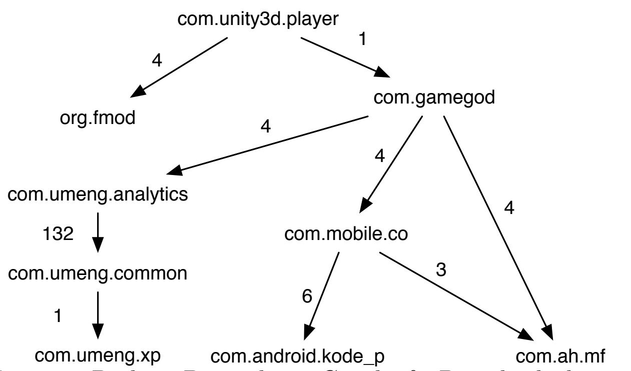

Figure 6: Package Dependency Graph of a Piggybacked app.

We then compute four metrics for estimating the relationships between packages in an app:

1. **weighted indegree**: In a directed graph, the indegree of a vertex is the number of headpoints adjacent to the vertex. In the PDG, the weighted indegree of a package corresponds to the number of calls that are made from code in other packages to methods in that package.

2. **unweighted indegree**: We compute the normal Indegree of a package in the PDG by counting the number of packages that call its methods. The reason why we take into account indegree as a metric is based on the assumption that hackers take the least effort to present the hook. As an example, *com.gamegod* in Figure 6 is actually the entry-point of the rider code, which has a smallest indegree for both weighted and unweighted indegree.

3. **maximum shortest path**: Given a package, we compute the shortest path to every other package, then we consider the maximum path to reach any vertex. The intuition behind this metric is based on our investigation with samples of piggybacked apps which shows that malware writers usually hide malicious actions far away from the hook, i.e., multiple call jumps from the triggering call. Thus, the maximum shortest path in rider module can be significantly higher than in carrier code.

4. **energy**: we estimate the energy of a vertex (package in the PDG) as an iterative sum of its weighted outdegrees and that of its adjacent packages. Thus, the energy of a package is total sum weight of all packages that can be reached from its code. The energy value helps to evaluate the importance of a package in the stability of a graph (i.e., how relevant is the sub-graph headed by this package?).

The above metrics are useful for identifying packages which are entry-points into the rider code. We build a **ranked list of the packages** based on a likelihood score that a package is the entry point package of the rider code. Let v_i be the value computed for a metric i described above ($i = 1, 2$ for in-degree metrics, the smaller the better; $i = 3, 4$ for others, the bigger the better) , and w_i the weight associated to metric i. For a PDG graph with n package nodes, the score associated to a package p, with our m metrics, is provided by formula (2).

$$s_p = \sum_{i=1}^{2} w_i * (1 - \frac{v_i(p)}{\sum_{j=0}^{n-1} v_i(j)}) + \sum_{i=3}^{4} w_i * (\frac{v_i(p)}{\sum_{j=0}^{n-1} v_i(j)}) \tag{2}$$

In our experiments, we weigh all metrics similarly (i.e., $\forall i, w_i = 1$). For each ranked packaged p_r, the potential rider code is constituted by all packages that are reachable from p_r. A hook is generally a code invocation from the carrier code to the rider code. Thus, we consider a hook as the relevant pair of packages that are interconnected in the PDG.

Finally, to increase accuracy in the detection of hooks we further dismiss such packages (in stand-alone hooks or in package-pair hooks) whose nodes in the PDG do not meet the following constraints:

- **No closed walk**: Because rider code and carrier code are loosely connected, we consider that a hook cannot be part of a directed cycle (i.e., a sequence of vertices going from one vertex and ending on the same vertex in a closed walk). Otherwise, we will have several false positives, since typically, in a benign app module (i.e., a set of related packages written for a single purpose), packages in the PDG are usually involved in closed walks as in the example of Figure 7.

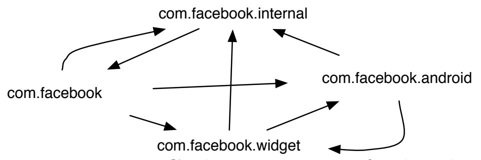

Figure 7: Excerpt PDG showing a set of related packages in the carrier code of *com.gilpstudio.miniinimo* (sha256 - *CB6534*)

- **Limited clustering coefficient**: A hook must be viewed as the connection link between carrier code and rider code via two packages. Since both packages belong to different (malicious and benign) parts of the app, they should not tend to cluster together in the package dependency graph as it would otherwise suggest that they are tightly coupled in the design of the app. To implement this constraint we measure the *local clustering coefficient* [18] of the vertex representing the carrier entry package. This coefficient quantifies how close its adjacent vertices are to being a clique (i.e., forming a complete graph). Given v, a vertex, and n, the number of its neighbors, its coefficient $cc(v)$

is constrained by formula (3)

$$cc(v) \begin{cases} < \frac{C_{n-1}^2}{C_n^2}, n \geq 2 \\ = 0, n < 2 \end{cases} \quad (3)$$

4. EVALUATION

We have implemented our approach for piggybacking detection and malware dissection. We now present the results of our evaluation campaign for the different steps of the approach.

4.1 Performance in Piggybacking Detection

Our machine learning-based detection of piggybacked apps is based on the Random Forests [19] ensemble learning algorithm. This algorithm, which operates by building a multitude of decision trees, was selected over simple decision trees because it is well known to correct for decision trees' habit of overfitting to the training dataset [20]. In our case, Random Forests are more suitable since our ground truth is limited by the criteria used to collect its sample apps.

Our evaluation of piggybacking detection is performed in two parts to answer two research questions:

RQ 1: Is the trained model accurate in discriminating piggybacked apps from non-piggybacked apps?

RQ 2: Can the classifier built on the collected ground truth generalize to other piggybacked apps?

To answer these questions we perform experiments in different settings, selecting the testing sets from the ground truth and in the wild (i.e., with market apps which are unlabelled).

4.1.1 10-Fold Cross Validation

We leverage the ground truth to directly measure the performance of our approach. To that end, we only train the classifier with a portion of the ground truth and use the remaining portion for testing the accuracy of the prediction. To ensure that our results are statistically significant and reduce variability depending on the partitioning of the dataset, we use 10-Fold cross validation which combines measures from several rounds. We are then interested in the measure of *recall* (i.e., the percentage of known piggybacked apps that are detected by the model) and *precision* (i.e., the percentage of all predicted piggybacked apps that are truly piggybacked). The harmonic mean of the two, called *F-measure*, gives an overall estimate of the performance. Our experiments show that our classification approach yields a performance of 97%, 97% and 97% respectively for precision, recall and F-measure. This high performance suggests that the considered features extracted in our approach are suitable for discriminating between piggybacked and non-piggybacked apps. However such extremely high performance are indicative of overfitting in the training model. We thus undertake to perform other evaluations in the following subsections.

4.1.2 Experiments in the Wild

In a second experiment, we consider all the ground truth to build a classifier that will be used to test a randomly selected subset of apps from our 2 millions dataset. The challenge is now to evaluate whether the predicted apps which are not part of our ground truth are indeed piggybacked malware. First we immediately verify based on VirusTotal labels that a classified piggybacked app is indeed flagged as malware. Second, for predicted apps that are indeed malicious we explore all other apps in an attempt to find its original counterpart. This simplifies the task of verification by having a baseline for comparison, beyond our limited hands-on experience with Android malware app writing.

Malicious piggybacking.

We randomly select 1000 apps from our initial dataset of 2 million apps, which consists of 650 goodware and 350 malware. The classification with this sample dataset yields a set of 290 apps as candidate piggybacked apps, where 130 of these apps are from the malware subset. These results do not presume of the performance of the approach. Instead, it may indicate that: (1) some apps, which are identified as piggybacked may indeed be benign. However, they present characteristics of clone or repackaged apps, which can be assimilated to "benign" piggybacking since the rider code is not malicious. This is the case for example when a legitimate adware library is inserted into benign apps; or (2) some apps, which are piggybacked apps presenting new malicious behaviours, had not yet been flagged by any anti-virus product at the time of analysis. We simply confirmed the plausibility of this explanation: we remind the reader that our initial set of apps is composed of 2 millions apps. After each app was downloaded, we immediately requested an analysis by VirusTotal anti-virus products. Thus, for many of the apps, the VirusTotal report is several months old. We have now requested an updated analysis to VirusTotal for the set of goodware apps which have been classified as piggybacked. Interestingly, we have found two of these apps are now labeled as malicious by at least one of the anti-virus products hosted by VirusTotal.

Previously unknown piggybacked apps.

We now verify whether the detected piggybacked apps are really piggybacked. To avoid the researcher bias that would come with manual verification, we proceed to automatically find the carrier app that will confirm the piggybacking. We focus on the malicious piggybacked and search the potential clones in a set of benign apps.

To find the original carrier apps of identified piggybacked apps, we proceed as in the PiggyApp [9] approach, and position all apps (i.e., over 1,4 million benign apps + 130 malicious piggybacked apps) in a high dimensional space where each app is represented as a vector of features which allow to distinguish it from the others. These features include permissions requested by the app, all actions declared in its manifests and all component names. The features are first represented into a vector of 0 and 1 values (where 0 indicates the absence of a feature in the app, and 1 its presence). The vectors, one for each app, are then projected into a metric space where the problem of detecting similar apps is reduced to a nearest neighbour search problem. The metric space is constructed using the Jaccard distance expressed by formula (4) to estimate the dissimilarity between the features of a pair of apps (a,b).

$$d_{Jaccard}(f_a, f_b) = \frac{|f_a \cup f_b| - |f_a \cap f_b|}{|f_a \cup f_b|} \quad (4)$$

Out of the 130 identified malicious piggybacked apps, we were able to find the original benign carriers of 53 apps (∼41%). Since 1) our search space, with benign apps, was limited by the available dataset and 2) it is possible that some carrier apps are already classified as malicious and thus not included in our search space, the remaining apps, for which we did not find the original counterparts, cannot be classified as false positives of our approach either.

Comparison with other approaches..

The closest work, which is comparable to ours is PiggyApp [9]. Unfortunately, the authors (1) do not release their dataset, preventing us to apply our approach and compare results, (2) do not give the values of some important parameters of their approach (e.g., not knowing the threshold of similarity for deciding that two primary modules are cloned) preventing us to implement their approach for comparison. We contacted the PiggyApp authors about these issues but did not hear back from them.

4.2 Performance in Malware Dissection

We now evaluate the second step of our approach which consists of automating the dissection of piggybacked malware apps to identify rider and hook code.

The output of the malicious rider identification step being a ranked list of packages, our evaluation consists in verifying the percentage of rider packages in the top 5 items (i.e., accuracy@5) in the list that are effectively correctly identified, and the proportion of rider code that is included in the list. For this experiment we have randomly selected 500 piggybacked apps from our collected ground truth (cf. Section 2). First, we automatically build the baseline of comparison by computing the `diff` between each of the selected piggybacked app and its corresponding original app. With this `diff`, we can identify the rider code and the hook.

Then, we apply our dissection approach to the 500 apps and compare the top ranked packages against the baseline. Our approach yields an accuracy@5 of 83% and an accuracy@1 of 68.4% for rider packages.

The manual analysis further provided some insights on how malware writers perform piggybacking at a large scale. Table 2 presents three samples of hook code (at the package level) which suggest that **piggybackers often connect their payload to the carrier via one of its included libraries**. Thus, malware can systematize the piggybacking operation by targeting apps that use some popular libraries. For example library package `com.unity3d.player` is the infection point in 53 (out of the 500) piggybacked apps. In 14 of those apps, the entry package of the rider code is `com.gamegod`.

Table 2: Three hook samples and their affected number of apps.

Hook	Affected Apps (#.)
com.unity3d.player → com.gamegod	14
com.unity3d.player → com.google.ads	7
com.ansca.corona → com.gamegod	4

5. OUTLOOK & DISCUSSION

Our prototype implementation has demonstrated promising results in automating the dissection of Android malware. First we have devised a ML classifier to detect piggybacked apps, then we have proposed an approach to localize the malicious payload at the granularity level of packages. We now discuss how the approach and datasets built in our work can be exploited in the hunt for malware in the Android ecosystem. Subsequently, we enumerate a few threats to validity.

5.1 Malware Detection

Our work was motivated by the need for building effective malware detectors for Android. After collecting snippets of malicious rider code from piggybacked apps, we explore their potential for improving malware detection approaches.

5.1.1 Basic Malware detection

In a first scenario, we consider the case of machine learning-based malware detection leveraging features of the identified rider code in our ground truth of piggybacked apps. The malware prediction in this case is a one-class classification problem as we only have knowledge on what features a malware should include. We first apply the classifier built with these new features on our ground truth. In 10-fold cross validation experiments, we recorded an accuracy of 91.6%. These results suggest that rider code features are effective in detecting piggybacked malware.

We further investigate the MalGenome dataset to determine the proportion of malicious apps which share the same malicious payloads with the piggybacked apps of our ground truth. To that end we consider the package dependency graph of each app of the MalGenome dataset and map them with the collection of rider package pairs collected in our ground truth. 125 MalGenome apps contain only 1 package. They are thus irrelevant for our study. Among those apps with several packages, 252 (i.e., 22.2%) contained rider code features of our ground truth. With a malware detection tool based on our rider code collection, we could have directly flagged such apps with no further analysis.

5.1.2 Malware family classification

In a second scenario, we consider the case of classifying malware to specific families based on the rider features. To that end, we consider the apps of our ground truth dataset and apply our dissection approach. We then collect the identified rider code of all apps and apply the Expectation-Maximization (EM) [21] algorithm[6] on the edges related to rider code in the app PDG to cluster them. This leads to the construction of 5 clusters of varying sizes. Our objective is then to investigate whether the clusters of rider code thus built are also related to specific malware families. To that end, we consider the labels[7] that anti-virus products provide after analysing the piggybacked apps corresponding to the rider code in each cluster.

We compute the Jaccard distance between the sets of labels for the different clusters. The results summarized in Table 3 reveals that the malware labels in a given cluster are distant from those of any other clusters. This suggests the dissected rider code contribute to malware of specific families.

Table 3: Jaccard distance (dissimilarity) of malware label sets between clusters built based on rider code. The first two rows show the number of apps and anti-virus labels in each cluster, respectively.

#. of Apps	10	236	27	7	37
#. of ≠ labels	22	269	51	5	27
	C_1	**C_2**	**C_3**	**C_4**	**C_5**
C_1	0	0.96	0.89	0.92	0.88
C_2	0.96	0	0.91	0.99	0.96
C_3	0.89	0.91	0	0.94	0.94
C_4	0.92	0.99	0.94	0	0.95
C_5	0.88	0.96	0.94	0.95	0

We also consider clustering the piggybacked apps based directly on the malware labels. The EM algorithm produces six clusters. We then compute the Jaccard distance between each of those clusters and the 5 clusters of apps previously constructed based on rider code. Table 4 summarizes the results which reveals that each cluster (built based on malware labels) is much closer to a single cluster (built based on rider code) than to any other clusters. The difference are not significantly high for clusters C_1 and C_4, two cases where the contained app sets are small. Nevertheless, these experiment results overall illustrate that the malicious code ungrafted from piggybacked apps indeed represent a signature of a malware family.

[6] This algorithm is able to decide itself an appropriate number of groups to cluster.

[7] An anti-virus label (e.g., *Android.Trojan.DroidKungFu2.A*) represents the signature identified in a malicious app.

Table 4: Jaccard distance between clusters of apps. MC_i is a cluster built based on anti-virus labels, while C_i is a cluster built based on rider code features.

#. of Apps	90	44	69	47	14	53
#. of ≠ labels	170	82	35	57	13	67
	MC_1	**MC_2**	**MC_3**	**MC_4**	**MC_5**	**MC_6**
C_1	0.90	0.90	0.84	0.87	0.97	0.91
C_2	0.43	0.70	0.88	0.79	0.96	0.85
C_3	0.91	0.90	0.93	0.89	0.93	0.24
C_4	0.99	0.94	0.95	0.97	0.94	0.96
C_5	0.96	0.93	0.73	0.94	0.55	0.95

5.2 Threats to Validity and Limitations

Our approach and the experiments presented in this work present a few threats to validity. First of all, the collected ground truth is constrained to specific types of piggybacking. However, with our experiments in the wild, we have demonstrated that the collected dataset is diverse enough to allow identifying new piggybacked apps. Second, we only focus on a specific kind of piggybacking where rider code is shipped with its own packages, i.e. the carrier and the rider are only linked by the hook. Third, our dissection is at the granularity level of packages.

6. RELATED WORK

The problem of piggybacked app detection is closely related to that of Software clone detection in general, and repackaged app detection in particular. Our work is also relevant to the field of malware detection in the wild.

6.1 Code clone & plagiarism detection

Traditional code clone detection approaches [22–26] work at the level of a fragment of code (or graphs/trees), the objective being to measure the similarity of two fragments of code (or graphs/trees). Generally, this code fragment is a method/function. Work however exists for detecting higher-level clones (e.g., file-level clones) [27]. Cesare et al. have also recently proposed Clonewise [28], an approach to detect package-level clones in software packages. Even if the notion of clone fragment, be it a method, file or package, could be very useful for app similarity measurements, it is not sufficient in the context of Android, since Android apps are composed of multiple components. In other words, two apps with similar code fragments are not necessarily similar. Note that mostly this is the case, mainly because of the intensive use of libraries in Android apps.

Closely related to our work is Clonewise which, to the best of our knowledge, is the first to consider clone detection as a classification problem. Our approach, also in contrast with state-of-the-art, considers piggybacking detection as a classification problem to enable a practical use in real-world settings.

6.2 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 [15, 16, 29, 30]. With DNADroid [16] Crussell et al. presented a method based on program dependency graphs comparison. The authors later built on their DNADroid approach to build AnDarwin [30] an approach that uses multi-clustering to avoid the scalability issues induced by pairwise comparisons. DroidMOSS [15] 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 [31] use frequencies of occurrence of various OpCodes to cluster Android malware into families. Finally, in [32], 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 [33] to compute a measure of similarity on the apps' resource files. Similarly, ResDroid [34] 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 [35] 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, practically all listed approaches perform similarity computations through pair-wise comparisons. 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, in our work, we rely on semantic features of apps to achieve better efficiency in detection.

6.3 Piggybacked app search and Malware variants detection

Cesare and Xiang [36] have proposed to use similarity on Control Flow Graphs to detect variants of known malware. Hu, Chiueh, and Shin [37] 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 [38] 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 [9] 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.

6.4 ML-based malware detection

In a recent study with antivirus products we have shown that malware is still widespread within Android markets. This finding is inline with regular reports from Anti-virus companies where they reveal that Android has become the most targeted platform by malware writers. Research on systematic detection of Android malware is nevertheless still maturing. Machine learning techniques, by allowing sifting through large sets of applications to detect malicious applications based on measures of similarity of features, appear to be promising for large-scale malware detection [4, 39–42].

Researchers use a diverse set of features to detect malware. In 2012, Sahs and Khan [4] built an Android malware detector with features based on a combination of Android-specific permissions and a Control-Flow Graph representation. Use of permissions and API calls as features was proposed by Wu et al [43]. In 2013, Amos et al [44] leveraged dynamic application profiling in their malware detector. Demme et al [1] also used dynamic application analysis to profile malware. Yerima et al [2] built malware classifiers based on API calls, external program execution and permissions. Canfora et al [3] experimented feature sets based on SysCalls and permissions.

Unfortunately, through extensive evaluations, the community of ML-based malware detection has not yet shown that current malware detectors for Android are actually efficient in detecting malware in the wild. Chief among the candidate reasons to this situation is the fact that features are "elaborated" by research teams based on the behaviour of specific malware families whose behavioural description has provided the intuitions for constructing the classifiers. Our work will allow for better classification of malware, e.g., through the implementation of multi-classifiers, taking into account the different ways that exist for writing malware (and indirectly the different structures and behaviours of malware).

7. CONCLUSION

We have proposed in this paper an approach for detecting and dissecting piggybacked apps to localize and collect malicious samples. We consider piggybacking detection as a classification problem and investigated a collected clean ground truth to infer the most reliable features to leverage for the machine learning experiments. Through extensive evaluations, we have demonstrated the performance of our approach in piggybacked detection and malware dissection. Finally, we have shown how collected malicious payload (i.e., rider code) information can be used to detect malicious apps. Further investigations revealed how rider code clusters correlate with malware signatures by anti-virus products.

Acknowledgments

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under the project AndroMap C13/IS/5921289.

8. REFERENCES

[1] 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.

[2] 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.

[3] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. A classifier of malicious android applications. In *Availability, Reliability and Security (ARES), 2013 eight International Conference on*, 2013.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*. IEEE, 2014.

[12] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Parameter Values of Android APIs: A Preliminary Study on 100,000 Apps. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[13] 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.

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

[15] 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.

[16] 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 âĂŞ ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 37–54. Springer Berlin Heidelberg, 2012.

[17] System permissions. http://developer.android.com/guide/topics/security/permissions.html. Accessed: 2015-08-23.

[18] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of /'small-world/' networks. *Nature*, 393(6684):440–442, 06 1998.

[19] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[20] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[21] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):pp. 1–38, 1977.

[22] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEENâĂŹS UNIVERSITY*, 115, 2007.

[23] B.S. Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95, Jul 1995.

[24] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377, Nov 1998.

[25] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[26] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDDâĂŹ06*, pages 872–881. ACM Press, 2006.

[27] H.A. Basit and S. Jarzabek. A data mining approach for detecting higher-level clones in software. *Software Engineering, IEEE Transactions on*, 35(4):497–514, July 2009.

[28] Silvio Cesare, Yang Xiang, and Jun Zhang. Clonewise âĂŞ 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, December 2006.

[40] Boyun Zhang, Jianping Yin, Jingbo Hao, Dingxing Zhang, and Shulin Wang. Malicious codes detection based on ensemble learning. In *Proceedings of the 4th international conference on Autonomic and Trusted Computing*, ATC'07, pages 468–477, Berlin, Heidelberg, 2007. Springer-Verlag.

[41] R. Perdisci, A. Lanzi, and Wenke Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 301–310, 2008.

[42] 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.

[43] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69, 2012.

[44] Brandon Amos, Hamilton Turner, and Jules White. Applying machine learning classifiers to dynamic android malware detection at scale. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 1666–1671, 2013.