Li L, Li D, Bissyandé TF *et al.* On locating malicious code in piggybacked Android apps. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 32(6): 1108–1124 Nov. 2017. DOI 10.1007/s11390-017-1786-z

On Locating Malicious Code in Piggybacked Android Apps

Li Li[1], Daoyuan Li[1], Tegawendé F. Bissyandé[1], Jacques Klein[1], Haipeng Cai[2], *Member*, *ACM*, *IEEE*
David Lo[3], and Yves Le Traon[1]

[1]*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg 2721, Luxembourg*
[2]*School of Electrical Engineering and Computer Science, Washington State University, Washington, WA 99163, U.S.A.*
[3]*School of Information Systems, Singapore Management University, Singapore 178902, Singapore*

E-mail: {li.li, daoyuan.li, tegawende.bissyande, jacques.klein}@uni.lu; hcai@eecs.wsu.edu; davidlo@smu.edu.sg
yves.letraon@uni.lu

Received April 20, 2017; revised October 13, 2017.

Abstract To devise efficient approaches and tools for detecting malicious packages in the Android ecosystem, researchers are increasingly required to have a deep understanding of malware. There is thus a need to provide a framework for dissecting malware and locating malicious program fragments within app code in order to build a comprehensive dataset of malicious samples. Towards addressing this need, we propose in this work a tool-based approach called HookRanker, which provides ranked lists of potentially malicious packages based on the way malware behaviour code is triggered. With experiments on a ground truth of piggybacked apps, we are able to automatically locate the malicious packages from piggybacked Android apps with an accuracy@5 of 83.6% for such packages that are triggered through method invocations and an accuracy@5 of 82.2% for such packages that are triggered independently.

Keywords Android, piggybacked app, malicious code, HookRanker

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 work[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 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 is 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 either which malicious actions are actually detected, or 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

Regular Paper
Special Section on Software Systems 2017
This work was supported by the Fonds National de la Recherche (FNR), Luxembourg under projects AndroMap C13/IS/5921289 and Recommend C15/IS/10449467.

Jiang[6] manually investigated 1 260 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 packages (e.g., privilege escalation or personal information stealing); 4) how malware exploits the permission system. The produced dataset named MalGenome, has opened several directions in the research of malware detection, most of which either have 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; the characterization provided in the study is too high-level to allow the inference of meaningful structural or semantic features of malware.

The ultimate goal of our work is to build an approach towards 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 malicious packages, is referred to as a piggybacked app. Our assumption that most malware is piggybacked of benign apps is confirmed with the MalGenome dataset where over 80% of the samples were built through repackaging. For simplicity, in this entire paper we refer to any code package injected via piggybacking as a "malicious" package. Actually, such a package may 1) directly contribute in implementing the malicious behaviour, 2) contribute in further hiding malicious operations to static analyzers, or 3) provide commodity functions (e.g., in the form of a library) which are exploited by piggybackers to facilitate payload hooking.

Identifying and extracting accurately malicious code in an app is however a challenging endeavour. In any case, a malicious behaviour can be implemented as an orchestration of different behaviour steps in several packages. To the best of our knowledge, state-of-the-art studies mainly leverage comparison-based approaches (either 1-to-1[9] or 1-to-n[10] comparison) to pinpoint malicious payloads. Approaches analysing solely a malware sample, to systematically identify packages which contribute to malicious behaviour implementation, are scarce. Our objective is therefore to propose a step towards helping analysts to readily identify malicious packages in Android apps without requiring the availability of other apps for comparison. To that end, we build HookRanker, a ranking approach which orders packages with regard to the likelihood of their malicious status. Overall, we make the following contributions.

• We propose an automated approach for locating hooks (i.e., code that either switches the execution context from benign to malicious code or triggers malicious code independently) within piggybacked apps. Our approach eventually yields two ranked lists of most probable malicious packages, which can benefit malware analysts to quickly understand how the malicious behaviour is implemented and how the malicious code is triggered. A key characteristic of our approach is that it does not require to have the original benign version of the piggybacked app, which is usually hard to harvest, in order to perform some form of difference analysis.

• We present a tool called HookRanker to automatically recommend potential malicious packages and components. Evaluations on a set of benchmark apps have demonstrated that HookRanker is efficient to locate malicious packages of piggybacked apps.

• We experimentally show that our work can immediately be leveraged, to some extent, by researchers and practitioners to build classifiers that output explainable results, i.e., when an app is flagged as a malware, one understands precisely that it exhibits features from a particular malicious package, and thus it is straightforward to indicate the relevant type/family of malware.

Reproducibility. We make our dataset and experimental results available online①.

This paper is an extended and improved version of a short paper[11] presenting preliminary results at the 2017 International Conference on Mobile Software Engineering and Systems (MobileSoft). In the previous version, we have explored solely $Type_1$ hook for piggybacked apps, although we have actually shown that there are in total two types of hooks (including $Type_1$ and $Type_2$ hooks (see Listing 1)). In this extension, in addition to $Type_1$ hook, which involves method calls for triggering piggybacked rider code, we further explore

①https://github.com/serval-snt-uni-lu/HookRanker, Nov. 2017.

```
//Activity for launching the app
public class com.unity3d.player.UnityPlayerProxyActivity extends android.app.Activity {
 protected void onCreate(android.os.Bundle) {
 $r0 := @this: com.unity3d.player.UnityPlayerProxyActivity;
 $r1 := @parameter0: android.os.Bundle;
 $b0 = 1;
 specialinvoke $r0.<android.app.Activity: void onCreate(android.os.Bundle)>($r1);
+ staticinvoke <com.gamegod.Touydig: void init(android.content.Context)>($r0);
 $r2 = newarray (java.lang.String)[2];
 $r2[0] = "com.unity3d.player.UnityPlayerActivity";
 $r2[1] = "com.unity3d.player.UnityPlayerNativeActivity";
 staticinvoke <com.unity3d.player.UnityPlayerProxyActivity: void
     copyPlayerPrefs(android.content.Context,java.lang.String[])>($r0, $r2);
}}

//Broadcast Receiver for listening PACKAGE_ADDED, CONNECTIVITY_CHANGE, and BOOT_COMPLETED events
+ public class com.mobile.co.UR extends AdPushReceiver {...}
```

Listing 1. Example of $Type_1$ and $Type_2$ hooks. This snippet is extracted from a real piggybacked app named apscallion.sharq2. The "+" sign indicates the code that was injected into the origin app.

$Type_2$ hook for piggybacked malicious apps, where the malicious rider code is triggered through the use of Android event system.

The remainder of this paper is organized as follows. Section 2 provides the necessary background information related to piggybacked apps, including the piggybacking terminology to which we will refer in this paper. Section 3 presents our approach for automatically locating malicious packages in piggybacked apps. We evaluate our work in Section 4 and discuss the threats to validity as well as outlook in Section 5. Section 6 discusses related work and Section 7 concludes this paper.

2 Preliminaries

We now provide preliminary details that are essential for understanding the purpose, techniques and key concerns of Android piggybacking. In particular, we first briefly introduce the terminology related to the piggybacking process in Subsection 2.1. Then, in Subsection 2.2, we present the Android app launch model, which is central to how malicious packages in piggybacked apps can be reached for triggering malicious behaviour. Next, we summarize in Subsection 2.3 the techniques that are leveraged by malware writers to graft piggybacking code with existing app code. Finally, in Subsection 2.4, we present the ground truth dataset that we use in this work to evaluate the effectiveness of HookRanker.

2.1 Piggybacking Terminology

We now introduce the necessary terminology to which we will refer in the remainder of this paper. Fig.1 shows the constituting parts of a piggybacked malware[2], which is built by taking a given original app, referred to in the literature as the carrier[12], and grafting malicious packages to it (also known as a piece of malicious code[3]), referred to as the rider. The malicious behaviour will be triggered thanks to the hook that is inserted by the malware writer to ensure the injected packages will be executed.

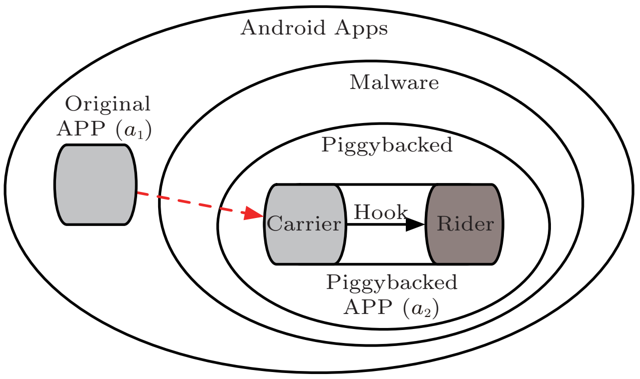


Fig.1. Piggybacking terminology.

It is also noteworthy that, in this work, we make a clear difference between piggybacking and repackaging, two terms that are frequently used in the literature. Indeed, unlike piggybacking, repackaging does not necessarily include a modification of the bytecode of a given

[2]In this work, we focus on piggybacked malicious apps, where the malicious state is ensured by the results of VirusTotal.

[3]To simplify the description, in this work, we consider all the injected code as malicious, even if the actual malicious payload is only some part of the added code.

Android app. Instead, repackaging may simply be performed to change the app certificate and thus switch the ownership. However, piggybacking always implies repackaging.

2.2 Android App Launch Model

Android apps are made up of four types of components:

• `Activity`, which represents the graphical interface of Android apps;

• `Service`, which is dedicated to performing time-intensive tasks in the background;

• `Broadcast Receiver`, which is used in waiting and resolving system as well as user-defined events;

• `Content Provider`, which provides a standard interface for other components/apps to access app data.

Unlike traditional Java applications, which include a single entry point (i.e., the *main*() function) to launch the program, Android apps contain multiple entry points through which some parts of the app code can be triggered: basically every component could be an entry point. This situation can be exploited by piggybackers as opportunities for triggering the execution of their injected malicious packages. Fig.2 summarizes typical examples of the common launch model of Android apps. It illustrates that in addition to the normal launch process (launcher), Android apps can actually be triggered through system events and user-defined events.

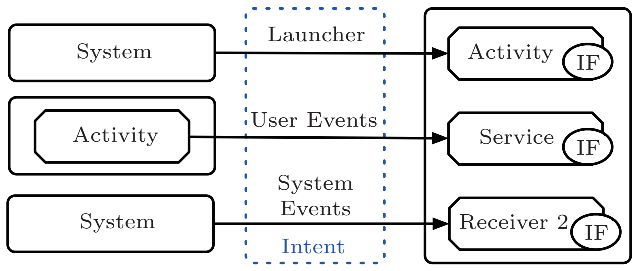


Fig.2. Examples of the Android app launch model. IF indicates intent filter.

Actually, the three aforementioned entry point types are based on the inter-component communication (ICC) mechanism. Each entry point (i.e., component) has to declare at least one[4] intent filter to specify how it could be launched. In order to be a launcher entry point, as shown in Listing 2, the launcher component (activity in this case) has to declare an intent filter with an action attribute named MAIN and a category attribute named LAUNCHER (lines 24∼25). Similarly, in order to be a system event-triggered entry point, a component (usually receiver) must declare intent filters to listen for some system events. When the declared system events are fired, the component will be triggered. Both a launcher entry point and a system event triggered entry point can be used to start an app, but they differ by the fact that a launcher entry point can also be triggered via user events, e.g., an intent object (with MAIN and LAUNCHER attributes filled) constructed with explicit targets in mind.

```
20 <manifest package="rapscallion.sharq2">
21 <application>
22 activity:".UnityPlayerProxyActivity"
23   intent-filter
24   action:"android.intent.action.MAIN"
25   category"android.intent.category.LAUNCHER"
26
27 receiver:"com.mobile.co.UR"
28   intent-filter
29   action:"android.intent.action.PACKAGE_ADDED"
30   data:"package"
31   intent-filter
32   action:"android.net.conn.CONNECTIVITY_CHANGE"
33   intent-filter
34   action:"android.intent.action.BOOT_COMPLETED"
35 </application></manifest>
```

Listing 2. Simplified manifest of app *apscallion.sharq*2.

2.3 Hook Types

Given the app launch model described above, we infer that there are two ways for piggybackers to hook their malicious code from the carrier code, i.e., to allow the triggering of the payload in their injected malicious packages. We refer to these two ways as $Type_1$ hooks and $Type_2$ hooks.

• $Type_1$ hook involves method calls that explicitly connect carrier code to rider code. In this case, we identify the hook via the point[5] where the carrier code is switched into the rider code in the execution flow. Listing 1 shows a snippet illustrating an example of $Type_1$ hook (line 8), which is inserted immediately at the beginning of the *onCreate*() method (line 7) of component *UnityPlayerProxyAct*. As shown in Listing 2, *UnityPlayerProxyActivity* is actually the app launcher, as indicated by the *MAIN* action and *LAUNCHER* category. When users launch the app, the first lifecycle method *onCreate*() of component *UnityPlayerProxyActivity* will be triggered. Consequently, the malicious packages (starting from class *com.gamegod.Touydig*) will immedi-

[4]It is possible to declare several intent filters. As shown in Listing 2, component com.mobile.co.UR declared three intent filters.

[5]In our implementation we focus on identifying the Java package.

ately be triggered (by calling the method *init*()), switching the current execution context to piggybacked code.

• $Type_2$ hook involves the use of the Android event system. Thus, the piggybacked code hooking is done via a component that is explicitly connected to any code of the original app. Instead, the (malicious) rider code will be triggered directly by system or user-defined events. Listing 1 also includes an example of $Type_2$ hook (line 16), where the whole component named *com.mobile.co.UR* is injected during piggybacking. Listing 2 illustrates the capabilities declared for this component, which is registered to listen to three different system events: 1) *PACKAGE_ADDED* will be fired when a new app is installed on the device; 2) *CONNECTIVITY_CHANGE* will be fired when a change related to network connectivity has occurred; 3) *BOOT_COMPLETED* will be fired after the booting process has completed. When any one of the aforementioned events is broadcasted, hook *com.mobile.co.UR* will be triggered and consequently the malicious packages will be executed.

It is worth noticing that thanks to the definition of piggybacked apps, which have been grafted with malicious packages, there will be no such case that neither $Type_1$ nor $Type_2$ hooks are applied to piggybacked apps.

2.4 Piggybacking Ground Truth Dataset

In this work, we leverage the ground truth that we built in previous work[13] to perform our investigation. This ground truth contains hundreds of pairs of piggybacked and associated benign apps which were collected from a large repository of millions of apps[14] crawled over several months from several markets such as the Google Play store, AppChina, and which was used for large-scale experiments[15-17]. Each pair (a_b, a_m) consists of a benign app (a_b) and a piggybacked malicious app (a_m, where a_b is the benign original counterpart of a_m). As shown in Fig.3, we carefully ensure that each pair of apps 1) have identical app package name[6], 2) are written by different authors[7], 3) have the same SDK version, and 4) have at least 80% similar code[18]. The malicious state of a given app is checked via VirusTotal[8], which hosts over 50 anti-virus products from providers like Kaspersky, McAfee.

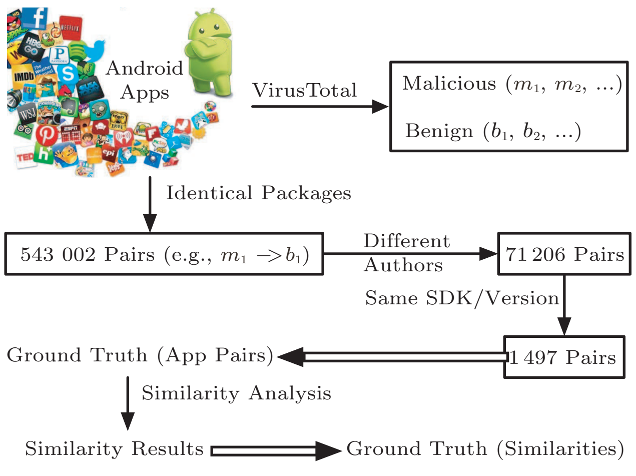


Fig.3. Ground truth building process.

3 Approach

Our primary objective of this work is to provide researchers and practitioners with 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 identification 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 fulfill this objective we require a set of reliable metrics to automatically identify malicious packages within a detected piggybacked app.

Given a set of piggybacked malware, we aim at identifying the hooks that trigger the execution of rider code and thereby ungrafting the malicious rider code from piggybacked malicious apps. As introduced in Section 2, there are two types of hooks leveraged by piggybackers. These two types of hooks are significantly different in terms of their behaviours, making it difficult to identify them through a single generic approach. Therefore, in this work, we present two separate techniques to identify the hooks. Fig.4 gives an overview of our approach, which takes as input a single Android app and outputs two recommended hook lists with most likely hook code being preferentially ranked. These two ranked lists can then be leveraged by applications or users to support the development of many other implications such as malware detection or app repairing. We

[6]App package name is specified by the package attribute of the manifest configuration (e.g., line 20 in Listing 2). Two apps with the same app package name cannot be installed on the same device.

[7]We do not consider cases where developers piggyback their own apps.

[8]http://virustotal.com, Oct. 2017. We take an app as malicious if at least one anti-virus product flags it as such.

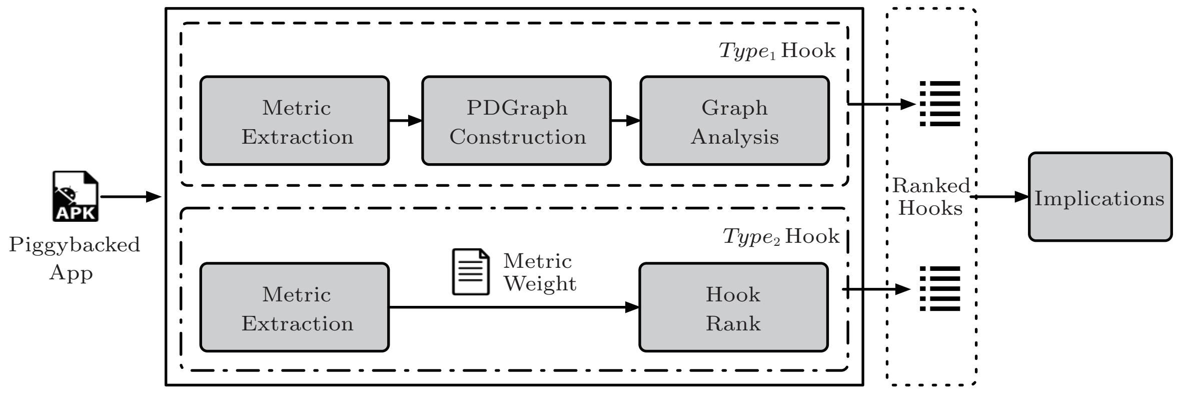


Fig.4. Overview of our approach.

now detail these two approaches in Subsection 3.1 and Subsection 3.2 respectively.

3.1 $Type_1$ Hook Identification

To automate this approach, we consider the identification of $Type_1$ hook as a graph analysis problem. Fig.5 illustrates the package dependency graph (PDGraph) of a piggybacked app (the same app as we used in Listing 1). PDGraph is a directed graph which makes explicit the dependency between packages. The values reported on the edges correspond to the times a call is made by code from 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.

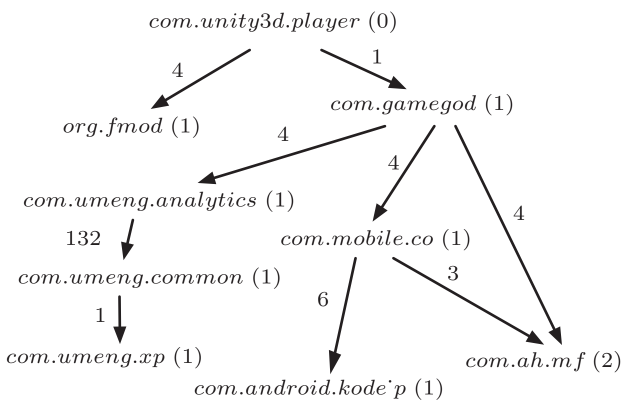


Fig.5. Package dependency graph of a piggybacked app. Numbers between parentheses indicate the unweighted indegree values while numbers near edge lines indicate weighted indegree values.

We now 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 PDGraph, 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 PDGraph 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 Fig.5 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, and 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., the multiple call jumps from the triggering call. Thus, the maximum shortest path in the rider module can be significantly higher than that in carrier code.

4) *Energy*. We estimate the energy of a vertex (package in the PDGraph) 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 subgraph 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 the others, the bigger the better), and w_i is the weight associated to metric i. For a PDGraph graph with n package nodes, the score associated to a package p, with our proposed metrics, is provided by (1).

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

In our experiments, we weigh all metrics similarly (i.e., $\forall i, w_i = 1$). For each ranked package p_r, the potential rider code is constituted by all packages that are reachable from p_r. A hook is generally a method invocation from the carrier code to the rider code. Thus, we consider a $Type_1$ hook as the relevant pair of packages that are interconnected in the PDGraph.

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 PDGraph 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 PDGraph are usually involved in closed walks as in the example of Fig.6.

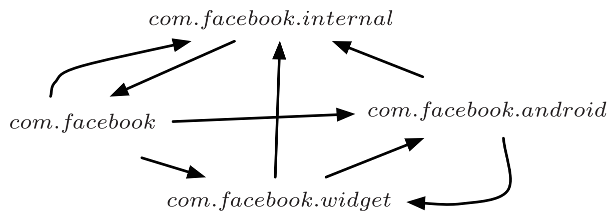


Fig.6. Excerpt PDGraph showing a set of related packages in the carrier code of *com.gilpstudio.miniinimo*.

• *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[19] of the vertex representing the carrier entry package. This coefficient quantifies how close its adjacent vertices are to be a clique (i.e., forming a complete graph). Given v, a vertex, and n, the number of its neighbors, the vertex's coefficient $cc(v)$ is constrained by (2).

$$\begin{cases} cc(v) < \frac{C_{n-1}^2}{C_n^2}, \text{ if } n \geqslant 2, \\ cc(v) = 0, \quad \text{otherwise.} \end{cases} \quad (2)$$

3.2 $Type_2$ Hook Identification

We leverage three metrics to assess the likelihood for a given app component to be a $Type_2$ hook. These metrics are designed based on the following Android concepts used in $Type_2$ hooking.

1) *Intent Filter*. In Android, an `Intent` is a specific type of object that is dedicated to support inter-component communication (ICC) within or across Android apps. Incoming Intents are then resolved based on `Intent Filter` declarations in apps' Manifest files. Components generally declare intent filters to describe their capabilities, i.e., what kind of actions they can perform (e.g., they can open audio files, view PDF files) or which types of events they are waiting for to engage in some processing (e.g., when bootup is completed, when Wi-Fi status changes). The more intent filters a component declares, the more intents it will catch, and consequently, and the more likely its code will be executed.

The example shown in Listing 2 presents such a case where a component, namely *com.mobile.co.UR*, in a piggybacked app, is associated with three unrelated intent filters. This suggests that piggybackers are maximizing the opportunities for triggering the execution of their injected malicious packages.

2) *Action*. In Android, `action` is dedicated to specifying the generic activity, such as view, that components can perform. Intent filters leverage this attribute to describe their capabilities. Generally, the more actions declared for a component, the most likely the component will be triggered.

3) *Category*. Together with `action` attribute, `category` provides additional information about the kind of intent that components should handle. Basically, category raises the bar for a given component to

resolve incoming intents because both actions and categories need to be matched. As a result, the more categories declared for a given intent filter (or component), the less likely the component will be executed.

The above information can be used as metrics for computing the likelihood scores of components to be used as $Type_2$ hooks. Based on these scores, we can build a ranked list of the components to recommend potential $Type_2$ hooks. Let m be the total number of intent filters declared by component c, f_i be the score of the i-th intent filter of component c, and w_i be the weight associated to f_i, the score of the component s_c can be computed through (3).

$$s_c = \sum_{i=1}^{m} w_i \times f_i. \tag{3}$$

We can further calculate f_i through (4), where p and q denote the number of actions and categories of f_i respectively, a_j and g_k stand for the score of the j-th action and the k-th category of f_i respectively while w_j and w_k are the weights associated to a_j and g_k, respectively.

$$f_i = \sum_{j=1}^{p} w_j \times a_j - \sum_{k=1}^{q} w_k \times g_k. \tag{4}$$

In our experiments, we give a base score 1 to all actions and categories appearing for intent filters of components, i.e., $\forall j, k, a_j = 1$ and $g_k = 1$. We also weight all metrics except action similarly, i.e., $\forall i, k, w_i = 1$ and $w_k = 1$. For metric action, because of its importance, we weight it through a mapping illustrated in Table 1. The "count" column shows the number of occurrences of the action indicated in the first column in all the piggybacked apps that we have considered. The weight is computed through the natural logarithm (base 10) of the count number (e.g., $4 = \lceil \log(1\,693) \rceil$).

Table 1. Mapping from Action to Its Weight

Action	Count	Weight
android.intent.action.PACKAGE_ADDED	1 693	4
android.net.conn.CONNECTIVITY_CHANGE	1 560	4
android.intent.action.USER_PRESENT	1 279	4
android.intent.action.CREATE_SHORTCUT	359	3
android.intent.action.PACKAGE_REMOVED	281	3
android.intent.action.BOOT_COMPLETED	239	3
android.intent.action.MAIN	118	3
android.intent.action.VIEW	36	2
android.intent.action.SIG_STR	26	2
android.provider.Telephony.SMS_RECEIVED	23	2
android.net.wifi.WIFI_STATE_CHANGED	22	2
android.intent.action.CHINAMOBILE_OMS_GAME	13	2
Other	-	1

3.3 Implementation

We now briefly introduce the implementation details of our tool-based approach called HookRanker. HookRanker is implemented on top of Soot[20], which is a well-known framework for analyzing and transforming Java and Android apps. HookRanker works at the Jimple level, where Jimple is an intermediate representation (IR) of Soot. The transformation from Android Dalvik bytecode to Jimple code is performed by a tool called Dexpler[21], which now has been integrated into Soot. The package dependency graph (PDGraph) we build for pinpointing $Type_1$ hooks is supported by GraphStream[22], which is a Java library for modeling and analyzing dynamic graphs. The reason why we select GraphStream to build the PDGraph is that it provides a toolkit, where a lot of common graph-based algorithms such as computing the clustering coefficients are already implemented and thus can be simply applied.

4 Evaluation

We now evaluate our approach that automates the dissection of piggybacked malware to identify rider and hook code. Our evaluation aims at answering the following research questions.

*RQ*1. How are $Type_1$ and $Type_2$ hooks distributed in piggybacked apps?

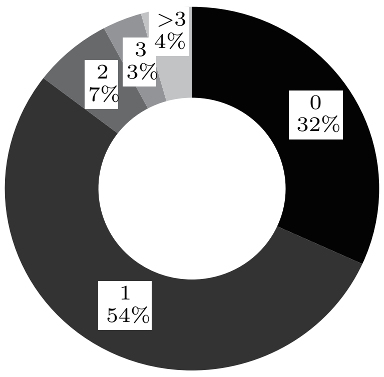
*RQ*2. Are our proposed metrics capable of locating $Type_1$ and $Type_2$ hooks in piggybacked Android apps? If so, what is the accuracy?

*RQ*3. Can we leverage rider code to hunt malware in the Android ecosystem?

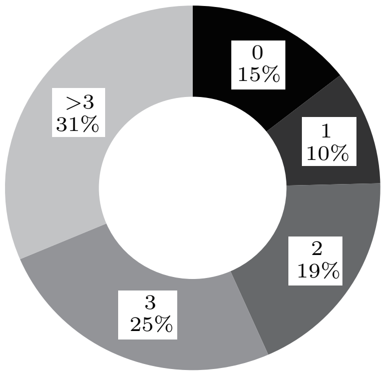
Experimental Setup. All the experiments discussed in this section are performed on a Core i7 CPU and on a Java VM with a maximum 8 GB heap size. Although the piggybacking dataset contains corner cases where the difference of apps in piggybacking pairs still does not clearly provide the ground truth of $Type_1$ and $Type_2$ hooks, we are able to consider from the dataset 500 pairs from which we could build a benchmark for our evaluation.

4.1 RQ1 — Distribution of Hook Types

Fig.7 illustrates the distribution of piggybacked apps on $Type_1$ and $Type_2$ hooks. As shown in Fig.7(a), 159 piggybacked apps (32%) do not contain any $Type_1$ hook, while the majority of piggybacked apps (54%) contain only one $Type_1$ hook.



(a)



(b)

Fig.7. Distribution of piggybacked apps on $Type_1$ and $Type_2$ hooks. The text in each fan (i.e., $\frac{x}{y\%}$) shows that $y\%$ of evaluated apps contain x hooks. (a) $Type_1$ hook. (b) $Type_2$ hook.

Fig.7(b) reveals that piggybackers are injecting more $Type_2$ hooks than $Type_1$ hooks. Indeed, over 80% of evaluated piggybacked apps have injected $Type_2$ hooks. The most common number of injected $Type_2$ hooks is three, which is used in 126 (25.2%) piggybacked apps. This preference was expected for two reasons. 1) On the one hand, piggybackers do not need to pay any effort for understanding the carrier code in benign apps since no connection between carrier and rider code is needed. 2) On the other hand, the more $Type_2$ hooks injected, the more likely the injected malicious packages will be executed. In contrast, the more $Type_1$ hooks are needed, the more effort piggybackers have to put to inject them safely. Instead, we find that, in several cases, piggybackers combine $Type_1$ and $Type_2$ hooks to increase the opportunities of executing malicious packages.

Fig.8 further compares the distribution of piggybacked apps on $Type_1$ and $Type_2$ hooks. The median number for $Type_1$ hook is 1 while for $Type_2$ is around 3. Clearly, $Type_2$ hooks are much more favored by piggybackers, compared with $Type_1$ hooks. We ensure this difference of median numbers between the datasets is significant by performing a Mann-Whitney-Wilcoxon (MWW) test. The resulting p-value confirms that the difference is significant at a significance level[9] of 0.001.

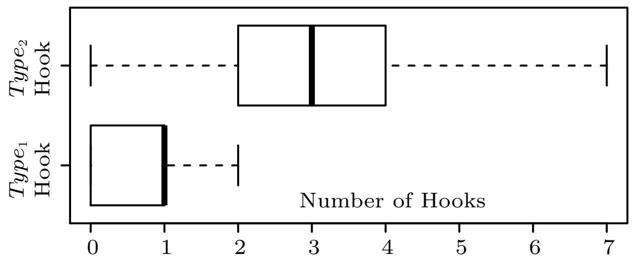


Fig.8. Distribution of piggybacked apps on $Type_1$ and $Type_2$ hooks.

*Answer to RQ*1. Both $Type_1$ and $Type_2$ hooks are commonly implemented in piggybacked apps. However, compared with $Type_1$ hooks, $Type_2$ hooks are much more favored by piggybackers.

4.2 RQ2 — Hook Identification

The output of our hook identification approach, namely HookRanker, consists of two ranked lists of potential hooks (packages[10] for $Type_1$ hooks and components for $Type_2$ hooks). Our evaluation consists in verifying the percentage of hooks in the top 5 items (i.e., accuracy@5) in the list that are correctly identified.

To support the verification, we first automatically build the baseline of comparison by computing the `diff` (difference) between each of the selected piggybacked apps and its corresponding original app. With this `diff` (difference), we can identify the rider code and the hook (including $Type_1$ and $Type_2$ hooks). Then, we apply our dissection approach by only considering the piggybacked apps[11], and compare the top ranked packages against the baseline. Table 2 enumerates the verification results reported by our approach. Our verification is performed in two cases: Match Any Hook and Match All Hooks. In the case of Match Any Hook,

[9]Given a significance level $\alpha = 0.001$, if p-value $< \alpha$, there is one chance in a thousand that the difference between the compared two datasets is due to a coincidence.

[10]Packages encompass sufficient information for analysts to quickly locate the relevant pair of packages that are interconnected in the PDGraph. For simplicity, we only consider packages in this work.

[11]We remind the readers that our goal is to identify hooks of piggybacked apps without knowing their original counterparts, i.e., the "diff" cannot be computed in practice.

where we consider an app verified as long as one of its hooks is located, HookRanker yields an accuracy@5 (we check the top 5 packages) of 89.4% for $Type_1$ hooks and an accuracy@5 (we only check the top 5 components) of 99.5% for $Type_2$ hooks if any one hook is matched. In the case of Match All Hooks, where we consider an app is verified if and only if all of its hooks are located, HookRanker yields an accuracy@5 of 83.6% for $Type_1$ hooks and an accuracy@5 of 82.2% for $Type_2$ hooks. For such apps that have more than five hooks, we consider them to be not verified.

Table 2. Hook Identification Results (Accuracy@5)

Type	Hook-Infected/Total Apps	Match Any Hook	Match All Hooks
$Type_1$	341/500	305 (89.4%)	285 (83.6%)
$Type_2$	428/500	426 (99.5%)	352 (82.2%)

Our manual analysis on the dissecting results further provides some insights into how malware writers perform piggybacking at a large scale. Table 3 and Table 4 present five samples of $Type_1$ hooks (at the package level) and $Type_2$ hooks (at the component level), respectively. Both tables show that some malicious packages are repeatedly injected into (different) Android apps. For example, *com.google.ads*, an AD-related package, has been injected into seven benign apps while package *com.fivefeiwo.coverscreen.SR* appears in 50 distinct piggybacked apps. This repeating phenomenon suggests that piggybacking could be performed in batches.

Table 3. Five Ranked $Type_1$ Hook Samples and Their Affected Number of Apps

$Type_1$ Hook	Number of Affected Apps
com.unity3d.player $\rightarrow$ *com.gamegod*	12
com.unity3d.player $\rightarrow$ *com.google.ads*	7
com.unity3d.player $\rightarrow$ *com.basyatw.bcpawsen*	5
com.ansca.corona $\rightarrow$ *com.google.ads*	3
com.g5e $\rightarrow$ *com.geseng*	2

Table 4. Five Ranked $Type_2$ Hook Samples and Their Affected Number of Apps

$Type_2$ Hook	Type	Number of Affected Apps
com.fivefeiwo.coverscreen.SR	Receiver	50
net.crazymedia.iad.AdPushReceiver	Receiver	48
com.kuguo.ad.MainReceiver	Receiver	48
com.zpvg.cvjaoyt.BawnHawn	Receiver	33
com.czvzoytyttq.fbmszy.Laoenawy	Receiver	33

Now, let us look one step deeper into the frequency of injected $Type_1$ hooks (in Table 3): piggybackers often connect their packages 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, package `com.unity3d.player` is the infection point in 65 (out of the 500) piggybacked apps. In 12 of these apps, the entry package of the rider code is `com.gamegod`.

We further summarize the type of located $Type_2$ hooks in Fig.9. Clearly, receiver is the most implemented component type of $Type_2$ hooks. This finding is actually what we expected, because receiver is much easier to trigger compared with other component types. Indeed, in addition to user-defined events, receiver can also listen to system events. Whenever a broadcast event arrives, receiver with the corresponding capability declared will be triggered and consequently propagate the execution to other code.

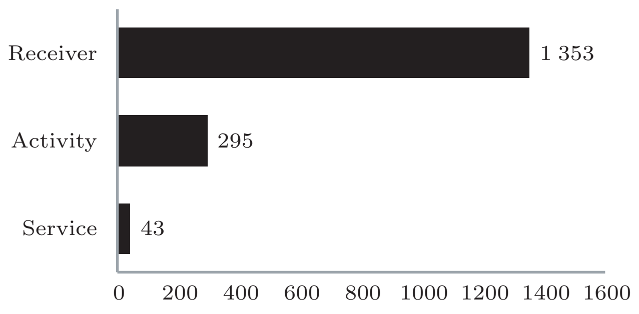


Fig.9. Distribution of the component type of $Type_2$ hooks.

As shown in Fig.9, after receiver, the second appearing component type for $Type_2$ hooks is activity. The fact that activity is more favored than service is actually surprising to us. Compared with activity, service does not need to be involved in user interfaces and thus can be executed stealthily. This feature should be more fit for the requirements of piggybackers. Therefore, we go one step deeper to investigate the reason why this happens. To this end, we summarize the capabilities leveraged by the newly injected activities. The most declared activity-based action appears to be *CREATE_SHORTCUT*, which is used for creating shortcuts for Android apps, resulting in an alternative way to launch the apps (the code executed in this way can be totally independent from the true app code). This fact suggests that piggybackers are intended to maximize the possibility of triggering the execution of their injected malicious packages.

*Answer to RQ*2. HookRanker is efficient in locating both $Type_1$ and $Type_2$ hooks. Our in-depth analysis on the located malicious packages further discloses that 1) piggybacking process is likely performed in batches; 2) Broadcast Receiver is the most adopted component type for implementing $Type_2$ hooks.

4.3 RQ3 — Rider-Based Malware Detection

After collecting snippets of malicious rider code from piggybacked apps, we now explore their potential for improving malware detection approaches.

4.3.1 Basic Malware Detection

In a first scenario, we consider the case of machine learning (ML) 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 consider features (malicious packages) that malware samples exhibit. For this experiment, we consider each malicious package as a distinct feature, e.g., package "*com.gamegod*" shown in Fig.5 is thus a feature in our feature set; if a given app has a package starting with "*com.gamegod*", we set the value of ("*com.gamegod*") feature to "TRUE" ("FALSE", otherwise). We first apply the classifier built with these new features on our ground truth. In 10-fold cross-validation experiments, using the RandomForest classification algorithm[23], we have recorded an accuracy of 91.6% in identifying Android malware. These results suggest that rider code features are effective in detecting piggybacked malware.

It is worth mentioning that our objective in this work is not to propose an ML-based malware detection approach that outperforms the state of the art. Instead, we simply show that collected malicious packages are recurrent and promising ingredients for discriminating malware from benign apps. Therefore, it is expected that the accuracy achieved by our approach may not be so good as the state-of-the-art ones. Nevertheless, we believe well-designed fine-grained features based on such collected malicious packages would lead to better results. However, to explore this interesting direction is out of the scope of this work and therefore we take it as our future work.

We further investigate the MalGenome dataset to determine the proportion of malicious apps which share the same malicious packages 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 one package. They are thus irrelevant for our study. Among those apps with several packages, 252 (i.e., 22.2%) contain 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.

4.3.2 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 packages of all apps where each package represents a distinct feature, and apply the Expectation-Maximization (EM)[24] algorithm on the edges related to rider code in the app PDGraph to cluster them. This leads to the construction of five clusters of varying sizes. Here the number of clusters (5) is directly computed by the EM algorithm, which is able to infer a suitable number of clusters to optimize the distance among clusters. Our objective is then to investigate whether the clusters of rider code are also related to specific malware families. In this paper, we consider the labels[12] that anti-virus products from VirusTotal provide after analysing the piggybacked apps corresponding to the rider code in each cluster. Other familial classification studies such as the one presented by Fan *et al.*[25], where frequent subgraph (to represent the common behavior or malware) is leveraged, could be also leveraged to achieve the same purpose (i.e., to build Android malware families).

$$d_{\mathrm{Jaccard}}(f_a, f_b) = \frac{|f_a \cup f_b| - |f_a \cap f_b|}{|f_a \cup f_b|}, \qquad (5)$$

$$f_a = \bigcup_{i=1}^{k} L_i. \qquad (6)$$

We compute the Jaccard distance (5) between the sets of labels for the different clusters. As an example, given a cluster a, f_a denotes a union set of anti-virus labels, which can be computed through (6), where L_i stands for a set of anti-virus labels that VirusTotal gives for app_i. The results summarized in Table 5 and Table 6 reveal that the malware labels in a given cluster are

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

distant from those of any other clusters. This suggests that the dissected rider code contributes to malware of specific families.

Table 5. Number of Apps and Anti-Virus Labels of the Clusters Built Based on Rider Code

	C_1	C_2	C_3	C_4	C_5
Number of apps	10	236	27	7	37
Number of anti-virus labels	22	269	51	5	27

Table 6. Jaccard Distance (Dissimilarity) of Malware Label Sets Between Clusters Built Based on Rider Code

	C_1	C_2	C_3	C_4	C_5
C_1	0.00	0.96	0.89	0.92	0.88
C_2	0.96	0.00	0.91	0.99	0.96
C_3	0.89	0.91	0.00	0.94	0.94
C_4	0.92	0.99	0.94	0.00	0.95
C_5	0.88	0.96	0.94	0.95	0.00

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 five clusters of apps previously constructed based on rider code. Table 7 and Table 8 summarize the results which reveal 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 is not significantly high for clusters C_1 and C_4, two cases where the contained app sets are small. Nevertheless, these experimental results overall illustrate that the malicious packages ungrafted from piggybacked apps indeed represent a signature of a malware family.

Table 7. Number of Apps and Anti-Virus Labels of the Clusters Built Based on Anti-Virus Labels

	MC_1	MC_2	MC_3	MC_4	MC_5	MC_6
Number of apps	90	44	69	47	14	53
Number of anti-virus labels	170	82	35	57	13	67

Table 8. Jaccard Distance of Malware Label Sets Between Clusters of Apps

	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

Note: MC_i is a cluster built based on anti-virus labels, while C_i is a cluster built based on rider code features.

*Answer to RQ*3. The located rider code is helpful for supporting malware detection approaches. Our empirical experiment further illustrates that malicious packages incorporate good features for grouping malicious apps into families.

5 Discussion

Our approach and the experiments presented in this work introduce a few threats to validity.

• First, our dissection is at the granularity level of packages. Additional efforts may apply if one wants to infer low-level artifacts (e.g., class or method). However, based on our experiments, it is relatively easy to infer the low-level method/class calls involved in a hook code (because of weak connection) once the corresponding packages have been identified.

• Second, in order to present a fast solution, we construct the PDGraph in a context-insensitive manner and we take no account of inter-component communication (ICC), reflective calls (including dynamically loaded code) and native code. This trade-off may result in false positives. In our future work, we plan to integrate IccTA[7] and DroidRA[26] into our approach for taking care of ICC and reflective calls.

• Third, HookRanker is implemented based on the assumption that malicious code and the original benign code are loosely coupled. However, this assumption may not always be true and hence may lead to a ranked list that is irrelevant to the desired malicious packages, unfortunately resulting in more efforts for security analysts to identify the real malicious code. However, as demonstrated by Li *et al.*[9], attackers (or piggybackers) usually want to make the least effort to inject their malicious payloads, generally resulting in only small changes being made, and therefore leading to loosely coupled benign and malicious packages. In other words, our assumption should remain valid for most piggybacked malicious apps, making our approach relevant for many cases.

• Fourth, our approach has no specific treatment to deal with obfuscation. Theoretically, HookRanker should not be impacted by basic obfuscation techniques such as renaming but would be impacted by advanced techniques such as control flow alterations. Nonetheless, deobfuscation approaches such as DeGUARD[27] and the one presented by Wang and Rountev[28] could be applied to limit the impact of obfuscation. So far, because of Android app packers such as Bangcle, Ijiami, which aim at preventing Android apps from being reverse engineered and further repackaged, HookRanker is also not able to tackle packed piggybacked malicious apps. Fortunately, state-of-the-art approaches including DexHunter[29] and PackerGrind[30] have already demonstrated promising results for extracting DEX code from those packed apps, making it still possible for HookRanker to tackle packed apps and thereby to locate malicious packages.

• Finally, it is hard to know whether a given malware is piggybacked from other apps (because of lacking ground truth) or is built from scratch by "bad" guys. As HookRanker will attempt to yield ranked lists of packages and components in any case, applying our approach to non-piggybacked malware may result in false alarms. In other words, HookRanker should not be applied to normal malware developed from scratch, because the enumerated potential hooks might mislead the analysis of security analysts. To mitigate this, we argue that there is a need to automatically infer piggybacked apps, even when the original app is not "known" (e.g., identifying piggybacked apps through machine learning based techniques[31] or symptoms based approaches[32]). Nonetheless, it is out of the scope of this paper to automatically identify piggybacked apps. We take it as our future work.

As for future work, we also plan to directly perform our graph analysis in the class or method level, where the fine-grained results could be more accurate for analysts to identify malicious behaviours and for rider-based malware classifiers. Furthermore, we would like to investigate other means (e.g., community detection on the built graph) to improve the accuracy of our approach. Finally, we plan to conduct a user study and consequently to understand to what extend HookRanker can help analysts dissect piggybacked apps, where their original counterparts are unknown.

Last but not the least, the findings of our approach, namely the hook and the rider code, could be used to boost many more implications. In addition to the one we have demonstrated in RQ3, where we have shown how our results can be leveraged for malware detection, another potential implication is to exploit the hook/rider code to develop an automatic app repair/blocking approach which disconnects the rider code or disassembles the hook so that the malicious payload would not be triggered. As demonstrated by Li *et al.*[9], third-party libraries are frequently compromised to include malicious payloads. Therefore, based on a whitelist of known libraries[15,33-35], it is possible to supplement this work with a comparison between released and in-app library code. If the in-app library code is substantially different from the known publicly released version, it hints on probable attack on the library code (e.g., hook introduction to trigger malicious code). Moreover, since our approach provides quantitative outputs (i.e., the rank), it could be utilized to rank Android apps based on the extent of the suspiciousness on their malicious status. This ranking can benefit app vetting processes for both end users and security analysts.

6 Related Work

In a recent study with anti-virus products, researchers have shown that malware is still widespread within Android markets[36]. This finding is in line 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[9]. 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[4,37-39].

Cesare and Xiang[40] proposed to use similarity on control flow graphs to detect variants of known malware. Chen *et al.*[10] presented an approach named MassVet that compares a submitted app with all those existing ones in a market, vetting Android apps based on the commonality in UI structures and differences in components. Eventually, their approach suspects a given app of being malicious based on unusual components. Our work however focuses on analyzing each app to highlight potential components which contribute to the malicious payloads. Hu *et al.*[41] 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[42] 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[43].

PiggyApp[12] is the work that focuses on piggybacked app detection. The authors improved over their previous work, namely DroidMoss[44], 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, PiggyApp builds its program dependency graph, and assigns weights to the edges in accordance to the degree of relationship between the packages. Then, it uses an agglomerative algorithm to cluster the packages and thereby to select the primary module of the app, which is further leveraged to highlight piggybacked apps by comparing with other selected primary modules. To escape the scalability problem with pair-wise comparisons, the authors of [12] relied on the vantage point tree data structure to partition the metric space and eventually to detect piggybacked apps. The identified piggybacked apps can be taken as the input of our approach.

Researchers use a set of diverse 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.*[45] In 2013, Amos *et al.*[46] 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. Zhang *et al.*[47] used weighted contextual API dependency graphs as program semantics to classify Android malware, where they leveraged graph similarity metrics to disclose homogeneous app behavior.

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 reason among the candidate ones 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. Furthermore, because most malware is actually piggybacked from benign apps, the ML-based features are probably similar to those extracted from benign apps, making them indistinguishable for ML-based malware detection (e.g., due to the multi-generation repackaging problem[48]). Indeed, as pointed out by Meng *et al.*[49], the current feature-based malware detection approaches are not enough because they cannot provide detailed information beyond their mere detection. They thus proposed an alternative approach that leverages semantic features (based on deterministic symbolic automaton (DSA)) to comprehensive Android malware and thereby to detect and classify them. As another example, Tian *et al.*[50] proposed an approach that leverages code heterogeneity analysis to detect repackaged Android malware. Given an Android app, they strategically partitioned its code structure into multiple dependence-based regions, where each region is a basic unit that will be independently classified on its behavioural features.

Our work differs from them in a way that we actually focus on extracting the malicious packages (rider code) from piggybacked apps. Based on the located rider code, our approach can be used to compliment those approaches by allowing them for better representation of features and 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 different structures and behaviours of malware).

7 Conclusions

We proposed in this paper an approach for dissecting piggybacked apps to locate and collect malicious samples. Through extensive evaluations, we demonstrated the performance of our approach, i.e., the precision of locating hook/rider code. We also experimentally showed that the collected malicious packages (i.e., rider code) can be leveraged to detect new malicious apps. Further investigations revealed that rider code clusters strongly correlated with the clusters characterized via malware signatures given by anti-virus products.

References

[1] Demme J, Maycock M, Schmitz J, Tang A, Waksman A, Sethumadhavan S, Stolfo S. On the feasibility of online malware detection with performance counters. In *Proc. the 40th Annual Int. Symp. Computer Architecture*, June 2013, pp.559-570.

[2] Yerima S Y, Sezer S, McWilliams G, Muttik I. A new Android malware detection approach using Bayesian classification. In *Proc. the 27th IEEE Int. Conf. Advanced Information Networking and Applications* (*AINA*), March 2013, pp.121-128.

[3] Canfora G, Mercaldo F, Visaggio C A. A classifier of malicious Android applications. In *Proc. the 8th Int. Conf. Availability, Reliability and Security* (*ARES*), September 2013, pp.607-614.

[4] Sahs J, Khan L. A machine learning approach to Android malware detection. In *Proc. European Intelligence and Security Informatics Conf* (*EISIC*), August 2012, pp.141-147.

[5] Symantec. 2015 Internet Security Threat Report: Attackers are bigger, bolder, and faster. https://www.symantec.com/connect/blogs/2015-internet-security-threat-report-attackers-are-bigger-bolder-and-faster, Oct. 2017.

[6] Zhou Y J, Jiang X X. Dissecting Android malware: Characterization and evolution. In *Proc. IEEE Symp. Security and Privacy* (*SP*), May 2012, pp.95-109.

[7] Li L, Bartel A, Bissyandé T F, Klein J, Le Traon Y, Arzt S, Rasthofer S, Bodden E, Octeau D, Mcdaniel P. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proc. the 37th Int. Conf. Software Engineering*, May 2015, pp.280-291

[8] Arp D, Spreitzenbarth M, Hübner M, Gascon H, Rieck K. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proc. Network and Distributed System Security Symp.* (*NDSS*), February 2014.

[9] Li L, Li D Y, Bissyande T F, Klein J, Le Traon Y, Lo D, Cavallaro L. Understanding Android app piggybacking: A systematic study of malicious code grafting. *IEEE Trans. Information Forensics and Security*, 2017, 12(6): 1269-1284.

[10] Chen K, Wang P, Lee Y, Wang X F, Zhang N, Huang H Q, Zou W, Liu P. Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-play scale. In *Proc. the 24th USENIX Conf. Security Symp.*, August 2015, pp.659-674.

[11] Li L, Li D Y, Bissyandé T F, Klein J, Cai H P, Lo D, Le Traon Y. Automatically locating malicious packages in piggybacked Android apps. In *Proc. the 4th IEEE/ACM Int. Conf. Mobile Software Engineering and Systems* (*MOBILESoft*), May 2017, pp.170-174.

[12] Zhou W, Zhou Y J, Grace M, Jiang X X, Zou S H. Fast, scalable detection of "piggybacked" mobile applications. In *Proc. the 3rd ACM Conf. Data and Application Security and Privacy*, February 2013, pp.185-196.

[13] Li L, Li D Y, Bissyandé T F D A, Lo D, Klein J, Le Traon Y. Ungrafting malicious code from piggybacked Android apps. Technical Report, University of Luxembourg, 2016.

[14] Li L, Gao J, Hurier M, Kong P F, Bissyandé T F, Bartel A, Klein J, Le Traon Y. AndroZoo++: Collecting millions of Android apps and their metadata for the research community. arXiv: 1709.05281, 2017. https://arxiv.org/abs/1709.05281, October 2017.

[15] Li L, Bissyandé T F, Klein J, Le Traon Y. An investigation into the use of common libraries in Android apps. In *Proc. the 23rd IEEE Int. Conf. Software Analysis, Evolution, and Reengineering* (*SANER*), March 2016, pp.403-414.

[16] Li L, Martinez J, Ziadi T, Bissyandé T F, Klein J, Le Traon Y. Mining families of Android applications for extractive SPL adoption. In *Proc. the 20th Int. Systems and Software Product Line Conf.*, September 2016, pp.271-275.

[17] Allix K, Bissyandé T F, Jérome Q, Klein J, State R, Le Traon Y. Empirical assessment of machine learning-based malware detectors for Android. *Empirical Software Engineering*, 2016, 21(1): 183-211.

[18] Li L, Bissyandé T F, Klein J. SimiDroid: Identifying and explaining similarities in Android apps. In *Proc. IEEE Trustcom/BigDataSE/ICESS*, August 2017, pp.136-143.

[19] Watts D J, Strogatz S H. Collective dynamics of 'small-world' networks. *Nature*, 1998, 393(6684): 440-442.

[20] Lam P, Bodden E, Lhotak O, Hendren L. The soot framework for Java program analysis: A retrospective. In *Proc. Cetus Users and Compiler Infastructure Workshop* (*CETUS2011*), October 2011.

[21] Bartel A, Klein J, Le Traon Y, Monperrus M. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *Proc. ACM SIGPLAN Int. Workshop on State of the Art in Java Program Analysis*, June 2012, pp.27-38.

[22] Dutot A, Guinand F, Olivier D, Pigné Y. GraphStream: A tool for bridging the gap between complex systems and dynamic graphs. In *Proc. ECCS*, October 2007.

[23] Breiman L. Random forests. *Machine Learning*, 2001, 45(1): 5-32.

[24] Dempster A P, Laird N M, Rubin D B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 1977, 39(1): 1-38.

[25] Fan M, Liu J, Luo X P, Chen K, Chen T Y, Tian Z Z, Zhang X D, Zheng Q H, Liu T. Frequent subgraph based familial classification of Android malware. In *Proc. the 27th IEEE Int. Symp. Software Reliability Engineering* (*ISSRE*), October 2016, pp.24-35.

[26] Li L, Bissyandé T F, Octeau D, Klein J. DroidRA: Taming reflection to support whole-program analysis of Android apps. In *Proc. the 25th Int. Symp. Software Testing and Analysis*, July 2016, pp.318-329.

[27] Bichsel B, Raychev V, Tsankov P, Vechev M. Statistical deobfuscation of Android applications. In *Proc. ACM SIGSAC Conf. Computer and Communications Security*, October 2016, pp.343-355.

[28] Wang Y, Rountev A. Who changed you? Obfuscator identification for Android. In *Proc. the 4th IEEE/ACM Int. Conf. Mobile Software Engineering and Systems*, May 2017, pp.154-164.

[29] Zhang Y Q, Luo X P, Yin H Y. DexHunter: Toward extracting hidden code from packed Android applications. In *Proc. the 20th European Symp. Research in Computer Security*, September 2015, pp.293-311.

[30] Xue L, Luo X P, Yu L, Wang S, Wu D H. Adaptive unpacking of Android apps. In *Proc. the 39th Int. Conf. Software Engineering*, May 2017, pp.358-369.

[31] Shao Y R, Luo X P, Qian C X, Zhu P F, Zhang L. Towards a scalable resource-driven approach for detecting repackaged Android applications. In *Proc. the 30th Annual Computer Security Applications Conf.*, December 2014, pp.56-65.

[32] Gonzalez H, Kadir A A, Stakhanova N, Alzahrani A J, Ghorbani A A. Exploring reverse engineering symptoms in Android apps. In *Proc. the 8th European Workshop on System Security*, April 2015, Article No. 7.

[33] Li M H, Wang W, Wang P, Wang S, Wu D H, Liu J, Xue R, Huo W. LibD: Scalable and precise third-party library detection in Android markets. In *Proc. the 39th Int. Conf. Software Engineering*, May 2017, pp.335-346.

[34] Ma Z A, Wang H Y, Guo Y, Chen X Q. LibRadar: Fast and accurate detection of third-party libraries in Android apps. In *Proc. the 38th Int. Conf. Software Engineering Companion*, May 2016 pp.653-656.

[35] Wang H Y, Guo Y. Understanding third-party libraries in mobile app analysis. In *Proc. the 39th IEEE/ACM Int. Conf. Software Engineering Companion*, May 2017, pp.515-516.

[36] Nagappan M Shihab E. Future trends in software engineering research for mobile apps. In *Proc. the 23rd IEEE Int. Conf. Software Analysis, Evolution, and Reengineering* (*SANER*), March 2016, pp.21-32.

[37] Kolter J Z Maloof M A. Learning to detect and classify malicious executables in the wild. *The Journal of Machine Learning Research*, 2006, 7: 2721-2744.

[38] Zhang B Y, Yin J P, Hao J B, Zhang D X, Wang S L. Malicious codes detection based on ensemble learning. In *Proc. the 4th Int. Conf. Autonomic and Trusted Computing*, July 2007, pp.468-477.

[39] Perdisci R, Lanzi A, Lee W. McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Proc. Annual Computer Security Applications Conf.*, December 2008, pp.301-310.

[40] Cesare S, Xiang Y. Classification of malware using structured control flow. In *Proc. the 8th Australasian Symp. Parallel and Distributed Computing*, January 2010, pp.61-70.

[41] Hu X, Chiueh T C, Shin K G. Large-scale malware indexing using function-call graphs. In *Proc. the 16th ACM Conf. Computer and Communications Security*, November 2009 pp.611-620.

[42] Jang J, Brumley D, Venkataraman S. BitShred: Feature hashing malware for scalable triage and semantic analysis. In *Proc. the 18th ACM Conf. Computer and Communications Security*, October 2011 pp.309-320.

[43] Linares-Vásquez M, Holtzhauer A, Poshyvanyk D. On automatically detecting similar Android apps. In *Proc. the 24th IEEE Int. Conf. Program Comprehension* (*ICPC*), May 2016.

[44] Zhou W, Zhou Y J, Jiang X X, Ning P. Detecting repackaged smart phone applications in third-party Android marketplaces. In *Proc. the 2nd ACM Conf. Data and Application Security and Privacy*, February 2012, pp.317-326.

[45] Wu D J, Mao CH, Wei TE, Lee HM, Wu KP. DroidMat: Android malware detection through manifest and API calls tracing. In *Proc. the 7th Asia Joint Conf. Information Security* (*AsiaJCIS*), August 2012, pp.62-69.

[46] Amos B, Turner H, White J. Applying machine learning classifiers to dynamic Android malware detection at scale. In *Proc. the 9th Int. Wireless Communications and Mobile Computing Conf.* (*IWCMC*), July 2013, pp.1666-1671.

[47] Zhang M, Duan Y, Yin H, Zhao Z R. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In *Proc. ACM SIGSAC Conf. Computer and Communications Security*, November 2014, pp.1105-1116.

[48] Li L, Bissyandé T F, Bartel A, Klein J, Le Traon Y. The multigeneration repackaging hypothesis. In *Proc. the 39th IEEE/ACM Int. Conf. Software Engineering*, May 2017 pp.344-346.

[49] Meng G Z, Xue Y X, Xu Z Z, Liu Y, Zhang J, Narayanan A. Semantic modelling of Android malware for effective malware comprehension, detection, and classification. In *Proc. the 25th Int. Symp. Software Testing and Analysis*, July 2016, pp.306-317.

[50] Tian K, Yao D F, Ryder B G, Tan G. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *Proc. IEEE Security and Privacy Workshops* (*SPW*), May 2016 pp.262-271.

Li Li is a research associate at Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg, and a honorary research associate at the CREST group, University College London, London. He received his Ph.D. degree in computer science from the University of Luxembourg, Luxembourg. His research interests are in the fields of Android security, static code analysis, and machine learning. Dr. Li received a Best Paper Award at the ERA Track of IEEE SANER 2016.

Daoyuan Li is currently working towards his Ph.D. degree at University of Luxembourg, Luxembourg. His research is mainly focused on machine learning, especially time series classification and its applications in smart buildings, IoT and FinTech. He loves (open source) software and writes code to get his thoughts straight or to prove a point. Previously he worked as technical director in a startup company in China and before that was a research scientist at Ericsson Research NomadicLab, Finland.

Tegawendé F. Bissyandé is a research scientist with Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg. He received his Ph.D. degree in computer science from the University of Bordeaux, Bordeaux, in 2013. His work is mainly related to software engineering, specifically empirical software engineering, reliability and debugging as well as mobile app analysis. His studies were presented in major conferences such as ICSE, ISSTA and ASE, and published in top journals such as Empirical Software Engineering and IEEE TIFS. He has received a Best Paper Award at ASE 2012, and has served in several program committees including ASE-Demo, ACM SAC, ICPC.

Jacques Klein is senior research scientist at the University of Luxembourg, Luxembourg, and at the Interdisciplinary Centre for Security, Reliability and Trust (SnT). He received his Ph.D. degree in computer science from the University of Rennes, Rennes, in 2006. His main areas of expertise are threefold: 1) mobile security (malware detection, prevention and dissection, static analysis for security, vulnerability detection, etc.); 2) software reliability (software testing, semi-automated and fully-automated program repair, etc.); 3) data analytics (multi-objective reasoning and optimization, model-driven data analytics, time series pattern recognition, etc.). In addition to academic achievements, Dr. Klein has also standing experience and expertise on successfully running industrial projects with several industrial partners in various domains by applying data analytics, software engineering, information retrieval, etc., to their research problems.

Haipeng Cai received his Ph.D. degree in computer science and engineering from the University of Notre Dame, Notre Dame, in 2015. He worked on computer graphics and visualizations during his previous graduate studies and was a software developer in Internet search services and embedded systems. He is currently an assistant professor in the School of Electrical Engineering and Computer Science at Washington State University, Pullman. His research interests are in software engineering and software systems in general with emphasis on program analysis and its applications for the quality, security and reliability of evolving software. He is a member of ACM and IEEE.

David Lo is an associate professor in School of Information Systems, Singapore Management University, Singapore. He is working in the intersection of software engineering and data mining research. He is an active researcher in the emerging field of software analytics that focuses on the design and development of specialized data analysis techniques to solve software engineering problems. He has received a number of international awards including two ACM Distinguished Paper Awards. He has contributed as an organizing committee member of many conferences; the upcoming ones include serving as a program co-chair of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME'18) and workshop co-chair of the 41st ACM/IEEE International Conference on Software Engineering (ICSE'19). He serves/served in the steering committee of the IEEE International Conference on Software ANalysis, Evolution and Reengineering (SANER), IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), and IEEE/ACM International Conference on Automated Software Engineering (ASE). He is an editorial board member of Empirical Software Engineering, Journal of Software: Evolution and Process, Information and Software Technology, Information Systems, and Neurocomputing.

Yves Le Traon is professor at University of Luxembourg, Luxembourg, where he leads the SERVAL (SEcurity, Reasoning and VALidation) research team. His research interests within the group include 1) innovative testing and debugging techniques, 2) Android apps security and reliability using static code analysis, and machine learning techniques, and 3) model-driven engineering with a focus on IoT and CPS. He has been General Chair of major conferences in the domain, such as the 2013 IEEE International Conference on Software Testing, Verification and Validation (ICST), and Program Chair of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS). He serves at the editorial boards of several internationally-known journals (STVR, SoSym, IEEE Transactions on Reliability) and is author of more than 160 publications in international peer-reviewed conferences and journals.