Automatically Locating Malicious Packages in Piggybacked Android Apps

Li Li*, Daoyuan Li*, Tegawendé F. Bissyandé*, Jacques Klein*, Haipeng Cai†, David Lo‡, and Yves Le Traon*
* Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
† Washington State University, Washington, USA
‡ School of Information Systems, Singapore Management University, Singapore
{li.li, daoyuan.li, tegawende.bissyande, jacques.klein, yves.letraon}@uni.lu hcai@eecs.wsu.edu davidlo@smu.edu.sg

***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 set of piggybacked apps, we are able to automatically locate the malicious packages from piggybacked Android apps with an accuracy of 83.6% in verifying the top five reported items.**

I. 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 smart home and office appliances. As we now heavily depend on mobile apps in various activities that pervade our modern lives, 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], [2], [3], [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 [6]. 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 the lack of accurate understanding of what constitutes a malicious code. In 2012, Zhou and Jiang [7] have manually investigated 1260 malware samples to characterize 1) their installation process, i.e., which social engineering-based techniques (e.g., repackaging [8], [9], [10]) 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); and 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 have either focused on detecting specific malware types (e.g., malware leaking private data [11], [12], [13]), or are exploiting features such as permissions in ML classification [14]. 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 for 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 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 [15], [5]. 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 [16]. 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. For simplicity, in this entire paper we refer to any code package injected via piggybacking as a "malicious" package[1].

[1]This package may directly contribute to implementing the malicious behaviour, or further to hiding malicious actions to static analyzers, or may simply include library code leveraged by piggybackers.

Accurately identifying and extracting 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 phases in several packages. To the best of our knowledge, the literature does not include any approach for systematically identifying packages which are responsible for the malicious behaviour of a malware. We propose in this work a step towards **helping analysts** readily identify malicious packages in Android apps. To that end, we build HookRanker, a ranking approach which orders packages with regards 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 switches the execution context from benign to malicious code) within piggybacked apps. Our approach eventually yields a ranked list 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 access to the original benign version of the piggybacked app, which is usually hard to harvest, in order to perform some form of *diff* analysis.
- We present a tool called HookRanker to automatically recommend potential malicious packages. Evaluation on a set of benchmark apps has demonstrated that HookRanker is efficient to locate malicious packages of piggybacked apps.

II. Hook Taxonomy

We now introduce the necessary terminology to which we will refer in the remainder of this paper. Figure 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** [17], and grafting to it malicious packages (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.

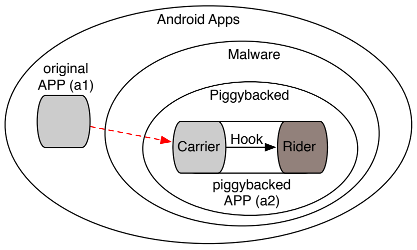

Fig. 1. Piggybacking Terminology [18].

[2]In this work, we focus on piggybacked malicious apps, where the status of each app has been confirmed 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.

By investigating into the Android app launch model, we observe 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 $type_1$ and $type_2$ hooks:

```
1 //Type 1 hook, through method invocation
2 public class UnityPlayerProxyActivity extends
      android.app.Activity {
3  protected void onCreate(android.os.Bundle) {
4   specialinvoke $r0.onCreate($r1);
5 + staticinvoke Touydig.init($r0);
6   $r2 = newarray (java.lang.String)[2];
7 }}
8 //Type 2 hook, through Broadcast Receiver
9 + public class UR extends AdPushReceiver {...}
```

Listing 1. An Example of Type$_1$ and $Type_2$ Hook. 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$_1$ hook involves method calls that explicitly connect carrier code to rider code. In this case, we identify the hook via the point where 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 5), which is inserted immediately at the beginning of the *onCreate()* method (line 4) of component *UnityPlayerProxyAct*. By calling the hook method (i.e., *init()*), the malicious packages (starting from class *com.gamegod.Touydig*) will immediately be triggered and thereby 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 not explicitly connected to any code of the original app. On the contrary, the (malicious) rider code will be triggered directly by system or user-defined events.

Based on our observation, piggybacked apps often feature both type$_1$ and type$_2$ hooks to ensure the execution of their malicious payloads. In this work, we tame type$_1$ hook only and keep type$_2$ hook as our future work. Our objective in this work is thus to automatically locate suspicious method calls for a given piggybacked malicious app and thereby to systematically extract malicious payloads that are injected into the piggybacked app.

III. Hook Identification

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 fulfil this objective we require a set of reliable metrics to automatically identify malicious packages within a detected piggybacked app. To the best of our knowledge, in the literature, there are no such works that have addressed this before.

To automate the identification approach, we consider the identification of type$_1$ hook as a graph analysis problem. Figure 2 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 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.

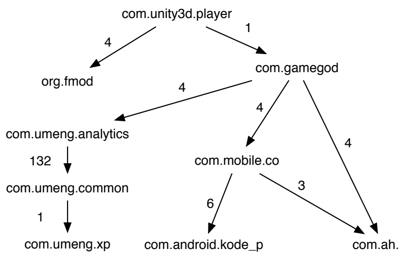

Fig. 2. Package Dependency Graph of a Piggybacked app.

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 edges pointing 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 Figure 2 is actually the entry point of the rider code, which has the 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 longest 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 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 sub-graph rooted at 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 PDGraph graph with n package nodes, the score associated to a package p, with our proposed metrics, is provided by formula (1).

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

In our experiments, we weight all metrics equally (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.

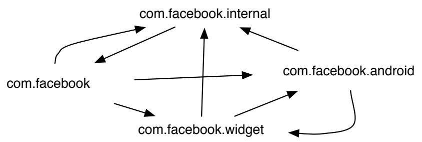

Fig. 3. A Partial PDGraph Showing a Set of Related Packages in the Carrier Code of *com.gilpstudio.miniinimo*.

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 Figure 3.
- **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 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 (2).

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

IV. 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$_1$ hooks distributed in piggybacked apps?

RQ 2: Is our proposed metrics capable of locating type$_1$ hooks in piggybacked Android apps? If so, what is the accuracy?

Experimental Setup. The experiments of this work are conducted on a benchmark of piggybacking pairs provided by Li et al. [15]. Because of some corner cases, where our tool fails to rank the potential hooks, we eventually consider a set of 500 pairs from which we could build a benchmark for our evaluation.

A. RQ1 - Distribution

Fig. 4 illustrates the distribution of piggybacked apps on type$_1$ hooks, where the median number of type$_1$ hook for the investigated apps is one. Among the 500 investigated piggybacked apps, 159 (32%) of them do not contain any type$_1$ hook, while the majority of piggybacked apps (54%) contains only one type$_1$ hook. This distribution demonstrates that piggybackers attempts to change as less as possible (one method call for over half of the investigated cases) for the original app in order to trigger the execution of their injected payloads.

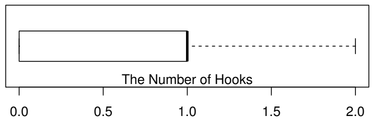

Fig. 4. Distribution of Type$_1$ Hooks.

B. RQ2 - Hook Identification

The output of our hook identification approach, namely HookRanker, is a ranked list of hooks (packages which 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.). 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` between each of the selected piggybacked apps and its corresponding original app. With this `diff`, we can identify the rider code and the hook. Then, we apply our dissection approach by only considering the piggybacked apps[4], and compare the top ranked packages against the baseline. Our verification is performed in two cases: *Match Any Hook* and *Match All Hooks*. In the case of *Match Any Hook*, 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$_1$ hook. 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$_1$ hook. For such apps that have more than five hooks, we consider them to be not verified.

Our manual analysis on the dissecting results further provided some insights into how malware writers perform piggybacking at a large scale. Table I presents five samples of type$_1$ hook at the package level. It shows 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**.

Now, let us look one step deeper into the frequency of injected type$_1$ hooks (in Table I): **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 those apps, the entry package of the rider code is `com.gamegod`.

TABLE I
TYPE$_1$ HOOK SAMPLES AND THEIR AFFECTED NUMBER OF APPS.

Type$_1$ Hook	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

V. Conclusion

We have proposed in this paper an approach for dissecting piggybacked apps to locate and collect malicious samples. Through extensive evaluations, we have demonstrated the performance of our approach, i.e., the precision of locating hook/rider code. We have also experimentally shown that piggybacking could be conducted in batches and piggybackers often connect their malicious payloads to the carrier via one of its included libraries.

VI. Acknowledgments

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects AndroMap C13/IS/5921289 and Recommend C15/IS/10449467.

[4]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.

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] 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.

[7] 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.

[8] 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.

[9] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: a scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 71–82. ACM, 2015.

[10] 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*, pages 56–65. ACM, 2014.

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014)*, 2014.

[12] 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.

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

[14] 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.

[15] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security (TIFS)*, to appear.

[16] Li Li, Tegawendé François D Assise Bissyande, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. Technical report, SnT, 2016.

[17] 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.

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

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