Understanding Android App Piggybacking

Li Li*, Daoyuan Li†, Tegawendé F. Bissyandé*, Jacques Klein*, Yves Le Traon*, David Lo†, Lorenzo Cavallaro‡
* Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg
† School of Information Systems, Singapore Management University
‡ Information Security Group, Royal Holloway, University of London

firstname.lastName@uni.lu; davidlo@smu.edu.sg; Lorenzo.Cavallaro@rhul.ac.uk

Abstract—The Android packaging model offers adequate opportunities for attackers to inject malicious code into popular benign apps, attempting to develop new malicious apps that can then be easily spread to a large user base. Despite the fact that the literature has already presented a number of tools to detect piggybacked apps, there is still lacking a comprehensive investigation on the piggybacking processes. To fill this gap, in this work, we collect a large set of benign/piggybacked app pairs that can be taken as benchmark apps for further investigation. We manually look into these benchmark pairs for understanding the characteristics of piggybacking apps and eventually we report 20 interesting findings. We expect these findings to initiate new research directions such as practical and scalable piggybacked app detection, explainable malware detection, and malicious code location.

I. INTRODUCTION

Thanks to a set of existing tools, Android apps can easily be modified by third parties [1], [2]. Malware writers can thus build on top of popular benign apps to rapidly spread new malware. Indeed, it would be more effective to simply mutate a popular benign app (e.g., by injecting some malicious code) for distributing malicious functionalities. The resulting mutant, which thus piggybacks a malicious payload, is referred to as a piggybacked app.

Fig. 1 illustrates constituting parts of a piggybacked app. The piggybacking process involves in selecting a given original app, referred to in the literature [3] as the carrier, and grafting to it a malicious code, known as the rider. The connection between carrier to rider is known as hook, which defines the point where the execution of malicious code can be triggered.

State-of-the-art works have mainly focused on detecting piggybacked apps (or cloned apps in general) through similarity comparison [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31]. However, pairwise comparison based approaches are not scalable for analyzing millions of Android apps that are now available in various markets. Besides, the comparison-based approaches also require that both benign and piggybacked apps are available in the evaluated app set. Instead of a brute-force comparison, a more practical solution is to leverage semantic features collected through a thorough understanding on piggybacking scenarios to tame the problem of piggybacked apps. Indeed, understanding Android app piggybacking could help in pushing further a number of research directions: 1) Practical detection of piggybacked apps (e.g., through machine learning based predictors); 2) Explainable detection of piggybacked apps (e.g., through fine-grained semantic features); and 3) Automatic localization of piggybacked malicious payloads (e.g., through graph-based analysis [32], [33]). Interested readers are encouraged to obtain more information from the journal publication (cf. [4]) of this extended abstract.

II. APPROACH

Our objective is to conduct a thorough dissection on piggybacked Android apps and thereby to have a deep understanding on how Android apps are piggybacked. The observed knowledge can then be used to invent advanced techniques for taming the Android app piggybacking problem. Although several approaches have been proposed to tackle this problem [34], [5], their associated datasets are not always released to public [35], [6], [31], [36]. In other words, the research on piggybacked apps is challenged by the scarcity of datasets and benchmarks. To this end, in this work, we first present an automate approach to systematically collect a set of trustable benchmarks (i.e., piggybacked apps) before conducting in-depth dissection on piggybacked apps.

A. Benchmark Collection

Our collection is based on AndroZoo [37], a large repository of millions of apps crawled over several markets including the official one named Google Play. As shown in Fig. 2, the benchmark collection is mainly done in three steps. We now briefly describe them respectively.

- **VirusTotal Classification.** First, we collect the malicious status of Android apps through the associated anti-virus scanning reports of VirusTotal. Based on the identified
malicious status, we divide the set of apps into two subsets: benign set and malicious set.

- **Irrelevance Filtering.** Second, in order to only concentrate on piggybacked app pairs, we filter out irrelevant results through a set of meta-data (including the unique app package name, app certificate and the app version) extracted from Android apps. We remind the readers that this step may miss a number of piggybacking pairs, but those that we have found will unlikely be false positives.

- **Similarity Inspection.** Finally, we conduct pairwise similarity comparison on the candidate pairs (remaining in the second step) to validate the correctness of piggybacked pairs. Given a pair of candidate apps \((a_1, a_2)\), we expect that the majority code of \(a_1\) should be part of \(a_2\) while \(a_2\) should also include new code to constitute its malicious payload.

The aforementioned three steps allow for a conservative identification of piggybacking pairs. We would like to emphasize that our focus was not to precisely detect all piggybacking pairs. Instead, we aimed for collecting a sufficient number of accurate pairs in order to be able to dissect and understand piggybacking processes. Therefore, as indicated before, our approach may have missed a number of piggybacking pairs, but those that we have found are unlikely to be false positives.

### B. Piggybacking Dissection

Based on the identified piggybacking pairs, we manually look into the difference between the original and piggybacked apps with an attempt to understand how piggybacking is done. In addition to the manual investigation, we also leverage some automated tools (e.g., Soot-based static analyzer) and scripts (e.g., Shell and Python) to facilitate our analysis. As an example, our similarity analysis approach (the similarity inspection step) is implemented in Java on top of Soot, a framework for analyzing and transforming Java/Android apps [38]. The comparison is eventually conducted at Soot’s Jimple level, where Jimple is a simplified representation of Android Dalvik bytecode. The representation is conducted by Dexpler [2], which now has been integrated as a plugin into Soot.

### III. FINDINGS

Our dissection explores several aspects of Android app piggybacking in order to answer the following three research dimensions: 1) Which app elements are manipulated by piggybackers? 2) How app functionality and behavior are impacted? and 3) Where malicious code is hooked into benign apps?

With these three research dimensions in mind, our dissection has eventually identified 20 interesting findings. Because of space limitation, we only highlight take-home messages of those findings. We recommend readers to read the detailed explanation of those findings in our journal publication [4]. The abstracted findings are as follows:

1. The realization of malicious behavior is often accompanied by a manipulation (i.e., adding/removing/replacing) of app resource files.
2. Piggybacking modifies app behavior mostly by tampering with existing original app code.
3. Piggybacked apps are potentially built in batches.
4. Piggybacking often requires new permissions to allow the realization of malicious behavior.
5. Some permissions appear to be more requested by piggybacked apps than non-piggybacked apps.
6. Piggybacking is probably largely automated.
7. Piggybacked apps overly request permissions, while leveraging permissions requested by their original apps.
8. Most piggybacked apps now include new user interfaces, implement new receivers and services, but do not add new database structures.
9. Piggybacking often consists in inserting a component that offers the same capabilities as an existing component in the original app.
10. Piggybacked apps can simply trick users by changing the launcher component in the app, in order to trigger the execution of rider code.
11. Piggybacking is often characterized by a naming mismatch between existing and inserted components.
12. Malicious piggybacked payload is generally connected to the benign carrier code via a single method call statement, making it possible to automatically locate grafted malicious payloads from piggybacked malicious apps [32], [33].
13. Piggybacking hooks are generally placed within library code rather than in core app code.
14. Injected payload is often reused across several piggybacked apps.
15. Piggybacking adds code which performs sensitive actions, often without referring to device users.
16. Piggybacking operations spread well-known malicious behavior types.
17. Piggybacked apps increasingly hide malicious actions via the use of reflection and dynamic class loading.
18. Piggybacking code densifies the overall app’s call graph, while rider code can even largely exceed in size the carrier code.
19. Piggybacked app writers are seldom authors of benign apps.
20. Piggybacking code brings more execution paths where sensitive data can be leaked.
REFERENCES


[34] Li Li, Tegawendé F Bissyandé, 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.


[38] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In CETUS, 2011.