Baseline Approach for Repackaged App Detection

A Supplement Document of the Research Paper Entitled “Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark [1]”

Li Li, Tegawendé F. Bissyandé, Jacques Klein

Our replication study conducted in the main research paper not only shows that our benchmark provides a promising means for supporting replication and comparison of state-of-the-art work, but also provides possible baselines for pairwise-based repackaged app detection approaches. Except for the replication results done by us, we also encourage the authors of other repackaging detection tools to provide their experimental results yielded based on running their tools on our benchmark dataset as baselines to facilitate comparisons of state-of-the-art work.

We remind the readers that our benchmark is built via a combination of an unsupervised learning-based approach (cf., Step (2) EM Clustering) and a pairwise similarity comparison-based approach (cf. Step (3) Candidate Pairing). Like many state-of-the-art approaches, which attempt to split the search space via a similar mechanism, we believe that this kind of mechanism does not really solve the scalability challenge for performing market-scale app repackage detection, although it might work in practice so far. Indeed, as the number of Android apps grows, even if some historical pairwise comparison results can be reused, the number of comparisons needed to be additionally conducted will increase in a non-linear way. Moreover, because the original app counterparts may not be available in the dataset, some repackaged pairs will never be revealed by this kind of approach (i.e., our benchmark construction approach actually suffers from this challenge).

In this section, to mitigate the aforementioned challenges, we propose a new repackaged app detection approach for Android to showcase the usefulness of the dataset and to encourage new research directions towards contributing to taming Android app repackaging without assuming that the original counterpart of a repackaged app is needed (i.e., this baseline approach is not for pairwise comparison-based approaches). To get intuitions on how repackaged apps are built, we consider samples from our RePack dataset and manually investigate how a repackaged app differentiates from its original counterpart. Building on the characteristics of repackaging that emerge, we build a machine learning classifier based on the following features:

Component Capability declarations. In Android, *Intents* are the primary means for exchanging information between components. These objects contain fields such as the `Component name` to optionally indicate which app component to deliver

L. Li is with the Faculty of Information Technology, Monash University, Australia.
T. Bissyandé and J. Klein are with the Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg.

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

Listing 1: Simplified view of the *manifest* of *se.illusionlabs.labyrinth.full*.

the object, some `data` (e.g., a phone number), or the `action`, which is a string that specifies the generic action to perform (such as *view* a contact, *pick* an image from the gallery, or *send* an SMS). When the Android OS resolves an intent which is not explicitly targeted to a specific component, it will look for all registered components that have *Intent filters* with actions matching the intent action. Indeed, Intent filters are used by apps to declare their capabilities (e.g., a Messaging app will indicate being able to process intents with the `ACTION_SEND` action). Our manual investigation into repackaged apps has revealed that they usually declare additional capabilities to the original apps to support the needs of the grafted code (i.e., attach to original app). Usually, such new capabilities are used for the activation of the malicious behaviour. For example, repackaged apps may add a new broadcast receiver with an intent-filter that declares its intention for listening to a specific event. Listing 1 illustrates an example of repackaged app from our RePack dataset. In this Manifest file, components *CAdR* (line 12) is inserted and accompanied with the capability declaration for handling the `PACKAGE_ADDED` system event (line 13). We have further noted that repackaging may add a component with a given capability which was already declared for another component in the carrier app. This is likely typical of repackaging since there is no need in a benign app to implement several components with the same capabilities (e.g., two PDF reader components in the same app). Thus, duplication of capability declarations (although for different components) may be indicative of repackaging.

App Permission requests. Because Android is a privilege-separated operating system, where each application runs with a distinct system identity, every app must be granted the necessary permissions for its functioning. Thus, every sensitive resource or API functionality is protected by specific permissions (e.g., the Android permission `SEND_SMS` must be granted before an app can use the relevant API method to send a SMS message). Every app can thus declare in its Manifest file which permissions it requires given its use of the Android API, via the `<uses-permission>` tag. Newly grafted code in a repackaged app often calls sensitive API methods or use sensitive resources that were not needed in the original app. For example, a game app can be repackaged to send premium rate SMS. Thus, to allow the correct functioning of their malicious apps, repackagers have to add new permission requests in the Manifest file. The app example from Listing 1 illustrates this practice. Permission `GET_TASKS` has been added (line 3) along with the grafted code. Moreover, this example further supports the finding that repackaging is done in an automated manner and with the least effort possible. Indeed, we note that a new entry requesting permission `WAKE_LOCK` (line 4) has been added in the repackaged app, ignoring the fact that this permission had already been declared (line 2) in the carrier app. Thus, we consider that duplicating permission requests can be indicative of repackaging as well.

Mismatch between Package and Launcher component names. Android apps include in their Manifest file an Application package name that uniquely identifies the app. They also list in the Manifest file all important components, such as the `LAUNCHER` component with its class name. Generally Application package name and Launcher component name are identical or related identically. However, when a malware writer is subverting app users, she/he can replace the original Launcher with a component from his grafted payload. A mismatch between package name and the launcher component name can therefore be indicative of repackaging. Again, the app example from Listing 1 illustrates such a case where the app's package (`se.illusionlabs.labyrinth.full`, line 1) and launcher (`com.loading.MainFirstActivity`, line 9) differ.

Package name diversity. Since Android apps are mostly developed in Java, different modules in the application package come in the form of Java packages. In this programming model, developer code is structured in a way that its own packages have related names with the Application package name (generally sharing the same hierarchical root, e.g., `com.example.*`). When an app is repackaged, the inserted code comes as separated modules constituting the raider code with different package names. Thus, the proportions in components that share the application package names can also be indicative of repackaging. Such diversity of package names can be seen in the example of Listing 1. The presented excerpt already contains three different packages. Since Android apps make extensive use of Google framework libraries, we systematically exclude those in our analysis to improve the chance of capturing the true diversity brought by repackaging.

Favoured sensitive APIs. Repackaged apps often graft code for implementing malicious behaviours, or triggering injected ad libraries. Such new behaviours may leverage on specific sensitive (i.e., permission-protected) APIs, which may then be symptomatic of repackaging. For example, *getActiveNetworkInfo()* API method for checking network status is largely favoured by repackaged apps in our dataset where it used by many ad libraries.

TABLE I: Feature value ranges for ML classification.

Category	Feature Type	Values
App Info	new capability	occurrence frequency
	duplicate capability	boolean
	new permission	occurrence frequency
	duplicate permission	boolean
Component Info	package diversity	int
	name mismatch	boolean
Code	sensitive API	occurrence frequency

To design our machine-learning classifier, we build, for each app, feature vectors with values inferred from the characteristics detailed above. Table I summarises the feature set. The classifier is expected to infer detection rules based on the training dataset that we provide through the collected RePack dataset. For *duplicate capability*, *duplicate permission* and *name mismatch* features, the possible values are `NO` and `YES`. For the feature *new capability*, we analyse each app and for each of its declared capabilities we indicate in its feature vector the number of times this capability type was added by piggybacking in the training data. The same procedure is applied for *new permission* and *sensitive API* features. We then apply Random Forests ensemble learning to find the correlation between feature vector values and repackaging.

A. Assessment

Sensitive APIs constitute the feature subset which will help the classifier to model common behaviour in repackaging payloads. The symptom-driven features that we have engineered contribute to increase the detection accuracy, as they hint on repackaging processes. We have computed, based on our RePack dataset, that each of such symptoms occurs in over 10% of repackaging processes. Our feature set finally consists of 521 features, including four boolean and int features presented in Table I and 107, 266, 144 features for repackaging favorite *new permissions*, *new capabilities* and *sensitive APIs*, respectively. We evaluate our classification-based approach on the constructed dataset using 10-Fold cross validation. Table II details the performance metrics. The recall is computed based on Formula 2 while the Precision and F-Measure are computed based on Formula 1 and Formula 3, respectively. Overall, we achieve 81% F-Measure for distinguishing a repackaged app from a non-repackaged one.

$$precision := \frac{TP}{TP + FP} \tag{1}$$

$$recall := \frac{TP}{TP + FN} \tag{2}$$

$$f - measure := 2 * \frac{precision * recall}{precision + recall} \tag{3}$$

TABLE II: Feature categories for ML classification.

Class	Precision	Recall	F-Measure
Non Repackaged	0.801	0.840	0.820
Repackaged	0.832	0.791	0.811

Although these performance scores are lower than many performances (up to 100%) recorded in the literature, they are obtained, to the best of our knowledge, from the first readily reproducible experiments with an approach that is not based on pairwise similarity computation.

Finally, building a classifier with our complete dataset, we apply it in the wild to the Genome dataset. We detect 64% of this dataset of malware as repackaged, about 15% less than the percentage reported by the authors. This suggests that, even with our minimal feature set we are able to detect most known repackaged apps. We expect that taking into account a variety of other repackaging symptoms can substantially improve the performance of the classifier.

REFERENCES

[1] L. Li, T. F. Bissyandé, and J. Klein, "Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark," 2019.