Scalable Android Application Repackaging Detection

Amirhossein Gharib and Ali A. Ghorbani

Faculty of Computer Science, University of new Brunswick

ABSTRACT

In recent years, as mobile smart device sales grow quickly, the development of mobile applications keeps accelerating, so does mobile app repackaging. Attackers can easily repackage an app and embed advertisements to earn money or modify a popular app by inserting malicious payloads into the original app. We propose a user interface based approach to mobile app repackaging detection. Android apps are user interaction intensive and event dominated, and the interactions between users and apps are performed through user interface, or views. In this work, we construct a graph for each app which represent possible users' navigation behavior across app views. Our system can detect repackaged apps at a large scale, both effectively and efficiently. Our experiments also show that the false positive and false negative rates are both very low.

What is repackaging?

Due to the structure of Android apps, attackers can easily decompile an app and get the source code. After modifying and adding malicious codes, they compile the app and distribute the repackaged app through different markets.

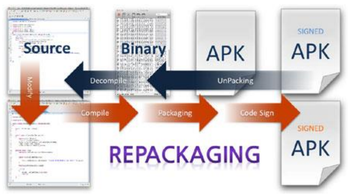

Figure 1. Repacking Cycle

System Architecture

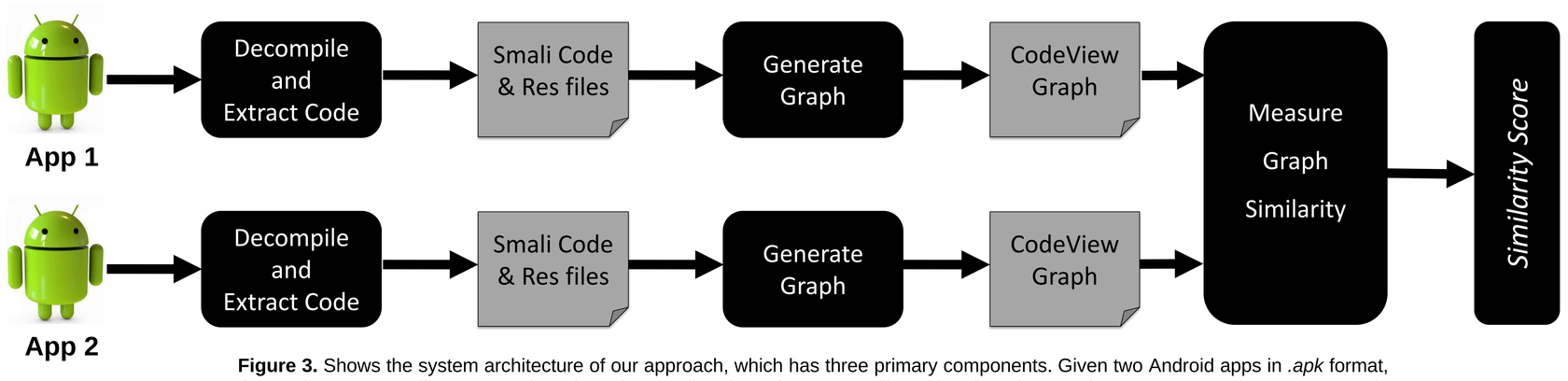

Figure 3. Shows the system architecture of our approach, which has three primary components. Given two Android apps in *.apk* format, the *Code Extractor* will extract and produce the Smali code and Resource files. After that, the *Graph Constructor* performs some static analysis to generate a graph for each app. Then, the *Graph Similarity* component compares two graphs and calculates a similarity score.

Problem Statement

The most fundamental challenge of app repackaging detection is to find features to characterize apps accurately. Plagiarists and malware writers tend to use obfuscation on the repackaged apps to ***evade detection***. First challenge is to design a detection scheme that is resilient against code ***obfuscation*** techniques. Second challenge is to build a detection tool that can perform detection in ***large scale*** scenarios.

Types of Attacks

- **Lazy attacks:** Simple changes such as different author name or different advertisement to earn credit.
- **Amateur attacks:** Adds, deletes and changes a small part of the app behaviour then applies automatic code obfuscation.
- **Malware attacks:** Adding malicious payload to the popular legitimate app to create a malicious app.

Figure 2. Repackaging Motivations and attacks

App's Graph Generation

We extract these information from Smali code and resource files:

1. **Nodes:** Collect all the activities that are associated with potential UI views. Each activity represents a Node.
2. **Node Features:** Number, types and layout of the visible components.
3. **Edges:** Represents the activity switch relationship among the set of views.

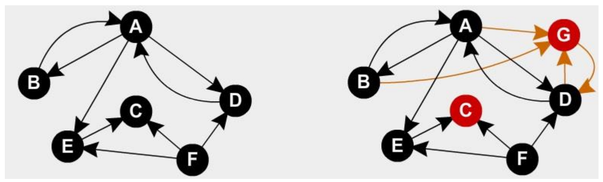

Figure 4. Left graph represents a legitimate app while the right one shows the app after an amateur attack.

System Evaluation

True Positive	1093	**Sensitivity**	96.98%
True Negative	165143	**Specificity**	99.98%
False Positive	22	**Precision**	98.02%
False Negative	34	**Accuracy**	99.96%

Table 1. True/False Matches and Accuracy using 729 labeled repackaged apps

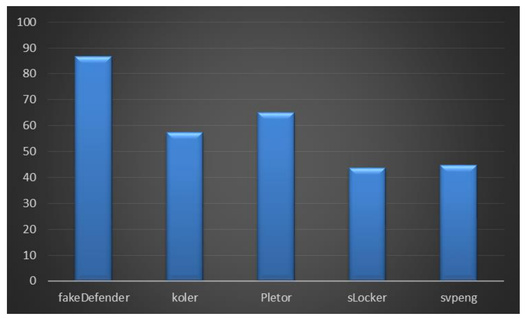

Figure 5. Ratio of repacked apps to all apps for five families in a *Ransomware* dataset captured from markets.

Conclusion

Our static user interface based detection method is:

- Resilient to various obfuscation techniques.
- Fast and Efficient for large scale experiments.
- Accurate with low false matches.

Future Work

- Add **more features** to Graph to reduce false matches.
- **Adopt *Big Data*** techniques to reduce overall execution time.
- Add a layer of **dynamic analysis** after the static one to reduce false positives and increase accuracy.