A Scalable Cloud-Based Android App Repackaging Detection Framework

Jinghua Li(✉), Xiaoyan Liu, Huixiang Zhang, and Dejun Mu

School of Automation, Northwestern Polytechnical University, Xi'an, China
jovistar@gmail.com, liuxyleo@163.com,
{zhanghuixiang,mudejun}@nwpu.edu.cn

Abstract. The problem of app repackaging has become a huge threat to the security of Android ecosystem. The massive amount of existing and developing apps makes a high demand on scalability of app repackaging detectors. In this paper, we propose a cloud-based app repackaging detection framework. It is designed to analyze and detect repacked Android apps in a large-scale way. The framework consists of three primary components: market monitor, app feature extractor and app similarity computer. Market monitor crawls all new and updated apps in specific alternative app markets periodically. Then, the multi-level features of apps are extracted by app feature extractor. App similarity computer computes the similarity score of two apps based on these features. A prototype system is implemented. The evaluation results demonstrate that the proposed cloud-based framework is highly scalable and effective for large-scale Android app repackaging detection.

Keywords: Android · App markets · Repackaging detection · Cloud computing

1 Introduction

In recent years, the industry of mobile device grows rapidly. Android has become the most popular mobile operating system in the world after years of developing. The emerging of millions of apps have changed the way people use their mobile devices.

However, it's easy to decompile and repack Android apps with existing decompiling tools. Taking advantage of this feature, hackers can easily decompile and inject malicious codes into existing popular apps. Those apps will be repacked and submitted to alternative app markets. Contrast to iOS ecosystem in which apps can be downloaded only from the official "App Store" [1], Android apps can be downloaded anywhere, including the official "Google Play" [2] and countless alternative app markets. Lots of users (especially those from China) install apps downloaded from various alternative markets on their Android devices. Thus, the repacked popular (yet malicious) apps will be downloaded and installed on users' devices. This situation results in lots of economic losses and privacy leaks. The problem of app repackaging has become a huge threat to the Android ecosystem.

A number of Android security researchers have flung themselves into the field of app repackaging detection. Some detection approaches [17, 18, 20] have been proposed

© Springer International Publishing Switzerland 2016
X. Huang et al. (Eds.): GPC 2016, LNCS 9663, pp. 113–125, 2016.
DOI: 10.1007/978-3-319-39077-2_8

and claimed to be effective. But few of them take consideration into the problem of scalability. As the number of Android apps in markets grows rapidly, the practicality of those approaches are questionable.

In this paper, a scalable cloud-based detection framework is presented, which utilizes a hierarchical similarity-based app repackaging detection approach. The framework monitors specific alternative app markets and searches for new or updated apps. Then, the metadata and binary apk (application package) files of those matched apps will be crawled for multi-level feature extraction. A hierarchical multi-level based approach will be applied to generate a similarity score for two apps. Firstly, the similarity scores of apps is applied to smali [3] files in packages. Then, the similarities of packages will be computed according to the files similarity scores. Finally, the similarity score of apps will be calculated based on the packages similarity and app-level metadata. The presented approach takes consideration of similarities of files, packages and app-level metadata.

Upon the proposed framework and detection approach, we implemented a simplified prototype system. All components run in containers. The implemented system was deployed in the Aliyun cloud [4] on purpose of performance evaluation. The experiment results demonstrate that the proposed app repackaging detection framework is able to analyze and detect large-scale apps in an effective, efficient and flexible way.

The main contributions of this paper are summarized in the following:

– A scalable cloud-based Android app repackaging detection framework is presented. It takes the power of flexible cloud computing to monitor alternative app markets, extracting app features and compute similarities of apps in a large-scale way.
– A hierarchical similarity-based repacked app detection approach is proposed. It computes the similarity of apps with app-level metadata, packages similarity and smali files similarity results.
– A prototype system is implemented and deployed in a real cloud computing environment with all components running in containers. The evaluation results prove the effectiveness and efficiency of the proposed framework.

The rest of this paper is structured as follows. In Sect. 2, the proposed framework and its components are introduced. In Sect. 3, the implementation of prototype system is described. In Sect. 4, we evaluate the prototype system. In Sect. 5, we discuss the limitations of our work and the future improvement. In Sect. 6, we introduced the related work on Android app repackaging detection. Finally, we conclude our work in Sect. 7.

2 Design

2.1 Architecture Overview

As the market share of Android grows, the number of Android apps newly released every day also increases. Moreover, lots of existing apps update periodically. Thus, the total number of Android apps in official and alternative markets is quite considerable. Under this circumstances, the scalability is rather important for Android repacked app detection.

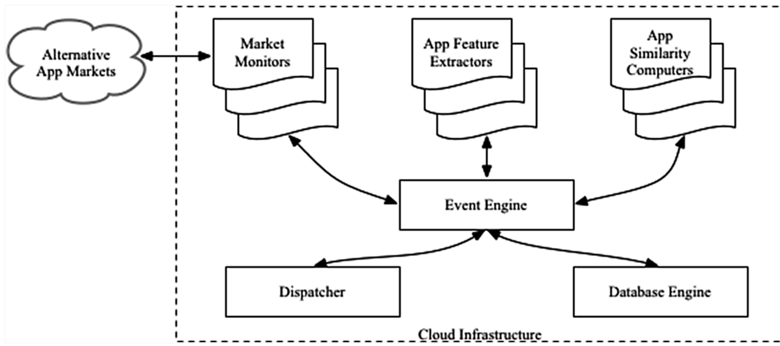

Fig. 1. The architecture overview of proposed framework.

The architecture overview of the proposed cloud-based framework for large-scale Android app repackaging detection is shown in Fig. 1. The market monitor checks the alternative app markets in watch list. When an app is added or updated in those markets, the market monitor crawls the app's metadata and downloads the corresponding apk file from the market. The app feature extractor takes the metadata and apk file as inputs for multi-level app features extraction. The app similarity computer computes the similarity scores of apps with these extracted features. The dispatcher distributes tasks to corresponding components. Besides, it monitors running load of the framework. The app metadata and extracted features are stored in a database engine. As the framework is designed to be event driven, there exists an event engine for events producing and consuming.

All components in the proposed framework are low-coupled, as all inter-components communications are accomplished through the event engine. Each component runs in respective computing instances in cloud.

2.2 Market Monitor

The key to fast app similarity computing is effective feature extraction in advance. To achieve this goal, all newly emerged or recently updated apps should be found and processed in time. Thus, the market monitor is constructed to monitor alternative app markets in the framework. Thinking of the resource limitation and spider politeness, currently, the market monitor scans all app markets only once a day. This frequency can be changed dynamically to meet different demands. As shown in Fig. 2, the workflow of market monitor consists of category monitoring, app metadata crawling and apk file downloading.

Category Monitoring. Category is defined as an app list in app markets. Market monitor is category-based, not market-based. This means each monitor just scans one app category in one market in the same time. With this mechanism, market monitor is able to scan different categories in the same market simultaneously without interference.

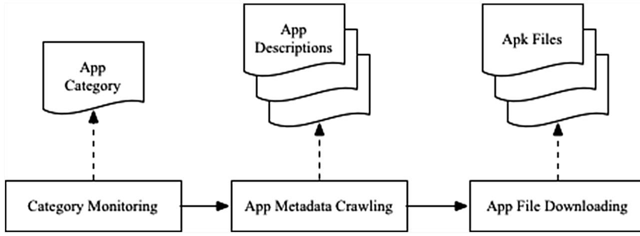

Fig. 2. The workflow of market monitor.

Every monitoring result will be kept as a screenshot of the category. When the screenshot changes, it means there are some newly uploaded apps in this category. After comparing with the previous screenshot, market monitor will mark these new apps as "NEW".

App Metadata Crawling. After scanning the app category, market monitor crawls each app's metadata (name, version, description, etc.) as an app screenshot to judge whether it is updated. Once the screenshot is found to be changed, the app will be marked as "UPDATED".

Apk File Downloading. In this step, the apk files of all apps marked as "NEW" and "UPDATED" will be downloaded from app markets. After that, all these binary files with all metadata will be sent to app feature extractor.

2.3 App Feature Extractor

A hierarchical similarity-based app repackaging detection approach is proposed. It consists of two steps: feature extracting and similarity computing. App feature extractor is used to extract multi-level features: app-level features, package-level features and file-level features in six steps as illustrated in Fig. 3.

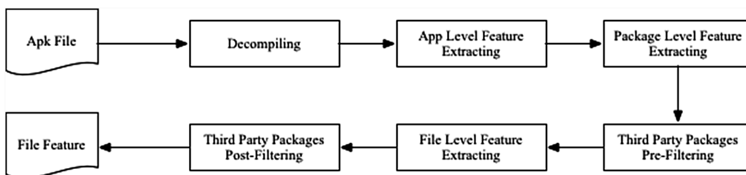

Fig. 3. The workflow of app feature extractor.

Decompiling. All downloaded apk files will be decompiled to get the file of "AndroidManifest.xml" and the file of author's signature certification. "Classes.dex" will be decompiled into smali files.

App Level Feature Extracting. App level feature includes app name, main activity, author signature and some other necessary elements.

Package Level Feature Extracting and Third Party Packages Pre-Filtering. Package-level feature consists of the name of package used in the app. The existing of third party packages may interfere the similarity computing result. Thus, we build a whitelist with a number of known third party packages. All those packages will be filtered during the pre-filtering step.

File Level Feature Extracting. There are three kinds of file-level features including the counts of specific methods, the counts of fields and the counts of opcodes in each smali file. Table 1 lists the main elements of file feature.

Third Party Packages Post-Filtering. We construct a post-filter to filter unknown third party packages according to the compound of some elements in file-level features.

Table 1. Main elements of file-level features.

Method related	private, public, static, final, Z, B, S, C, I, J, F, D, L, [
Field related	private, public, static, final, Z, B, S, C, I, J, F, D, L, [, V
Opcode related	goto, packed-switch, sparse-switch, if-eq, if-ne, if-lt, if-ge, if-gt, if-le, if-eqz, if-nez, if-ltz, if-gez, if-gtz, if-lez, cmpl-float, cmpg-float, cmpl-double, cmpg-double, cmp-long, add-type, sub-type, mul-type, div-type

2.4 App Similarity Computer

In the app repackaging detection approach, there are three thresholds: app similarity threshold, package similarity threshold and file similarity threshold. Once the similarity score of two apps/packages/files exceeds the corresponding threshold, the apps/packages/files are deemed to be similar. As shown in Fig. 4, there are three steps in app similarity computing.

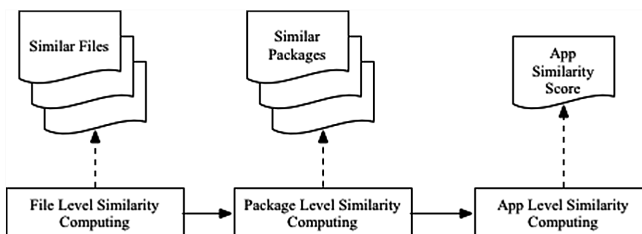

Fig. 4. The workflow of app similarity computer.

File Level Similarity Computing. All elements in the file feature will be transformed into integral values. These values make up of a file feature vector. The similarity score of two files is the cosine distance of the vectors:

$$Sim_F(File_a, File_b) = \text{cosine}(File_Vec_a, File_Vec_b) \tag{1}$$

Here, $File_Vec_a$ and $File_Vec_b$ represent the file feature vector. If the score exceeds the file similarity threshold, the two files are similar.

Package Level Similarity Computing. The package similarity score is computed based on all the similarity scores of files in two packages:

$$Sim_P(Package_a, Package_b) = \frac{File_Num_{P,sim}}{\min(File_Num_{P,a}, File_Num_{P,b})} \tag{2}$$

Here, $File_Num_{P,sim}$ is the number of similar files in two packages, while $File_Num_{P,a}$ and $File_Num_{P,b}$ are the number of files in respective packages. Two packages with a similarity score of exceeding the package similarity threshold are similar.

App Level Similarity Computing. After all packages similarity in two apps have been computed, the final similarity score of two apps are:

$$Sim_A(App_a, App_b) = \frac{File_Num_{A,sim}}{\min\left(File_Num_{A,a}, File_Num_{A,b}\right)} + \alpha \tag{3}$$

Here, $File_Num_{A,sim}$ is the number of actual similar files (only those similar files in similar packages) in two apps. $File_Num_{A,a}$ and $File_Num_{A,b}$ are the total file numbers in each app (not considering those files in packages filtered as third party packages). α is the weighted similarity result of app metadata. If the similarity score exceeds the app similarity threshold, the two apps are similar. Currently, the app similarity threshold is just for reference.

3 Prototype Implementation

We have implemented a simplified prototype of the proposed Android app repackaging detection framework. Redis [5] is used as an event engine for events producing and consuming. By utilizing the container technology, the prototype system can be easily and quickly deployed on any kind of cloud computing infrastructures. For evaluation purpose, we have deployed the whole prototype system in the Aliyun cloud. The architecture of the prototype system is illustrated in Fig. 5.

Database Engine and Event Engine. Redis, an in-memory data structure store, is used both as database engine and event engine.

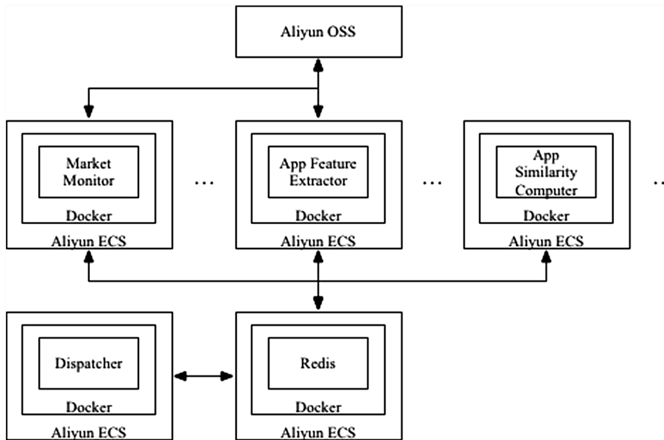

Fig. 5. The architecture of implemented prototype.

Dispatcher. All tasks are partitioned and distributed to the corresponding components. Celery [6], the distributed task queue, is the foundation of dispatcher. It also monitors running load of the prototype.

Market Monitor. This component crawls specific markets' web pages to search for newly added or updated apps periodically (like a day). All metadata and apk files of the qualified apps will be downloaded and stored. Currently, this component builds upon the outstanding open source spider: Scrapy [7].

App Feature Extractor. Once added or updated apps are crawled and downloaded, this component decompiles the corresponding apk files to extract metadata and smali files' features.

App Similarity Computer. Unlike other components, this component is external event-driven. It provides a simple interface for apps similarity computing demands.

Containerization. In consideration of simplicity and speed of system deployment, container technology is applied in the prototype implementation. More specifically, Docker [8], the most popular container is used. All system components: Redis, market monitor, app feature extractor and app similarity computer run in respective containers.

Scalability. Once the dispatcher finds the system has been under high load for a period of time, more computing instances of primary components will be started to accelerate the processing. Vice versa, some instances will be shut down while the load is continuously low.

Deployment. We have deployed our prototype system in the Aliyun cloud. While Aliyun Container Service [9] is still testing, several computing instances of Aliyun ECS [10] are used to host all containers running system components. To ease the sharing of apk files and decompiled files, Aliyun OSS [11] is used to host all files.

4 Evaluation

4.1 Experiment Setup

Two implemented prototype system (with scalability disabled and enabled) were deployed in the Aliyun cloud. For comparison, we also ran a prototype system in a local laptop. The three experiment environments are illustrated in Table 2.

Table 2. The cloud and local experiment environments.

Aliyun cloud		Local laptop	
Region	Qingdao	CPU	Intel i5 2.7 GHz
ECS number	7 (scalability disabled)	Memory	8 GB
	7–12 (scalability enabled)		
ECS CPU	2 Cores	Hard drive	1 TB
ECS memory	2 GB		
ECS hard drive	20 GB		
OS	Ubuntu 14.04 64Bit	OS	Ubuntu 14.04 64Bit
Docker number	8 (scalability disabled)	Docker number	5
	8–13 (scalability enabled)		

Table 3. The app markets in the watch list.

App markets	Gfan, Appchina, Anzhi, Pc6, 3310, Hiapk, Liqu, Cnmo

Table 4. The number of apps crawled.

Category	Number of apps
Entertainment	14204
Finance	13986
Shopping	13873

The watch list of market monitor contains eight app markets shown in Table 3. Considering the purpose of evaluating, only three categories were monitored and crawled: entertainment, finance and shopping.

During the experiment, there were a total number of 42063 apps crawled by the system as shown in Table 4.

4.2 Results

Table 5 lists the average time consuming of several operations of the prototype system in cloud and at local. Actually, the performances of one CPU core in both environments are close, which is proven by the result of operation No.3. Only one CPU core will be used when app level feature is being extracted. However, since the other operations are all CPU-bound, the systems deployed in cloud are much faster.

Furthermore, Table 5 shows that the implemented framework is quite scalable. While the system with scalability enabled is working hard on time-consuming tasks

Table 5. The average time consuming of several operations.

No	Operation	Aliyun cloud (Scalability disabled)	Aliyun cloud (Scalability enabled)	Local laptop
1	Crawling all three app categories in all eight markets	59.38 s	60.13 s	300.21 s
2	Crawling metadata of all 42063 apps in all eight markets	1634.59 s	1038.30 s	12030.43 s
3	Extracting app level feature of one app	0.32 s	0.30 s	0.34 s
4	Extracting package level features of all packages in one app	1.42 s	1.20 s	11.34 s
5	Extracting file level features of all files in one app	28.98 s	19.32 s	160.39 s
6	Computing the similarity score of two apps	4.95 s	3.44 s	21.89 s

(like operation No.2 and No.5), more computing instances are activated. With more computing resources, the total time of processing decreases a lot.

We manually analyzed hundreds of crawled apps crawled and collected 100 groups of highly similar (same or repacked with minor modifications) to evaluate the hierarchical similarity-based app repackaging detection approach. Another 100 groups were also selected, each of which consists of eight totally different apps.

Since the app similarity threshold is just for reference, we use a two-tuples: <package similarity threshold, file similarity threshold> in the experiment. Currently, the default app similarity threshold is set to be 0.8. As shown in Fig. 6, with <0.8, 0.9>,

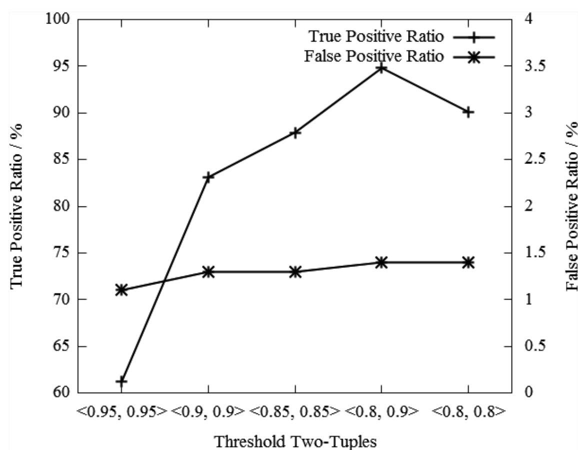

Fig. 6. The true positive ratios and false positive ratios of the proposed approach with different thresholds.

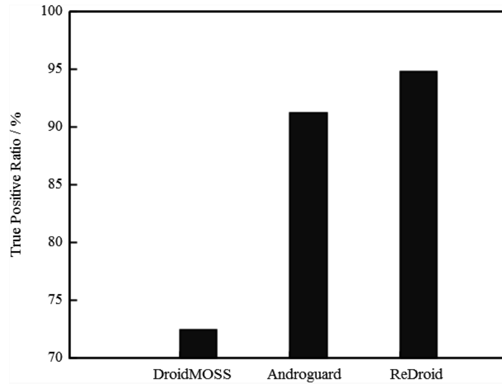

Fig. 7. The contrasting detection rates.

Table 6. The result of obfuscation resilient measurement.

Number of original apps	50
Number of obfuscated apps	50
Number of similar app pairs	47
Average similarity score	0.8349

the proposed approach achieved true positive ratio of 94.8 % and false positive ratio of 1.4 %. Actually, the false positive ratio is rather stable.

For comparing purpose, Fig. 7 shows the detection rates of "Androguard" [12], "DroidMOSS" [13] (implemented by ourselves) and our approach (with <0.8, 0.9>) called "ReDroid". As seen in Fig. 7, "ReDroid" outperforms the other approaches.

Obfuscation resiliency is a key measurement for app repackaging detection approach. To evaluate the robustness of the proposed approach against obfuscation, 50 different apps were selected and obfuscated. Then, the similarity scores of the original apps and obfuscated apps were computed. If the score exceeds the default app similarity threshold (0.8), the two apps are thought of being similar. As shown in Table 6, the proposed approach can detect repacked apps in an obfuscation resilient way. Moreover, the average similarity score indicates that there is still room for improvement against obfuscation.

5 Limitations and Future Work

Benefiting from the cloud-based architecture, the proposed framework is able to crawl and process Android apps in time on a large-scale. Constructed on the container cloud-based platform, all components can be deployed with flexibility. The proposed hierarchical similarity based approach is effective and efficient for app repackaging detection.

Nevertheless, during evaluation of the implemented prototype, a few limitations have come up which may influence the practicality of the framework. Thus, here lists some future work to avoid those limitations.

Under specific occasions, our framework fails to recognize some third-party packages which interfere the result of similarity computing. To solve this problem, an optimized third-party package self-learning and recognizing method is under developing. With the help of machine learning, the method is able to learn and recognize unknown third-party packages.

Currently, the similarity computing result comes just with similarity score, which is not intuitive for researchers. Thus, we are trying to present the result in a visual way with full details of computing procedure.

The biggest limitation is that the current implemented prototype is unsuitable for real-time app similarity computing. With the massive amount of existing apps in markets and in consideration of the explosion growth of new apps, the current similarity computing method requires too many computing resources. Though it can be achieved with continuous cloud resources investment, the cost is too high. A real-time cluster-based similarity computing method with proper cost is the key to solve this cloud resources black hole problem.

6 Related Work

ANDRUBIS [14] is an automated Android apps analysis system with both static and dynamic analysis. It is designed to provide Android security researchers a large-scale platform to analyze Android malwares. Users can submit Android apps through web site and mobile app provided by ANDRUBIS. ANDRUBIS has analyzed more than 1 million Android apps. A lot of details including network behavior of Android malwares are also presented.

A lightweight real-time Android malware detection framework: AndRadar [15] is proposed and implemented to identify potential malware in alternative Android app markets. It takes advantage of the metadata of apps (including the package name and developer's certificate fingerprint) to build a powerful app tracking framework. With AndRadar, the researchers are able to monitor the third-party markets efficiently in a real-time way.

DNADroid [16] is a large-scale Android clone apps searching tool against app cloning in app markets. Based on specific app attributes, suspicious cloned app candidates are selected and grouped for next robust clone code searching. The results claimed that the false positive rate of DNADroid is very low with a relatively high detection performance.

PiggyApp [17] is a fast and scalable system intended to find repacked apps in app markets. A module decoupling approach with semantic features extraction is introduced to decouple the primary modules of apps. The search approach for apps with similar modules is fast and efficient. However, the extracted feature fingerprint is not quite representative. Thus, the accuracy of detection system may be interfered.

ViewDroid [18] is also an app repackaging detection system. It extracts birthmarks as features from user interface components of apps. With these features, view graphs

are constructed for next repackaging detection. The evaluation results show that ViewDroid is robust, effective and highly resilient to code obfuscation. However, the graph comparison time grows much with the increment of the number of graph pairs..

DIVILAR [19] is a virtualization-based approach for Android app's self-protection. By hooking into the Dalvik VM, DIVILAR is able to composite the virtual instruction and Dalvik bytecode execution. DIVILAR is robust enough for both static and dynamic analysis, thus, effective for preventing app repackaging. However, the hooking mechanism severely limits the usage of DIVILAR since it requires a huge change in the underlying system.

DroidSim [20] is mainly designed to detect code reuse in Android Apps, especially for repackaged apps and malware variants detection. Compared to the other approaches, it's more accurate and robust. Component-based control flow graph (CB-CFG) is extracted to compute similarity score by DroidSim. An App can be uniquely represented with CB-CFG and author information. The evaluation results show that DroidSim is effective and robust for both repacked apps and malware variants detection. Yet, as it is claimed to be effective, the efficiency of DroidSim is almost not mentioned. The consuming time of construction of CB-CFG and graph-based pair-wise comparison is unclear.

As so many app repackaging detection and analysis approaches have been proposed, a framework for evaluating those algorithms [21] was presented. It is mainly designed to evaluate the performance of anti-obfuscation of detection algorithms. The experiment results show that the framework is able to evaluate detection algorithms broadly and deeply. The framework will present a full view about the strength and weakness of the evaluated algorithm after the evaluation.

7 Conclusion

In this paper we present a scalable cloud-based Android app repackaging detection framework. It is composed of three primary components: market monitor, app feature extractor and app similarity computer. Market monitor observes specific alternative app markets. Once there are apps newly added or updated in these markets, the metadata of apps will be crawled with the downloading of corresponding apk file for feature extraction and similar apps searching. App feature extractor extracts multi-level features. App similarity computer computes the similarities of apps as demanded.

Taking the advantages of cloud computing, the proposed framework is appropriate for large-scale app markets monitoring, features extracting and similarity computing. By using the cloud-based container technology, the implemented prototype is built to be scalable. The evaluation results demonstrate that this framework is able to detect repacked apps in a scalable, fast and effective way.

References

1. Apple App Store, December 2015. https://itunes.apple.com/us/genre/ios/
2. Google Play, December 2015. https://play.google.com/store/

3. Smali, December 2015. https://github.com/JesusFreke/smali/
4. Aliyun, December 2015. http://www.aliyun.com/
5. Redis, December 2015. http://redis.io/
6. Celery, December 2015. http://www.celeryproject.org/
7. Scrapy, December 2015. http://scrapy.org/
8. Docker, December 2015. http://docker.com/
9. Aliyun Container Service, December 2015. http://www.aliyun.com/product/containerservice/
10. Aliyun ECS, December 2015. http://www.aliyun.com/product/ecs/
11. Aliyun OSS, December 2015. http://www.aliyun.com/product/oss/
12. Androguard December 2015. https://github.com/androguard/androguardd
13. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party Android marketplaces. In: Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY), pp. 317–326. ACM (2012)
14. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., van der Veen, V., Platzer, C.: ANDRUBIS-1,000,000 apps later: a view on current Android malware behaviors. In: Proceedings of 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS) (2014)
15. Lindorfer, M., et al.: AndRadar: fast discovery of Android applications in alternative markets. In: Dietrich, S. (ed.) DIMVA 2014. LNCS, vol. 8550, pp. 51–71. Springer, Heidelberg (2014)
16. Crussell, J., Gibler, C., Chen, H.: Attack of the clones: detecting cloned applications on Android markets. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 37–54. Springer, Heidelberg (2012)
17. Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S.: Fast, scalable detection of "Piggybacked" mobile applications. In: Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY), pp. 185–196. ACM (2013)
18. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: ViewDroid: towards obfuscation-resilient mobile application repackaging detection. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks (ACM WiSec), pp. 25–36. ACM (2014)
19. Zhou, W., Wang, Z., Zhou, Y., Jiang, X.: DIVILAR: diversifying intermediate language for anti-repackaging on Android platform. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY), pp. 199–210. ACM (2014)
20. Sun, X., Zhongyang, Y., Xin, Z., Mao, B., Xie, L.: Detecting code reuse in Android applications using component-based control flow graph. In: Cuppens-Boulahia, N., Cuppens, F., Jajodia, S., Kalam, A.A.E., Sans, T. (eds.) SEC 2014. IFIP AICT, vol. 428, pp. 142–155. Springer, Heidelberg (2014)
21. Huang, H., Zhu, S., Liu, P., Wu, D.: A framework for evaluating mobile app repackaging detection algorithms. In: Huth, M., Asokan, N., Čapkun, S., Flechais, I., Coles-Kemp, L. (eds.) TRUST 2013. LNCS, vol. 7904, pp. 169–186. Springer, Heidelberg (2013)