Detecting Malicious Behaviors in Repackaged Android Apps with Loosely-Coupled Payloads Filtering Scheme

Lulu Zhang, Yongzheng Zhang(✉), and Tianning Zang

Institute of Information Engineering,
Chinese Academy of Sciences, Beijing 100093, China
{zhanglulu, zhangyongzheng, zangtianning}@iie.ac.cn

Abstract. Recently, the security problem of Android applications has been increasingly prominent. In this paper, we propose a novel approach to detect malicious behaviors in loosely-coupled repackaged Android apps. We extract and modify the FCG of an app based on its loosely-coupled property, and divide it into several sub-graphs to identify primary module and its related modules. In each remaining sub-graph, API calls are added and sensitive API paths are extracted for dynamic instrumentation on top of APIMonitor. The experiments are conducted with 438 malwares and 1529 apps from two third-party Android markets. Through manual verification, we confirm 5 kinds of malwares in 16 apps detected by our approach. And the detection rate of collected malwares reaches 99.77 %. The reduction rate of monitored functions reaches 42.95 % with 98.79 % of malicious functions being successfully saved. The time spent on static and dynamic analysis is 74.9 s and 16.0 s on average.

Keywords: Android security · Malicious behaviors · Payloads filtering · Dynamic instrumentation

1 Introduction

The development and popularity of Android mobile system is extremely rapid in recent years due to its open-source nature and the popularity of its app markets. However, malwares against Android devices has been increasingly rampant at the same time. The huge damage caused by Android malwares should never be tolerated. The Symantec Security Report [3] shows the security risks on Android app markets in the first half of 2013 and points out that the volume of malwares has reached almost 275,000 in June 2013 and its 2013 Annual Report indicates Android system is the number one target for malware.

For this end, it is imperative to design a precise and efficient approach to detect malicious behaviors in Android apps. We propose HunterDroid, an efficient and accurate system for detecting malicious behaviors in repackaged Android applications based on following key assumptions. Firstly, most malicious payloads are unnaturally added to legitimate apps so, they are usually loosely coupled with legitimate modules. Secondly, primary module of a legitimate app can be identified for it generally consists of activities decorated with certain APIs to enhance user interactions.

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2015
J. Tian et al. (Eds.): SecureComm 2014, Part I, LNICST 152, pp. 454–462, 2015.
DOI: 10.1007/978-3-319-23829-6_31

To evaluate the effectiveness of our approach, we conduct experiments with 438 malwares from Zhou [5] and 1529 apps from two third-party Android markets in China. Through manual verification, we confirm that 5 different kinds of malwares in 16 apps have been detected by our approach. Besides, the detection rate of collected malwares reaches 99.77 %. And the accuracy of primary module identification reached 94.14 %, 98.79 % of function nodes in malicious payloads are saved, and we reduce the number of function nodes to be monitored during instrumentation by 42.95 %. As for performance, the execution time caused in static analysis phase and dynamic instrumentation phase is 74.9 and 16.0 s on average respectively, which is reasonably negligible for an offline analysis tool.

In this paper, we make the following major contributions.

- We propose a novel payloads filtering scheme to detect malicious behaviors triggered by malicious payloads in repackaged Android apps.
- We instrument our samples on top of APIMonitor to cover complete sensitive API paths in malicious payloads.
- We implement a prototype system, HunterDroid, and conduct extensive experiments to evaluate its effectiveness and accuracy with 438 real-world malwares and 1529 apps from two third-party Android markets. And we confirm 5 kinds of malwares in 16 samples.

The rest of the paper is organized as follows. In Sect. 2 we describe our motivation and the overall architecture of our prototype system. In Sect. 3 we give detailed system design. After that, we present our evaluation results in Sect. 4. And we discuss the limitations and further work of our system in Sect. 5. Then we present the related work in Sect. 6 and finally we conclude in Sect. 7.

2 Motivation and Overall Architecture

As Zhou et al. indicates 86 % of their collected malware samples are repackaged versions of legitimate applications by adding malicious payloads, our system is designed to be an automated tool combined with static and dynamic analysis approaches to identify malicious behaviors in repackaged Android apps.

2.1 Motivation

There are three major analysis approaches to detect malicious behaviors in Android apps: static analysis, dynamic analysis and the combination of static analysis and dynamic analysis.

Static analysis focuses on finding certain malicious patterns based on pre-defined signatures. If pre-defined signatures are not complete, it is difficult to discover new-born malwares. Therefore, we propose to identify legitimate payloads in an app and analyze the remaining payloads further, so we need to distinguish legitimate payloads from non-legitimate payloads.

As for dynamic analysis, it is common to instrument Android system or app samples and then execute samples to analyze runtime behaviors. Google offers several

useful testing and debugging tools for monitoring and automatic execution like Monkeyrunner, Monkey, Logcat, there are still some problems to solve. For example, though Logcat is designed for recording runtime logs, it is hard for analysts to quickly recognize malicious behaviors from massive logs without setting proper filtering rules. Therefore, we think it is time-saving to filter unrelated function calls during static analysis before instrumentation.

2.2 Overall Architecture

Our prototype system, HunterDroid, detects malicious behaviors in repackaged Android apps in four steps, and the overall architecture is shown in Fig. 1.

Preparation. We obtain FCG with the help of Androguard and modify it based on the loosely-coupled property in repackaged Android apps.

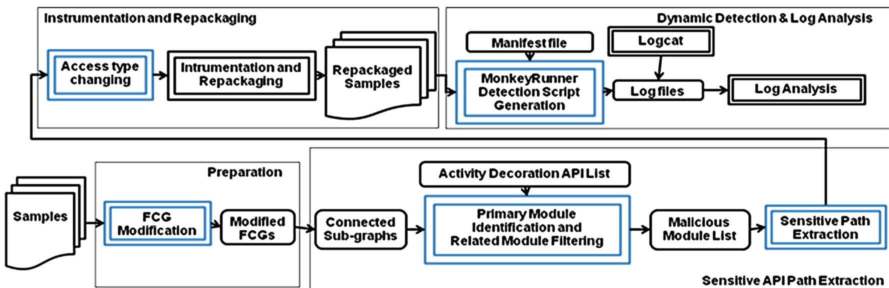

Fig. 1. Overall Architecture of HunterDroid

Sensitive API Path Extraction. We obtain sub-graphs in FCG by DFS and identify the primary module based on that most legitimate apps consist of activities that invoke its decoration APIs for user interactions. As for its related modules, they are filtered by computing the correlation values between them and the primary module. Finally we add API calls to each remaining module and extract sensitive API paths.

Instrumentation and Repackaging. This step is on top of APIMonitor, the major extension focuses on instrumenting sensitive API paths that contain both API nodes and self-defined function nodes rather than merely API nodes.

Dynamic Detection and Log Analysis. Each instrumented and repackaged application is executed by Monkeyrunner which is feed with an automatically built script based on distinct application manifest file.

3 System Design

In this section, we present detailed system design, including the modification of original FCG, sensitive API paths extraction, instrumentation and repackaging, and dynamic detection.

3.1 Modification of Original FCG

Though legitimate payloads and malicious payloads are loosely-coupled, they may be linked by few edges. Meanwhile, app developers tend to arrange package hierarchy according to the functions of different classes and malicious payloads often contain packages with relatively unrelated names. Based on these facts, we design a tactics to divide loosely-coupled sub-graphs into separated parts, as is shown in formula 1, where X and Y represent the function node set in a package, Edge(X, Y) represents the set that contains edge starts with X and ends with Y, pkg_sim(X, Y) is simply defined as follows. For instance, the similarity of "android.content" and "android.net.http" is 1/min(2,3) = 0.5, 2 and 3 represent the level number of "android.content" and "android.net.http", and 1 is the shared "android". We remove edges across X and Y if their CV value is less than T.

A proper threshold T should be able to separate the primary module and malicious modules without producing too many modules. By counting the number of separated modules with different thresholds, we find this tactics works well when T is 0.3.

$$CV(X,Y) = \max\left(\frac{|Edge(X,Y)|}{|Y|}, \frac{|Edge(Y,X)|}{|X|}\right) * P \tag{1}$$

3.2 Sensitive API Path Extraction

Definition 1. A sensitive API path is a call path in FCG that starts from a function node that is never called by any other function node and ends with an API that could be used maliciously.

Firstly, we formally define sensitive API path as Definition 1 and identify the primary module based on the following assumption: legitimate apps tend to enhance user interaction with activity decoration API which is presented in Definition 2, while malicious payloads tend to hide their behaviors by few or even no user interactions. Therefore, we build a list containing activity decoration APIs. By ranking the occurrences of these APIs in each module we identify the primary module.

Definition 2. An activity decoration API is a member method of an activity whose function is to modify the appearance of this activity by setting views.

Then we compute the correlation value between remaining modules and primary module by counting the elements belonging to primary class set occur in other modules. If exist, these modules are treated as related modules and filtered as well. Finally, we extend the remaining modules with APIs and extract sensitive API paths. Figure 2 depicts the whole payloads filtering process in static analysis.

3.3 Instrumentation and Repackaging

Samples are instrumented and repackaged based on APIMonitor, which is a promising tool by instrumenting an app with a user-defined API list and an android database. The major extension can be divided into following two aspects.

```
Algorithm 1 Payloads filtering scheme
Input:  Function call graph (FCG) of an app
Output: A list of self-defined functions and APIs
 1:  package_node_map = create_pkg_node_map(FCG)
 2:  for each pair<elem1,elem2> in package_node_map
 3:    if ( cohesion_value(elem1,elem2) <= threshold )
 4:      remove_edges_across(FCG,elem1,elem2)
 5:    end if
 6:  end for

 7:  sub-graphs = create_connected_subgraphs_by_DFS(FCG)
 8: compute_occurrence_of_decoration_APIs(sub-graphs)
 9:  primary_graph = select_the_max_occurrence(sub-graphs)

10:  filtered_graphs={ primary_graph}
11:  FCG = FCG - { primary _graph}
12:  while contain_related_graph(FCG,filtered_graphs)
13:    related_graph = select_graph_related(FCG,filtered_graphs)
14:    filtered_graphs = filtered_graphs ∪{related_graph}
15:    FCG = FCG - {related_graph}
16:  end while
17:  remove_nodes_and_edges(FCG,filtered_graphs)
18:  extend_remaining_graphs_with_APIs(FCG)
19:  save_nodes_to_file(FCG)
```

Fig. 2. The whole payloads filtering algorithm

Firstly, we notice that some sensitive APIs are widely used among all kinds of payloads. As APIMonitor does not consider where and how a sensitive API is invoked during instrumentation, we replace the API list with our sensitive API path file, only APIs invoked in remaining payloads are instrumented.

Secondly, we instrument self-defined functions to better understand the malicious working mechanism. However, the way APIMonitor instruments functions is to replace the original method with an exterior stub method in a newly-defined stub class, so we need to change the access type of original methods from private to public to ensure correct instrumentation.

3.4 Dynamic Detection

Our dynamic detection is executed by a distinct component-based python script for each sample, which enumerates all activities, receivers and services as well as their intent-filter information declared in manifest file.

In the script, Monkeyrunner starts activities with “android.intent.action.MAIN” first. Then it starts all remaining components by sending various fake events such as SMS_RECEIVED, BATTERY_CHANGED and BOOT_COMPLETED according to their own intent-filters. Each activity is exercised randomly by Monkey with user inputs and events. Finally, runtime logs are saved for further manual analysis.

4 Evaluation

In this section, we present our evaluation on the effectiveness and accuracy of HunterDroid. All experiments are conducted on an Intel Core machine with a four-core CPU 3.40 GHz CPU and 24 GB memory. a real Android device (HTC T329T with OS

version 4.1.1) is used to conduct dynamic detection. We first evaluate its effectiveness and accuracy in recognizing malicious payloads. Next, we measure the time performance of HunterDroid. Finally, we present its detection results.

4.1 Evaluation

The malware dataset consists of 438 malwares from Zhou [4] and 1529 samples crawled from two third-party Android markets in July 2013, we name them Eoe and Mumayi respectively.

To evaluate the accuracy of identifying primary modules, we manually verify each sample in Malware and 200 apps randomly sampled from Eoe and Mumayi. Our result shows 18 out of 638 samples are not correctly identified. So the accuracy of identifying primary module is 94.14 %.

Next, we evaluate the effects of our filtering payloads scheme, we define FN as the percentage of malicious function nodes in filtered function nodes and FP as the percentage of legitimate function nodes in monitored function nodes. Our experiment results show that the FN of Malware reaches as low as 1.21 %, i.e. 98.79 % of malicious function nodes are correctly saved for instrumentation. And the detection rate of malwares reached 99.77 %, there is only one sample from DroidKungFu3 is not detected for its malicious payloads is incorrectly labeled as primary module. As for FP, it reaches 18.22 %, as there are lots of callback nodes, which will not be instrumented anyway, so the real FP is lower.

In terms of effectiveness, we calculate the reduction rate of function node number, call edge and sub-graph after payload filtering process. Figure 3 depicts the difference between number of function nodes before and after filtering payloads process for all samples. The average reduction rate of function nodes is 42.95 %, which means almost half of function nodes are filtered before being instrumented.

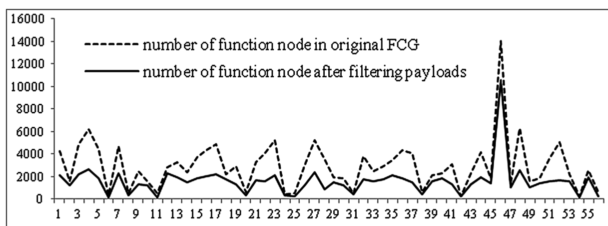

Fig. 3. The reduction rate of function nodes for all families or categories, x axis is the family id or category id, y axis is the number of function node

In order to evaluate the performance of our system, we record the execution time of static filtering scheme and dynamic instrumentation for each app. The time cost in filtering process is 74.9 s on average, and the average time for instrumentation is 16.0 s. Therefore, the total time to get a newly instrumented app is around 91 s on average, which is reasonably acceptable for an offline tool. As for detection results, by

inspecting the saved logs, we manually confirm 5 kinds of malwares in 16 apps, which further indicates the effectiveness of HunterDroid.

5 Discussion and Future Work

In this work, we focus on identifying malicious payloads in repackaged Android applications and our prototype system HunterDroid demonstrates some promising results. Now we discuss some limitations and improvements in future work.

Due to the incomplete list of decoration APIs, there are about 5 % apps that we cannot identify their primary modules. So in the future work, we will extend the decoration API list with APIs provided by other classes related to user interaction, like classes in “android.view”.

In addition, we resort to manually analyzing logs to discover malicious behaviors. As the log files are well-formatted in the future work, we will resort to some clustering and classification algorithms to better understand malicious behaviors in Android apps.

6 Related Work

HunterDroid is an approach to detect malicious behaviors in repackaged Android apps. In the field of Android malware detection, approaches can be divided into three aspects: dynamic detection, static detection and a combination of static and dynamic approaches.

Dynamic detection approach tracks the sensitive behaviors at runtime by instrumenting app code or Android system [1]. Enck et al. [7] proposed a dynamic taint tracking analysis system to track multiple sources of sensitive data and analyze at runtime within Android virtualized execution environment. Zhang [2] et al. implemented VetDroid for reconstructing sensitive behaviors from a permission usage perspective.

On the other hand, static detection approaches mainly focus on identifying the possible malicious behaviors with the help of reachability analysis and program slicing [1]. Grace et al. [11] developed RiskRanker aimed at root exploits detection, permission analysis and data-flow analysis. Crussel et al. [8] proposed AnDarwin to detect similar Android applications based on semantic information of app codes.

As for Approaches combined with static analysis and dynamic detection, Yang et al. [1] proposed AppIntent to identify whether data transmission is by user intention or not by building and executing GUI manipulation sequences generated by event-space constraint graph. Zheng et al. [19] proposed SmartDroid to reveal UI-based event trigger condition based on function call graph and activity switch paths.

Our approach belongs to the third aspect. It takes the loosely-coupled property of repackaged apps into account to extract malicious payloads in form of sensitive API paths before instrumentation. Compared with RiskRanker [10] and DroidRanger [9], we pay attention to the features legitimate apps share commonly and treat the remaining payloads as suspicious rather than detecting malwares by pre-defining malicious symptoms. In comparison with Andarwin [8] and Peng [4], our focus is to

detect malicious behaviors rather than cluster static features or calculate risk scores, and leave users to make decision.

7 Conclusion

In this paper, we present HunterDroid, a system for detecting malicious behaviors in repackaged Android applications. Based on our key assumptions that added payloads are mostly loosely-coupled with original codes and legitimate apps are commonly well decorated to enhance user interactions, we propose an efficient payload filtering scheme to locate malicious payloads for dynamic instrumentation. We have implemented a prototype HunterDoid and evaluated its effectiveness and accuracy with 438 real-world malwares and 1529 apps from third-party markets. Our experiments successfully verify 5 kinds of malwares in 16 apps in the wild detected by HunterDroid. And the detection rate of our malware dataset reaches 99.77 %. The reduction rate of monitored functions reaches 42.95 % with 98.79 % of malicious functions being successfully saved. The static analysis only takes 74.9 s and 16.0 s for instrumentation for each app on average.

Acknowledgment. The authors would like to thank the anonymous reviewers for their helpful comments for improving this paper. This work is partially supported by National Natural Science Foundation of China (NSFC) under contracts (No. 61303170, No. 61070185 and No. 61100188) and Knowledge Innovation Program of The Chinese Academy of Sciences under contracts (No. XDA06030200), and The National High Technology Research and Development Program of China under contracts (No. 2012AA012803 and No. 2013AA014703) and The National Key Technology R&D Program under contracts (No. 2012BAH46B02).

References

1. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P.,Wang, X.S.: AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, pp. 1043–1054. ACM Press, New York (2013)
2. Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang X.S., Zang B.: Vetting undesirable behaviors in android apps with permission use analysis. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, pp. 611–622. ACM Press, New York (2013)
3. android–madware–and–malware–trends http://www.symantec.com/connect/blogs/android-madware-and-malware-trends
4. Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Molloy, I. Using probabilistic generative models for ranking risks of android apps. In: Proceedings of the 2012 ACM conference on Computer and Communications Security, pp. 241–252. ACM Press, New York (2012)
5. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 95–109. IEEE Press, New York (2012)
6. Felt, A.P., Chin, E., Hanna, S., Song, D.,Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 627–638. ACM Press, New York (2011)

7. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the android permission specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 217–228. ACM Press, New York (2012)
8. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.: TaintDroid: an information-flow tracking system for real-time privacy monitoring on smartphones. In: OSDI, vol. 10, pp. 1–6 (2010)
9. Crussell, J., Gibler, C., Chen, H.: Scalable semantics-based detection of similar android applications. In: ESORICS (2013)
10. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium, pp. 5–8 (2012)
11. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: RiskRanker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th International Conference on Mobile systems, Applications, and Services, pp. 281–294. ACM Press, New York (2012)
12. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, pp. 239–252. ACM Press, New York (2011)
13. Wang, Z., Jiang, X., Cui, W., Wang, X., Grace, M.: ReFormat: automatic reverse engineering of encrypted messages. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 200–215. Springer, Heidelberg (2009)
14. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: mining api-level features for robust malware detection in android. In: Zia, T., Zomaya, A., Varadharajan, V., Mao, M. (eds.) SecureComm 2013. LNICST, vol. 127, pp. 86–103. Springer, Heidelberg (2013)
15. Yang, C., Yegneswaran, V., Porras, P., Gu, G.: Detecting money-stealing apps in alternative android markets. In: Proceedings of the 2012 ACM conference on Computer and Communications Security, pp. 1034–1036. ACM Press, New York (2012)
16. Shabtai, A., Fledel, Y., Elovici, Y.: Automated static code analysis for classifying android applications using machine learning. In: 2010 International Conference on Computational Intelligence and Security (CIS), pp. 329–333. IEEE Press, New York (2010)
17. Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., Siemens, C.E.R.T.: DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. Gottingen, Germany (2014)
18. Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S.: Fast, scalable detection of piggybacked mobile applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy, pp. 185–196. ACM Press, New York (2013)
19. Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., Zou, W.: Smartdroid: an automatic system for revealing UI-based trigger conditions in android applications. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 93–104. ACM Press, NewYork (2012)