International Journal of Internet, Broadcasting and Communication Vol.8 No.4 11-18 (2016)
http://dx.doi.org/10.7236/IJIBC.2016.8.4.11

IJIBC 16-4-2

Android App Reuse Analysis using the Sequential Hypothesis Testing

Jun-Won Ho

Department of Information Security
Seoul Women's University
621 Hwarangro, Nowon-Gu, Seoul, South Korea
jwho@swu.ac.kr

Abstract

Due to open source policy, Android systems are exposed to a variety of security problems. In particular, app reuse attacks are detrimental threat to the Android system security. This is because attacker can create core malign components and quickly generate a bunch of malicious apps by reusing these components. Hence, it is very imperative to discern whether Android apps contain reused components. To meet this need, we propose an Android app reuse analysis technique based on the Sequential Hypothesis Testing. This technique quickly makes a decision with a few number of samples whether a set of Android apps is made through app reuse. We performed experimental study with 6 malicious app groups, 1 google and 1 third-party app group such that each group consists of 100 Android apps. Experimental results demonstrate that our proposed analysis technique efficiently judges Android app groups with reused components.

Keywords: *Android; app reuse; sequential hypothesis testing*

1. Introduction

Android applications are widely used in various fields such as games, science, finance, and health etc. Although Android applications make it easier for people to perform their tasks in these fields, they have various security problems due to the open source policy. In particular, Android app reuse attacks are very dangerous in the sense that attacker can generate core malware components and spawn a large number of Android malwares by reusing these components, leading to devastating harm on Android ecosystem. To protect Android ecosystem from these attacks, it should be able to perceive whether a group of Android applications reuses some components.

To fulfill this need, we propose an app reuse analysis technique based on the sequential hypothesis testing (SHT). The SHT is a statistical decision mechanism developed by Wald [15]. It makes a decision with a few number of samples while accomplishing low error rates. In our proposed technique, we take a group of Android apps and use the SHT to decide whether a group comprises of apps with reused components. More

Manuscript Received: Aug. 31, 2016 / Revised: Sep.13, 2016 / Accepted: Sep. 25, 2016
Corresponding Author: jwho@swu.ac.kr
Tel: +82-2-970-5607, Fax: +82-2-970-5981
Department of Information Security, Seoul Women's University

specifically, the SHT treats a sample as the fraction of the equal dex byte portions between two apps distinctly and randomly picked up in an app group to the entire dex bytes. The sequential hypothesis testing takes a series of samples and it goes toward the acceptance of null (resp. alternate) hypothesis each time the sample value is below (resp. above or equal to) a predefined fraction of reused components. Once it accepts alternate (resp. null) hypothesis, it decides that Android apps in app group do (resp. not) contain the reused portions. In the sense that the SHT requires on an average a few number of samples to reach a decision, the SHT-based analysis will quickly and accurately determine app groups containing reused portions when compared to calculating the similarity of every pairs of apps in app groups. Additionally, there is no limitation on target apps to which our proposed technique is applied. In other words, it can be applied to any arbitrary groups of apps regardless of whether apps are already classified as malicious or not.
We evaluate the performance of our SHT-based analysis by using real Android data set. This data set consists of six malicious app groups collected in [1] and one official app group collected from google play store and one unofficial app group collected from third-party market. According to the evaluation results, our proposed analysis determines the existence of reused sections with at most 5.5 samples on an average in five malicious app groups, one google and one third-party app group while it requires on an average 9 samples in one malicious app group. Moreover, the evaluation results exhibit that malicious app groups are apt to have more reused components than app groups from google and third-party markets.

The rest of paper is organized as follows. In Section 2, we present related work regarding Android app reuse. In Section 3, we describe our proposed analysis technique to determine whether a set of Android apps contains reused components. In Section 4, we provide the results of experiments through which we evaluate our proposed analysis. In Section 5, we make a conclusion the paper.

2. Related Work

In this section, we present the related work with respect to Android app reuse/repackaging/clone.
Chen et al. proposed a signature-based detection scheme for app clone attacks [2]. In [3], geometry property of dependency graphs is used to discover similar code portions in two apps. In [4], AnDarwin tool is developed to find out the similarity of Android apps by using semantic information. DNADroid is proposed to calculate the similarity of Android apps by utilizing program dependency graphs in [5]. Gonzalez et al. proposed a DroidKin system that utilizes the nature of binary and meta data associated with Android apps to detect app similarity [6]. Hanna et al. developed a scalable Juxtapp tool that discerns Android app reuse [7]. In [8], ImageStruct system is proposed to check image similarity from Android app repackaging. Kim et al. proposed a dynamic detection technique against reused Android apps [9]. In this technique, the app reuse can be identified through monitoring the API calls between apps and mobile system. In [10], Kullback-Leibler Divergence (KLD) is used to unearth the repackaged Android malwares. Shao et al. leverages core resources linked with codes to find out the repackaged Android apps [11]. In [12], runtime user interface information is leveraged to hunt out Android app clones. Sun et al. utilizes component-based control flow graph to reveal Android code reuse [13]. In [14], layout resources are utilized to discern visually alike apps. In [16], Android apps are roughly examined for app clone detection at the first stage and they are explored in detail at the second stage. Zhang et al. designed ViewDroid system in which the user interactions between users and apps are made use of finding out app reuse [17]. In [18], FSquaDRA was proposed to dig out Android application repackaging rooted on the resources files. In [19], AppInk is proposed to uncover Android app repackaging with watermarking technique. Zhou et al. developed DroidMOSS system in which a fuzzy hashing method is adapted to identify Android app reuse [20].

Although our proposed technique is closely relevant to the aforementioned repackaging/clone/reuse detection work in android apps, it is distinct in the target to which the work is applied. Specifically, the aforementioned work mainly focus on detection of whether a single app is cloned or not. On the other hand, our proposed technique analyzes the existence of the reused components in an arbitrary group of apps. Hence, it will be efficient to apply our proposed scheme to explore the reused app group. Moreover, with the aid of the SPRT, it does not need to look into the entire app pairs in a group, but a few number of app pairs in order to make a decision with regard to the reused app group.

3. App Reuse Analysis using the SHT

The sequential hypothesis testing (SHT) is deemed to be a statistical decision process in which it could accurately makes a decision with a few number of samples [15]. We adapt the SHT to our app reuse analysis as follows: Let $E_m (m \geq 1)$ denote the fraction of the equal portions between the dex bytes of distinctly and randomly selected two apps in a set of apps to the entire dex bytes. If two apps have different size, we calculate the equal portions on the basis of the smaller size. We define a Bernoulli random variable F_m as follows:

$$F_m = 0 \quad \text{if } E_m < E^*$$

$$F_m = 1 \quad \text{if } E_m \geq E^*$$

The success probability ι of the Bernoulli distribution is given by

$$\iota = \Pr(F_m = 1) = 1 - \Pr(F_m = 0) \qquad (1)$$

We have preset thresholds ι_0, ι_1, and E^* such that $\iota_0 < \iota_1$ and E^* represents the threshold for the fraction of app reuse. Based on ι_0 and ι_0, we define null hypothesis H_0 that a set of apps does not contain at least E_m percentage reused components over the entire codes. We also define alternate hypothesis H_1 that a set of apps contains at least E_m percentage reused components over the entire codes. When $\iota \leq \iota_0$, H_0 is likely accepted. On the other hand, if $\iota \geq \iota_1$, H_1 is likely accepted.

On the basis of the property of H_0 and H_1, we describe how the SHT makes a decision regarding app reuse from the *m* observed samples, where F_m is treated as a sample. The log-probability ratio on *m* samples G_m is given as:

$$G_m = \ln \frac{\Pr(F_1, \ldots, F_m \mid H_1)}{\Pr(F_1, \ldots, F_m \mid H_0)}$$

We assume that F_m is independent and identically distributed. This is reasonable from the perspective that E_m is calculated from distinctly and randomly selected two apps and thus it is independent of other values.

By the i.i.d. assumption, G_m can be expressed as:

$$G_m = \ln \frac{\prod_{c=1}^{m} \Pr(F_c \mid H_1)}{\prod_{c=1}^{m} \Pr(F_c \mid H_0)} = \sum_{c=1}^{m} \ln \frac{\Pr(F_c \mid H_1)}{\Pr(F_c \mid H_0)} \qquad (2)$$

Let v_m denote the number of times that $F_c = 1$ in the *m* samples. Then Equation 2 is converted to

$$G_m = v_m \ln \frac{\iota_1}{\iota_0} + (m - v_m) \ln \frac{1-\iota_1}{1-\iota_0} \qquad (3)$$

where $\iota_0 = \Pr(F_c = 1 \mid H_0)$, $\iota_1 = \Pr(F_c = 1 \mid H_1)$.

Then, the SHT for H_0 against H_1 is given by:

- $v_m \le a_0(m)$: accept H_0 and terminate the SHT.
- $v_m \ge a_1(m)$: accept H_1 and terminate the SHT.
- $a_0(m) < v_m < a_1(m)$: continue the SHT with another observation.

Where α' is a user-configured false positive error is rate and β' is a user-configured false negative error rate.

$$a_0(m) = \frac{\ln \frac{\beta'}{1-\alpha'} + m \ln \frac{1-\iota_0}{1-\iota_1}}{\ln \frac{\iota_1}{\iota_0} - \ln \frac{1-\iota_1}{1-\iota_0}} \qquad a_1(m) = \frac{\ln \frac{1-\beta'}{\alpha'} + m \ln \frac{1-\iota_0}{1-\iota_1}}{\ln \frac{\iota_1}{\iota_0} - \ln \frac{1-\iota_1}{1-\iota_0}}$$

Once the SHT accepts H_0 or H_1 , it newly restart and repeats the above process with a new series of samples

4. Experimental Study

In this section, we first describe experiment environments to evaluate our proposed analysis and then present the experiment results.

4.1. Experiment Environments

We performed our experiments in Santoku virtual machine installed on iMac computer. The reason why we adopt virtual machine for experiments is to prevent app reuse analysis task from affecting the host machine. We also used the Android malware data set collected in [1]. More specifically, as shown in Table 1, we employed 6 malicious app groups gathered by Arp et al. [1] and 1 official app group from google play store and 1 unofficial app group from third-party market. We set the number of apps in a group to 100. Even though we apply our proposed analysis to these eight app groups, there is no limitation on target apps to which it is applied. In other words, it works regardless of whether apps are already classified as malicious or not. Rather, it can be applied to any arbitrary groups of apps.

Table 1. Android application group in data set

Label	Android app group
Google	Android apps collected from official google market.
Third	Android apps collected from unofficial third-party market.
BaseB	BaseBridge malicious app group.
DroidK	DroidKungFu malicious app group.
FakeI	FakeInstaller malicious app group.
GinM	GingerMaster malicious app group.
Icon	Iconosys malicious app group.
Opfake	Opfake malicious app group.

Table 2. Analysis evaluation metrics.

Average Number of Samples	Average number of samples needed for the SHT to reach a decision.
SHT Decision Fraction	Fraction of H_0 and H_1 decisions made by the SHT.

We also configure $E^* = 0.15, 0.3.$ In terms of the SHT configurations, we set $\alpha' = \beta' = 0.01$. We also configure $\iota_0 = 0.1, \iota_1 = 0.9.$ Given 50 samples for each app group such that a sample is defined as the fraction of the equal portions between two randomly selected apps' dex bytes to the entire dex bytes, the sequential hypothesis testing(SHT) initiates and restarts itself each time it reaches a decision. We repeat this process 100 times and present an average of the 100 executions in the following section.

4.2. Experiment Results

We present the evaluation results of our SHT-based analysis using the metrics defined in Table 2.
In terms of average number of samples, when $E^* = 0.15,$ our SHT-based analysis technique on an average makes a decision with at most 5 samples in all app groups. When $E^* = 0.3,$ it requires on an average at most 5.5 samples to reach a decision in all app groups except FakeInstaller malicious app group, in which it demands 9 samples to reach a H_1 decision on an average. This means that our SHT-based analysis quickly determines whether app groups take in reused components.

When $E^* = 0.15,$ the SHT decision fractions are shown in Figure 1. We observe that all SHT decisions in BaseBridge, FakeInstaller, Iconosys, and Opfake malicious app groups are H_1 . This indicates that the probability of app reuse in these malicious app groups is 1. Putting it in different way, Android malwares belonging to these four groups share at least 15% of the entire codes with other malwares in the same group. We also see that 84% and 96% of SHT decisions in DroidKungFu and GingerMaster malicious app groups are H_1 , respectively. This signals that the proportion of reused components in DroidKungFu (resp. GingerMaster) malicious app group is at least 15% with probability 0.84 (resp. 0.96), respectively. On the other hand, we observe that 100% of SHT decisions in both Google and Third-party group is H_0. This signifies that the cloned fractions of Android apps in these groups are less than 15%.

Figure 2 shows the SHT decision fractions when $E^* = 0.3$. We clearly see that the fractions of H_1 decision in all app groups are smaller than the ones in case of $E^* = 0.15$.

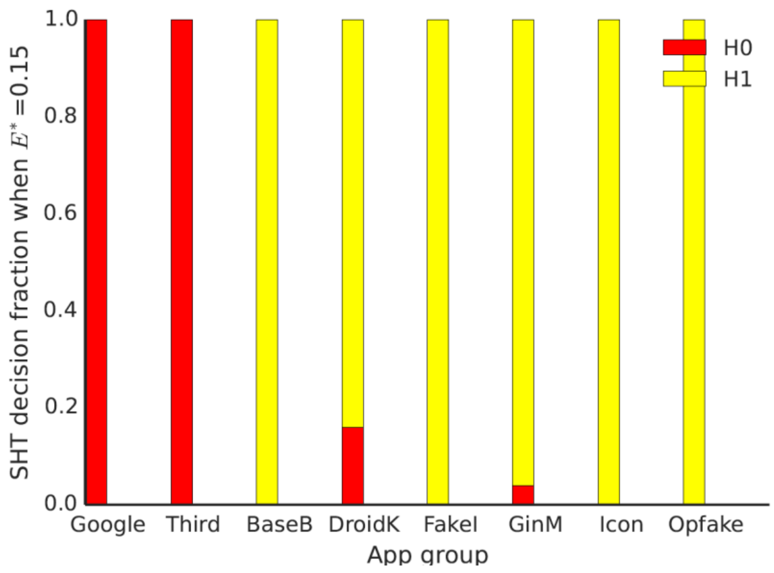

Figure 1: Fraction of SHT decisions (H_0 ,H_1) in seven app groups when $E^* = 0.15$.

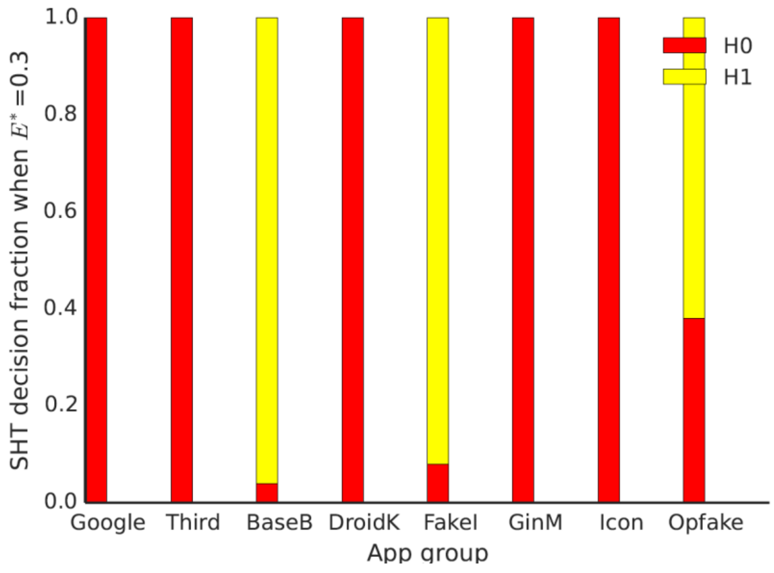

Figure 2: Fraction of SHT decisions (H_0 ,H_1) in seven app groups when $E^* = 0.3$.

In particular, all decisions are H_0 in Google, Third-party, DroidKungFu, GingerMaster, and Iconosys app groups, meaning that the reused fractions of Android apps in these groups are less than 30%. On the other hand, the fractions of H_1 decision in BaseBridge and FakeInstaller malicious app groups are still at least 92%. From this observation, it is asserted with at least probability 0.92 that Android malwares in

BaseBridge and FakeInstaller groups are generated with reused components, whose fraction is at least 30% of the whole codes.

5. Conclusions

In this paper, we proposed Android app reuse analysis technique based on the sequential hypothesis testing. By taking advantage of the fast decision property in the sequential hypothesis testing, we quickly determine whether a given app group contains apps with reused components. In particular, we demonstrate that Android malwares tend to be spawned through app reuse by applying our proposed analysis technique to real Android app groups consisting of six malicious groups and one official google group and one unofficial third-party group.

Acknowledgement

This work was supported by a research grant from Seoul Women's University (2015).

References

[1] Daniel Arp, Michael Spreitzenbarth, Malte Huebner, Hugo Gascon, and Konrad Rieck. Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket, In 21th Annual Network and Distributed System Security Symposium (NDSS), February 2014

[2] Chen, J., Alalfi, M. H., Dean, T. R., and Zou, Y. Detecting Android Malware Using Clone Detection. Journal of Computer Science and Technology, 30(5), 942-956, 2015.

[3] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In International Conference on Software Engineering (ICSE), pages:175-186, 2014.

[4] Crussell, Jonathan, Clint Gibler, and Hao Chen. Scalable semantics-based detection of similar Android applications. In ESORICS 2013, Springer Berlin Heidelberg, 2013.

[5] Crussell, Jonathan, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In ESORICS 2012, 2012

[6] Gonzalez, H., Stakhanova, N., and Ghorbani, A. A. DroidKin: lightweight detection of Android apps similarity. In International Conference on Security and Privacy in Communication Networks, pages:436-453, Springer International Publishing, September 2014.

[7] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: a scalable system for detecting code reuse among Android applications. In DIMVA'12 Proceedings of the 9th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 2012.

[8] Jiao, S., Cheng, Y., Ying, L., Su, P., and Feng, D. A Rapid and Scalable Method for Android Application Repackaging Detection. Information Security Practice and Experience, pages:349-364, Springer International Publishing, 2015.

[9] Kim, D., Gokhale, A., Ganapathy, V., and Srivastava, A. Detecting plagiarized mobile apps using API birthmarks. Automated Software Engineering, pages:1-28, 2015.

[10] H. Shahriar and V. Clincy Kullback-Leibler Divergence Based Detection of Repackaged Android Malware. Journal of Information Security Research, Vol. 6, Num. 1, March 2015.

[11] Shao, Y., Luo, X., Qian, C., Zhu, P., and Zhang, L. Towards a scalable resource-driven approach for detecting repackaged Android applications. In Proceedings of the ACM Annual Computer Security Applications Conference, pages:56-65, December 2014.

[12] Soh, C., Tan, H. B. K., Arnatovich, Y. L., and Wang, L. Detecting clones in Android applications through analyzing user interfaces. In IEEE 23rd International Conference on Program Comprehension (ICPC), pages:163-173, May 2015.

[13] Sun, X., Zhongyang, Y., Xin, Z., Mao, B., and Xie, L. Detecting code reuse in Android applications using component-based control flow graph. In ICT Systems Security and Privacy Protection, pages:142-155, Springer Berlin Heidelberg, 2014.

[14] Sun, M., Li, M., and Lui, J. DroidEagle: seamless detection of visually similar Android apps. In Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, June 2015.

[15] A. Wald. Sequential Analysis. Dover, 2004.

[16] Wang, H., Guo, Y., Ma, Z., and Chen, X. WuKong: a scalable and accurate two-phase approach to Android app clone detection. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, pages:71-82, July 2015.

[17] Zhang, F., Huang, H., Zhu, S., Wu, D., and Liu, P. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In Proceedings of the ACM conference on Security and privacy in wireless & mobile networks, pages: 25-36, July 2014.

[18] Zhauniarovich, Y., Gadyatskaya, O., Crispo, B., La Spina, F., and Moser, E. FSquaDRA: fast detection of repackaged applications. In Data and Applications Security and Privacy XXVIII, pages:130-145, Springer Berlin Heidelberg, 2014.

[19] W. Zhou, X. Zhang, and X. Jiang. AppInk: watermarking Android apps for repackaging deterrence. In ASIA CCS, pages:1-12, May 2013.

[20] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY), 2012.