“Do you want to install an update of this application?” A rigorous analysis of updated Android applications

Ahmet Ilhan AYSAN
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
Email: aysan@hacettepe.edu.tr

Sevil SEN
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
Email: ssen@cs.hacettepe.edu.tr

***Abstract*—Attackers have been searching for security vulnerabilities in Android applications to exploit. One of these security vulnerabilities is that Android applications could load codes at runtime. This helps attackers to avoid being detected by static analysis tools. In this study, we have done a rigorous analysis to see how attackers employ updating techniques in order to exploit this vulnerability, and to assess the security risks of applications using these techniques in the markets. A comprehensive analysis is carried out on nearly 30,000 applications collected from three different Android markets and two malware datasets. Both static and dynamic analysis techniques are employed to monitor malicious activities in such applications. As a result, we found 70 new malicious applications from Google Play. Our work is the first study which monitors updating behaviours of applications during their execution. This analysis allows us to analyse suspicious applications deeply and to develop better security solutions.**

Keywords*—*Android, mobile malware, static analysis, dynamic analysis, update attacks, dynamic code loading.

I. INTRODUCTION

Android architecture provides a mechanism for developers to update their applications after their installations completed on the device. The updating mechanism allows attackers to load malicious payload or to change the application completely at runtime. Therefore it helps attackers to hide their malicious activities from the analysis carried out in the market stores. Detecting these types of malicious activities is one of the biggest problems the market stores face with. Moreover, these types of applications usually do not follow the updating policy of the application markets. After installation, application fetches malicious payload from servers determined by the application developer. In April 2013, Google Play declared that “An app downloaded from Google Play may not modify, replace or update its own APK binary code using any method other than Google Play’s update mechanism” [8] . However the reality is different from its policy. Even Facebook, one of the most popular applications in Google Play, still updates itself by using its own servers. Furthermore, Amazon[1] and SlideMe[12] stores do not set a policy about updating applications from unknown servers.

In this study, we analysed updated applications from three different Android markets: Google Play [7], SlideMe [12] and AppsApk [4]. We also investigated malwares using updating techniques in the publicly available malware datasets, namely Malgenome [29] and Drebin [14]. Both static and dynamic analysis are carried out to reveal malicious applications using updating techniques in the market stores. At first, we investigated suspicious applications by applying signature-based analysis. Secondly, dynamic analysis techniques are performed on each application in order to reveal malicious applications which hide themselves from static analysis techniques by using evasion techniques such as obfuscation, encryption, and other similar means.

Our main contributions could be summarized as follows:

- We have done a rigorous analysis on mobile applications which use the updating techniques, upgrading, silent installing and dynamic class loading. We analysed nearly 30,000 applications collected from three different markets and two malware datasets.
- Our work is the first large-scale analysis (signature-based and dynamic analysis) that uncovers malicious applications using updating techniques. Even though there has been a limited amount of research on statically analysing malwares using dynamic class loading ([26], [22], [21]), we also applied dynamic analysis techniques in order to investigate updating applications evading from static analysis. Our up-to-date analysis shows the dramatic increase in the number of applications using dynamic class loading. Moreover, 70 new malicious applications from Google Play not detected by VirusTotal and Google Bouncer are discovered as a result of this analysis.
- We also investigate the mechanisms that trigger malicious applications. The results show that triggering is one of the evasive strategies effectively applied by attackers. As far as we know, there is no study focusing on triggering mechanisms to reveal applications updating themselves. The time-based triggering techniques have largely increased the number of applications to analyse. It also allows us to find new updated attacks which are not revealed by the well-accepted analysis study previously [29].

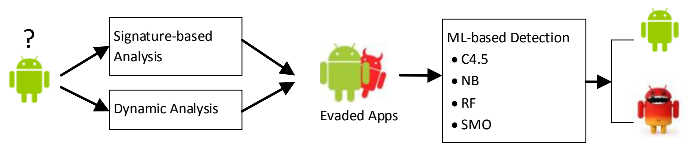


Fig. 1. The conceptual schema of the analysis

The remainder of this paper is organized as follows: the updating techniques are presented in Section 2. Our methodology and the triggering mechanisms are introduced in Section 3, the analysis results are discussed thoroughly in Section 4 and the related work is discussed in Section 5. Finally, we conclude our work in Section 6.

II. ANDROID UPDATE TECHNIQUES

Most of the application stores use packet manager to manage the installation of applications' updated/new versions. Application managers usually check applications whether they need to install new package or not. Typically, Android OS developers employ the updating techniques explained below:

Upgrading: When a new version is ready in the store, the packet manager represents the new applications to the user or triggers the automatic update. If the application name, the permissions and the application signature are the same with the previous version, the update mechanism is triggered implicitly. Otherwise the installation process is committed explicitly for users who might not have superuser privileges.

Silent Installing: This technique is only applicable in rooted devices. Users need to have root privileges in order to perform installation without any approval. Therefore, an attacker has an opportunity to install malicious application without user approval. In this study, this mechanism is aptly called silent updating. An attacker uses *"pm install"* command in order to start installation.

Dynamic Class Loading: Android applications are written originally in Java and compiled into the .dex file. Android applications have powerful flexibility that developers can load applications (.jar and .apk files) from any server at runtime [5]. Since dex file has limitation up to 64K reference size, developers typically use the dynamic class loader to overcome this limitation. Specifically, they divide the application into several files and each file is dynamically loaded during the execution by using the *DexClassLoader* class.

III. METHODOLOGY

We firstly examine the malwares in the publicly available malware datasets, Malgenome [29] and Drebin [14]. Secondly, we examine Android applications downloaded from three popular markets, Google Play, SlideMe and AppsApk. Signature-based and dynamic analysis techniques are carried out on both the applications, and the downloaded files by these applications at runtime.

We firstly determine the applications using the Android updating techniques defined in the previous section. Here, only a static analysis is carried out. This initial analysis is called *signature-based analysis*. Then, we carry out a dynamic analysis on all applications in order to detect malwares evaded from signature-based analysis. Since updating techniques could be encrypted in the bytecode, all applications are sent to the dynamic analysis. Our research mainly focuses on finding malicious applications using the updating techniques as an evasive strategy. Please note that, signature-based analysis is only used to find out applications that have explicit signatures for updating as given in Section 2. The most important part of our work is the dynamic analysis. We especially analyse the applications which avoid being identified by the signature-based analysis, and perform malicious activities at runtime. These evaded applications are further explored by machine learning (ML)-based detection system in order to reveal unknown malicious applications as shown in Figure 1. The ML-based detection system works on dynamic features of applications.

A. Signature-based Analysis

This initial analysis classifies applications according to whether they are using the updating techniques or not. We firstly disassembled applications into the *.smali* files using Android apktool [2]. After the disassembling step, we dissected applications whether they have updating features in their code or not. This signature-based analysis brings out potentially dangerous applications which are using upgrading, silent installing and dynamic class loading techniques. We search for keywords related to the API calls defined according to the characteristics of each updating technique given in Section 2. We ensured that an application including the complete signature of any of these three updating techniques in its code is tagged for further analysis. For instance, usage of the *intent.setDataAndType* is not enough to determine the application as suspicious and using upgrading technique. Besides, it should include *Intent Class* and trigger the *startActivity*. We use "startActivity(Landroid/content/Intent", "setDataAndType" and "application/vnd.android.package-archive" signatures in order to detect the usage of the upgrading technique. For the silent installing technique, the signatures "Ljava/lang/Runtime;->exec" and "pm install" are searched for. The "DexClassLoader;->loadClass" signature is considered to indicate the usage of the dynamic class loading technique.

B. Dynamic Analysis

An attacker could conceal his malicious activities by using obfuscation and encryption techniques. Therefore, we develop a dynamic analysis tool in order to overcome the limitations of signature-based analysis. What makes our work unique is to analyse applications in order to monitor their updating behaviours during their execution. To achieve our goal, we use DroidBox[6], one of the mostly used dynamic analysis tools. In order to force applications to update themselves at runtime, we add extra features to DroidBox, and propose a new triggering mechanism called time-based triggering.

Event-based Triggering: Droidbox uses *MonkeyRunner* [11] to generate events in order to analyse application behaviours. However, *MonkeyRunner* is not enough to trigger applications to load the payload at runtime. Besides, some applications wait for some events to occur in order to trigger

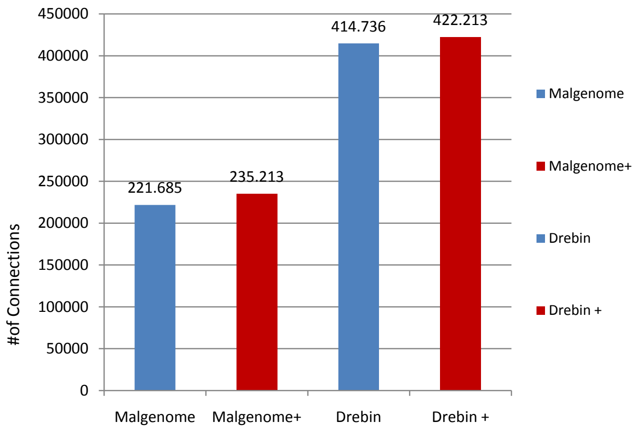


Fig. 2. The number of connections activated by applications in the malware datasets.

updating. In order to overcome this limitation of *MonkeyRunner*, we use *Monkey* [10] which is an UI/Application exerciser. While *MonkeyRunner* generates random events, then sends them to the Android device or emulator, *Monkey* generates UI events by running on an Android device or on an emulator. Thus, we could force the application automatically to click OK button in the pop-up dialogue in order to start downloading new application or loading payload dynamically by using Monkey. We have also added multi-thread ability to the Droidbox. In addition, we have limited dynamic analysis to run only 10 minutes.

Time-based Triggering: Many researchers have pointed out that one of the important weaknesses of dynamic analysis techniques for mobile devices is to inspect applications for a limited period of time. The applications are generally executed for 10 minutes due to efficiency constraints. Therefore, attacker could exploit this weakness by controlling the time that malicious code will be executed. Android uses Java *java.lang.System* package to get the current time. It is observed that 59% of applications from Google Play uses *java.lang.System.currentTimeMillis()* method in their packages. To eliminate this limitation, we add a module to the DroidBox in order to change system date during the execution and set the time forward. A dramatic increase in the number of downloaded files is observed owing to this triggering mechanism in the results. However, the time-based triggering might not be adequate to detect some malicious applications using static, dynamic and hypervisor heuristics [25] in order to evade from dynamic analysis.

In dynamic analysis, we also explore how many malicious files are downloaded and how many connections are opened during runtime. So, we logged all the downloaded files and IP connections. We firstly work on malware datasets, then analyse the applications in the stores in order to see whether they download similar files and/or they connect to the same servers as malwares do. The IP addresses that applications connect to is forwarded to IpVoid [9] in order to check whether these IP addresses are listed in malicious blacklists. IpVoid uses 39 different technologies to decide whether given IP addresses communicate with malicious servers or not.

IV. EVALUATION

A. *Analysis of Malware Datasets*

We firstly analyze the malware datasets publicly available [29], [14]. Furthermore, we investigate the effect of the time-based triggering on the number of downloaded files by malwares during the runtime. The dynamic analysis reveals that the time-based triggering is the most effective method in Malgenome and Drebin datasets. The number of downloaded .apk files is increased by 92% for the Malgenome dataset, 53% for the Drebin dataset with time-based triggering. A relatively small increase in the number of downloaded .dex files is also observed in the results (6% for Malgenome dataset and 28% for Drebin dataset).

In addition, we analyse the impact of time-based triggering on the number of connections malwares make. Figure 2 shows that the number of connections is increased by 6% for the Malgenome dataset, 2% for the Drebin dataset. “+” symbol in the figure indicates time-based triggering. We find that 4 of the connected servers from Malgenome dataset, and 30 of the connected servers from Drebin dataset are listed in malicious domain lists. Moreover a new C&C server, used by 244 malwares belonging to five malware families in the Drebin dataset and 178 malwares in the Malgenome dataset, is discovered. We observe a huge amount of communication between this C&C server and the malicious applications.

Zhou and Jiang [29] divides malwares into four groups according to the techniques they applied to install malwares on mobile phones : repackaging, update attacks, drive-by-downloads and others. They said that 4 malware families performing update attacks are exist in the dataset. These families are BaseBridge, DroidKungFuUpdate, AnserverBot, and Plankton. However, our runtime analysis shows that 5 families (275 applications) from Malgenome datasets download runnable Android applications that are tagged as malicious by the VirusTotal. The results are shown in Table I. Most of the downloaded files are the same whereas they are the member of the same family. For example, “mainmodule.jar” malicious payload are seen 165 times in AnserverBot and 78 times in BaseBridge family. However, we found out that DroidKungFuUpdate and Plankton families do not perform any updating attacks. These families seem not to connect to malicious servers anymore. For example, applications from Plankton family try to reach the following link “http://schemas.android.com/apk/res/com.planktond”, which is no longer accessible. In our analysis, the three additional families found are DroidKungFu1, Droid- KungFu3 and DroidKungFu4. With the help of time-based triggering, we are able to find more updated attacks than the previous analysis [29]. These results emphasize the importance of dynamic analysis in order to detect malwares using update techniques.

B. *Analysis of Application Stores*

We select three popular application stores for analysing malicious applications using update mechanism as shown in Table II. We crawled all free applications from application stores (SlideMe : 1,469 applications, AppsApk : 3,560 applications) between August 2013 and February 2014. We downloaded 20,000 applications randomly from Google Play that represents nearly 2% of the Google Play store. While

TABLE I. THE ATTACK FAMILIES USING UPDATE TECHNIQUES IN THE MALGENOME DATASET

Family	**Number**	**Percentage**
AnserverBot	183	98%
BaseBridge	78	64%
DroidKungFu1	2	6%
DroidKungFu3	11	4%
DroidKungFu4	1	1%
Total	**275**	**22%**

TABLE II. THE RESULTS OF THE SIGNATURE-BASED ANALYSIS

	Google Play	**SlideMe**	**AppsApk**
Silent Installing	21 (0.1%)	1 (0.06%)	15 (0.4%)
Upgrading	1,127 (5.6%)	66 (4.4%)	402 (11.3%)
Dynamic Class Loading	660 (3.3%)	98 (6.6%)	94 (2.6%)
Total Number of Updated Applications	1,808 (9%)	165 (11.2 %)	511 (14.3%)
Total Number of Applications	20,000	1,474	3,563

downloading applications from SlideMe and AppsApk stores was straightforward, we developed a tool which uses Android Market API [3] for downloading applications from Google Play.

Signature-based Analysis: We found that most of the applications especially adwares use dynamic class loading, since it is easily manageable at runtime. For instance, while upgrading needs to make lots of changes on the device, this technique allows users to download new files straightforwardly. We found 3,480 adware applications from Google Play using the dynamic class loading technique. Silent installing is the least used updating techniques among developers, since it requires root privileges to update. Finally, we found that 10% of applications by these three market stores on average are using update techniques. This shows how insecure the application stores are. Please note that the adwares are not included in Table II.

Grace et al. [21] shows 3.90% of 118,000 applications is using code loading techniques. Sebastian Poeplau et al. [26] finds out that 5% of 1632 applications from Google Play using code loading techniques. Both of the results could include adwares since there is no information on their study about adwares. Our analysis detects 19.60% of 25,000 applications from three markets datasets using this updating technique. If we exclude adware applications, this number is decreased down to 3.40%. The usage of update techniques against each market can be seen clearly in Table II. Our results show that there is a substantial increase in the number of applications using the updating techniques, especially dynamic class loading in the last few years. While some developers apply these techniques to overcome the reference size limit, attackers could easily use them in order to download their malicious code.

Dynamic Analysis: Figure 3 shows the percentage of applications using update techniques in the store datasets. We found that 2% of Google Play and 1% of SlideMe datasets evaded signature-based analysis and downloaded runnable applications at runtime. We found 453 applications in the dataset collected from GooglePlay and SlideMe evade the signature-based analysis. However for the AppsApk dataset, the number of applications downloading runnable applications are less than the number of applications using the updating techniques according to the signature-based analysis. One of the reason is that the dynamic analysis is only executed for a limited period of time. Secondly, specific events might not be generated to trigger the update with the dynamic analysis. Moreover, malicious applications could hide themselves with the realization of running on an isolated environment. Attackers commonly use static, dynamic and hypervisor heuristics [25] in order to evade from dynamic analysis. These techniques might be used to detect running environment of the application.

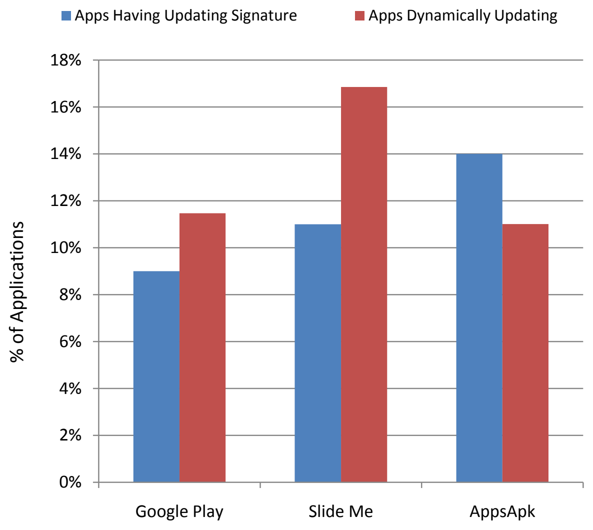
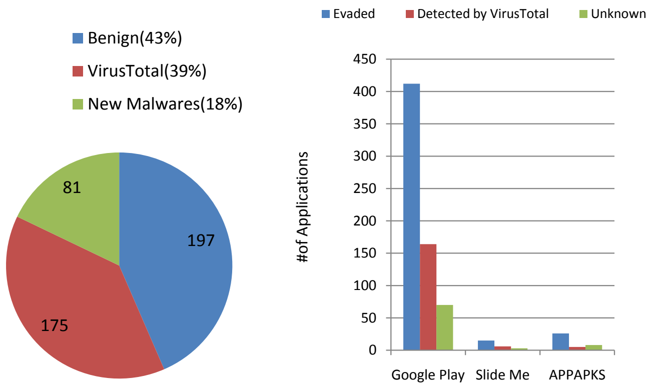


Fig. 3. The percentage of applications using update techniques in the store datasets.



(a) Evaded Apps. (b) Evaded Apps by Market Stores.

Fig. 4. New malwares not detected in the Signature-based Analysis.

We combined the results of signature-based and runtime analysis to search for applications using updating techniques stealthily. We found 453 applications in the dataset collected from the application stores evade the signature-based analysis by using some techniques such as obfuscation, encryption, and the like and, update themselves at runtime. We found out that these samples do not contain any updating signature in the bytecode. However, they could download executable files during runtime. 36% of applications out of these updated applications are detected to be malicious by VirusTotal [13]. The remaining applications are analysed dynamically and some representative features are extracted. The collected features are sent to the ML-based approach proposed in [24] . This approach differentiates malicious applications from benign applications using machine learning (ML) techniques (C4.5, Naive Bayes, Random Forest and SMO). The models work on dynamic features of applications collected using DroidBox. Here, the application is accepted to be malicious if more than the three ML techniques detect the application as malicious. As a result, 81 (18%) new malicious applications are discovered. 70 applications out of 412 applications in Google Play, 6 applications out of 15 applications in SlideMe, 5 applications out of 26 applications in AppsApk are found to perform malicious activities as presented in Figure 4.

We manually analysed newly found malicious applications by our system in order to verify the results of ML-based approach. We deeply analysed malicious applications especially found in Google Play. We observed some interesting results that three applications access to *deviceId, IMEI and IMSI* numbers from the downloaded payload at runtime. One of them has stored IP address in the downloaded payload bytecode and this IP address is in the IpVoid blacklists. Another application tries to execute the following Linux commands for privilege escalation: *chmod, chown and mount*. These malicious applications are not detected by VirusTotal since the malicious payload of these applications are downloaded at runtime.

We also observed that even a download file does not have .dex extension, it might contain a runnable code inside. This could be one of the techniques that attacker uses to hide itself from security mechanisms. Some downloaded .dex files use the following extensions : .epub (1), .data (4), .tmp (15) and .zip (292).

We also investigated the connections that applications made at runtime, and sent the IP addresses they connected to IpVoid. IpVoid tags an IP address as malicious if two or more vendors agree that the given IP address is in the blacklists. Besides, it puts a warning tag if one vendor ensures that given IP address is in the blacklists. We found that 26 malicious from Google Play dataset build connections with the blacklisted IP addresses. Moreover these applications are also found malicious by the dynamic analysis.

C. Interesting Observations

One of the most interesting malwares in our study is using phishing techniques to deceive users. It warns user that application requires Adobe Flash Player in order to run the application. Even if the user approves to install the application, the following error is displayed on the device display "App not installed. This app is not compatible with your phone". However, two malwares are installed with the user approval anyway : ''com.adobe.flash.apk'' and ''ado be.flash.new.apk''. Another interesting application which is downloaded from Google Play (apkv2:air.albinoblacksheep. shoot:1:4.apk) communicates with a C & C server. This application sends the IMEI MD5 hash sum of the device to the server. After successful communication, the server sends message to the client ({''code:200,action:hi''}). Moreover, this application reads four different process information and accessed to system memory information (/proc/meminfo) six times. Furthermore, this application uses the AES algorithm with the key "0123456789abcdef". Only one anti-virus in VirusTotal identifies this application as malware.

V. RELATED WORK

Even though many researchers have been working on mobile malware security, there is no complete solution to this complex problem. Many of the studies focus on the analysis of permissions in order to protect mobile devices against malwares. Kirin [20] proposed an approach which terminates the installation of an application if suspicious permissions are requested by the application. Zhou *et al.* [30] compares the permissions requested by an application with the permissions in the mobile malware samples. Yuan Zhang *et al.* [28] also analyses the permissions in order to identify privacy leakage.

Andromaly [27] employs machine learning techniques in order to differentiate malicious applications from benign ones. The feature set used is obtained by employing dynamic analysis. There are also other proposals based on dynamic analysis, such as AppGuard [15] which uses program traces, and Crowdroid [16] which monitors system calls. TaintDroid [19] monitors privacy sensitive information with taint tracking, MADAM [18] monitors application behaviours both at the user level and at the kernel level.

There are also malware detection techniques based on static analysis available for mobile devices. Chin *et al.* [17] proposed ComDroid in order to detect applications' vulnerabilities by analysing inter-application communications. RiskRanker [21] proposed a two-level analysis. High-risk and medium-risk applications are determined in the first-order analysis, and applications employing using obfuscating, encryption or dynamic class loading techniques are extracted among these risky applications in the second-order analysis. However, RiskRanker only employs static analysis, does not analyse downloaded files at runtime. Grace et al. [22] showed that dynamically code loading is dangerous since an attacker remotely controls the application and injects suspicious payload after installation. Hence malicious applications could easily bypass static analysis techniques by modifying their code at runtime. Sebastian Poeplau *et al.* [26] presented a static analysis tool in order to detect code loading techniques. Furthermore they showed that these code loading techniques introduce vulnerabilities that could be exploited in order to shift a benign application to a malware. Dominik Maier *et al.* [23] showed that a malware can easily bypass VirusTotal scanners. They developed an application which has benign and malicious parts and malicious part is loaded during runtime by using the dynamic code loading technique.

Even though there are static analysis techniques proposed to analyse applications susceptible to malicious code loading in the literature, our work focuses on both signature-based and dynamic analysis of applications. Furthermore, this work presents an analysis of applications employing all update techniques, not only dynamic class loading.

VI. CONCLUSIONS

In this paper, we have done a rigorous analysis on mobile applications which use the updating techniques, upgrading, silent installing and dynamic class loading. We analysed nearly 30,000 applications collected from three different markets and two malware datasets. Our work is the first large-scale analysis (signature-based and dynamic analysis) to uncover malicious applications using updating techniques. After our analysis, 9%

of applications from Google Play have updating signatures that are detected by signature-based analysis. As a result of the runtime analysis, we found that nearly 11% of applications collected from Google Play use Android update mechanism. We found out that nearly 2% of applications downloaded from Google Play evade the signature-based analysis and download runnable applications at runtime. Moreover, 70 new malicious applications from Google Play not detected by VirusTotal and Google Bouncer are discovered as a result of this analysis.

We also investigated the techniques to trigger applications in order to download files, or to fetch code at runtime. The time-based triggering technique is employed to increase the number of updated applications to analyse. It also allows us to find out more updated attacks missed by the previous analysis in the literature [29]. This technique show us how attackers could effectively evade even from dynamic analysis techniques. They could also employ other evasive strategies such as obfuscation and encryption.

To sum up, this study shows the real number of applications using updating techniques, the malicious activities performed by updated applications such as the downloaded file types, the connected IP lists and allows researcher to develop better security solutions. In the future, we are planning to enhance security solutions to detect such update attacks. Moreover, we aim to carry out more investigation on triggering mechanisms in order to make such applications to update themselves.

VII. ACKNOWLEDGEMENTS

This study is supported by the Scientific and Technological Research Council of Turkey (TUBITAK-115E150). We would like to thank TUBITAK for its support. We also would like to thank VirusTotal for their support.

REFERENCES

[1] Amazon. (Visited April 2015) [Online]. Available: https://developer.amazon.com/public/support/faq.

[2] Android Apktool. (Visited April 2015) [Online]. Available: https://code.google.com/p/android-apktool/.

[3] Android Market API. (Visited April 2015) [Online]. Available: http://code.google.com/p/android-market-api.

[4] AppsApk. (Visited April 2015) [Online]. Available: http://www.appsapk.com/android/all-apps/.

[5] DexClassLoader. (Visited April 2015) [Online]. Available: http://android-developers.blogspot.com.tr/2011/07/custom-class-loading-in-dalvik.html.

[6] Droidbox. (Visited April 2015) [Online]. Available: https://code.google.com/p/droidbox/.

[7] Google Play. (Visited April 2015) [Online]. Available: https://play.google.com/store/apps.

[8] Google Play Update Policy. (Visited April 2015) [Online]. Available: https://play.google.com/about/developer-content-policy.html.

[9] IpVoid. (Visited April 2015) [Online]. Available: http://www.ipvoid.com/.

[10] Monkey. (Visited April 2015) [Online]. Available: http://developer.android.com/tools/help/monkey.html.

[11] Monkey Runner. (Visited April 2015) [Online]. Available: http://developer.android.com/tools/help/monkeyrunner-concepts.html.

[12] SlideMe. (Visited April 2015) [Online]. Available: http://slideme.org/.

[13] Virus Total. (Visited April 2015) [Online]. Available: https://www.virustotal.com/.

[14] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[15] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard-real-time policy enforcement for third-party applications. Technical report, 2012.

[16] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[17] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[18] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. Madam: a multi-level anomaly detector for android malware. In *Computer Network Security*, pages 240–253. Springer, 2012.

[19] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014.

[20] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[21] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proc. of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[22] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012.

[23] D. Maier, T. Muller, and M. Protsenko. Divide-and-conquer: Why android malware cannot be stopped. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pages 30–39. IEEE, 2014.

[24] H. B. Ozkan, E. Aydogan, and S. Sen. An ensemble learning approach to mobile malware detection. Technical report, Department of Computer Engineering, Hacettepe University, 2014.

[25] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[26] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proc. of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, volume 14, pages 23–26, 2014.

[27] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. Andromaly: A behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38:161–190, 2012.

[28] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[29] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. *2012 IEEE Symposium on Security and Privacy*, (4):95–109, May 2012.

[30] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, pages 5–8, 2012.