IET Information Security

Research Article

IET Journals
The Institution of Engineering and Technology

Analysis of dynamic code updating in Android with security perspective

ISSN 1751-8709
Received on 21st November 2017
Revised 19th June 2018
Accepted on 1st November 2018
E-First on 20th February 2019
doi: 10.1049/iet-ifs.2018.5316
www.ietdl.org

Ahmet I. Aysan[1], Fatih Sakiz[1], Sevil Sen[1] ✉

[1]*Department of Computer Engineering, Hacettepe University, Ankara, Turkey*

✉ *E-mail: ssen@cs.hacettepe.edu.tr*

Abstract: Attackers have been searching for security vulnerabilities to exploit in Android applications. Such security vulnerabilities include Android applications that could load code at runtime which helps attackers avoid detection by static analysis tools. In this study, an extensive analysis is conducted in order to see how attackers employ updating techniques to exploit such vulnerabilities and to assess the security risks of applications in the marketplace using these techniques. A comprehensive analysis was carried out on nearly 30,000 applications collected from three different Android markets and two malware datasets. Static, dynamic and permission-based analyses were employed in order to monitor malicious activities in such applications, and new malicious applications using updating techniques were discovered in Google Play. The results show that applications employing code updating techniques are on the rise. It is believed that this is the first study of its kind to monitor updating behaviours of applications during their execution. This analysis allows us to deeply analyse suspicious applications and thereby develop better security solutions.

1 Introduction

The Android architecture provides a mechanism for developers to update their applications after their installations completed on the device. The code 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 market stores. Detecting these types of malicious activities is one of the biggest problems that market stores face.

Moreover, these types of applications usually do not follow the updating policy of the application markets. After installation, applications fetch their 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' [1]. However, reality differs from the policy. Even Facebook, one of the most popular applications in Google Play, still updates itself by using its own servers. Furthermore, many markets such as Amazon [2] and SlideMe [3] stores have no policy on updating applications from unknown servers.

In this study, updated applications from three different Android markets were analysed: Google Play [4], SlideMe [3], and AppsApk [5]. Investigations also included malware using code updating techniques in publicly available malware datasets, namely Malgenome [6] and Drebin [7]. Both static and dynamic analyses were carried out to reveal malicious applications using updating techniques in market stores. Suspicious applications were investigated by applying signature-based analysis, and then dynamic analysis techniques were performed on each application in order to reveal malicious applications which hide from static analysis techniques through evasion techniques such as obfuscation, encryption, and other similar means. Furthermore, the permission-based analysis was carried out to explore the behaviours of dynamically loaded code. It was found that some permission is only used by malicious files downloaded at runtime execution. These permissions could be employed as distinguishing features in order to differentiate updated attacks from benign adware.

Mechanisms that trigger malicious applications were also investigated. A new method called time-based triggering is introduced. To the researcher's knowledge, there is no published study in the literature that has focused on triggering mechanisms to reveal applications updating themselves. The time-based triggering techniques have largely increased the number of applications to analyse, which allows the finding of newly updated attacks not revealed through current methods of analysis from existing studies [6]. The results support that triggering is one of the evasive strategies effectively applied by attackers.

The aim of this study is to analyse update attacks extensively and to present their characteristics. The main contributions of this study could be summarised as follows:

- This is the first large-scale analysis (signature-based and dynamic analyses) that uncovers malicious applications using updating techniques [8]. Triggering of updating behaviour of applications is also explored. This study extends this work [8] with permission-based analysis for better profiling of updating attacks. As far as the researchers are aware, this is the first extensive study to analyse permissions from the updating perspective, which could help to detect update attacks.
- New malicious applications using code updating techniques were discovered in the markets. Detailed analyses of these applications are presented. This study established that some such applications, which are not detected by any anti-virus systems at the time of analysis, could be detected by the new versions of some anti-virus systems. Furthermore, they are still in the official market. The analysis helps researchers to see the evolution of update attacks in the market.
- A proof-of-concept malware is developed and uploaded to the official Android market to demonstrate market stores' vulnerability to update attacks.

The remainder of this paper is organised as follows: updating techniques are presented in Section 2. Methodology and triggering mechanisms are introduced in Section 3; analysis results are thoroughly discussed in Section 4. The related work is discussed in Section 5 and the study concludes in Section 6.

2 Android update techniques

Most application stores use packet manager to manage the installation of new versions of applications' or updates.

Fig. 1 *Conceptual schema of the analysis*

Table 1 Updating signatures

Method	Signature
upgrading	startActivity(Landroid/content/Intent)
	setDataAndType
	application/vnd.android.package-archive
silent installing	pm install
	Ljava/lang/Runtime;-> exec
dynamic class loading	DexClassLoader;-> loadClass

Application managers usually check applications as to whether or not they need to install a new package. Typically, Android OS developers employ the updating techniques that follow:

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

Silent installing: This technique is only applicable to rooted devices. Users need to have root privileges in order to perform the installation without approval. Therefore, an attacker has an opportunity to install malicious applications without user approval. In this study, this mechanism is aptly called 'silent updating'. An attacker uses '*pm install*' command in order to start the installation. According to a recent Kaspersky security bulletin [9], the most popular and dangerous Trojans of 2016 employed this technique for installing new apps on devices.

Dynamic class loading: Android applications are originally written in Java and compiled into the .dex file. Android applications have significant flexibility, enabling developers to load applications (.jar and .apk files) from any server at runtime [10]. Since dex files are limited 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 execution by using the *DexClassLoader* class.

3 Methodology

First, the malware samples in publicly available malware datasets, Malgenome [6] and Drebin [7], were examined. Secondly, Android applications downloaded from three popular markets, Google Play, SlideMe, and AppsApk, were then examined. Signature-based and dynamic analysis techniques were conducted on both applications and the downloaded files at runtime.

The first step determined applications using the Android updating techniques, as defined in the previous section. Here, only a static signature-based analysis was carried out. Then, the dynamic analysis was conducted on all applications in order to detect malware that evaded signature-based analysis, since updating techniques could be encrypted in the bytecode. This research mainly focuses on finding malicious applications using updating techniques as an evasive strategy. Please note that the signature-based analysis is only used to find applications that have explicit signatures for updating as given in Section 3. The most significant part of this study is the dynamic analysis. Applications were especially analysed which avoided identification in the signature-based analysis and perform malicious activities at runtime. These evading applications were further explored by a machine learning (ML)-based detection system in order to reveal unknown malicious applications (see Fig. 1). The ML-based detection system works on the dynamic features of applications.

3.1 Signature-based analysis

This initial analysis classifies applications according to whether or not they use the updating techniques. Applications were firstly disassembled into *.smali* files using Android apktool [11]. After the disassembling step, applications were dissected in order to ascertain whether or not they contain updating features in their code.

This signature-based analysis highlights potentially dangerous applications employing upgrading, silent installing, or dynamic class loading techniques. Keywords were searched for related to the application programming interface (API) calls defined according to the characteristics of each updating technique (see Section 2). Applications including the complete signature of any of these three updating techniques in its code were tagged for further analysis. Signatures of each technique are presented in Table 1.

3.2 Dynamic analysis

An attacker could conceal his malicious activities by using obfuscation and encryption techniques. Therefore, a dynamic analysis tool was developed in order to overcome the limitations of the signature-based analysis. What makes this work unique is the analysis of applications in order to monitor their updating behaviours during execution. To achieve this, DroidBox [12], one of the mostly used dynamic analysis tools, was employed. In order to force applications to update at runtime, extra features were added to DroidBox, and a new triggering mechanism called time-based triggering was proposed.

Event-based triggering: DroidBox uses *MonkeyRunner* [13] to generate events in order to analyse application behaviours. However, *MonkeyRunner* alone cannot trigger applications to load the payload at runtime. Besides, some applications wait for certain events in order to trigger updating. To overcome this limitation of *MonkeyRunner*, a user interface/application exerciser called *Monkey* [14] was used. *Monkey* generates streams of simulated user and system events by running on an Android device or emulator. Thus, applications could be automatically forced to click the pop-up dialogue 'OK' button in order to start the new application download or loading a payload dynamically by using Monkey. Multi-thread ability was also added to the DroidBox. In addition, the dynamic analysis was limited to execute for only 10 min.

Time-based triggering: Many researchers have pointed out a significant weakness of dynamic analysis techniques for mobile devices as the limited time period for application inspections. Applications are generally executed for 10 min due to efficiency constraints. Therefore, attackers could exploit this weakness by controlling the time when malicious code will be executed. Android uses Java *java.lang.System* package to obtain the current time. It is observed that 59% of applications from Google Play use the *java.lang.System.currentTimeMillis()* method in their packages. In order to eliminate this limitation, a module was added to the DroidBox in order to change the system date during execution and set the time forward. Test results show a noticeable increase in the number of downloaded files observed owing to this trigger mechanism. However, time-based triggering might not be adequate to detect some malicious applications using static, dynamic and hypervisor heuristics in order to evade detection by dynamic analysis.

In dynamic analysis, the number of malicious files downloaded and how many connections opened during runtime was also explored, and thereby all the downloaded files and internet protocol (IP) connections were logged. Work was first performed on malware datasets, then the applications in stores were analysed in order to see whether or not they downloaded similar files and/or connected to the same servers as the malware. IP addresses that applications connected to were forwarded to IpVoid [15] in order to check whether or not these IP addresses were listed on malicious blacklists. IpVoid uses 39 different technologies to decide whether or not any given IP address communicates with malicious servers.

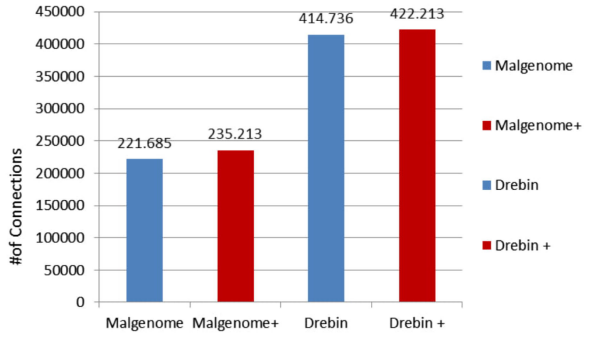


Fig. 2 *Number of connections activated by applications in the malware datasets*

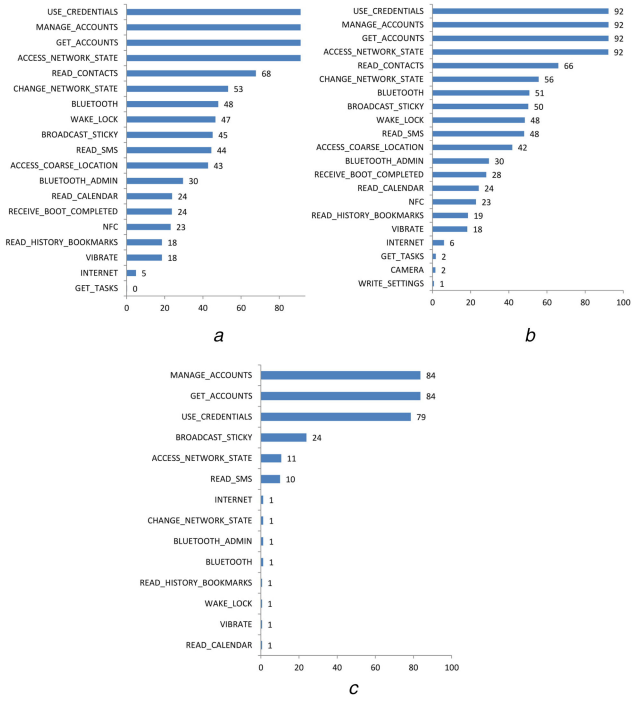


Fig. 3 *Permissions used by downloaded files at runtime in malicious and top applications*

(a) Permissions of downloaded files by Malgenome dataset (876), ***(b)*** Permissions of downloaded files by Drebin dataset (916), ***(c)*** Permissions of downloaded files by Benign dataset (159)

3.3 Permission analysis

Permissions are one of the important security mechanisms in Android. It is known that malware writers mainly request more permissions on average than benign applications do. From the point of view of an updated attack, attackers might well have to request more permissions needed for the code to be loaded dynamically at runtime. Therefore, a permission analysis was carried out for completeness of this study.

In order to detect these types of malicious applications, unused permissions of applications in the manifest file were analysed; then it was investigated as to whether or not these unused permissions were required for file downloads later on. Therefore, suspicious applications could be found which have prepared permissions in the AndroidManifest.xml file for usage in the future. In this study, PScout [16] API-permission list is employed in order to extract the real permission list of applications (the permissions used in the code).

4 Evaluation

4.1 Analysis of malware datasets

The publicly available malware datasets [6, 7] were first analysed. Furthermore, the effect of time-based triggering on the number of files downloaded by malware during the runtime was investigated. Dynamic analysis revealed time-based triggering to be very effective on the Malgenome and Drebin datasets, with the number of downloaded APK files increasing by 92 and, 53%, respectively. A relatively small increase in the number of downloaded DEX files was also observed (6% for Malgenome; 28% for Drebin).

In addition, the impact of time-based triggering on the number of connections made by malware was analysed. Fig. 2 shows that the number of connections increased by 6% for Malgenome and 2% for Drebin. The '+' symbol in Fig. 2 indicates time-based triggering. Results show that four of the connected servers from the Malgenome dataset and 30 from the Drebin dataset were listed as malicious domains. Moreover, a new C&C server, used by 244 malware samples belonging to five malware families in the Drebin dataset and 178 malware samples in the Malgenome dataset, was discovered. A vast amount of communication was observed between this C&C server and the malicious applications.

Zhou and Jiang [6] divided malware into four groups according to the techniques they applied to install malware on mobile phones: repackaging, update attacks, drive-by-downloads, and others. They stated that four malware families performing update attacks exist in the dataset: BaseBridge, DroidKungFuUpdate, AnserverBot, and Plankton. However, runtime analysis from this study's experiment showed that five families (totalling 275 applications) from the Malgenome dataset download runnable Android applications were tagged as malicious by VirusTotal (see Table 2). Most of the downloaded files were the same, even though they are members of the different families. For example, 'mainmodule.jar' malicious payload was seen 165 times in AnserverBot and 78 times in the BaseBridge family. However, there were no update attacks for either the DroidKungFuUpdate or Plankton families, which no longer seem to connect to malicious servers. For example, applications from the Plankton family attempt to reach 'http://schemas.android.com/apk/res/com.planktond', which is no longer accessible. Results of the analysis show the three additional families found were DroidKungFu1, DroidKungFu3, and DroidKungFu4. With the help of time-based triggering, more updated attacks can now be found compared to the previous analysis techniques [6]. These results emphasise the importance of dynamic analysis in order to detect malware using update techniques.

Table 2 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

Permission analysis: Extra permissions to execute extra payload downloaded during runtime were found in 20% of samples of the Malgenome dataset and 5% of samples of the Drebin dataset. These applications were obtained as a result of comparison of the permission list extracted in the static analysis and the permission list of the downloaded files extracted in the runtime analysis. These applications are members of the AnserverBot and BaseBridge families (see Fig. 3). Results were also compared to the permission analysis with the benign applications. For the comparison, 1260 of the top applications were selected randomly from the official store; among them, 159 have executable payload downloaded during runtime. In addition, 2.3% of samples in the top application dataset were found to have extra permissions to execute extra payload downloaded during runtime. After a detailed analysis, it was ensured that these applications use ad libraries.

Fig. 3 shows that MANAGE_ACCOUNTS, GET_ACCOUNTS, USE_CREDENTIALS, and BROADCAST_STICKY permissions are widely used in all datasets. However, malware datasets more commonly used ACCESS_NETWORK_STATE, READ_CONTACTS,

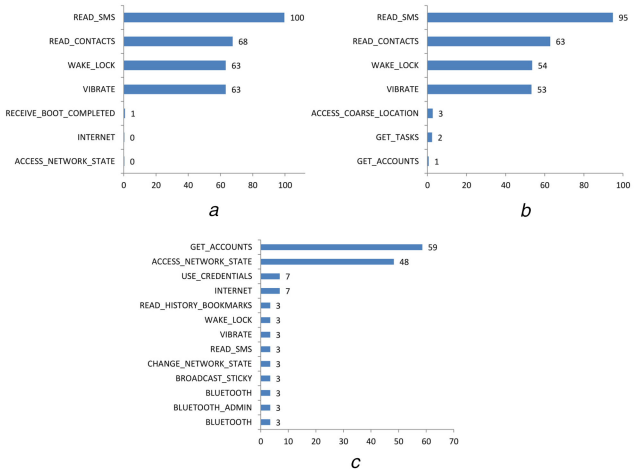


Fig. 4 *Permissions only used by downloaded files at runtime in malicious and top applications*

(a) Permissions of downloaded files by Malgenome dataset (256), ***(b)*** Permissions of downloaded files by Drebin dataset (293), ***(c)*** Permissions of downloaded files by Benign dataset (29)

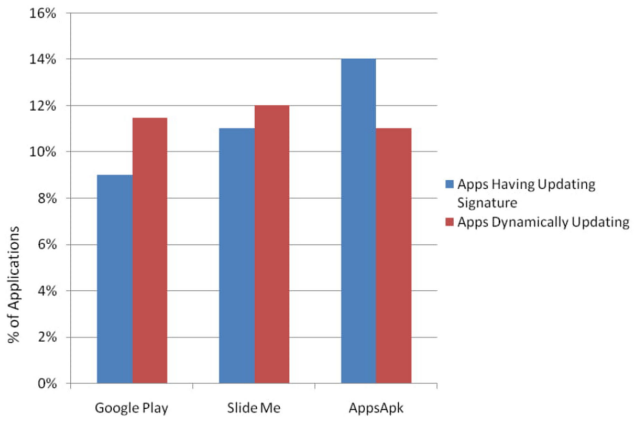


Fig. 5 *Percentage of applications using update techniques in the store datasets*

CHANGE_NETWORK_STATE, BLUETOOTH, WAKE_LOCK, READ_SMS, and ACCESS_COARSE_LOCATION permissions than benign datasets. Fig. 4 shows the permissions only used by downloaded files at runtime. According to the results, over-privileged usage of READ_SMS, READ_CONTACTS, WAKE_LOCK, and VIBRATE could be used as a sign of malicious application in static analysis, which could be added as distinguishing features to detection systems.

4.2 Analysis of application stores

Three popular application stores were selected for analysing malicious applications using the update mechanism (see Table 3). All free applications available from the application stores (SlideMe: 1469 applications, AppsApk: 3560 applications) were crawled between August 2013 and February 2014. A total of 20,000 applications were randomly downloaded from Google Play, representing nearly 1% of the Google Play store at that time. While downloading applications from SlideMe and AppsApk stores was straightforward, a tool was developed which uses Android Market API [17] for downloading applications from the Google Play store.

Even though these applications are generally supported by the old versions of Android due to their collection times, there are still 60% of devices running old versions of Android [18]. Therefore, malware targeting these devices due to having more exploitable vulnerabilities [19] is still being studied. There are recent valuable datasets introduced that covers such malware [18]. These malware samples are still threats to mobile devices, and they are still in the wild as shown in the results (Section 4.3). Moreover, this recent analysis, by analysing the same applications in time, allows us to see how update attacks are evolved in Google Play, which presents valuable finding for researchers.

Table 3 Results of the signature-based analysis

	Google Play	SlideMe	AppsApk
silent installing	21 (0.1%)	1 (0.06%)	15 (0.4%)
upgrading	1127 (5.6%)	66 (4.4%)	402 (11.3%)
dynamic class loading	660 (3.3%)	98 (6.6%)	94 (2.6%)
all updated applications	1808 (9%)	165 (11.2%)	511 (14.3%)
all applications	20,000	1474	3563

Signature-based analysis: Most of the applications found, especially adware, use dynamic class loading since it is easily manageable at runtime. For instance, while upgrading requires making considerable changes on the device, this technique allows users to download new files straightforwardly. A total of 3480 adware applications were found from Google Play using the dynamic class loading technique. Silent installing was the least-used updating technique among developers since it requires root privileges to update. Finally, 10% of applications on average were found by these three market stores to use update techniques; showing how insecure and vulnerable the application stores are. Note: adware are excluded from Table 3.

Grace *et al.* [20] showed that 3.90% of 118,000 applications used code loading techniques, whilst Sebastian Poeplau *et al.* [21] found that 5% of 1632 applications from Google Play used code loading techniques. Both results could have included adware since there was no information on their studies with regard to adware. The current study's analysis detected 19.60% of 25,000 applications from three markets datasets using this updating technique. If adware applications were to be excluded, this number decreases down to just 3.40%. Results show a substantial increase in the number of applications using updating techniques, especially dynamic class loading, over the last few years. While some developers apply these techniques to overcome the reference size limit, attackers could also easily use them in order to download malicious code.

Dynamic analysis: Fig. 5 shows the percentage of applications using update techniques in the store datasets. It was found that 2% of Google Play and 1% of SlideMe datasets evaded signature-based analysis and downloaded runnable applications at runtime. A total of 453 applications from the application stores datasets were found to evade signature-based analysis; however, for the AppsApk dataset, the number of applications downloading runnable applications was less than the number of applications using the updating techniques according to the signature-based analysis. One reason is that dynamic analysis is only executed for a limited period of time. Secondly, specific events might not be generated to trigger the update through dynamic analysis. Moreover, malicious applications could hide with the realisation of running in an isolated environment. Attackers commonly use static, dynamic and hypervisor heuristics in order to evade from the dynamic analysis [22]. These techniques might be used so as to detect the running environment of the application.

The results of signature-based and runtime analysis were combined in order to search for applications using updating techniques stealthily. A total of 453 applications from application store datasets were found to evade signature-based analysis by using techniques such as obfuscation, encryption, and updating themselves at runtime. These samples were found not to contain any updating signature in the bytecode; however, they could download executable files during runtime. Of these updated applications, 36% were detected to be malicious by VirusTotal [23]. The remaining applications were analysed dynamically and some representative features extracted. The collected features were sent to the ML-based approach proposed in [24]. This approach differentiates malicious applications from benign applications using ML techniques (C4.5, Naive Bayes, Random Forest, and sequential minimal optimization (SMO)). The models work on the 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

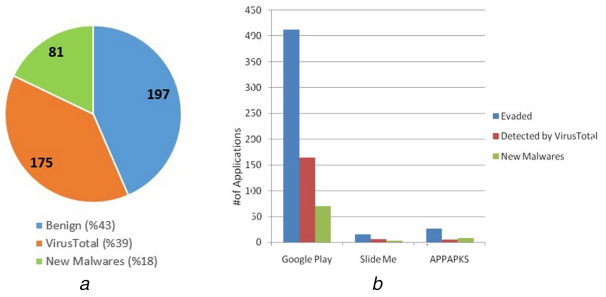


Fig. 6 *New malware not detected in the signature-based analysis*
(a) Evaded Apps, ***(b)*** Evaded Apps by market stores

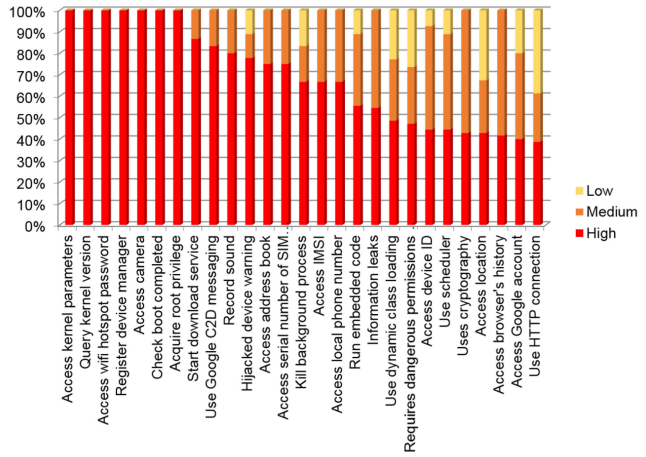


Fig. 7 *Suspicious activities and their percentage distribution into threat levels*

(18%) new malicious applications were discovered. 70 applications out of 412 applications in Google Play, six applications out of 15 applications in SlideMe, and five applications out of 26 applications in AppsApk were found to perform malicious activities (see Fig. 6). The newly found malicious applications were manually analysed in order to verify the results of the ML-based approach. Malicious applications especially found in Google Play were deeply analysed and the analysis results are presented in the next subsection.

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

Connections that applications made at runtime were also investigated, and the IP addresses they connected with were sent to IpVoid. IpVoid tags an IP address as malicious if two or more vendors agree that the given IP address is blacklisted. Additionally, it places a warning tag if one vendor denotes the IP address as being blacklisted. In total, 26 applications from the Google Play dataset were found to build connections with blacklisted IP addresses.

4.3 Newly discovered malware

In this section, the aforementioned 70 newly discovered malicious applications from Google Play detected through ML-based detection were deeply analysed to confirm their maliciousness. Both static and dynamic analyses were conducted on these applications, and 27 suspicious activities (see Fig. 7) were monitored. According to the suspicious activities they carried out, the applications were grouped into three threat levels: low, medium, and high by using expectation–maximisation clustering. The suspicious activities and their percentage distribution into threat levels are shown in Fig. 7. For instance, it is shown that all applications in the high-level threat category (100%) access kernel parameters. Such suspicious activities are summarised as follows:

First of all, the updating behaviour of the applications was analysed. It was observed that all of them tried to load at least one class through DexClassLoader. In addition, each application tried to install a seemingly new version of itself within the first 2 s after they started running. Moreover, most of the applications (~88.5%) requested HTTP connections to various addresses.

The results of dynamic analysis also provide evidence of the sneaky nature of malicious applications. For example, many applications (~35.7%) access the file that stores wireless hotspot passwords (/data/misc/wifi/wpa supplicant.conf). Other important targets of malicious applications are operating system and memory. They could access kernel parameters (/proc/cmdline), which is read by init process after the kernel boots in order to set system properties accordingly (~35.7%), or query the version of the running kernel (/proc/version) (~37.14%). A few of them check if the boot process is completed or not. Almost all applications access the file (/proc/meminfo) which provides detailed information of the RAM usage of the system and few of them (~8.57%) kill background processes.

Some applications try to access information that uniquely identifies the user and/or the device in the network. For example, some of them query the device ID (~38.5%), SIM card serial number (5.7%), local phone number (~8.57%) and international mobile subscriber identity (~4.3%), which is a unique international identification of mobile terminals and users. While some applications (~15.7%) leak such information via the network and only one application asks for root privilege.

An interesting service used by some of the applications (10%) is Google cloud-to-device (C2D) messaging service, which enables developers to communicate with installed applications via messages. The messages sent from an app server are distributed to applications installed on Android devices through a connection server owned by Google itself. Even though the application is not running at the time of message delivery, it will be invoked since the messages are delivered by the Android OS. According to SecureList [25], attackers could exploit this service and turn it into a C & C channel. Moreover, both antivirus software and the users are unable to block this message delivery since it is considered a system activity performed by the OS. The service could be used to disseminate links and/or commands in order to perform malicious activities. Even though the messaging service has been shut down as of 20 October 2015, and replaced by Google Cloud Messaging and firebase cloud messaging (FCM), both services could be misused in a similar manner, according to the National Institute of Standards and Technology Mobile Threat Catalogue [26]. An analysis of a malware which utilises FCM to communicate with a C & C server can be found in [27].

This study has shown that analyses of malicious applications could cause unexpected behaviour during dynamic analysis. The malicious applications could try taking full control of the emulator and if successful, they could circumvent normal operations necessary for proper dynamic analysis. For example, during analyses of nine applications, the emulator gave the following warning message: 'WARNING: Device: This app might have hijacked the device!'.

These newly discovered malicious applications were also tested in two online tools: Akana from MobiSec Lab [28] and NVISO ApkScan [29]. They both perform static and dynamic analysis of .apk files, and provide detailed reports about the applications, in addition to utilising VirusTotal [3]. NVISO ApkScan's results showed that 15 out of 70 applications showed 'suspicious activity' and one application was identified as 'confirmed malicious'. On the other hand, Akana has three threat levels: low, medium, and high. For each application, their threat levels are produced along with the probability of occurrence. Akana's results showed that 66 out of 70 applications had a significant probability of low-level threat occurrence while three applications had a significant probability of medium-level threat occurrence and one application had a significant probability of high-level threat occurrence. On the other hand, this current study's results showed that 26 out of 70 applications fell into the high-level threat category, 13 as medium-level and 31 as low-level threat categories. Detailed results of the analysis are shown in Fig. 8. Since the results of dynamic analysis

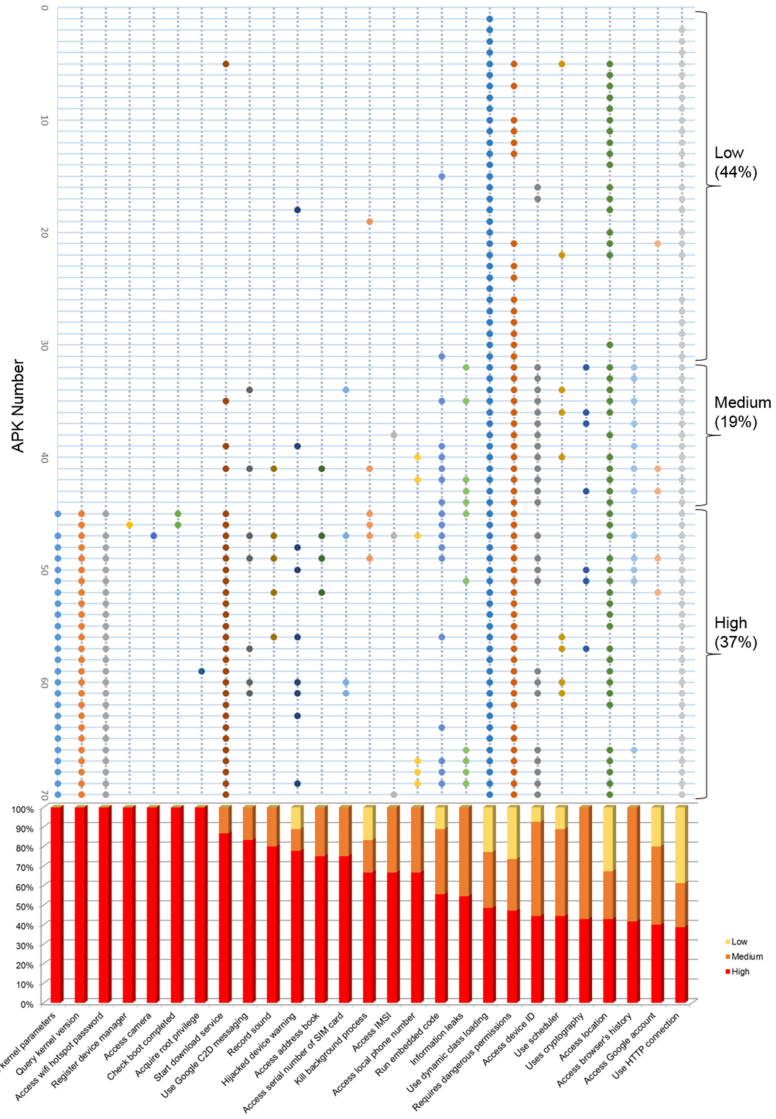
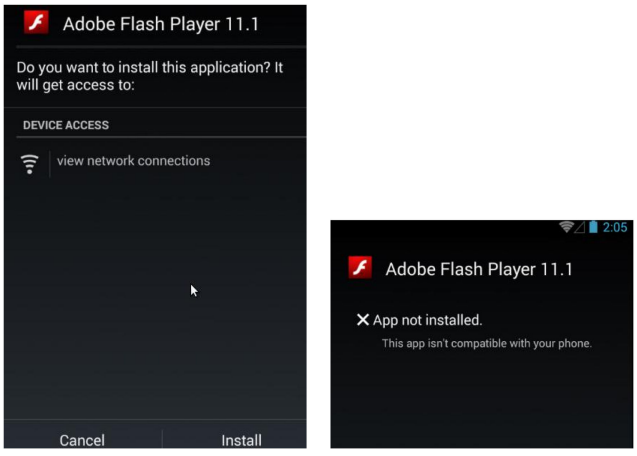


Fig. 8 *Newly discovered malicious applications with respect to considered parameters along with their percentage distributions*



a *b*

Fig. 9 *Malware needing so-called adobe flash player*
(a) Installation screen, ***(b)*** Not compatible screen

depend on inputs generated which could trigger a myriad number of activities, different tools could yield different results as shown here. For example, by utilising DroidBot [30] in the detailed analysis more malicious activities were observed to be triggered.

Interesting applications were observed during the analysis. Some applications (~4.3%) hid their launcher icons in order to evade detection from users. If the activities/permissions of an application do not conform to its design goal(s), the inconsistency could discredit the application even though it performs unsuspicious activities and therefore requires extra attention. For example, one interesting application which is a simple face-swapping application gets busy with encryption and sending/receiving information to three different IP addresses within the first 40 s after it starts running without any user input. After watching closely, it was revealed that the information sent by the application includes the international mobile equipment identity (IMEI) number of the device.

One of the most interesting malware found in this current study uses phishing techniques to deceive users. It warns the user that the application requires Adobe Flash Player in order to run the application. Even if the user approves the installation, the following error is displayed on the device display 'App not installed. This App is not compatible with your phone' as shown in Fig. 9. However, two malware samples were installed with the users' unbeknown approval: 'com.adobe.flash.apk' and 'adobe.flash.new.apk'.

Another interesting application 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 a message to the client ({'code:200,action:hi'}). Moreover, this application reads four different process information and access to system memory information (/proc/meminfo) six times using the advanced encryption standard (AES) algorithm with the key '0123456789abcdef'. Only one anti-virus in VirusTotal identified this application as malware.

Table 4 Updated attacks in Google Play (2017)

Applications at Google Play	Detected by our approach				Detected by VirusTotal
	High	Medium	Low	Total	
removed	8	5	5	18	9
exist with same version	9	3	12	24	1
exist with updated version	2	4	3	9	3
exist with unknown version	7	1	11	19	3
Total	26	13	31	70	16

Table 5 Effects of novel features on new malware

Family	Family size	SAFEDroid (code & API), %	SAFEDroid (all), %	SAFEDroid (all) with new features, %
BaseBridge	22	63.6	72.7	77.3
DroidKungFu	193	94.8	95.3	95.3
Plankton	614	2.1	14.5	17.9
DREBIN	4432	70.2	61.86	71.6

It should be noted that at the time of the initial analyses, 70 new applications were found that had not been detected by VirusTotal. However, a recent submission of these applications to VirusTotal showed that some of these applications are now identified as malware by at least one antivirus. While some of them are removed from Google Play in time, the remaining are still as yet undetected by Google Bouncer as shown in Table 4. Please note that versions of some applications could not be checked due to their unavailability in our country anymore. This table shows how attacks are evolved at Google Play in time. It also underlines that how evasive update attacks can be.

4.4 Discussion

In this study, a detailed analysis of benign and malicious software that updates itself has been carried out. Furthermore, malicious software that is not detected by security mechanisms on Google Play has been analysed in detail. As a result of these analyses, some characteristics of update attacks are obtained. For example, permission-based analysis has shown that some over-demanded permissions in the manifest file are frequently used by malicious payload downloaded at runtime. It is believed that the results of this analysis will be useful for malware analysts. It will also help in the development of malware detection systems. To illustrate this, some of the features obtained from this analysis are added to an existing static analysis-based detection system.

A recent detection system based on structural features called SAFEDroid is employed as the base system. In that study [31], different classifiers were trained using different combinations of features and the results compared. The results show that the combination of code-based features and API calls produces the lowest error rate. The performance of this combination is quite close to the combination of all features. Hence, both combinations were evaluated on the new malware. The MalGenome dataset [6] was used for training and validation. The Drebin dataset [7], which is a larger dataset than MalGenome [6], was used for evaluation. It should be noted that all applications that exist in Malgenome were removed from the test dataset before the evaluation took place. Hence the results show the performance of the system on new malware families and new variants of existing malware.

The distinctive permissions obtained from Figs. 3 and 4 are added to the SAFEDroid system. The same experimental set used by SAFEDroid was employed. Since newly added features belong to more than one feature groups (manifest-based and code-based), all features of SAFEDroid plus these new features are used for training (Table 5). The new detection system shows a similar performance on the Malgenome dataset with a small decrease in the false positive rate (0.7%). However, the results show that new features show a noticeable positive effect on detecting new variants of update attacks and new malware. The detection rate was increased approximately by 10%. When the newly added features are evaluated based on the information gain method, some of these new features (such as the READ_SMS permission in the manifest file, the READ_SMS and ACCESS_COARSE_LOCATION permissions used in the code) are observed to be very highly effective.

This proof-of-concept experiment shows the analysis carried out in this study could be useful for detecting update attacks. Please note that different trade-offs between detection and the false positive rate could be discovered by using different combinations of features. Furthermore, the detection system could be improved by adding dynamic features of update attacks in the future.

4.5 Security against update attacks in application stores

In this study, a proof-of-concept update attack is developed. A repackaged space game application called Spicy Space Defender was developed and successfully submitted to the official Android market. In the game, the ships aim to travel as long as they can among enemy ships in space. In order to bypass Google Play's security mechanisms, a clean version of the game application was first uploaded; then its infected version, which downloads additional malicious code after installation on the device, was upload. According to the recent Kaspersky's security bulletin [9], it has become one of the techniques that attackers employ in order to evade the current market's security mechanisms. The attackers generally upload a clean app at first, then provide a few clean updates, and finally upload an infected version.

The malware uses dynamic class loading technique in order to load malicious payload at runtime. When a game player travels for a particular distance in space, the dynamic code is loaded. So, it is difficult to trigger the code loading by using input generation tools based on a random exploration strategy, which is the most frequently used tools to test Android apps [32]. Furthermore, the server name that the code is going to be downloaded from, the name of the package to be downloaded, and the class name containing the malicious code in the package is embedded within an image in the application package. The keywords dalvix.system.DexClassLoader and loadClass are also embedded in the figure. Therefore, this study was able to invoke these suspicious methods by using reflection. Reflection is also employed in other parts of the code in order to prevent the code from initiating the download from being too obvious. Before uploading the application to Google Play, the application is scanned on VirusTotal to ensure it was undetectable by any antivirus system. The application was uploaded to the market in February 2017 and has remained there since.

The malicious code downloaded simply sends packets to a victim whose IP address was also downloaded from the server. The number of packets is a parameter controlled by the server. The time that the packets are to be sent to the victim node could also be specified. Hence, a distributed denial-of-service attack (DDoS) could be implemented through a lot of users downloading such an application. It has been reported that Google Play even has malicious applications that have been installed more than 100,000 times [9].

This proof-of-concept application has shown that a malicious application, by using reflection and dynamic code loading techniques, could still bypass security mechanisms in the markets and become successfully uploaded. The markets are as yet largely ineffective against such evasion techniques and detecting such updated attacks. Even though Google Bouncer is known to perform dynamic analysis, it did not connect to the server used in this experiment to get the package before approving the application. Please note that for ethical reasons, the malicious code in the server is replaced not to harm users/devices.

5 Related work

Even though many researchers have worked on mobile malware security, there is no complete solution to this complex problem. Many studies have focused on the analysis of permissions in order to protect mobile devices against malware. Kirin [33] proposed an approach which terminates the installation of an application if suspicious permissions are requested by the application. Zhou *et al.* [34] compare the permissions requested by an application with the permissions in the mobile malware samples. Yuan Zhang *et al.* [35] also analyses the permissions in order to identify privacy leakage. Sen *et al.* [31] take into consideration also the number of used/unused dangerous permissions in the code in order to detect malware.

Andromaly [36] employs ML techniques in order to differentiate malicious applications from the benign. The feature set used was obtained by employing a dynamic analysis. There are also other proposals based on dynamic analysis, such as AppGuard [37] which uses program traces, Crowdroid [38] which monitors system calls, TaintDroid [39] which monitors privacy sensitive information with taint tracking, and MADAM [40] which monitors application behaviours both at the user and kernel levels.

There are also malware detection techniques based on static analysis available for mobile devices. Chin *et al.* [41] proposed ComDroid in order to detect applications' vulnerabilities by analysing inter-application communications. RiskRanker [20] proposed a two-level analysis with high-risk and medium-risk applications determined in the first-order analysis, and applications employing obfuscating, encryption or dynamic class loading techniques extracted among these risky applications in the second-order analysis. However, RiskRanker only employs static analysis and does not analyse downloaded files at runtime.

Grace *et al.* [42] showed that dynamic code loading is dangerous since an attacker can remotely control the application and inject suspicious payload after installation. Hence malicious applications could easily bypass static analysis techniques by modifying code at runtime. Sebastian Poeplau *et al.* [21] 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.

Maier *et al.* [43] showed that malware can easily bypass VirusTotal scanners by developing an application with both benign and malicious parts, with the malicious part loaded at runtime using the dynamic code loading technique. Xue *et al.* [44] also showed that benign code could evolve into more evasive malware by using dynamic code loading. While it has been shown that malicious applications often make use of dynamic code loading [45], this conflicts with another recent extensive analysis [46]. According to the analysis of one-million apps submitted to Andrubis [46], dynamic code loading was not seen as an indicator of malicious behaviour any more due to its rising popularity among goodware.

There are also proposals for protection mechanisms against code injection attacks, which exploit vulnerabilities introduced by dynamic code loading. Grab'n Run [47] proposed a code verification protocol and introduce a library for secure implementation of dynamic code loading. StaDyna [48] proposed an approach which expands the method call graph of an application by capturing additional codes loaded at runtime through dynamic code loading and reflection. However, these models are triggered manually and are therefore unsuitable for automatic analysis. They extended their study by proposing StaDART, which utilises ArtDroid in order to avoid modifications to the Android framework, unlike StaDyna [49]. Furthermore, DroidBot [30] is employed for triggering malicious activities.

To the best of the researchers' knowledge, the only work on detecting updated attacks was presented recently by Mercaldo *et al.* [50]. Mercaldo *et al.* identified update attacks by analysing four malware families, and localises the portion of code that implements downloading by using formal methods.

Even though there has been limited research on statically analysing malware using dynamic class loading [20, 21, 42], this current study also applied dynamic analysis techniques and permission-based analysis in order to investigate updating applications evading static analysis. Furthermore, all update techniques are explored, not just dynamic class loading. Moreover, new update attacks were found in the wild and analysed in this study.

6 Conclusions

This study presents an extensive study of dynamic code updating in Android. Nearly 30,000 applications collected from three different markets and two malware datasets were deeply analysed with both static and dynamic analyses performed. Permission analysis, which analyses over-privileged permissions in order for use in downloaded code at runtime, was also conducted. This first time permission analysis shows that some dangerous permission is requested only for use by a downloaded code of malicious applications.

This work has been the first large-scale analysis to uncover malicious applications using updating techniques on Android. As a result of static and dynamic analyses, suspicious applications have been extracted. Even though these applications do not have updating signatures in their code, they are able to load malicious code at runtime. When these applications are fed into the malware classifier, some were found to be malicious. To confirm their maliciousness, these malicious applications were then deeply analysed. Analysis showed that all applications fell into one of three threat categories (low, medium, or high). It was observed that some were detected as malicious by some commercial antiviruses over time; however, others still remain undetected in the official market. The proof-of-concept update attack was also successfully uploaded to Google Play.

To summarise, this study has extensively analysed code updating applications. Both malicious and benign applications were taken into consideration, and the important characteristics of both obtained. The authors believe that this analysis will help other researchers to develop solutions to address update attacks, which are shown to be one of the biggest security threats that Android faces with.

7 Acknowledgments

This study was supported by the Scientific and Technological Research Council of Turkey (TUBITAK-115E150). The authors would like to thank TUBITAK for its support and also Yilmaz Degirmenci for his help in developing the proof-of-concept update attack application.

8 References

[1] Google Play Update Policy. Available at https://play.google.com/about/developer-content-policy.html, accessed July 2017
[2] Amazon. Available at https://developer.amazon.com/public/support/faq, accessed April 2015
[3] SlideMe. Available at http://slideme.org/, accessed July 2017
[4] Google Play. Available at https://play.google.com/store/apps, accessed July 2017
[5] AppsApk. Available at http://www.appsapk.com/android/all-apps/, accessed July 2017
[6] Zhou, Y., Jiang, X.: 'Dissecting android malware: characterization and evolution'. 2012 IEEE Symp. on Security and Privacy, San Francisco, CA, USA, May 2012, no. 4, pp. 95–109
[7] Arp, D., Spreitzenbarth, M., Hübner, M., *et al.*: 'DREBIN: effective and explainable detection of android malware in your pocket'. Proc. ISOC Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, 2014

[8] Aysan, A.I., Sen, S.: 'Do you want to install an update of this application? A rigorous analysis of updated android applications'. 2015 IEEE 2nd Int. Conf. on Cyber Security and Cloud Computing (CSCloud), New York, NY, USA, 2015, pp. 181–186

[9] K. Lab.: 'Kaspersky security bulletin 2016', 2016. Available at https://securelist.com/kasperskysecurity-bulletin-2016-executive-summary/76858/

[10] DexClassLoader. Available at http://androiddevelopers.blogspot.com.tr/2011/07/custom-class-loading-in-dalvik.html, accessed July 2017

[11] Android Apktool. Available at https://code.google.com/p/android-apktool/, accessed July 2017

[12] Droidbox. Available at https://code.google.com/p/droidbox/, accessed July 2017

[13] Monkey Runner. Available at http://developer.android.com/tools/help/monkeyrunner-concepts.html, accessed July 2017

[14] Monkey. Available at http://developer.android.com/tools/help/monkey.html, accessed July 2017

[15] IpVoid. Available at http://www.ipvoid.com/, accessed July 2017

[16] Au, K.W.Y., Zhou, Y.F., Huang, Z., *et al.*: 'PScout: analyzing the android permission specification'. Proc. of the 2012 ACM Conf. on Computer and Communications Security, Raleigh, NC, USA, 2012, pp. 217–228

[17] Android Market API. Available at http://code.google.com/p/android-market-api, accessed July 2017

[18] Wei, F., Li, Y., Roy, S., *et al.*: 'Deep ground truth analysis of current android malware'. Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment, Bonn, Germany, 2017, pp. 252–276

[19] Buchka, N.B., Kuzin, M.: 'Attack on Zygote: a new twist in the evolution of mobile threats'. Available at https://securelist.com/attack-on-zygote-a-new-twist-in-the-evolution-of-mobilethreats/74032/, accessed November 2017

[20] Grace, M., Zhou, Y., Zhang, Q., *et al.*: 'Riskranker: scalable and accurate zero-day android malware detection'. Proc. 10th Int. Conf. on Mobile Systems, Applications, and Services, Ambleside, United Kingdom, 2012, pp. 281–294

[21] Poeplau, S., Fratantonio, Y., Bianchi, A., *et al.*: 'Execute this! Analyzing unsafe and malicious dynamic code loading in android applications'. Proc. 20th Annual Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, 2014, vol. 14, pp. 23–26

[22] Petsas, T., Voyatzis, G., Athanasopoulos, E., *et al.*: 'Rage against the virtual machine: hindering dynamic analysis of android malware'. Proc. Seventh European Workshop on System Security, Amsterdam, Netherlands, 2014, p. 5

[23] Virus Total. Available at https://www.virustotal.com/, accessed July 2017

[24] Ozkan, H.B., Aydogan, E., Sen, S.: 'An ensemble learning approach to mobile malware detection'. Technical report, 2014

[25] Kaspersky SecureList, GCM in Malicious Attachments. Available at https://securelist.com/gcm-in-malicious-attachments/57471/, accessed September 2017

[26] Command-and-control messaging evades traffic analysis. Available at https://pages.nist.gov/mobile-threatcatalogue/application-threats/APP-29.html, accessed November 2017

[27] Stefanko, L.: 'Turn the light on and give me your passwords!'. Available at https://www.welivesecurity.com/2017/04/19/turn-lightgive-passwords/, accessed November 2017

[28] Akana, MobiSec Lab. Available at http://www.mobiseclab.org/akana/Intro.html, accessed September 2017

[29] ApkScan, NVISO. Available at https://apkscan.nviso.be/, accessed September 2017

[30] Li, Y., Yang, Z., Guo, Y., *et al.*: 'DroidBot: a lightweight UI-guided test input generator for android'. Proc. 39th Int. Conf. on Software Engineering Companion, Buenos Aires, Argentina, 2017, pp. 23–26

[31] Sen, S., Aysan, A.I., Clark, J.A.: 'SAFEDroid: using structural features for detecting android malwares'. Proc. 13th EAI Int. Conf. on Security and Privacy in Communication Networks (SECURECOMM 2017), Niagara Falls, Canada, 2017 (to appear)

[32] Choudhary, S.R., Gorla, A., Orso, A.: 'Automated test input generation for android: are we there yet?(e)'. 2015 30th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE), Washington, DC, USA, 2015, pp. 429–440

[33] Enck, W., Ongtang, M., McDaniel, P.: 'On lightweight mobile phone application certification'. Proc. 16th ACM Conf. on Computer and Communications Security, Chicago, IL, USA, 2009, pp. 235–245

[34] Zhou, Y., Wang, Z., Zhou, W., *et al.*: 'Hey, you, get off of my market: detecting malicious apps in official and alternative android markets'. Proc. 19th Annual Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, 2012, pp. 5–8

[35] Zhang, Y., Yang, M., Xu, B., *et al.*: 'Vetting undesirable behaviors in android apps with permission use analysis'. Proc. 2013 ACM SIGSAC Conf. on Computer & Communications Security, Berlin, Germany, 2013, pp. 611–622

[36] Shabtai, A., Kanonov, U., Elovici, Y., *et al.*: 'Andromaly: a behavioral malware detection framework for android devices', *J. Intell. Inf. Syst.*, 2012, **38**, pp. 161–190

[37] Backes, M., Gerling, S., Hammer, C., *et al.*: 'AppGuard-real-time policy enforcement for third-party applications'. Technical report, 2012

[38] Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: 'Crowdroid: behavior-based malware detection system for android'. Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Chicago, IL, USA, 2011, pp. 15–26

[39] Enck, W., Gilbert, P., Chun, B.-G., *et al.*: 'TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones', *Commun. ACM*, 2014, **57**, (3), pp. 99–106

[40] Dini, G., Martinelli, F., Saracino, A., *et al.*: 'MADAM: a multi-level anomaly detector for android malware'. Computer Network Security, St. Petersburg, Russia, 2012, pp. 240–253

[41] Chin, E., Felt, A.P., Greenwood, K., *et al.*: 'Analyzing inter-application communication in android'. Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services, Bethesda, MD, USA, 2011, pp. 239–252

[42] Grace, M.C., Zhou, W., Jiang, X., *et al.*: 'Unsafe exposure analysis of mobile in-app advertisements'. Proc. Fifth ACM Conf. on Security and Privacy in Wireless and Mobile Networks, Amsterdam, Netherlands, 2012, pp. 101–112

[43] Maier, D., Muller, T., Protsenko, M.: 'Divide-and-conquer: why android malware cannot be stopped'. 2014 Ninth Int. Conf. on Availability, Reliability and Security (ARES), Fribourg, Switzerland, 2014, pp. 30–39

[44] Xue, Y., Meng, G., Liu, Y., *et al.*: 'Auditing antimalware tools by evolving android malware and dynamic loading technique', *IEEE Trans. Inf. Forensics Sec.*, 2017, **12**, (7), pp. 1529–1544

[45] Maier, D., Protsenko, M., Müller, T.: 'A game of droid and mouse: The threat of split-personality malware on android', *Comput. Secur.*, 2015, **54**, pp. 2–15

[46] Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., *et al.*: 'Andrubis-1,000,000 apps later: a view on current android malware behaviors'. Proc. 3rd Int. Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), Wroclaw, Poland, 2014

[47] Falsina, L., Fratantonio, Y., Zanero, S., *et al.*: 'Grab'n run: secure and practical dynamic code loading for android applications'. Proc. 31st Annual Computer Security Applications Conf., Los Angeles, CA, USA, 2015, pp. 201–210

[48] Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., *et al.*: 'Stadyna: addressing the problem of dynamic code updates in the security analysis of android applications'. Proc. 5th ACM Conf. on Data and Application Security and Privacy, San Antonio, TX, USA, 2015, pp. 37–48

[49] Ahmad, M.: 'Mobile application security in the presence of dynamic code updates'. PhD thesis, University of Trento, 2017

[50] Mercaldo, F., Nardone, V., Santone, A., *et al.*: 'Download malware? No, thanks. How formal methods can block update attacks'. 2016 IEEE/ACM 4th FME Workshop on Formal Methods in Software Engineering (FormaliSE), Austin, TX, USA, 2016, pp. 22–28