Securing mobile devices: malware mitigation methods

Malware on mobile handsets has always been a point of concern for its users. With the widespreadadoption of smartphones and tablets and the emergence of centralized application markets it startedto represent a significant threat. This situation has led to the development of defence methods forsecuring mobile devices coming from operating system developers, antivirus vendors and securityresearchers. In this paper we focus on the solutions proposed by security researchers which includeboth methods inherited from malware detection on personal computers and new methods specific tomobile device environment. This paper gives an overview of the history and development of mobilemalware and provides a survey of the methods proposed for malware mitigation on mobile devicesin the last years.

Keywords: mobile devices, malware, malware mitigation

1 Introduction

Mobile devices became ubiquitous in the last years. Modern tablets and smartphones provide many useful services such as internet browsing, maps, social network clients, internet banking in addition to standard mobile functionality including phone calls, SMS and Bluetooth. The data used and stored in these services is often highly sensitive and therefore desired by the attackers.

If we take a look at the mobile threats history, it can be easily seen that the attackers always preferred the most popular mobile OS. The first mobile worm Cabir appeared in June 2004. It was targeting Symbian OS. Even though Cabir was not initially designed to harm users (it was spreading via Bluetooth transmitting a special .sis file), later it was used for spreading various malware, such as Pbstealer, which was stealing contacts from phone books. According to [1], the main types of mobile malware in 2004-2006 were trojans designed for financial gain, e.g. Mosquit sending SMS to premium phone numbers, and vandal trojans designed to make harm or disable the device, e.g. Skuller. Other functionalities implemented in malware of this period included infecting files, enabling remote control of the smartphone, disabling system and third-party applications, installing other malicious programs, blocking memory cards, stealing data. The malware had the capability to self-propagate via Bluetooth, MMS and, later, via removable media. As reported in [2] the new malware appeared in 2007-2009 which was aimed at damaging user data, e.g. Delcon, Deladdr; disabling operating system security mechanisms; calling paid services, e.g. Smofree, Pornidal. Polymorphic mobile worm PMCryptic was encountered in 2008. Till 2009 Symbian was the most affected by mobile malware.

After that Java 2 Platform, Micro Edition became the most popular target for mobile malware attacks. Interestingly, though, no new techniques have been encountered in the development of mobile malware. The most prevalent type of malware was SMS-trojan sending messages to premium-rate numbers. The numbers and the text to send were under control of remote malicious server in some versions of these

trojans, e.g. Sejweek. Other threats included trojans stealing online banking access data and authentication codes for online banking transactions, spyware trojans stealing other privacy-sensitive information, trojans making phone calls to premium-rate numbers. The first mobile malware for Android and iOS platforms was also encountered in this period [3].

Android platform gained its popularity in 2011 and since then most of the discovered mobile malware was aimed at this system. The attackers are mainly interested in stealing data (including financial), using premium-rate services and establishing control channels [4, 5, 6]. In 2013 the growth in the number of mobile banking trojans was reported [6].

First mobile malware was self-spreading via Bluetooth, MMS, a vulnerability in the system or it was downloaded by a user from the Internet resource usually after receiving some kind of advertisement. When the centralized application markets emerged, the attackers got even simpler way to spread the mobile malware. Official marketplaces provide more or less thorough review of all the submitted applications, e.g. Android market uses scanning tool Google Bouncer. More proper review provided by application stores in iOS and Windows Phone considerably improves the situation with exposing dangerous applications to users. However, several malware samples were found even in the official Apple App Store, e.g. Find and Call app.

Several marketplaces exist for each mobile platform: official one and some alternative marketplaces. Infection rate in alternative marketplaces is an order of magnitude higher than in the official marketplace, e.g. about 0.02% of apps in the official Android Market (in present Google Play Store) and 0.20% to 0.47% of apps in alternative marketplaces are revealed as malicious[7].

We survey the defense methods proposed for the mitigation of mobile malware threats in this paper. It extends paper [8] which surveys mobile malware detection methods with some new detection approaches. We also considered a separate group of mobile malware prevention methods in this paper. Most of the reviewed methods are intended for the Android OS as the vast majority of malware targets this platform.

2 Mobile malware specifics

When the first mobile malware emerged, malware detection methods, historically developed for desktop computers first, had to adapt. Desktop computers and mobile devices have similar hardware and software running inside. Therefore, security methods for computers and smartphones have a lot in common; however, there are some specific aspects of mobile malware detection that have to be taken into account. Ramu compares and contrasts the aspects of mobile-specific and Desktop PC security in [9].

One of the main differences is that smartphones have rather limited resources: their computational power and memory capacity is usually much smaller compared to Desktop PCs. Some resource-intensive software applications that run on Desktop PCs (including anti-malware applications) may not run on mobile devices due to these constraints.

Mobile devices have communication methods specific to them. Malware in mobile networks can propagate using SMS, MMS, Bluetooth in addition to spreading through traditional IP-based applications and e-mail like in Desktop PCs. The messaging services are used in payment systems and in advertising, therefore they can also be used in making money for the attackers.

Other specific aspects Ramu mentions is the presence of mobile network environment and the difference in user interface – as mobile devices' screens are rather small, some standard security mechanisms like indicators in browsers, CAPTCHA are not applicable to them.

As already mentioned malicious functionality is a bit wider for mobile devices. Here is a list of actions implemented in malicious applications encountered in mobile malware up to date.

Stealing user's data This category includes leaks of various user's data: passwords, account numbers, location, contacts, messages, phone call history, data in social networks, IMEI, IMSI, photos, videos. It also includes sensory malware which uses the information from onboard sensors such as accelerometer data, to derive sensitive information (e.g. user input) [10]. Leaked information then can be used legally to generate an advertisement; a spyware can use it to generate a database of mobile users; similarly, a personal spyware can use it to spy on a particular user.

Examples: Plangton (2011), Svpeng (2013).

Annoying advertising Most applications have embedded advertisements. However, sometimes such ads disturb from using the application too aggressively. This type of malicious applications is called adware.

Examples: Android Airpush (2007).

Use of premium-rate services This type of malware sends SMS or makes phone calls to premium-rate numbers without user's consent.

Examples: OpFake, FakeInst, Obad trojans (2013).

Sending spam SMS/MMS messages OpFake and FakeInst trojans are also designed to send spam SMS messages with a malicious link to contacts from the victim's contact list.

Establishing remote access channels Remote access channels are used to control the infected device and to organize the infected devices in a botnet. They can also be used for a targeted attack aimed at a particular victim.

Examples: Brador (2004), Svpeng trojan (2013).

Locking the OS functionality or user data This type of malware is called locker. Lockers can be used for ransom or simply to harm the owner of the device. They can use encryption to lock the data or change the device password.

Examples: Cardblock (2005), Simplocker (2014).

Changing or deleting user data, system or third party applications This type of malicious activity is usually implemented by so called vandal trojans, they often do not generate a direct financial gain to the attackers.

Example: Skuller (2004).

To achieve their goals, malicious applications often include additional functionality, such as:

- root exploits – attacks on lower layers of the software stack which grant the attacker root privileges;
- confused deputy attacks concerning malicious apps, which leverage unprotected interfaces of benign system and 3rd party apps (denoted deputies) to escalate their privileges [10];
- collusion attacks concerning malicious apps that collude using covert or overt channels in order to gain a permission set which has not been approved by the user [10].

Some malware detection methods rely on detection of this additional functionality.

3 Mobile malware detection methods

We survey the methods proposed for detection of malicious mobile applications in this section. We selected the most relevant methods for the review: the majority of the surveyed methods were published in proceedings of the recent conferences concerning mobile malware detection.

We grouped individual detection methods based on the techniques they utilize, the techniques that define each class of detection methods are explained in further detail in subsections 3.2 to 3.7 below. The classification is not unique, some methods can be included in several classes; however, all attempts were made to include methods in the most relevant groups. The individual detection methods are discussed paying a particular attention to their notable features and the underlying mechanisms. Subsection 3.1 details the criteria of comparison.

Table 1 provides general information for each of the reviewed methods. It serves for answering the question who and how can use the detection methods. Table 2 provides information on the methods evaluation. The NI in a cell stands for No Information, and a cell with "-" means that the corresponding parameter can not be measured in a detection method.

3.1 Criteria of comparison

1. Target audience

   The detection method is proposed either for the *mobile phone owners* or *mobile malware analysts* depending on the information provided by analysis and requested resources.

2. The analyzed object

   Most of the observed methods analyze *individual applications* for the presence of malicious activity (e.g. detect potential data leaks). Others analyze *the whole system state*, e.g. energy consumption, to detect malicious behavior.

3. Detection approach

   In *misuse detection* the observed behavior is compared to the predefined abnormal behavior. Conversely, *anomaly detection* identifies departures from the predefined normal behavior of the system.

4. Malicious actions detected by the system

   Possible malicious application functionality is discussed in section 2. For the methods intended for detection of specific malicious activity we listed the corresponding malicious actions. For the methods relying on detection of some features characteristic for malware we listed the corresponding features. Table 3 gives the values of this criteria for each of the reviewed methods.

5. The results of analysis

   The representation of analysis results varies in observed methods. Generally, the result can be an explicit recommendation on the analyzed object being malicious or not; or the information for further detailed analysis. Some methods produce a numeric value, a score, where higher scores generally indicate a higher confidence that the analyzed application is malicious. Sometimes this score is binary which means the method classifies applications as either *malicious or benign*. There is a group of methods which aim to give some kind of *security advice* to users or the *information valuable for malware analysts*.

6. The place of analysis

For each method we considered the place where the analysis is supposed to be deployed. The methods proposed for mobile handset owners usually run on *mobile handsets*. However, due to the limited resources a part of analysis is usually running on a *remote server* or in the *cloud* or it can be distributed across a *group of mobile devices*.

7. The moment of analysis execution

   For the methods proposed for mobile device owners the analysis can be executed either at the time of *installation* or at the time of *running*. The analysis provided at the time of installation of the suspect application, after it has been downloaded, usually prevents device owners from using dangerous applications. Analysis at the time of running is usually aimed at detecting and preventing sensitive data leaks. For the methods proposed for malware analysts the analysis can be executed only on Desktop PC without actual interaction with mobile device (denoted as *independent* in Table 1).

8. Supported operating system

9. Computational overhead

   Mobile malware detection and prevention systems introduce additional overhead which can be critical. Several types of overhead are distinguished: time, memory and energy. Time overhead is measured by comparing the time of different operations on an original system with the time of the same operations on the protected system. Memory overhead is a percentage of additional memory required by the running protected system (it also could be given as an absolute value). Energy overhead is measured by comparing the energy consumption of the system running the implemented malware detection method with the energy consumption of the original system. General formula for computational overhead is as follows:

   $Overhead = (\frac{V_{mod}}{V_{orig}} - 1) * 100\%,$

   V_{mod} is the corresponding value measured on a modified system and V_{orig} is the same value measured on original system.

10. Evaluation dataset

    Evaluation dataset is the data used for analysis, both malware and benign applications. This information is useful as researchers often use completely different datasets which means that the metrics of the implemented methods can not be compared directly. Besides, it helps to understand the scope of the provided evaluation.

11. False positives and false negatives rate (only for the methods which give the explicit recommendation on the analyzed subject being malicious)

    False positives rate is defined as a percentage of the benign applications being incorrectly identified as malicious by the detection system. False negatives rate is defined as a percentage of the malicious applications being identified as benign by the detection system.

12. The processing rate of the analysis

    For the methods analyzing individual applications we consider the average time required for the analysis of a single app.

3.2 Static analysis

Static analysis methods analyze the applications content without making use of its execution or emulation. They often include decompiling as the first step of analysis and then build data structures reflecting the features of an interest.

Egele et al. propose the technique called PiOS [11] which is aimed at detecting leaks of privacy-sensitive data. The analysis consists of Objective-C binary decryption and building of the control flow graph representing the analyzed application. The data leak paths are the paths originating from privacy-sensitive resources (the functions obtaining them) and ending with transmission over the network. The obtained graph is checked for the presence of such paths. PiOS was tested on a number of applications and revealed that most applications did not perform serious data leaks. However, transmitting device ID without user's knowledge was seen in many applications.

Social-AV is an example of social collaboration approach to detection [12]. The main idea of the method is to distribute a database of malware signatures among a group of devices supposed to belong to a group of friends in a social network. It helps to reduce memory resources consumed by the malware detection system. The authors of Social-AV also distinguish between more or less useful (often encountered) signatures. They introduce a "hot" set of signatures and this set is held by every device in a group. The other part of signatures database is divided into "cold" sets and distributed among the phones so these sets differ for each phone in a group. When a new file f is downloaded, this file is first prefiltered, then it is matched against a "hot" set of signatures and then against a "cold" one. In case no match was encountered the system requests a friend device to provide the same analysis. Each device also has a list of its friends devices with the numbers of signatures stored in each device's "cold" set. Social-AV is based on an open source anti-malware software, ClamAV. The experiments performed by the authors show that an AV agent of Social-AV consumes about 55% of memory consumed by a traditional anti-malware software with the same detection capability.

With the growth of smartphone's resources, the need in social collaboration for malware detection goes down. Such methods are no longer proposed by researchers. Instead, more sophisticated methods are increasingly in use, the ones that go beyond signature-based detection.

Grace et al. [13] classify applications that exploit software vulnerabilities as high-risk. Applications that cause users financial loss or steal sensitive information without exploiting software vulnerabilities are classified as medium-risk. Their tool RiskRanker aims at detecting applications which pose such risks by implementing signature-based analysis to detect known exploits and leveraging symbolic execution for the detection of user-unintended actions. RiskRanker also implements second-order analysis such as detecting native code execution, background dynamic code loading, use of encryption and decryption methods. Behavior that exhibit a combination of such properties is considered high-risk.

Rosen et al. [14] propose the system which builds application security profiles – AppProfiler. Based on the existing research and their own analysis the authors built a knowledge base which maps API call to the fine-grained descriptions of its privacy-related behavior and the aims of using the corresponding API. The API-calls extracted from the decompiled application are matched with the rules from the knowledge database. Then the set of the matched rules is processed in order to get the security profile. AppProfiler gives users access to the profiles through a web page and the Android application with the same name.

Most recent static analysis methods either include the dynamic part to enhance the analysis, or provide tools for verifying the results, e.g. AppIntent.

AppIntent proposed by Yang et al. [15] is devoted to detecting user-unintended private data leaks. Static taint analysis is used to detect the paths of privacy-sensitive data propagation and possible leaks, this step of the algorithm produces an event-space constraint graph with critical events (events with at least one instruction from the event handler method in a given data propagation path) and essential events corresponding to the necessary preconditions of the critical events. Guided symbolic execution is imple-

mented to generate the chains of events and the input data leading to transmission of privacy-sensitive data. AppIntent includes utility to simplify the analysts work: the analyzed application is repackaged concerning the above mentioned chains. Upon execution, the application automatically replays the malicious behavior according to the chain of events, the corresponding GUI elements are highlighted and the user is notified of data leakage. This utility also helps to verify the results of the static analysis and to avoid possible false positives.

The analysis in DroidSIFT system [16] includes building behavior graphs which consist of the API calls and relations between them. The derived graphs are then compared with the behavior graphs of malicious and benign applications and the similarity vector is built. The obtained feature vector is used in anomaly detector and signature-based classification. The authors of DroidSIFT also propose two methods of deploying the analysis scheme: the replacement of the vetting process or the public service.

3.3 Dynamic analysis

The methods based on dynamic analysis implement some kind of application execution either in the virtual environment or on a real device. The information obtained during execution can be analyzed at different abstraction levels to produce an understanding of application behavior.

TaintDroid [17] proposed by Enck et al. implements dynamic taint tracking integrating four granularities of taint propagation: file-level, message, method and variable. The mobile application is executed using the Dalvik virtual machine. Specific taint analysis marking the data originating from privacy-sensitive sources is performed during the execution. The impacted data should be identified before leaving a taint synk and getting transferred to a malware author.

To reduce the required resources several dynamic analysis systems utilize static techniques before executing the main part of analysis. For example, the prerequisites of potentially dangerous actions can be defined statically.

Zheng et al. [18] presented the system SmartDroid intended to automatically detect the UI-based trigger conditions required to expose sensitive behavior of Android malware. SmartDroid builds Activities Call Graph and Function Call Graph (FCG) of the analyzed application based on its smali code, then the dynamic analysis tracking the execution of sensitive API is performed. The dynamic part of analysis implements traversing of the UI elements in all emerging Activities excluding the wrong Activities not containing any sensitive API-calls. The sequences of UI events preceding the suspected API calls are recorded. SmartDroid modifies Android emulator in order to log the sensitive API names and parameters. SmartDroid demonstrated its effectiveness in revealing UI-based trigger conditions of the sensitive behavior for some malicious applications.

DroidRanger [7] applies a combination of static and dynamic analyses techniques in order to classify an application as either malicious or benign. DroidRanger comprises two detection engines, footprint-based and heuristic-based, aiming to detect known malware and zero-day malware correspondingly. The applications requiring dangerous permissions are considered potentially dangerous and further matched against the known malware behavioral footprints. The behavioral footprints comprise privacy-related information from the apps manifest, used packages, application code structure and FCG. Dynamic monitoring is applied to the applications with dynamic code loading. Calls to Android Framework APIs and some system calls are logged in process of dynamic monitoring and then manually analyzed. When a new zero-day is found, DroidRanger extracts its behavioral footprint and adds it to the known malware footprints.

Dynamic analysis scheme VetDroid considers the use of permissions [19]. The analyzed application is loaded into the Application Driver which extracts Activities and Services declared in the application manifest and executes them in a sandbox for a predefined time period. If some event handler is registered, Application Driver generates the corresponding events (new SMS, location changed, etc.) The user

Method	Proposed user	Analyzed object	Detection approach	Representation of result	Place of analysis	Moment of analysis	Supported OS
Static analysis							
PiOS [11]	malw. analyst	app	misuse	possible data leaks	remote server	independent	iOS
Social-AV [12]	device owner	untrusted data e.g app	misuse	malicious or benign	group of mobile devices	installation (obtaining data)	not specified
RiskRanker [13]	malw. analyst	app	misuse	malicious or benign	remote server	independent	Android
AppProfiler [14]	device owners, malw. analysts	app	misuse	application profile	remote server	independent	Android
AppIntent [15]	malw. analyst	app	misuse	possible data leak paths	mob. device + remote server	independent	Android
DroidSIFT [16]	device owners, malw. analysts	app	both	not specified	remote server	installation / independent	Android
Dynamic analysis							
TaintDroid [17]	device owner	app	misuse	possible data leaks	mobile device	running	Android
SmartDroid [18]	malw. analyst	app	misuse	UI-based trigger conditions for malware	remote server	independent	Android
DroidRanger [7]	malw. analyst	app	misuse	malicious or benign	remote server	independent	Android
VetDroid [19]	malw. analyst	app	misuse	function call graph	mobile device	running	Android
AppsPlayground [20]	malw. analyst	app	misuse	report	separately in emulator	running	Android
iDMA [21]	malw. analyst	app	misuse	malicious or benign, suspicious methods	mobile device	running	iOS
CopperDroid [22]	malw. analyst	app	misuse	the observed behavior	remote server	running	Android
Permission analysis							
Kirin [23]	device owner	app	misuse	assumption of the danger	mobile device	installation	Android
WHYPER [24]	device owner	app	misuse	description and required permissions correspondence	remote server	independent	Android
Machine learning							
DroidAPIMiner [25]	device owners, malw. analysts	app	misuse	malicious or benign	mobile device	installation	Android
MAST [26]	malw. analyst	app	misuse	rank of the danger	remote server	independent	Android
Machine learning by Shabtai et al. [27]	device owners	system state	anomaly	malicious or benign	mobile device + remote server	running	Android
Dendroid [28]	malw. analyst	app	misuse	malware family a sample belongs to	remote server	independent	Android
Drebin [29]	malw. analyst	app	misuse	malicious or benign / explanation	remote server	independent	Android
Battery life monitoring							
Battery life monitoring by Kim et al. [30]	device owner	system state	anomaly	malicious or benign	mobile device + remote server	running	Windows Mobile
VirusMeter [31]	device owner	system state	anomaly	alarm of possible malware	mobile device	running	Symbian
Cloud-based detection							
Paranoid Android [32]	device owner	system state	both	malicious or benign	device + cloud resources	running	Android
Crowdroid [33]	device owner	group of apps	misuse	malicious or benign	device + remote server	running	Android
Damopoulos et al. [34]	malw. analyst	app	anomaly	signal of suspicious activity	device + cloud resources	running	iOS

Method	Computational overhead	Evaluation dataset	FP	FN	Processing rate
Static analysis					
PiOS [11]	NI	825 apps from Apple's iTunes store; 582 apps from Cydia store	NI	NI	NI
Social-AV [12]	NI	809MB of data (text, image, pdf, etc.)	5.25% for prefilter	NI	NI
RiskRanker [13]	NI	118318 apps from 15 marketplaces; 133 samples from contagio repository	0%	9%	$\approx$ 1.03 s
AppProfiler [14]	NI	$\approx$ 80000 applications from Android Market, Contagio	16-23%	10-15%	NI
AppIntent [15]	NI	1000 top free apps from Google Play; 750 malicious apps	static: $\approx$ 50%, dynamic: 0%	NI	static: 3.3 m; dynamic : 5-134 m
DroidSIFT [16]	NI	13500 apps from Google Play, McAfee; 2200 apps from Android Malware Genome	5.5% for anomaly detection	2% for anomaly detection	$\approx$ 175.8 s
Dynamic analysis					
TaintDroid [17]	time: +3-29% memory: +3.5%	30 popular apps from Android Market	NI	NI	-
SmartDroid [18]	NI	19 apps from Antiy and Contagio repository	NI	NI	$\approx$ 37.63 s
DroidRanger [7]	NI	204040 free apps from several marketplaces; Geinimi, ADRD, Pjapps, Bgserv, DroidDream, zHash, BaseBridge, DroidDreamLight, Zsone, jSMSHider	0%	4.2%	$\approx$ 0.08 s (footprint-based)
VetDroid [19]	time +32.294% memory +14.110%	1249 top apps from Android Market; GGTracker, SMSReplicator, TapSnake malware	-	-	predefined
AppsPlayground [20]	NI	3968 apps from Android Market; Malware: FakePlayer, DroidDream, DroidKungFu	-	-	NI
iDMA [21]	NI	iKee-B, iSAM, Messages app	0% (Rand. Forest)	0% (Rand. Forest)	NI
CopperDroid [22]	time: + 20-32%	2900 samples from contagio, McAfee	-	-	NI
Permission analysis					
Kirin [23]	NI, lightweight	311 top apps from Android Market	1.6%	NI	NI, very fast
WHYPER [24]	NI	16001 apps from Google Play Store	NI	NI	NI
Machine learning					
DroidAPIMiner [25]	NI	16000 apps from Android Market and 8987 malicious apps	2.2%	1%	NI
MAST [26]	NI	36710 apps from SoftAndroid, Ndoo, Anzhi	NI	NI	0.45 s
Machine learning by Shabtai et al. [27]	Memory: 7027-7287 KB; CPU (learning): +13 - 14.5%; (idle): +0.0 - 1.72%	10 self-written malware, 5 malicious apps infected with PJApps, Geinimi, DroidKungFu-B, popular benign apps	13%	0%	NI
Dendroid [28]	NI	malicious samples from Android Malware Genome Project	5.74%	NI	NI
Drebin [29]	NI	123456 benign apps, 5560 malware	1%	4.1-6.1%	10 s
Battery life monitoring					
Battery life monitoring by Kim et al. [30]	NI	WiFifaker, Cabir, Mabir, Commwarior, Lasco	1 - 3%	0 - 5%	-
VirusMeter [31]	time +1.5% (linear regression)	FlexiSPY, Cabir; 100 clean data samples	5 - 22%	1.4 - 27.4%	-
Cloud-based detection					
Paranoid Android [32]	energy: +15-30%	data from phone users	NI	NI	NI
Crowdroid [33]	NI	60 self-written apps; PJApps, HongTouTou	0% for self-written	0% (self), 25% (HongTouTou)	NI
Damopoulos et al. [34]	CPU: + 20-45%, memory: +62-78%	iKee-B, iSAM, Messages apps and others	NI	NI	0.04-0.39 s

interaction is simulated using Android Monkey. Still, the full coverage of the program behavior is not guaranteed by the system. The module analyzing permission use extracts all the E-PUPs (the points of explicit permission use emerging on requesting a resource) and I-PUPs (implicit permission use points) and their connections. The I-PUPs are identified by the function-level taint analysis. The detected behavior is stored in the log file by the Log Tracer module. The log file is analyzed by the Behavior Profiler module which builds the FCG for the further offline analysis. VetDroid modifies Linux kernel, Binder driver and Dalvik VM to implement such an analysis.

The first dynamic analysis system implementing taint tracking on Android was TaintDroid. It was later used in other dynamic analysis systems like DroidBox [35], Mobile-Sandbox [36], AppsPlayground [20]. Nevertheless TaintDroid can not be directly used in automated analysis because it supposes manual execution, its successors implemented user interaction in some way. The simplest way of imitating user's actions is executing Android Monkey Tool which generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events [37]. Such interaction is used by VetDroid. More sophisticated methods like AppsPlayground consider the user interface of the application and its context in emulating possible user's interaction, which serves for a better code coverage.

One of the system based on TaintDroid is a multimodule dynamic analysis system AppsPlayground [20]. This system comprises several detection techniques, code exploration techniques and disguise techniques. The taint analysis is based on TaintDroid and is used for the detection of possible leaks. AppsPlayground monitors sensitive API usage on the middleware level and known exploits on the kernel level. To create a more realistic analysis environment various disguise techniques are used. Other used techniques are intended to provide provide better code coverage. These are the code exploration techniques such as fuzz testing, generation of fake events and intelligent execution which models user interaction with the displayed windows.

Damopoulos et al. proposed a software tool iDMA for dynamic monitoring and analyzing iOS applications [21]. CyDetector component of iDMA is able to detect specific system calls which can be used to acquire access to user's private data. CyDetector implements signature-based detection approach. The authors also employed machine learning techniques for analyzing the monitored methods executed by the analyzed app. They note that the Random Forest algorithm turned out to be the most promising of all the implemented machine learning algorithms.

CopperDroid system [22] implements dynamic analysis using modified version of Android emulator to track system calls executed by the app which are then processed by the unmarshalling Oracle component to retrieve high-level abstractions (IPC/ICC and RPC interactions). The system implements data-flow analysis to consider dependencies between observed system calls. The analysis also includes app stimulation with the events triggering the malicious behavior, which are statically obtained from the application code and the manifest.

3.4 Permission analysis

Permission analysis systems consider the information in application description and its manifest.

Kirin [23] defines conditions on the security configurations of applications under which they are considered potentially risky. These conditions refer to the requested permissions and action strings declared in the manifest, and if at least one condition is held, Kirin notifies the device owner of a dangerous app. Kirin should be used by Android Application Installer.

WHYPER [24], designed by Pandita et al., verifies whether the permissions required by the application correspond to its description. Natural language processing techniques are used in the analysis of descriptions. Parsed descriptions are then matched with the semantic graphs which represent the permissions.

Method	Detected malicious actions
Static analysis	
PiOS [11]	data leaks
Social-AV [12]	any known malware specified by a signature
RiskRanker [13]	known root exploits, use of premium-rate services
AppProfiler [14]	malware executing considered privacy-relevant API calls
AppIntent [15]	data leaks
DroidSIFT [16]	depends on the dataset
Dynamic analysis	
TaintDroid [17]	data leaks
SmartDroid [18]	sensitive behavior: possible data leaks, unintended sending SMS/MMS
DroidRanger [7]	any malware with a known behavior profile; particularly: sending SMS to premium-rate numbers, monitoring SMS
VetDroid [19]	data leaks
iDMA [21]	accessing user's private data
CopperDroid [22]	not specified
Permission analysis	
Kirin [23]	data leaks, disabling antivirus software, SMS spam, control channel through SMS
WHYPER [24]	privacy leaks: illegally reading contacts, calendar; illicitly recording audio
Machine learning	
DroidAPIMiner [25]	any malware using API more often encountered in malicious samples than in benign ones
MAST [26]	any malware with the corresponding values of considered features: dangerous permissions, intent filters, etc.
Machine learning by Shabtai et al. [27]	data leaks (self-updating malware)
Dendroid [28]	any malware belonging to malware families used at the learning stage
Drebin [29]	behavior using suspicious API calls and permissions, suspicious communications
Battery life monitoring	
Battery life monitoring by Kim et al. [30]	malware with predefined energy-consumption signatures
VirusMeter [31]	malware implementing actions that can drain battery significantly
Cloud-based detection	
Paranoid Android [32]	Detection methods are not fully specified. The authors consider malware with known signatures, data leaks and various exploits
Crowdroid [33]	any malware using API that can be distinguished from API used by benign app in clustering
Damopoulos et al. [34]	illegitimate use of system services, unauthorized users

Table 3: Mobile malware detection methods comparison (III)

It can be seen that most methods are intended to detect data leaks or some predefined malware.

3.5 Machine learning

We also distinguished a group of machine learning methods employing standard machine learning algorithms. They comprise a learning stage where the corresponding detection model is built and a classification stage. The corresponding methods can be regarded as static or dynamic analysis methods using machine learning as a tool. However, we decided to consider such methods a separate group, because the most significant part of analysis here is provided by machine learning techniques.

Aafer et al. [25] compared API used by malicious and benign Android applications and revealed characteristic API with some conditions on parameters. These API calls are encountered in malicious applications at least 6% more often than in benign ones. Authors' proposed DroidAPIMiner performs the following analysis:

1. Analysis of the bytecode. A vector of the used APIs is formed on this step.

2. Classification using one of the following classifiers: ID5 DT, C4.5 DT, KNN, SVM. The best

results have been shown by KNN classifier.

This step requires classification models built by training the classifiers with the corresponding vectors for malicious and benign applications.

Aafer et al. also used the vectors of required permissions for the classification but the permission model did not appear to be robust enough when applied to the applications requiring permissions not used the training dataset.

Chakradeo et al. [26] proposed the application triage system MAST with the aim to cut the required resources and the costs of preventing malware in large application markets. The main goal of the system is to identify the most suspicious applications for the first-order thorough analysis. The proposed analysis is lightweight. The authors extract the following application features: specific required permissions and Intent filters, embedded zip-archives, the presence of native code. Next, the distances (the similarity) between the analyzed application and the groups of malicious applications are computed using Multiple Correspondence Analysis. The obtained values are used to compute the general rank of the application danger.

Shabtai et al. utilize traffic analysis to detect malicious applications [27]. A special subclass of malicious applications is considered, namely self-updating applications that accomplish malicious activity some time after the installation, e.g. on the application update or after dynamic loading of a precompiled code. Based on the precomputed model of normal traffic statistics the observed behavior of the application can be classified as either normal or anomalous. Authors used Decision Table and REPTree classification algorithms as they showed the best results. If the sequence of application states is classified as anomalous, the analyzed application is considered malicious. This detection method is implemented as an Android application on a user's device communicating with a remote server.

Dendroid [28] analyzes the code structure of the applications using text mining techniques. First, Suarez-Tangil et al. extract code chunks from a population of malicious applications with the help of Androguard tool [38]. Next, based on the term frequency-inverse document frequency statistics family feature vectors are built. The most specific to a malware family code chunks are given the highest values in the feature vector. Hierarchical cluster and linkage analyses are then used to further study malware families and associated feature vectors. For a newly analyzed application Dendroid idenfies the malware family whose vector is the closest to its feature vector with respect to the cosine similarity score.

Drebin [29] method utilizes SVM for malware classification. It uses a broad range of features: obtained statically hardware components requested by the app, the requested permisssions, filtered intents, app components, suspicious and restricted API calls, network addresses. The analysis also includes explanation in case the app is considered malicious.

Looking at the methods listed above it can be seen that machine learning techniques used for mobile malware detection have been evolving in the last few years: from using relatively simple features like permissions or APIs to more complex features that analyze the code structure of applications.

3.6 Battery life monitoring

The presence of malicious applications on a mobile device can lead to additional energy consumption. This is an important consideration particularly given limited battery capacity of mobile handsets.

Malware detection framework proposed by Kim et al. [30] is based on detection of energy-depletion threats. The framework consists of a power monitor and a data analyzer components. The power monitor collects and stores the power consumption history. The data analyzer generates power signature by extracting a unique pattern from the history. This power signature is matched with the signatures in the database (*a priori* signatures). Similarities between power signatures are measured by the χ^2-distance. The data analyzer can be deployed on either the host mobile handset, or a remote server/data-sync PC

to reduce the overhead. According to the experimental results on an HP iPAQ running a Windows Mobile OS, the framework achieves significant (up to 95%) storage-savings without losing the detection accuracy, and a 99% detection rate in classifying mobile malware.

VirusMeter [31] considers a list of used services in monitoring battery power usage. Based on the predefined power consumption model and the list of used services, VirusMeter calculates the hypothetical (expected) power consumption. An alarm is generated if the difference between the actual power consumption and the expected one exceeds some predefined threshold. The system builds its own user-specific power model where the amount of consumed power is computed based on the standard actions of the user, their durations and also environmental factors. These factors in combination are used in three different approaches: linear regression, neural networks and decision trees. The three approaches are evaluated separately, showing the highest detection rate for the neural network approach and the highest false positive rate for the linear regression approach.

Hoffman et al. studied the energy consumption on modern smartphones [39]. They conducted several experiments with both malware and generated additional overhead, the results indicate that additional energy consumption is insignificant and can be compared to the noise introduced by unpredictable user and environment interactions. They concluded that battery life monitoring is mostly not satisfactory for malware detection on modern smartphones.

3.7 Cloud-based detection

Limited resources of mobile devices lead to the development of cloud-based detection methods. Such methods propose schemes for efficient analysis transferring data from the device to analysis place.

Paranoid Android (PA) [32] proposed by Portokalidis et al. is a cloud-based malware protection technique. On the phone, a tracer records all the data that is necessary to replay the execution of an application. Using the encrypted channel this data is further transmitted to a cloud. The execution is replayed on the cloud by a replayer which can apply several security checks in parallel. In principle, there is no limit on the number of attack detection techniques that can be applied simultaneously. When an attack is detected PA warns the user using a special signaling channel beyond the control of the attacker. PA uses a network proxy allowing an interception and temporal storage of inbound traffic before the replayer retrieves the corresponding data. This helps to significantly reduce data transmitting overhead imposed by the detection method. However, the significant drawback of the system is that PA implements the tracer module in the user space thus reducing battery life by about 30%.

Crowdroid [33] system is behavior-based. It consists of a Crowdroid client application and the central server. Client apps collect the executed system calls for each application and transmit it to the analysis server. Upon receiving such traces the server forms system call feature vector and, ultimately, a dataset of such vectors for every app. Each dataset is then clustered using a partitional clustering algorithm (particularly, k-means algorithm). Using the clusterization information Crowdroid could alert the users when one of their applications shows an abnormal trace.

Damopoulos et al. implemented a framework for anomaly-based IDS on iOS smartphones [34]. The framework includes several components: event sensors, system manager, detection engines on both the host and the cloud, response manager. The detection methods implement machine learning algorithms (particularly, Random Forest). The authors conclude that deploying particular detection methods either on host or in the cloud depends on the preferable timeliness and performance of the resulting system.

4 Mobile malware prevention methods

In this section we discuss the malware prevention methods proposed for mobile environment. Such methods are designed not to detect but rather to prevent possible malicious actions although some of them can be used as detection methods as well.

4.1 Criteria of comparison

Some of the criteria used for comparison of malware detection systems cannot be applied to malware prevention systems. First, most malware prevention systems are implemented on the device itself, i.e. they are proposed for mobile device users. Second, most prevention systems analyze all applications simultaneously. However, the processing rate of analyzing an application can not generally be determined, as the prevention system is analyzing it all the time while it is running, which makes it difficult to set the baseline. These remarks are not applied to application repackaging prevention systems. Such systems can be used by application markets to automatically enhance the security of the provided applications.

It is also worth noting that most malware prevention systems have to modify some of the OS components to be effective. This is likely to restrict their appeal to a typical smartphone user.

Table 4 reviews individual prevention systems across the following criteria:

1. Malicious actions that the system is designed to prevent;
2. The components of the OS that are modified by the prevention system;
3. Evaluation dataset;
4. Computational overhead;
5. Supported operating system.

4.2 Prevention of data leaks

AppFence proposed by Hornyack et al. [40] implements privacy controls to protect user data from exfiltration by malicious applications. AppFence combines data shadowing with exfiltration blocking techniques. Data shadowing prevents applications from accessing sensitive information by substituting it with empty or generated data. Exfiltration blocking is used to prevent the transmission of sensitive data over the network. AppFence system is based on TaintDroid. Hornyack et al. measured side effects of AppFence by recording visual output of application execution with and without policy control and they found that only 66% of applications are executed without any side effects.

Zhou et al. propose TISSA system that offers the protected mode for the applications [41]. The phone user can restrict access to private resources (location, phone state, contacts, phone calls) for each application. Different types of protection are available in TISSA: anonymized data, bogus data and empty data; each one representing data used instead of original data in the protected applications. The system introduces minor changes in the Android Framework in addition to a special TISSA application.

ProtectMyPrivacy is a system designed for the jailbroken iOS telephones [42]. ProtectsMyPrivacy provides functionality for setting security restrictions for different applications in order to prevent data leaks. The original data requested by the application can be substituted with the bogus one. There is a module sending formed security settings to a remote server. It is also possible to request recommended security settings based on the settings obtained from other ProtectMyPrivacy users.

Method	Mitigated malicious actions	Modified system components	Evaluation dataset	Computational overhead	Supported OS
Prevention of data leaks					
AppFence [40]	data leaks	Resource Manager, file system components	50 apps from Android market	NI	Android
TISSA [41]	data leaks	Privacy-aware components	24 free apps from Android market	NI	Android
ProtectMyPrivacy [42]	data leaks	various methods used to access private information, Springboard	225685 apps	NI	iOS
Application repackaging					
Aurasium [43]	data leaks, unintended use of premium-rate services, root exploits	No	3491 apps from lisvid.com, store, 1260 malicious apps	time +14-35%, NI for memory overhead	Android
Application repackaging by Livshits et al. [44]	data leaks	No	100 apps from WP Store	NI	Windows Phone
Mandatory access control					
FlaskDroid [10]	root exploits, data leaks, sensory malware, confused deputy, collusion attacks	Package Manager, Services, Content Providers	synthetic tests, WhatsApp, Facebook, mempodroid exploit	basic policy: time +37% memory +1.2%	Android
XManDroid [45]	confused deputy and collusion attacks	Package Manager, Reference Monitor	50 third party applications, self-developed malware	time for ICC +50%, time for read access: +2.4% - 48%, NI for memory overhead	Android

Table 4: Mobile malware prevention methods comparison

4.3 Application repackaging

Aurasium is a tool for repackaging Android applications with the aim to enhance users security and privacy. Aurasium is capable of intercepting virtually all framework APIs and enforcing many classes of security and privacy policies on them [43]. A part of Aurasium policy enforcement is implemented as a native library. Aurasium also modifies the apps java code with the help of apktool [46]. The resulting repackaged application displays warning messages to user encountering interception of any dangerous API. Repackaging applications implies the problem with signing the resulting apps. Aurasium solves it by matching the developers signatures with their own generated signatures and resigning the apps correspondingly.

Livshits and Jung propose static analysis technique for Windows Phone applications in order to make the requests for accessing user private data mandatory [44]. Analysis includes building of the control flow graph and inserting the checks immediately preceding the access to the sensitive resources. This technique is also proposed for the application market.

4.4 Mandatory access control (MAC)

Mandatory access control with specifically defined security policies can also be used for mobile malware prevention. SE Android [47] implements MAC on kernel and middleware level demonstrating the effectiveness in preventing known root exploits and application vulnerabilities exploitation.

Bugiel et al. proposed XManDroid [48] and its enhanced version [45]. It implements policy enforcement in the middleware level by applying policy rules to the analyzed ICC (Inter-Component Communicaton). Policy decisions are cached and reused for identical ICCs. Policy rules are based on dangerous combinations of permissions granted to the applications that can lead to confused deputy or collusion

attacks. To implement MAC on the kernel level TOMOYO Linux v 1.8 is employed. To enable communication between the two layers, the authors wrote a native library with access to TOMOYO's interfaces.

Another security architecture proposed by Bugiel et al. is FlaskDroid [10]. FlaskDroid is inspired by SELinux (SE Android) while extending it at the middleware level. MAC on these two levels, kernel and middleware, is dynamically synchronized. FlaskDroid is designed to support several policies of different stakeholders simultaneously (e.g. app developer, phone user) reconciliating them if necessary. Security policies can also be configured by User Policy App included in the FlaskDroid.

5 Conclusion

Only about a half of the detection methods discussed in our review is supposed to be utilized by device owners (see Table 1). It reveals that much work remains to be done in building security solutions and malware detection tools ready to be used by phone users.

Battery life monitoring malware detection methods are no longer viable, as proved by Hoffman et al. [39]. The increased resources of the latest mobile devices are gradually moving aside cloud-based detection methods, although they are still relevant for a thorough malware analysis.

Systems based on permission analysis consume minimal resources and therefore they are the most suitable for the deployment on mobile devices. At the same time, they are too imprecise to be used separately from the other malware detection tools.

Machine learning methods suit well for mobile malware detection, especially application ranking. However, it should be noticed that the significant drawback of such methods is that they rely extensively on the learning dataset and as the new malware emerges, the corresponding teaching models should be renewed as well.

The most thorough analysis can be achieved by the use of dynamic methods, however, such detection approach is also the most resource-consuming (see Table 2). Dynamic checks can be used in combination with the static analysis to achieve more precise results. Most of the dynamic analysis methods reviewed in this paper are proposed for malware analysts and are not generally available to the wider public. It also can be noticed that there is a tendency for creating sophisticated dynamic analysis tools taking into account the GUI of the analyzed application and its semantics for achieving a better code coverage.

In general, it can be seen that machine learning combined with static and dynamic mobile malware detection methods are the most widely used. We believe that various combinations of such techniques can be used for efficient mobile malware detection in future.

Mobile malware prevention systems, in general, provide a better coverage of mitigating malicious actions. As reviewed in Table 4, the prevention methods implemented in running systems require modification of OS components which might be considered a significant drawback by a typical user. Application repackaging methods do not require modification in OS but they depend essentially on the decompiling tools which can fail to obtain a proper representation of the analyzed application. Besides, there are certain problems with signing the resulting repackaged applications. The methods implementing mandatory access control are the most advanced of all the discussed methods and provide the widest coverage of the prevented malicious actions.

The techniques of automatic anti-malware verification applied by application markets are gradually enhanced, therefore we believe that high-quality market checks are central for users' security. Additional checks by antivirus applications installed on a device can be useful. Mandatory manual analysis of all the submitted applications before exposing them to users would bring the best results in application security, however, it is unfeasible for most application markets due to a huge number of the supplied apps. To counteract some of the limitations of the automatic application checks, other measures such as enhancing OS security and introducing mobile malware prevention systems can be taken. These two