Digital Investigation 14 (2015) 17–35

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Andro-AutoPsy: Anti-malware system based on similarity matching of malware and malware creator-centric information

Jae-wook Jang [a], Hyunjae Kang [b], Jiyoung Woo [a], Aziz Mohaisen [c], Huy Kang Kim [a, *]

[a] Graduate School of Information Security, Korea University, Republic of Korea
[b] Enterprise Risk Services, Deloitte Anjin LLC, Republic of Korea
[c] Computer Science and Engineering Department, State University of New York at Buffalo (SUNY Buffalo), USA

ARTICLE INFO

Article history:
Received 12 October 2014
Received in revised form 24 June 2015
Accepted 27 June 2015
Available online 16 July 2015

Keywords:
Similarity matching
Profiling
Android malware
Malware classification
Certificate

ABSTRACT

Mobile security threats have recently emerged because of the fast growth in mobile technologies and the essential role that mobile devices play in our daily lives. For that, and to particularly address threats associated with malware, various techniques are developed in the literature, including ones that utilize static, dynamic, on-device, off-device, and hybrid approaches for identifying, classifying, and defend against mobile threats. Those techniques fail at times, and succeed at other times, while creating a trade-off of performance and operation. In this paper, we contribute to the mobile security defense posture by introducing Andro-AutoPsy, an anti-malware system based on similarity matching of malware-centric and malware creator-centric information. Using Andro-AutoPsy, we detect and classify malware samples into similar subgroups by exploiting the profiles extracted from integrated footprints, which are implicitly equivalent to distinct characteristics. The experimental results demonstrate that Andro-AutoPsy is scalable, performs precisely in detecting and classifying malware with low false positives and false negatives, and is capable of identifying zero-day mobile malware.

Introduction

The explosive growth in the number of mobile devices running the Android platform has attracted the attention of malware creators because a vast amount of private information (e.g., contacts, short messages, and e-mails) is usually stored on these devices. The availability of this information in many mass-market mobile devices renders them a desirable target for malware creators, making the security of mobile devices one of the most important, yet challenging, areas of research.

Mobile as well as traditional malware analysis for detection and classification falls into two broad types: dynamic and static. Dynamic analysis aims to provide methods for effectively and efficiently extracting the unique patterns of each malware family based on its behavior. If malware programs (or samples) behave in a unique way under a specific condition, this type of analysis fails to detect them because it does not recognize their intended malicious behavior. Dynamic analysis techniques, on the other hand, analyze malware on an emulator or a mobile device without the human interaction, providing autonomous installation and execution. On one hand, this type of analysis has several limitations with respect to

* Corresponding author. Tel.: +82 2 3290 4898.
E-mail addresses: changkr@korea.ac.kr (J.-w. Jang), janetk1004@gmail.com (H. Kang), jywoo@korea.ac.kr (J. Woo), mohaisen@buffalo.edu (A. Mohaisen), cenda@korea.ac.kr (H.K. Kim).

http://dx.doi.org/10.1016/j.diin.2015.06.002
1742-2876/

analyzing malware embedding updates, drive-by downloads, or C&C (Command & Control) attacks (Zhou and Jiang, 2012). Furthermore, for capturing the unique behavior of malware accurately, dynamic analysis needs to roll back the emulator or mobile device into its clean state whenever the analysis of a piece of malware is complete. On the other hand, using static analysis, strings of bytes associated with malware samples are discovered through reverse engineering and used as a signature for identifying malware. In spite of effective characteristics, static techniques are often prone to high false positive rates because of the evolution in the code basis and code repackaging often associated with malware. Static analysis techniques require more efforts in reverse engineering to generate reliable and meaningful signatures. Furthermore, recent and new malware families have utilized embedding obfuscation techniques, such as proguard, which change the calling order or the method names of variables and functions, and thus hinder the transformation of the malware itself into source code, making such defenses ineffective.

Despite the efforts of antivirus (AV) vendors, the amount of malware is increasing exponentially. According to a report by McAfee, 2.47 million new pieces of mobile malware and a total of 3.73 million pieces of malware appeared in 2013 (McAfee, 2013). Between the end of 2012 and 2013, the total amount of malware increased by nearly 200%. To address this trend, AV vendors analyze a large number of malware samples every day in order to prevent their widespread dissemination and to guide users on disinfection and risk management by classifying malware into broad families. However, malware-centric analysis, including both static and dynamic analysis, is limited, and is not keeping pace with the trends increasing numbers of malware and their families. Existing malware analysis method focuses on codes and functions of malware. In particular, static analysis takes a long time to parse meaningful code patterns in disassembled or decompiled codes, and dynamic analysis requires an irritating amount of analyst-guided pre-analysis time for operations such as roll-back of emulator or mobile devices.

The added contents reads as follows: The “Trojan horse defense” surfaced in 2003 in several cybercrime cases brought in the United Kingdom is a good example of how our work can relate to the digital investigation community; the incident pertains to the claim that a malware creator ran malicious codes on victim's device without the device owner's consent (Brenner et al., 2004). The defense attributed actions to malware and has presented a challenging issue to the forensics community: the accurate assessment and investigation to determine whether a person is innocent or guilty. Technically, investigators needed to determine if a system was compromised, and if so what are the implications of such compromise and what unapproved activity was the device doing. To this end, our proposed method incorporates malware creator information as well as malware-centric information to attribute malware. Our method identifies direct evidence of malicious behaviors of malware creators to detect malware created by them and analyzes the malicious behavior and attacker's intent.

To overcome the drawbacks inherent in previous malware-centric methods and help investigators answer these questions, we propose a novel and feature-rich anti-malware system based on profiling, called Andro-AutoPsy. Our system is a novel hybrid malware detection and classification method based on similarity matching of profiles. Our proposed profiling system, which comprises mobile devices and a remote server, is analogous to criminal profiling. In the real world, criminal profiling, also known as offender profiling, is a methodology that is intended for helping investigators accurately predict and profile the characteristics of unknown criminal subjects or offenders (Kocsis, 2009; Nykodym et al., 2005; Rogers, 2003). We adopt criminal profiling methodology in the malware analysis domain. In order to respond to malware more efficiently and effectively, malware analysts need to check the target of an attack, since it reflects the attack's intent of the malware creator. Such process tries to achieve the end goals by answering the following questions. 1) What do malware creators want to obtain? 2) How do malware creators attack the victim? 3) What do malware creators need for an attack? By answering these questions, analysts can understand malware creators' attack pattern.

For understanding the intent of malware creator, we exploit integrated footprints, including opcodes in *.smali*, meta-data in *Androidmanifest.xml*, and the serial number of a certificate as feature vectors for malware characterization. We observe that a) malware samples have unique malicious behavior patterns and characteristics, b) the malicious behavior of malware samples is determined by operation codes (opcodes) and requires a particular permission set, and c) such an opcode set influences the behavior of the malware. To operate our system at scale, we represent malware characteristics using Bayer's profiling as described in Bayer et al. (2009). We prepare a representative profile that combines multiple features. For that, and for each malware family, we characterize it by integrated footprints using static analysis features. Then, by comparing the profiles, we detect and classify malware samples into similar groups.

Contribution:

1. We propose an “Integrated Malware Analysis System” which considers malware-centric information as well as malware creator-centric information. Using the serial number of a certificate simplifies the process of malware detection and classification.
2. We demonstrate the operational relevance of our system. Our system enables AV vendors to react to many species of malicious samples by quickly and efficiently conducting similarity matching between these and previously detected samples. Our system facilitates the detection of new malware, including existing malware's variants and zero-day exploits. This is further highlighted through in-depth experiments using real-world malware samples. Our system implements an efficient malware detection and classification method. Despite using static analysis, it requires only 72 s/MB to detect and classify malware into similar groups.

Table 1
Various malware detection/classification methods in previous works.

Approach	Method	Feature	Previous works
Detection on mobile device	Permission	Permission	Enck et al. (2009) Pearce et al. (2012)
	Footprint	System resources	Bugiel et al. (2012) Shabtai and Elovici (2010)
		Taint tracing	Enck et al. (2010)
		Event log, System call	Bose et al. (2008)
Detection outside mobile device	Permission	Permission	Peng et al. (2012) Wang et al. (2013)
	Footprint	System call	Lin et al. (2013)
		System call, Disassembled code	Blasing et al. (2010)
		System call, Interaction log	Reina et al. (2013)
		System call	Zheng et al. (2013)
		System/API call, Taint tracing	Rastogi et al. (2013)
	Permission + Footprint	Permission, API call	Arzt et al. (2014) Cao et al. (2015) Yang et al. (2012) Yang et al. (2015)
		Permission, API call, System call, XML information, Disassembled code	Spreitzenbarth et al. (2013) Weichselbaum et al. (2014) Yan and Yin (2012) Zhou et al. (2012)
		Permission, Network traffic	Vidas et al. (2014)
Hybrid	Footprint	System call	Burguera et al. (2011) Isohara et al. (2011)
		Function call	Schmidt et al. (2009)

3. Our proposed method can accommodate additional features that depict the unique footprint patterns of malware. While our method mainly utilizes the static analysis technique for malware classification, it is also flexible in utilizing dynamic analysis technique to capture the malicious behavior.

Organization

The rest of this paper is organized as follows. In Section Related work, the related work is reviewed. In Section Profiling modeling, we present our profiling system for malware analysis. Data exploration to find meaningful features for anti-malware system is presented in Section Data exploration. In Section Andro-AutoPsy: An Anti-Malware System, we present our anti-malware system, Andro-AutoPsy. In Section Performance evaluation, the performance evaluation results are provided. In Section Limitations, we discuss the limitation of our proposed method. Finally, we discuss future research directions and conclude with Section Conclusion and Future Work.

Related work

Based on where the scan and monitoring of the mobile malware takes place, malware analysis methods can be classified into three types: detection methods on the mobile device, detection methods outside the mobile device, and hybrid detection methods. We classify the studies in the literature based on the type of the malicious behavior into permission- and footprint-based methods. Footprint-based methods include system call-based, API call-based, disassembled code-based, and XML information-based methods. The detection methods on a mobile device scan malicious behavior patterns on the device and return the analysis results to the user. However, these approaches do not consider resource constraints of the mobile device, which may affect their usability and the user experience upon the operation of such systems: low computing power and battery life are two fundamental properties.

The detection methods applied outside the mobile device execute detection algorithms on an emulator or a real device running the targeted applications, and conduct static or dynamic analysis to determine the nature of the applications. While these approaches do not need to consider the aforementioned resource constraints, they cannot respond quickly to new malware families. To overcome the drawbacks of both approaches, hybrid approaches have been introduced in mobile malware analysis. The client module on the mobile device is instrumented to collect information related to the applications installed on it and to send the information to a remote server. The remote server then analyzes log files using their detection algorithms, while not impeding usability or degrading the user experience at the mobile device. Table 1 summarizes the various malware detection or classification methods in the literature falling under those categories. In the following, we elaborate on some of the related work in each category.

Detection methods on mobile devices

Previous works in this category have introduced malware detection methods that can execute relevant applications on devices, providing online detection.

Enck et al. (2009) proposed the Kirin security service, which performs lightweight certification of applications to mitigate malware at installation time. Kirin examined the requested permissions of applications, compared them

with self-defined security rules, and determined whether or not malicious activities were executed. Requested permission-based methods rely almost completely on the permissions given in a manifest file, *Androidmanifest.xml*. However, application developers tend to declare an excessive number of permissions in a manifest file, although the application does not in fact need them all. Thus, these methods' ability to detect and classify malware with a high accuracy is limited. Pearce et al. (2012) introduced AdDroid; they separated advertising permissions for the Android platform. In AdDroid, the host application and the core advertising code run in an isolated environment, where applications using AdDroid no longer send sensitive information to advertisement servers. However, AdDroid very rarely reacts to information leakage unrelated to advertisement, which applies to the majority of mobile malware.

Bugiel et al. (2012) proposed Xmandroid, a system-centric and policy-driven runtime monitoring system that regulates communications between applications. According to a heuristic analysis, the authors identified attack patterns and classified malicious applications. Shabtai and Elovici (2010) proposed Andromaly, a behavior-based detection framework for Android-based mobile devices. Andromaly is a host-based intrusion detection system that continuously monitors various resources and classifies malicious applications using a machine-learning algorithm. These proposed methods, however, require a significant hardware capacity (e.g., CPU, RAM, and battery life) in order to monitor all resources comprehensively. Enck et al. (2010) proposed Taintdroid, an extension to the Android mobile-phone platform that tracks the flow of sensitive information through third-party applications. If tainted data leave the Android device, Taintdroid provides a report logging the leaked data, where it was sent and which application leaked it. Taintdroid focuses on information leakage. An emulator, such as Droidbox, embeds Taintdroid and tracks information leakage. Bose et al. (2008) proposed a signature-based detection method for the Symbian operating system. The method constitutes a two-stage mapping technique consisting of an extraction and a representation process that construct signatures at run-time from the monitored system events and system calls. The method uses temporal logic to detect malicious activity over time that matches a set of signatures represented as a sequence of events. However, the method needs to obtain root privileges to access a kernel, and requires sufficient hardware capacity to extract system calls and convert related features into signatures.

Detection methods outside mobile devices

Previous studies in this category introduced malware detection methods that execute the relevant applications outside the device, providing offline detection. These methods execute their detection algorithms on an emulator or a real device other than the host device, and thus they are not subject to the constraints of real devices and do not impede usability or degrade the user experience.

Peng et al. (2012) used probabilistic generative models for risk scoring schemes, ranging from the simple Naïve Bayes to advanced hierarchical mixture models. Their proposed methods compute a real risk score of Android applications based on the requested permissions, and differentiate between malware and benign applications. Wang et al. (2013) proposed DroidRisk, a framework for quantitative security risk assessment of Android permissions and applications based on requested permissions. By using the quantitative risk levels of applications, they showed that a reliable risk signal could be generated to warn potential malicious activities. However, application developers tend to declare an excessive amount of permissions in a manifest file, requiring the method to rely on other criteria to achieve higher detection and classification accuracy.

Lin et al. (2013) proposed a System Call Sequence Droid (SCSdroid), which adopted the thread-grained system call sequences invoked by applications. SCSdroid first captures the system call sequence, extracts the common subsequences of malware, and adopts Bayes theorem to detect malware. SCSdroid then extracts system calls by exploiting Strace (Strace, 2013); however, Strace executes its functionality after installation. Thus, SCSdroid cannot detect malicious behavior during the installation process, and depends on the functionality of Strace. Blasing et al. (2010) proposed an Android Application Sandbox (AASandbox), which enables static and dynamic analysis on the Android platform. In the static analysis phase, AASandbox decompresses installation files and disassembles intended executable files, and then, compares them with pre-defined malicious patterns. In the dynamic analysis phase, it hijacks system calls for logging and builds a frequency table of system calls. However, the dynamic analysis methods based on the frequency of system calls need a more elaborate and redefined process in order to improve their detection or classification accuracy; the function name of the system call as well as the arguments used in the system call need to be considered. Reina et al. (2013) introduced Copper-Droid, an approach built on top of QEMU to automatically perform dynamic analysis of Android malware. CopperDroid conducts a unified analysis to characterize low-level OS-specific and high-level Android-specific behaviors (e.g., information leakage, sending SMSs) by observing and analyzing system call invocations, and IPC and RPC interactions. Zheng et al. (2013) introduced a systematic approach, called DroidAnalytics, a signature-based analytic system for automatically collecting, managing, and analyzing malware. Their system allows analysts to retrieve, associate, and reveal malicious logics to the opcode level. They only focused on detecting malware, leaving a lot to be desired in classifying and clustering malware into families. Rastogi et al. (2013) proposed AppsPlayground, a framework for automatic dynamic analysis, which executes a suspicious application on an emulator built on top of QEMU, and determines whether or not malicious activities are carried out by tracking information leakage and monitoring sensitive API and system calls.

Arzt et al. (2014) proposed a static taint-analysis system, FlowDroid, which is context-, flow-, object-, and field-sensitive and lifecycle-aware taint analysis for Android applications. By handling Android-specific characteristics such as lifecycle of an application or callback methods, FlowDroid successfully conducted information flow analysis. Cao et al. (2015) proposed a novel static

analysis tool, EdgeMiner, which is to systematically address the challenge of implicit control flow transitions. Their system applied automated program analysis methods to identify the callbacks and their registration methods on the Android platform. Yang et al. (2012) introduced a systematic approach, called Money-Guard, to detect stealthy money-stealing applications in the Android market. Money-Guard checks API calls and billing-related permissions to detect stealthy money-stealing malware, but cannot identify various malicious behavioral patterns, except for malware sending premium-rate SMSs. Yang et al. (2015) proposed a malware detection method, AppContext, based on the context of a security-sensitive behavior. AppContext constructed a self-defined call graph from an apps binary through static analysis and generated the complete contexts as leveraging identified activation events and context components. Then, AppContext classified the security-sensitive behaviors by using the extracted contexts, and detected malware. Spreitzenbarth et al. (2013) proposed Mobile-Sandbox, a static and dynamic analyzer for Android applications, like in AASandbox. In the static analysis phase, Mobile-Sandbox parses a manifest file, decompiles the application, and checks whether or not suspicious permissions have been used. In the dynamic analysis phase, they execute the application on Droidbox, log every operation of the application, and record native library calls executed by the processes. They extract the native library calls by exploiting ltrace (ltrace, 2014); ltrace executes its functionality after installation process is completed. Weichselbaum et al. (2014) introduced ANDRUBIS, a fully automated analysis system based on static and dynamic analysis approach, to analyze an Android application. In the static analysis phase, ANDRUBIS extracted information from a manifest file and its actual bytecode. In the dynamic analysis phase, ANDRUBIS executed the application in an emulated environment. While executing the application, ANDRUBIS monitored its actions at both the Dalvik VM and the system level. Yan and Yin (2012) proposed DroidScope built on top of QEMU and allowed the OS-level and Java-level semantic views to be reconstructed simultaneously. They analyzed malware by collecting native/Dalvik instruction traces, API-level activity, and information leakage. Zhou et al. (2012) proposed DroidRanger, which identifies malicious behavior through both a permission-based behavioral footprint scheme for the detection of known malware and a heuristic-based filtering scheme for the detection of zero-day malware. Vidas et al. (2014) presented A5, a fully automated analysis system based on static and dynamic analysis approach. In the static analysis phase, A5 extracted information from a manifest file and its actual bytecode. In the dynamic analysis phase, A5 executed malware in a sandbox environment, recorded network threats, and generated network intrusion detection system signatures.

Hybrid methods

In hybrid detection methods, clients collect information related to applications installed on the device and send the information to a remote server. The remote server then analyzes the log files using the proposed detection algorithm. This approach compensates for the drawbacks of the online and offline detection methods. However, users have to agree in advance on which client module will be used to send user information to the remote server.

Burguera et al. (2011) proposed a lightweight client called Crowdroid, which monitors system calls, makes a frequency table using the system calls, and sends them to a centralized server. The remote server then identifies malicious behavior in a statistical manner and detects malware utilizing a *K*-means algorithm. Crowdroid extracts system calls by exploiting Strace. However, Strace executes its functionality after installation. Thus, Crowdroid cannot detect malicious behavior during the installation process, and depends on the functionality of Strace. Isohara et al. (2011) proposed a kernel-based behavior analysis system that consists of a system call log collector on an Android device and a log analyzer on a remote server. The client collects system calls generated at the installation time and sends the logs to a remote server. The remote server then compares patterns in the logs with 16 pre-defined patterns. Since the pre-defined behavior patterns focus mainly on malicious behaviors, such as restricted information leakage, jailbreak, and abuse of root privileges, their system cannot detect malicious behavior such as sending a premium-rate SMS and calling premium-rate code. Furthermore, their approach does not guarantee sufficient scalability.

Schmidt et al. (2009) proposed a collaboration mechanism for Android platform security comprising a log collector on the device and a remote analyzer. In their proposed system, the client monitors the behavior of the malicious application at installation time, runs an analysis based on the similarity of the function call set used, exchanges the results of the analysis with neighboring devices, and performs collaborative malware detection.

Profiling modeling

Overview

In the real world, the criminal profiling process uses two approaches: inductive and deductive. Inductive profiling is a bottom-up approach and relies on generalizing behavioral patterns from a statistical analysis of the data of convicted offenders. The profilers make generalized behavior patterns by analyzing the correlation between a crime and a criminal's characteristics (e.g., job, education level, and income level). Deductive profiling, on the other hand, does not rely on generalities from sample groups. Deductive profiling is a top-down approach based on deductive logic and argues from the specific to the general (Rogers, 2003). As illustrated in Fig. 1, we adopt the Federal Bureau of Investigation (FBI) profiling approach, following the inductive profiling, for mobile malware analysis. Our profiling method consists of a four-stage process: 1) data assimilation, 2) malicious behavior definition, 3) malware attack scenario sketch, and 4) profile generation.

Data assimilation: In this stage, we gather malware-centric and malware creator-centric information from multiple resources (e.g., AV technical reports, web-

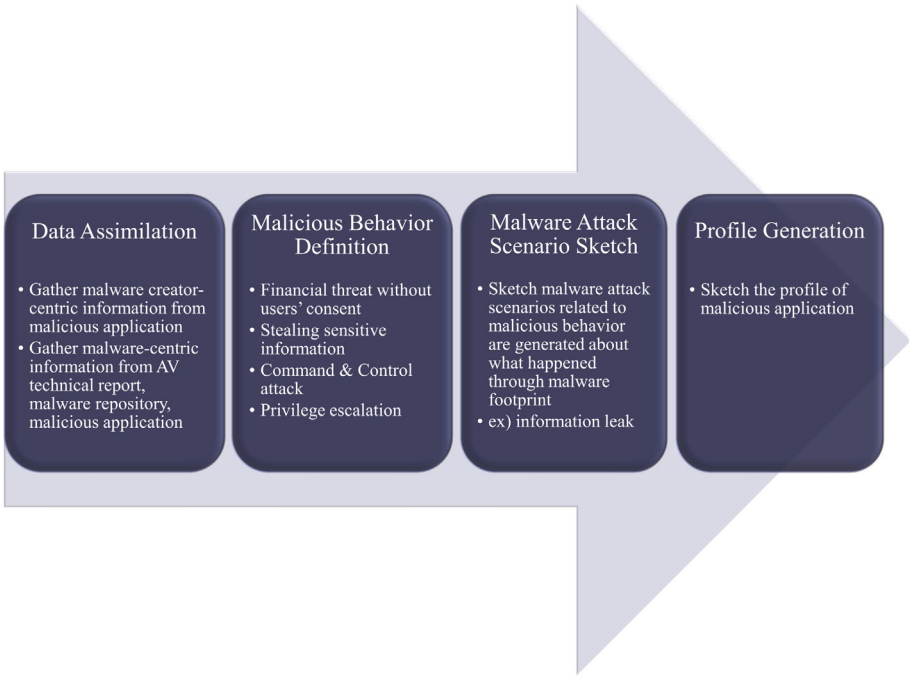

Fig. 1. Overall malware analysis process of Andro-AutoPsy.

crawling, malware repositories, community sites, and malware). Table 2 shows the feasible information parsed from an executable file of the malware.

Malicious behavior definition: Endpoint mobile devices can be infected by malware through many routes, usually as a downloaded application from the official market, but also through visits to alternative markets, spam, malicious SMSs, and malware-bearing advertisements (McAfee, 2013). Behaviors of the malware after invading a victim's device can be categorized according to the main purpose of the creator's attack. Malicious behaviors are classified into privilege escalation, remote control, financial charge, and information collection (Zhou and Jiang, 2012). The malicious behaviors are listed as monetization, information stealing, mobile botnet, and root privilege acquisition (Seo et al., 2014). Through examining these behaviors, we define malicious behaviors that can be used straightforwardly in our anti-malware system. As shown in Table 3, the malicious behaviors we proposed in this paper are categorized into malware creator-centric and malware-centric features. The malware creator-centric feature is built as the suspect list of the malware-creator. The malware-centric features include financial fraud by hiding SMS notification from the user, C&C attack by hiding SMS notifications from the user, the usage of system commands on root privilege for leveraging forged files, the leakage of sensitive information, and the likelihood ratio of critical permission.

Malware attack scenario sketch: We define and summarize distinct behavioral characteristics of malware at the malicious behavior definition stage. At the malware attack scenario sketch stage, we generate a scenario of the events that occurred during malware execution and analyze the relation between the malicious behavior of malware and the damage level at the victim's side (e.g., information leaks, damage by premium-rate without users' consent). We draw a flowchart based on the scenarios that utilizes a rule-based algorithm, as illustrated in Fig. 2.

Table 2
Candidate features for Malware analysis as a result of data assimilation.

Category	Collected information	Resources
Malware creator-centric	Certificate information, Hash digest of a certificate	CERT.* or alias.*
Malware-centric	Package name, Package version, Minimum SDK version, Requested permission	Android Manifest.xml
	API-related permission, Content provider-related permission, Intent-related permission, URL information, System command, Malicious API sequence, Usage of dynamic loading method, Usage of crypto method	Opcode of disassembled code
	Existence of forged file	Asset, res, or lib folder
	Hash digest of .dex, Hash digest of AndroidManifest.xml	MANIFEST.MF

Table 3
Proposed features of malicious behaviors.

Category	Proposed features of malicious behaviors
Malware creator-centric	The number of malware family or variant created by one Serial Number > Threshold (T_S)
Malware-centric	Concealing C&C management SMS, Concealing confirmation code SMS for premium-rate, Usage of system command AND Existence of forged files, Likelihood ratio of critical permission > Threshold (T_L), Leakage of system and personal information

Profile generation: We finally construct the sketch of malware at the profile generation stage. We explain the process of profile generation in more detail in the following subsection.

Profile generation

In the literature of traditional malware research related to personal computers operating Microsoft Windows, Bayer et al. (2009) proposed a method for scalable behavior-based malware clustering. The method contributes to the theoretical foundations of malware analysis by discussing behavior-based profiling formally. Given the relevance of that study to ours, we review Bayer's definitions of profiling for the completeness of our presentation, and incorporate details specific to our proposed system in the following.

Definition (profiling). A profiling P is defined by four tuples as $P = (O, OP, \Gamma, \Delta)$, where O is the set of all objects and OP is the set of all operations, which is represented in nested dictionary form as {name: {means: attribute}}. $\Gamma \subseteq (O \times OP)$ is a relation assigning one or several operations to each other, and $\Delta \subseteq ((O \times OP), (O \times OP))$ represents the sequence-unrelated set, which is equivalent to integrated footprints.

Object: An object represents an abstract functionality that malware samples need for executing the malicious behavior. We classify objects into a malware creator-oriented object type and a malware-oriented object type. The malware creator-oriented object type constitutes the identification of the malware creator, and the malware-oriented object type constitutes malicious behavior. We formally define the objects as follows.

```
Object      ::= Object-type
Object-type ::= Identification of malware creator |
                Malicious behavior
```

Operation: An operation represents a concrete malicious behavior. Formally, an operation comprises operation-name, operation-means, and operation-attribute. Operation-name acts as the identifier for malicious behavior. Operation-means is the attack method of malware, such as usage of the system command and the existence of forged files. Operation-attribute is a meaningful value that the malware wants to obtain. When we define the usage of system commands for leveraging forged files as operation-name, operation-means is a system command, and operation-attribute is *chmod* and *su*. We formally define an operation as follows.

```
Operation       ::= { Operation-name : { Operation-means :
                      Operation-attribute } }

Operation-name  ::= Information of malware creator |
                    Usage of system command for leveraging
                    forged files | Concealing received SMS |
                    Retrieving sensitive information |
                    Critical permission set | Likelihood
                    ratio of critical permission

Operation-means ::= The serial number of a certificate |
                    System command | Forged files |
                    Malicious receiver intent filter |
                    Malicious API | Requested critical
                    permission | API-related critical
                    permission ...| etc.
```

For example, we represent the profile (Δ) of a malware as follows: (Identification of malware creator) $\times$ ({Information of malware creator: {Serial number of a certificate: '93:6e:ac:be:07:f2:01:df'}}), (Malicious behavior) $\times$ ({Critical permission set: {Requested critical permission: {'RECEIVE_SMS', … ,'SEND_SMS'}}}).

Data exploration

In order to estimate the behavior pattern of malware, we adopt the serial number of a certificate, malicious API sequences, permission distributions, usage of system

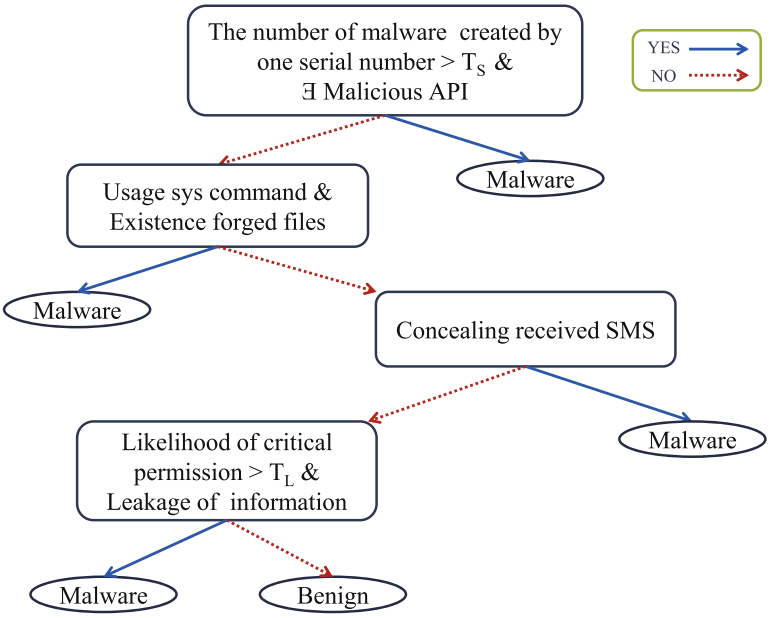

Fig. 2. If-then rule-base for malicious behavior detection.

commands, existence of forged files, URL information, usage of dynamic loading, usage of crypto methods, and hash digests of files as feature vectors. We determine metrics for malware analysis according to the data exploration. We used 9990 malware samples and 109,193 benign samples in our experiments.

Serial number of a certificate: When released to the Google Market, an application is signed with the application creator's private key, and a standard certificate of the public key is generated. There are blanks for writing the creator's name, organization, and location, when generating a certificate. However, the application creator can fill the blanks with false information, since the process has no confirmation steps. The certificate receives a unique serial number according to the RFC 2459,[1] X.509 standard. Thus, one can check whether certificates are the same or not by comparing the serial number. We notice that there may be certain serial numbers that are detected in many malware samples. We explore the serial number of a certificate as a feature. With the dataset of malware samples, we extracted the serial number of the certificate in each sample and found the distribution of the serial numbers. A total of 1834 serial numbers were observed in all the malware samples, but only 86 unique serial numbers comprised 70% of the samples. Surprisingly, 50% of malware samples were assigned 17 serial numbers. This means that malware creators frequently use particular serial numbers. Among the 1864 serial numbers, 370 numbers generated more than 2 families or variants of malware. For a fast primary screening, we made a blacklist of 370 serial numbers. This list takes much less time in screening than the more inclusive list of 1864 serial numbers. Moreover, '93:6e:ac:be:07:f2:01:df' and 'b3:99:80:86:d0:56:cf:fa' were deleted from the list, since they constitute a serial number of a standard test key for native applications that are built on a device or an emulator. As a result, a serial number blacklist having 368 numbers was constructed.

Malicious API sequence: We explore the Application Programming Interface (API), as documented in Android SDK, as a feature. The API is a set of functions provided to control the principal actions of Android platform conveniently. It is much more efficient to consider certain APIs often used by malware than to extract all the APIs from the source code of an application. Seo et al. (2014) analyzed malware samples and determined the malicious APIs frequently used by malware. They listed malicious APIs and compared the usage frequency of their malware samples and benign applications. We manually collected additional APIs by checking all the APIs in Android SDK that might operate in a similar way to the malicious APIs determined by Seo et al. (2014). These APIs are involved in collecting the user's personal information or the device information, accessing Web sites, sending and deleting SMS, accessing and reading the content provider, etc. The details of the malicious APIs we defined are listed in the appendix. A malicious API sequence represents the behavior pattern of malware. Although malicious profiles seem to be similar to each other, the APIs used in malware vary according to the malware creators. Therefore, we chose malicious API sequences as a feature for malware analysis.

Permission distribution: We explore the permission distribution as a feature. About 100 permissions are provided on the Android platform to inform the user of what actions will be performed and which resources will be accessed by the application to be installed. Sarma et al. (2012) compared two datasets, applications from the Android Market and malicious applications obtained from malware repositories, and analyzed the distribution of permissions requested by each dataset. They determined 26 risky permissions that are critical in terms of security and privacy. We also used only the aforementioned 26 permissions in our system. However, one

[1] http://www.rfc-editor.org/rfc/rfc2459.txt.

of the permissions, INTERNET, was excluded, since this permission is required by most of the applications. We considered that the relatively small difference in the ratio of permissions of Android Market applications and malware does not play an important role in detecting malware. INSTALL_PACKAGES, a permission usually used for installing a new package downloaded from a server, is included instead. While requested permissions are notified before installation, there are other methods of extracting permission specification by analyzing an API call-graph (Felt et al., 2011; Au et al., 2012). The requested permissions declared in *Androidmanifest.xml* are not in fact necessary for the application functionality. Au et al. (2012) brought the current permission system into question, and stated that it was incomplete. For this reason, the study specified the list of permissions required for every API call and provided the permission mappings. In our system, we leveraged only 26 critical permissions when applying PScout mapping which is a mapping table that lists up permissions required to use in an API (Au et al., 2012), along with the requested permission. We named this feature API-related permission. Benign applications and malware applications have different tendencies of requesting permissions. Malware often requests more permissions than benign applications, or often requests permissions that have risks related to privacy or monetary problem. In Sarma et al. (2012), it is stated that there is clearly a difference between the two groups in terms of the frequency of requesting permissions, such as SEND_SMS or READ_PHONE_STATE. We analyzed the distribution of the critical permissions in each benign and malware sample to calculate the likelihood of the permissions. The distribution of the critical permissions depends on the application's category (benign or malware).

In Table 4, the percentages of critical permissions for each category, requested and API-related, in benign samples and malware samples are presented. The permissions in Table 4 are listed in accordance with the percentage value and the difference in values of benign and malware. Using the probabilities and applying a Naïve Bayes classifier, we can calculate the likelihood of the permissions for each category. Peng et al. (2012) used the critical permissions, applying Naïve Bayes models to score the risk of an application. The permissions should be relatively independent in order to multiply each probability of permission. In Au et al. (2012), it was shown that most of the Android permissions have no correlation with other permissions from the perspective of API usage. They recognized that only 15 pairs of permissions have a dependence level among all the permission sets. Most of the critical permissions we used are not included in these pairs; only one pair (ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION) is high correlated each other. In order to reduce the complexity of computation, we also assume that they are relatively independent from the perspective of API usage.

Let n and m be number of applications and number of critical permissions respectively. The permission vector for application i is $a_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,m})$, where

$$a_{i,j} = \begin{cases} 1 & \text{if application } i \text{ uses critical permission } j \\ 0 & \text{otherwise} \end{cases}$$

which are independent variables because all critical permissions are relatively independent. In addition we put $c_i \in \{\text{benign, malicious}\}$ which indicates the category of application i.

Then,

$$P(c_i|a_i) = P(c_i|a_{i,1}, a_{i,2}, \ldots, a_{i,m}) = \prod_{j=1}^{m} P(c_i|a_{i,j})$$

Using Bayes' Theorem, the conditional probability of category c_i given variable $a_{i,j}$, which informs about the usage of the critical permission, can be written as:

$$P(c_i|a_{i,j}) = \frac{P(a_{i,j}|c_i) \cdot P(c_i)}{P(a_{i,j})}$$

Then, the ratio of probabilities is calculated as:

$$\frac{P(malicious|a_{i,j})}{P(benign|a_{i,j})} = \frac{P(a_{i,j}|malicious) \cdot P(malicious)}{P(a_{i,j}|benign) \cdot P(benign)}$$

We assume $P(c_i = malicious) = P(c_i = benign)$; i.e., there is no information about the category of the application, and therefore the application is supposed to have a variable of any category value following a uniform distribution. By multiplying the probabilities of m permissions, the likelihood ratio Λ is:

$$\Lambda(a_i) = \frac{P(c_i = malicious|a_i)}{P(c_i = benign|a_i)} = \prod_{j=1}^{m} \frac{P(a_{i,j}|c_i = malicious)}{P(a_{i,j}|c_i = benign)}$$

If one of the conditional probabilities is zero, then the whole multiplication becomes zero. To avoid the case that a denominator becomes zero, the conditional probabilities are calculated using the Laplace estimator (Leung, 2007),

$$P(a_{i,j}|c_i) = \frac{\sum_{i=1}^{n} a_{i,j} + 1}{n + 2}$$

When the likelihood of a malicious application increases as compared to that of the benign application, the value of Λ increases. Malware can be detected by comparing the

Table 4
Critical permission distribution in benign and malware samples (%).

Permission	Requested permission		API-related permission	
	Benign	Malware	Benign	Malware
ACCESS_COARSE_LOCATION	19.55	52.31	22.17	56.32
ACCESS_FINE_LOCATION	18.92	45.65	19.92	55.83
SEND_SMS	1.44	42.28	1.49	35.69
READ_PHONE_STATE	25.96	96.85	12.20	69.94
RECORD_AUDIO	3.00	16.29	2.84	15.58

likelihood ratio using some predefined threshold value T_L. Therefore, we choose the likelihood ratio (Λ) as a feature for malware detection.

Intent: Android applications do not have the unique entry that programs usually have on other operating systems. Android applications are made up of Android components: activity, service, broadcast receiver, and content provider. Activity is a UI component related to the screen, while service is a background process that is invisible to the user. Broadcast receiver waits for the signals from the system and wakes up the appropriate activities. Content provider plays the role of an intermediate unit for sharing data between applications. These four components work individually, and each component delivers messages to other components to allow cooperation. The delivered message is called "intent". Intent transfers from one activity to other activities, containing specific instructions about what the application wants. We checked the intent-specific information to detect malware that conceals SMS notifications. These malware receive a notification of SMS with the highest priority, and then prevent the delivery of the message to other applications. This malicious behavior can be found by searching the intent filters in *Androidmanifest.xml*.

Usage of system commands: We explore the usage of system command as a feature. The commands commonly used by malware are listed by Seo et al. (2014). We rearranged the list by excluding the commands used in only a few malware samples of our dataset. We found that '*chmod*', '*insmod*', '*su*', '*mount*', '*bash*', '*killall*', '*reboot*', '*mkdir*', '*getprop*', '*ln*', and '*ps*' are commands often used by malware, and they run on rooted Android devices. If these strings are extracted from the source code of an application, we mark the application as a malware. The commands are executed after the malware obtains the root privilege of the device. In addition, our list contains *gingerbreak* and *rageaginstthecage* that are root exploit codes.

Existence of forged files: When the extension of a file is different from the magic number of its header information, we regard the target file as a forged file. If the actual magic number of the header information is *.elf*, *.apk*, or *.jar*, although the extension of a target file is that of a graphics file, such as *.jpg*, *.gif*, and *.png*, we call the target file a forged file. In order to evade the detection methods of AV vendors, mobile malware, like traditional malware, hides script, including the execution of malicious behavior, in a normal-looking file, and executes the malicious behavior by loading that file. However, some benign applications change the extension of the updating file (e.g., .elf, .zip) to another extension (e.g., .so, .dat) for security reasons. In that case, the false positives due to file forging tend to be high. In order to reduce the false positives, we consider the intersection of the usage of a system command and the existence of a forged file as a feature for malware detection. Forged files containing executable exploit codes are usually located in the assets, lib, and res folder, and malware needs system commands for executing the exploit code (Seo et al., 2014). Therefore, the intersection of the usage of system commands and the existence of forged files is a good metric for detecting malware that overcomes the aforementioned shortcoming.

MISC (**miscellaneous**): We explore other footprints as features: URL information, usage of dynamic loading, usage of crypto methods, and hash digests of files. We cannot generalize URL (or destination) information as a malware detection rule, since the difference in one bit (or one letter) represents a different destination; i.e., that can effect only perfect string matching. Accordingly, URL information as a feature has no significant effect on the detection of malware.

The Android platform supports dynamic loading methods for binary and native machine code. Although dynamic loading methods for other binary files (.dex or .jar) may be misused for loading malicious code, benign applications leverage dynamic loading methods for flexible memory allocation or the extension of dynamic functionality during runtime execution. In order to enhance the performance or for other capabilities, some application developers use native code on JNI (Java Native Interface). JNI is a programming framework that enables a Java Virtual Machine (JVM) to interact with native applications and libraries written in the C language. Dynamic loading methods for native machine code appear in benign applications and malware. We found the crypto methods in benign samples, since application creators use these methods for ensuring data confidentiality and integrity, while malware creators use them for concealing victims' hijacked information. Thus, the usage of crypto methods as a feature has no significant effect on the detection of malware. Hash digest based on malware detection methods is prone to high false negatives, since the difference in a single bit makes a new file name different from its origin. In short, none of the features described in this subsection is sufficient to allow the detection of malware.

Summary: As a result of data exploration, we choose five footprints as features: the serial number of a certificate, malicious API sequence, permission distribution (critical perm ission set, likelihood ratio), intent, and the intersection of the usage of system commands and the existence of forged files.

Andro-AutoPsy: an anti-malware system

In the following, we review the design and operation of Andro-AutoPsy, a hybrid system for malware analysis. In subsection Overview, we provide an overview of our system. In subsection Integrated footprints extraction process, we review the extraction process of integrated footprints used for our profiling phase. In subsection Decision process, we review the decision process for determining the nature of an application based on the profiling.

Overview

As illustrated in Fig. 3, we propose a hybrid anti-malware system that consists of a client application on a mobile device and a profiling and analysis system on a remote server. A client application on the mobile device collects the installed application's information, and sends it to a remote server; the client application sends only application-centric information such as the hash digest of *.apk* files and package name. If the remote server cannot crawl that application, the client application sends the application package file (*.apk*) to the remote server. The remote server analyzes the malicious application and decides whether or not it is malicious, based on integrated footprints.

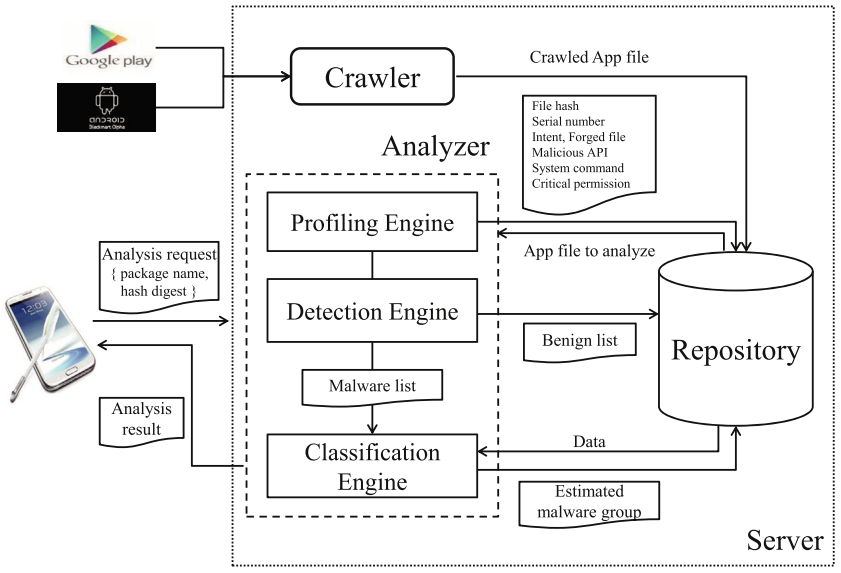

Fig. 3. Overall procedure of Andro-AutoPsy.

The remote server consists of three components: a crawler, a repository, and an analyzer. The crawler component crawls applications acquired from repositories, such as official and alternative markets. The crawled applications are then passed to the repository component, which runs a duplication test by comparing the hash digests of the *.apk* files with each other. If the crawled application is a duplicate, it is discarded; otherwise, the repository component sends the application to the analyzer component. After completing the analysis, the analyzer component sends the analysis results to both the repository component and the client application. Upon receiving the analysis results from the remote server, the client application displays the result on the screen to the user. The repository component searches in its database as soon as the repository component receives an analysis request from the client. If the repository component does not have analysis results that satisfy the client's request, it fetches the crawler component. The analyzer component has two processes: an integrated footprints extraction process and a decision process. The integrated footprints extraction process is composed of a profiling engine, and the decision process is composed of two engines: a detection engine and a classification engine. In the following, we review the extraction and decision processes.

Integrated footprints extraction process

Profiling Engine: The Profiling Engine (PE) disassembles an *apk* file, which is an Android package that comes as a compressed file that usually contains four folders (*META-INF*, *lib*, *res*, and *assets*), and three files (*AndroidManifest.xml*, *classes.dex*, and *resources.arsc*). The serial number is extracted from the *META-INF* folder which includes information of the serial number of each application and the hash digest of each file. The PE extracts the package name, requested permission, component name and intent in *AndroidManifest.xml*, and disassembles the *classes.dex* file into *.smali* code in the form of opcode by exploiting Apktool (Apktool, 2010). Apktool is known-well integrated solution that can decode resources to nearly original form. We used the latest version of Apktool. Our system sorts parsed components in ascending order. For efficiently searching and parsing significant information, our system searches only files with the same component name and all files in the folder including the files with the same component name. In addition, our system checks whether or not a forged file exists in *assets*, *res*, and *lib* folders. Malware behaves maliciously, by hiding malicious script in forged files. For example, malware disguises itself as a benign application: if the user clicks the fake application icon, it executes the malicious script hiding in the forged files (Paganini, 2013). Often, such malware replaces the legitimate banking application to steal sensitive information and monitor victim's device in the background. By following the codes used by components, the system extracts malicious APIs, system commands, and API-related permissions. When retrieving forged files, our system compares the file extension with the magic number of the file header, and decides whether or not it is forged. After capturing the integrated footprints of a malicious application, the PE creates the profile of each malware as a dictionary structure of the Python language for an efficient membership test, as explained in Section Profile generation, and passes it to the *Detection Engine*.

Decision process

The decision process consists of two modules: detection engine, classification engine. In the following we elaborate on each of those modules.

Detection engine

Our detection engine (DE) decides whether a given application is malicious or not based on the behavior patterns. The DE contains detection rules which are composed of the serial number list of malware creator, rule of checking the usage of system commands for leveraging forged files, rule of checking concealing received SMS notification, rule of checking leakage of sensitive information, and likelihood ratio of requested and API-related

critical permission. Also, our engine contains rules of detecting smishing (SMS phishing) applications.

The pseudo code in Algorithm 1 shows how the DE decides whether it is malicious or not. The detection algorithm starts by comparing the serial number of each application against the blacklist for a fast scanning. In our dataset, there are applications that are signed by the same serial number in a blacklist but do not exploit any malicious APIs as in Section Data exploration. In this case, we exclude these applications to avoid over-detection.

Secondly, the algorithm checks the usage of the system commands for leveraging forged files. These commands, which can be run on a rooted device, are usually found in malicious codes. The next detection step is finding malware that conceals received SMS notification. The purpose of malware that behaves in this manner is to subscribe to premium services that confirm and notify using SMS, or by receiving an SMS for a command and control (C&C) message. These applications use API methods, such as *getOriginatingAddress()*, *getMessageBody()*, and the *getDisplayMessageBody()* of *SmsMessage* class, to receive SMSs. In addition, they request highest priority for SMS receiving intent, and call *abortBroadcast()* to hide a notification of SMSs sent to other applications and the users.

The step checks whether or not an application uses the aforementioned methods and an intent filter to detect malicious behavior. Similarly to premium-rate SMS, a smishing application receives SMSs for C&C from a remote server, sends hijacked sensitive information (e.g., contacts, SMS content, call logs, certificate for financial transaction in South Korea). In this case, the smishing application

Algorithm 1: Malware detection algorithm

Input : Application $= \{A_1, A_2, ..., A_n\}$
Output : {malicious, benign}

Initialization:
B $= \{SN_1, SN_2, ..., SN_m\}$ ▷ Serial number blacklist
SN_{A_k} ▷ Serial number of Application k
API_{A_k} ▷ API of Application k
MAPI $= \{MAPI_1, MAPI_2, ..., MAPI_k\}$ ▷ Malicious API dictionary
for $i \leftarrow 1$ **to** n **do**
  if $SN_{A_i} \in B$ *AND* $API_{A_i} \in MAPI$ **then**
    $A_i \leftarrow$ malicious;
  else
    if $API_{A_i} \notin MAPI$ **then** $A_i \leftarrow$ benign;
    else if $\exists$ *system commands in* A_i *AND* $\exists$ *forged files in* A_i **then**
      $A_i \leftarrow$ *malicious*;
    end
    else if $\exists$ *Code to conceal received SMS notification in* A_i **then**
      $A_i \leftarrow$ malicious;
    end
    else if $\exists$ *Behavior pattern of smishing in* A_i **then**
      $A_i \leftarrow$ malicious
    end
    else if *Likelihood ratio of requested critical permission* $> T_L$ *AND* *Likelihood ratio of API-related critical permission* $> T_L$ *AND* *Calibrator to reduce false positives* **then**
      $A_i \leftarrow$ malicious;
    end
    else
      $A_i \leftarrow$ benign;
    end
  end
end

Table 5
Similarity metric for each behavior factor.

Behavior factor	Contents	Similarity metric	Update
Malicious API sequence	Information retrieving-related API (System info., Personal info.), Transmission-related API (Data network, SMS, Call), Dynamic loading-related API	Needleman-Wunsch algorithm [0, 1]	X
Usage of system commands for leveraging forged files	Privilege escalation commands, File/Directory ownership commands, Execution commands, Process commands	Jaccard coefficient [0, 1]	O
Usage of critical permission	Requested critical permissions, API-related critical permissions	Levenshtein distance [0, 1]	O

conceals the received SMS notification after the malicious behavior is executed.

In the final step, the algorithm calculates the likelihood ratio under the given critical permissions. Two likelihood ratios are obtained using the requested critical permissions and the API-related critical permissions. If the values are both higher than a threshold T_L, then the application is determined to be malware. To compensate for the limitation of the permission-based detection method, the algorithm checks whether or not an application sends SMSs, calling *abortBroadcast()*, or transmits two more sensitive items of information, such as bookmark, device ID, phone number, serial number of SIM card, and location of the device. The use of APIs collecting more than two kinds of information and transmitting information is determined to be a sufficiently suspicious behavior, concluding that the application is malicious.

Classification engine

The appropriate similarity metrics are applied to different types of behavior factors. The classification engine (CE) computes the similarity score between the profile of a malicious application and the representative profile of each malware family. The CE then assigns the malicious application to the most similar group. The representative profile of each malware family has to depict the unique and common behavior patterns of each malware family. Then, the CE chooses one of the methods for updating the representative profile as follows:

1. **AutoPsy-INT**: The first update method is the intersection. The representative profile for each malware family is updated by the intersection of the profiles of members in each subgroup. In the update method of AutoPsy-INT, as the number of members of each malware family increases, the representative profiles decrease.
2. **AutoPsy-UNI**: The second update method is the union. The representative profile for each malware family is updated by the union of the profiles of the members in each subgroup. In the update method of AutoPsy-UNI, as the number of members of each malware family increases, the representative profiles increase.

We define the similarity score as the weighted sum of the similarity of three behavior factors. The similarity score between the profile of a malicious application and a representative profile for each malware family is given by:

$$S = \sum_i w_i \cdot BFS_i \quad where \sum_i w_i = 1, \tag{1}$$

where BFS_i and w_i are the similarity and weight of behavior factor i, respectively. The behavior factor similarity (BFS) is composed of three parts: similarity of malicious API sequence, usage of system commands for leveraging forged files, and usage of critical permission set, including requested and API-related permissions. We chose to set the weight (w_i) at 1/3, which is the arithmetic mean.

Table 5 shows the similarity metric to apply to each behavior factor. We compute the similarity score for each behavior factor as follows:

1. We compute the similarity score for a malicious API sequence. According to a pre-defined malicious API dictionary, we create a unique API sequence of each malicious application. In order to compare the malicious API sequence of the target with the others, we transform the API method into ASCII (American Standard Code for Information Interchange) code. The transformed letter sequence represents the behavior pattern of each malicious application. In bioinformatics, a sequence alignment algorithm is a method of arranging the DNA, RNA, or protein to identify the regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences (Mount, 2004). Sequence alignment algorithms are classified into global alignment and local alignment algorithms. The global alignment algorithm attempts to find the longest path between vertices $(0,0)$[2] and (n,m) in the edit graph, whereas the local alignment algorithm attempts to find the longest path among pairs between arbitrary vertices (i,j) and (i',j') in the edit graph. In the case of malware analysis, the similarity comparison between malicious API sequences refers to the similarity of the global sequence rather than the local sequence. We adopted a global alignment algorithm, the Needleman-Wunsch algorithm (Needleman and Wunsch, 1970), in our study, which uses a dynamic programming

[2] Let X and Y be two strings; the length of X is n and that of Y is m. In order to leverage dynamic programming, we apply tabulation on an (n, m) matrix of the edit distance.

algorithm to find the optimal global alignment of two sequences. The value of the similarity score is $[0, 1]$.

2. We compute the similarity score for the usage of malicious system commands by applying the Jaccard coefficient. The Jaccard coefficient is defined as the number of elements in an intersection divided by the number of elements in a union. The order of the malicious command is not under consideration. We define the malicious system commands as: '*chmod*', '*insmod*', '*su*', '*mount*', '*bash*', '*killall*', '*reboot*', '*mkdir*', '*getprop*', '*ln*', *and* '*ps*'. The value of the similarity score is [0, 1].
3. We compute the similarity score for the usage of critical permissions as the average of the similarity for requested critical permissions and API-related critical permissions. We calculate the similarity of the critical permission set by applying the Levenshtein distance. This metric calculates the minimum number of character edits required to make two strings same. It is meaningless to consider the order of critical permissions, and therefore, we applied the Levenshtein distance after sorting the strings. A value of similarity is calculated as the number of edits over the maximum length of two strings. The value of the similarity score is $[0, 1]$.

The CE classifies a malicious application into the group with the highest similarity score, which is at least 0.70. We assume that 0.70 is a sufficiently high score to determine that two signatures are similar based on empirical tests. Whenever a new malware sample is queued into our anti-malware system for inspection, the CE continuously updates the representative profile according to the pre-chosen update method.

Performance evaluation

In the following, we demonstrate the performance of Andro-AutoPsy by highlighting aspects of its implementation and testing it on various real-world mobile malware samples.

Implementation

Our anti-malware system is composed of a mobile device and a remote server. The client application was installed on a mobile device (Vega IM-A870S) running on Android 4.1.2, and three components – a crawler, a repository, and an analyzer – were installed on the remote server. The remote server had an Intel(R) Xeon(R) X5660 processor and 8 GB of RAM with a 64-bit Microsoft Windows 7 Enterprise operating system. We performed all experiments in a hypervisor-based virtualization environment.[3] We implemented our anti-malware system using Python high-level programming language (as scripts). The client component on the mobile device was implemented in the form of an application and communicated with the remote server. The crawler component sent the package name to GooglePlay and downloaded the target application. The repository component stored the profile of each application in a database. The remote server was composed of the PE, DE, and CE. Among these, the PE was implemented as Python script coupled with Apktool.

Table 6
The summary of malware and benign samples.

Category	Family	Quantity	Family	Quantity
Malware (9990)	AdWo	2807	DroidKungFu	112
	Boxer	2138	Mseg	103
	Dowgin	936	SmsReg	103
	AirPush	495	FakeBattScar	99
	Gappusin	487	GingerMaster	94
	SMStado	288	FakeNotify	86
	Counterclank	276	PremiumSMS	63
	Wapsx	273	JiFake	53
	OpFake	240	Boqx	52
	SMSAgent	202	DroidDream	51
	Kuguo	192	Plankton	49
	Ropin	192	DroidRooter	46
	SmsPay	183	SMSSend	46
	Youmi	173	Agent	20
	Kmin	119	Utchi	12
Benign (109,193)	Application	91,304	Game	17,889

Experimental setup

For performance evaluation, 9990 malware samples[4] consisting of 30 malware families were collected from January 2013 to April 2014 through malware repositories such as VirusShare (VirusShare, 2014), Contagio (Contagio, 2011), and 109,193 benign samples were collected through GooglePlay for the same periods. In the real world, malware comprises a small fraction of all Android applications, and therefore, it is reasonable to use a larger set of benign samples to mimic a realistic scenario. Duplicate malware samples were eliminated according to SHA 256, and duplicate benign samples were also eliminated according to SHA 256. Establishing a ground truth for malware samples and their labels is a difficult problem. A common way used in the literature for establishing a consensus is by limiting studied samples to those detected by at least a certain number of scanners. To this end, we choose malware samples diagnosed by at least 10 antivirus vendors included in the VirusTotal dataset (VirusTotal, 2013). We used the textual description of malware produced by F-Secure (F-Secure, 2013). The statistics of the dataset we used are shown in Table 6.

Our system was configured with $T_L = T_S = 1$. It is reasonable to set the likelihood ratio of critical permission (T_L) to 1, which means the likelihood of malware is greater than that of benign applications. It is also reasonable to set T_S to 1, according to data exploration in Section Data exploration.

For the validation of our work, we used 5-fold cross-validation to evaluate the performance in our experiments. The k-fold cross-validation randomly partitions the dataset into k equal size sub-sample set. A single sub-sample set was used as the validation data for testing the model, while the other $k - 1$ sub-sample sets were used as

[3] VMWare ESXi; http://www.vmware.com/.

[4] Our dataset is available at http://ocslab.hksecurity.net/andro-autopsy.

training data. We repeated the cross-validation process five times and averaged the performance over the five experiments.

In the machine learning literature, there is no single answer for what is best for the value k in a k-fold-cross-validation scheme. In particular, overfitting is a side effect of a larger k, which results in a higher precision than a smaller k for the particular dataset used in an experiment. In the security literature, both 5 and 10 are used, and 5 is widely used when evaluating systems under aggressive settings. To that end, we choose $k = 5$, knowing that $k = 10$ would provide higher precision.

Experimental results and analysis

Our performance evaluation focused on the effectiveness and the efficiency of malware detection and classification. We demonstrate that our system performs precisely in detecting and classifying malware families.

Effectiveness of malware detection

We demonstrate that our proposed method provides an effective metric for distinguishing malware from benign samples. Table 7 is a confusion matrix that shows the performance of the detection algorithm. As a result, 535 benign samples, corresponding to 0.05% of all benign samples, were detected as malware, and 125 malware samples, corresponding to 1.25% of all malware samples, were detected as benign. When designing an anti-malware system, one important factor, which we should also consider, is its ability to discriminate between malware and benign applications. Anti-malware systems must detect malware with small errors in terms of false positive and false negative.

Explaining false detections: As shown in Table 8, to find the reason for which some of the benign samples were determined as malicious in our system, we conducted an in-depth analysis. Interestingly, we found that some benign samples had suspicious codes that exploited system commands with forged files (71), related to the malicious behavior of concealing received SMSs (84). Moreover, we found that other benign samples had a pattern of smishing (8), and a likelihood ratio of critical permission of more than T_L (14). To understand whether or not other anti-malware systems and scanners considered these benign applications to be malware, we uploaded these suspected GooglePlay samples to VirusTotal (VirusTotal, 2013) and checked the scanning results of various anti-virus vendors. We found that 161 out of the suspicious benign samples (accounting for about 30.1%) were diagnosed as malware. The high rate of misclassification of benign applications is, however, understandable, given the various potential reasons for such an infiltration of gray area applications into the market place. Malware creators can exploit hijacked or fraudulent developer accounts of GooglePlay by buying a verified developer account for 100$, and easily acquire malicious code generation kits from the black market (Krebs, 2013). We also conducted an in-depth analysis to understand the false negatives produced by Andro-AutoPsy. Most of the false negatives bypassed our defined pattern rules. When we adjusted tight rules for reducing false negatives, our system produced more false positives, which represents the trade-off between false positives and false negatives. In the case of some false negatives, our system could not find malicious behavior in parsed footprints. However, the false negative rate is low; about 1%.

Author-based feature and fast scanning: Additionally, we evaluated our proposed method based on the only serial number in a certificate to demonstrate the effectiveness of serial number-based detection. 7311 malware samples, corresponding to 73.18% of all the malware samples, were filtered as malicious, and 358 benign samples, corresponding to 0.32% of all the benign samples, were unfortunately filtered as malware. Further, we conduct a multi-factor analysis to determine what feature sets are relatively more significant in detecting malware. We measure the detection decline rate by each feature attribute set against the 9990 malware samples. Table 9 shows how much each feature attribute has an effect on detecting malware. The base case we use (case 0) corresponds to the scenario where all feature sets are utilized. In the first case (case 1; which means the feature set indicated by the attribute set is removed) the rate is degraded by 34.83% compared to the base case, which corresponds to the largest decline; highlighting that this feature set is the most

Table 7
Confusion matrix of malware detection.

Category		Actual class	
		Malware	Benign
Estimated class	Malware	9865	**535**
	Benign	**125**	108,658

Bold text indicates the number of misclassified samples in our experiments.

Table 8
The number of detected samples for each detection rule.

Detection rule	Malware	Benign
Blacklist of serial number	**7311**	**358**
Usage of the sys' commands for leveraging forged files	37	71
Concealing received SMS	275	84
Pattern of smishing	1	8
Likelihood ratio	2241	14
Total detected malware	**9865**	**535**

Bold text indicates the effect of author-based feature in our experiments.

Table 9
The change of detection decline rate by each feature attribute set (e.g., malware).

Case	Feature attribute set	Detect	Decline rate (%)
0	SN[a], API[b], Perms[c], Intent, Commands[d]	9865	–
1	API, Perms, Intent, Commands	6429	34.83
2	SN, API, Intent, Commands	7624	22.72
3	SN, API, Perms, Commands	9590	2.79
4	SN, API, Perms, Intent	9828	0.38

[a] Serial number of a certificate.
[b] Malicious API sequence.
[c] Permission distribution.
[d] Intersection of the usage of system commands and the existence of forged files.

Table 10
Classification performance for 9990 malware and 109,193 benign samples.

Category	Family	AutoPsy-INT		AutoPsy-UNI	
		FPs	FNs	FPs	FNs
Malware (9990)	AdWo (2807)	596	295	556	253
	Boxer (2138)	177	161	190	43
	Dowgin (936)	253	249	246	150
	AirPush (495)	339	92	343	114
	Gappusin (487)	121	232	102	259
	SMStado (288)	32	16	37	6
	Counterclank (276)	84	184	84	196
	Wapsx (273)	90	103	88	103
	OpFake (240)	76	47	56	115
	SMSAgent (202)	26	6	26	3
	Kuguo (192)	105	99	92	85
	Ropin (192)	111	119	103	120
	SmsPay (183)	43	74	26	56
	Youmi (173)	86	102	64	73
	Kmin (119)	16	0	11	0
	DroidKungFu (112)	23	60	21	27
	Mseg (103)	64	15	41	18
	SmsReg (103)	42	21	16	18
	FakeBattScar (99)	1	2	3	4
	GingerMaster (94)	10	69	10	71
	FakeNotify (86)	58	1	43	1
	PremiumSMS (63)	4	31	4	33
	JiFake (53)	25	12	12	19
	Boqx (52)	11	31	11	34
	DroidDream (51)	24	28	24	41
	Plankton (49)	8	34	8	38
	DroidRooter (46)	23	11	25	15
	SMSSend (46)	15	15	24	15
	Agent (20)	2	13	2	13
	Utchi (12)	5	0	5	0
Benign (109,193)		125	535	125	535
Average		84	86	77	79

* FPs, FNs refer to False Positives, False Negatives.

significant. The rest of the feature sets are shown in a descending order with their contribution to the accuracy.

Effectiveness of malware classification

We demonstrate that our proposed method provides an effective metric for classifying malware samples into similar subgroups. Table 10 shows that Andro-AutoPsy performs precisely in classifying malware families, having about 80 false positives and false negatives, regardless of the update method. Malware families, such as *SMStado*, *SMSAgent*, *Kmin*, *FakeBattScar*, and *Utchi*, were classified with low false positives rate and false negative rate. However, the performance for classifying malware families, such as *Counterclank*, *GingerMaster*, and *Plankton*, was low.

Some factors may have affected the classification performance of our system. *GingerMaster* contains the *gingerbreak* root exploit code, which is located on a forged file. Since the malicious behavior of *GingerMaster* constitutes only privilege escalation, if there are other malware families containing *gingerbreak* root exploit code, our system could not definitely classify each other. *Plankton* repackaged in a legitimate application runs a background service. This service fetches a drive-by download attack; a remote server sends the victim a malicious URL that points to a *jar* file with executable codes. Since our system could not analyze a downloaded payload directly and could find only system commands to execute exploit code, our system could not classify effectively malware families with system commands. *Counterclank* does not have distinctive malicious behavior. Our system misclassifies it, but perfectly detects it as malware.

Effectiveness of detecting zero-day malware

We demonstrate the effectiveness of detecting zero-day malware detection. We define an application as a zero-day malware if it has malicious behavior and cannot be detected by AV vendors. The malicious behavior of zero-day malware is up-to-date smishing. We leveraged 16 malware samples offered by the Korea Internet Security Agency (KISA). We uploaded them (as samples) to the VirusTotal, and checked the scanning results of various anti-virus (AV) vendors, such as F-Secure, Kaspersky, ClamAV, and Avast. We noted that none of the submitted samples was reported as malware when we conducted our experiment. The results of our experiment using Andro-AutoPsy showed that it captured smishing characteristics and performed precisely in detecting 15 malware samples, corresponding to 93.8% of 16 zero-day malware samples. The missed smishing sample leveraged only the method of the DevicePolicyManager class, such as *Android.app.admin.DevicePolicyManager*; →isAdminActive(), and did not send hijacked sensitive information (e.g., contacts, SMS content, call logs, certificate for financial transaction in Korea), so our system was thus unable to detect that sample as malware.

Efficiency of malware classification

Our proposed system takes only 72 s/MB to detect and classify each malware. The majority of this time is consumed creating the profile; it takes only 0.3 s on average to classify malware into similar groups. Our system has the limitation that the analysis time cannot be reduced, since it is dependent on Apktool. We need to directly parse feature vectors in binary code for faster analysis.

Limitations

Andro-AutoPsy has a few limitations, since it uses integrated footprints as feature vectors and employs static analysis techniques to capture malware's behavior footprint. First, it is difficult for our system to analyze malware embedding anti-malware analysis techniques, such as packing and binary code encryption. A code packing technique transforms a program into a packed program by compression or encryption (Kang et al., 2007). Unfortunately, recently reported smishing applications embedded code packing technique, such as Apkprotect (Apkprotect, 2013), and therefore, our system cannot unpack and analyze this malware. Moreover, since our system has been implemented with Apktool, malware that specifically undermines Apktool will likewise affect our system. Second, our system has a limitation in analyzing malware embedding encryption methods for binary code such as DES. In this case, our system cannot disassemble the executable file of the malware, and thus, fails to analyze it. However, these drawbacks are common in static analysis methods and are addressed in the literature at some expense. When we

combine the dynamic analysis component in Andro-AutoPsy in future work, we can overcome this drawback.

In Vidas and Christin (2013), it is stated that there are some alternative markets solely to distribute malware. Some of these markets actually sign all applications using their private key; that means that the serial numbers are same. If a large set of samples including malware are sourced from these types of markets, our serial number-based detection is going to be heavily biased. This drawback is resolved by incorporating other features as our system does. Moreover, an active adversary may downgrade the results of our system by considering the forgery of a certificate (e.g., private key leakage or the vulnerability of the signature algorithm). It would only affect the part corresponding to the fast scanning, and other features utilized for our system act to mitigate the disadvantage. Furthermore, low-level certificate-related features stay unchanged, despite the forged signature.

Conclusion and future work

In this paper, we proposed Andro-AutoPsy, an anti-malware system based on similarity matching of malware-centric and malware creator-centric information. Using Andro-AutoPsy, we classified malware samples into similar subgroups by exploiting the profiles extracted from integrated footprints, which are implicitly equivalent to distinct behavior characteristics. Andro-AutoPsy is capable of distinguishing benign and malicious applications and classifying malicious applications into similar behavior groups. Furthermore, Andro-AutoPsy is capable of detecting zero-day threats, which are missed by antivirus scanners. In particular, our system only misses 1 smishing application of 16 zero-day smishing applications reported in South Korea.

Our experiments demonstrated that Andro-AutoPsy performs precisely in detecting and classifying malware families with low false positives and false negatives, regardless of the update method. Our system hence enables AV vendors to react to many species of malicious samples by detecting and matching them with previous ones effectively and efficiently. Our experimental results indicate that it takes 72 s/MB to analyze a malware sample on average, with many opportunities for improving on the scalability using vertical expansions (utilizing the full array of parallelism and virtualization).

There are several directions that we will pursue in the future. First, we will integrate a malware analysis system with static and dynamic analysis methods and provide a free online service to at the Korea University Malware Center. Furthermore, we will implement an automatic anti-malware system for analyzing the malware embedding anti-malware analysis techniques, such as packing and binary code encryption, noted in Section Limitations.

Acknowledgments

This research was supported by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2015-H8501-15-1003) supervised by the IITP (Institute for Information & communications Technology Promotion). In addition, this work was also supported by the ICT R&D Program of MSIP/IITP. [14-912-06-002, The Development of Script-based Cyber Attack Protection Technology].

Appendix A. Malicious API list

Table 11
Malicious API list which we defined

Category	Class	Method(field, URI, string)
Retrieving system information (11)	TelephonyManager	getDeviceId()
		getLine1Number()
		getNetworkOperator()
		getSimOperatorName()
		getSimSerialNumber()
		getSubscriberId()
		getCallState()
	UUID	toString()
	WifiInfo	getMacAddress()
	WifiManager	getConnectionInfo()
		getWifiState()
Retrieving personal information (19)	LocationManager	getLastKnownLocation()
		requestLocationUpdates()
	–	content://mms-sms
		content://sms
		content://browser/bookmarks
	ContentResolver	query(), delete()
	Browser	BOOKMARKS_URI
	CallLog.Calls	CONTENT_URI
	Contacts.Phones	CONTENT_URI
	CommonDataKinds.Phone	CONTENT_URI
	Contacts	CONTENT_URI
	Audio.Media	getContentUriForPath()
	Images.Media	getContentUri()
		EXTERNAL_CONTENT_URI

(*continued on next page*)

Table 11 (*continued*)

Category	Class	Method(field, URI, string)
		INTERNAL_CONTENT_URI
	Video.Media	getContentUri()
	–	getContentResolver()
	Uri	parse()
Sending or Receiving SMS (10)	SmsManager	getDefault()
		sendTextMessage()
		createFromPdu()
		getDisplayMessageBody()
		getMessageBody()
		getOriginatingAddress()
		getUserData()
	gsm.SmsManager	sendTextMessage()
		createFromPdu()
		getDisplayMessageBody()
Calling (2)	telephony.ITelephony	endCall()
	–	NEW_OUTGOING_CALL
Recoding (3)	AudioRecord	startRecording()
	MediaRecorder	start(), stop()
Data transmission (7)	HttpURLConnection	getOutputStream()
	URLConnection	getInputStream()
		getOutputStream()
	ssl.HttpsURLConnection	getOutputStream()
	client.HttpClient	execute()
	client.DefaultHttpClient	execute()
	JSONObject	put()
Device policy management (3)	DevicePolicyManager	isAdminActive(), lockNow()
	DeviceAdminReceiver	–
Dynamic loading (8)	AssetManager	getAssets()
	DexClassLoader	loadClass()
	SecureClassLoader	–
	URLClassLoader	–
	Runtime	exec(), getRuntime()
	System	load(), loadLibrary()
Encryption (4)	crypto.Cipher	doFinal()
		getInstance()
	crypto.KeyGenerator	generateKey()
	SecretKeySpec	–
Reflection (2)	Class	getDeclaredMethod()
	reflect.AccessibleObject	setAccessible()
ETC (13)	PendingIntent	getBroadcast()
	–	abortBroadcast
	FileOutputStream	–
	ZipOutputStream	close()
	PackageManager	setComponentEnabledSetting()
	Environment	getExternalStorageDirectory()
	Environment	getExternalStorageState()
	String	equalsIgnoreCase(), split()
	ActivityManager	restartPackage()
	AudioManager	setVibrateSetting()
		setRingerMode()
	Context	getSystemService()

References

Apkprotect. Apkprotect. Apkprotect – Android APK security Protection/Encryption/Guard. 2013. http://www.apkprotect.com/. accessed Mar. 19, 2014.

Apktool. Android-apktool. android-apktool – a tool for reverse engineering Android APK files. 2010. http://code.google.com/p/android-apktool/. accessed Mar. 19, 2014.

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation; 2014. p. 259–69. PLDI '14.

Au KWY, Zhou YF, Huang Z, Lie D. PScout: analyzing the android permission specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security; 2012. p. 217–28. CCS '12.

Bayer U, Comparetti P, Hlauschek C, Kruegel C, Kirda E. Scalable, behavior-based malware clustering. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium; 2009. NDSS '09.

Blasing T, Batyuk L, Schmidt A-D, Camtepe S, Albayrak S. An Android application sandbox system for suspicious software detection. In: 5th International Conference on Malicious and Unwanted Software (MALWARE); 2010. p. 55–62.

Bose A, Hu X, Shin KG, Park T. Behavioral detection of malware on mobile handsets. In: Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services; 2008. p. 225–38. MobiSys '08.

Brenner SW, Carrier B, Henninger J. The Trojan horse defense in cybercrime cases. Santa Clara Comput High Technol Law J 2004;21(1):1.

Bugiel S, Davi L, Dmitrienko A, Fischer T, Sadeghi A-R, Shastry B. Towards Taming privilege-escalation attacks on android. In: Proceedings of the 19th Annual Symposium on Network and Distributed System Security; 2012. NDSS '12.

Burguera I, Zurutuza U, Nadjm-Tehrani S. Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st

ACM Workshop on Security and Privacy in Smartphones and Mobile Devices; 2011. p. 15–26. SPSM '11.
Cao Y, Fratantonio Y, Bianchi A, Egele M, Kruegel C, Vigna G, et al. EdgeMiner: automatically detecting implicit control flow transitions through the android framework. In: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS); 2015. NDSS '15.
Contagio. Contagio mobile-mobile malware mini dump. 2011. accessed Mar. 19, 2014, http://contagiominidump.blogspot.kr/.
Enck W, Gilbert P, Chun B-G, Cox LP, Jung J, McDaniel P, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on operating systems design and implementation; 2010. p. 1–6. OSDI'10.
Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security; 2009. p. 235–45. CCS '09.
F-Secure. F-secure, 25 years of the best protection in the world. 2013. http://www.f-secure.com/en/web/labs_global/. accessed Mar. 19, 2014.
Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security; 2011. p. 627–38. CCS '11.
Isohara T, Takemori K, Kubota A. Kernel-based behavior analysis for android malware detection. In: 7th International Conference on Computational Intelligence and Security (CIS); 2011. p. 1011–5.
Kang MG, Poosankam P, Yin H. Renovo: a hidden code extractor for packed executables. In: Proceedings of the 2007 ACM Workshop on Recurring Malcode; 2007. p. 46–53. WORM '07.
Kocsis RN. Applied criminal psychology: a guide to forensic behavioral sciences. Charles C Thomas Publisher; 2009.
Krebs B. Mobile malcoders pay to (Google) play. Krebs on security. March 2013. http://bit.ly/1kranE5.
Leung KM. Naive Bayesian classifier. Polytechnic University Department of Computer Science/Finance and Risk Engineering; 2007.
Lin Y-D, Lai Y-C, Chen C-H, Tsai H-C. Identifying android malicious repackaged applications by thread-grained system call sequences. Comput Secur 2013;39:340–50.
ltrace. The Debian Project. 2014 (accessed 19.03.14), https://packages.debian.org/search?keywords=ltrace.
McAfee. McAfee Labs threat s report, Fourth Quarter 2013. 2013 (accessed 19.03.14), http://www.mcafee.com/au/resources/reports/rp-quarterly-threat-q4-2013.pdf.
Mount DW. Bioinformatics: sequence and genome analysis. 2nd ed. Cold Spring Harbor Laboratory Press; 2004.
Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol 1970;48(3):443–53.
Nykodym N, Taylor R, Vilela J. Criminal profiling and insider cyber crime. Digit Investig 2005;2(4):261–7.
Paganini. Android Wroba banking trojan targeted Korean users. October 2013 (accessed 19.03.14), http://securityaffairs.co/wordpress/19041/malware/android-wroba-trojan-korea-banks.html.
Pearce P, Felt AP, Nunez G, Wagner D. AdDroid: privilege separation for applications and advertisers in android. In: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security; 2012. p. 71–2. ASIACCS '12.
Peng H, Gates C, Sarma B, Li N, Qi Y, Potharaju R, et al. Using probabilistic generative models for ranking risks of android apps. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security; 2012. p. 241–52. CCS '12.
Rastogi V, Chen Y, Enck W. AppsPlayground: automatic security analysis of smartphone applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy; 2013. p. 209–20. CODASPY '13.
Reina A, Fattori A, Cavallaro L. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: Proceedings of the 6th European Workshop on System Security (EUROSEC). Prague: Czech Republic; April 2013.
Rogers M. The role of criminal profiling in the computer forensics process. Comput Secur 2003;22(4):292–8.
Sarma BP, Li N, Gates C, Potharaju R, Nita-Rotaru C, Molloy I. Android permissions: a perspective combining risks and benefits. In: Proceedings of the 17th ACM Symposium on Access Control Models and Technologies; 2012. p. 13–22. SACMAT '12.
Schmidt A-D, Bye R, Schmidt H-G, Clausen J, Kiraz O, Yuksel K, et al. Static analysis of executables for collaborative malware detection on android. In: IEEE International Conference on Communications, 2009; 2009. p. 1–5. ICC '09.
Seo S-H, Gupta A, Mohamed Sallam A, Bertino E, Yim K. Detecting mobile malware threats to homeland security through static analysis. J Netw Comput Appl 2014;38:43–53.
Shabtai A, Elovici Y. Applying behavioral detection on android-based devices. 2010. p. 235–49.
Spreitzenbarth M, Freiling F, Echtler F, Schreck T, Hoffmann J. Mobile-Sandbox: having a deeper look into android applications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing; 2013. p. 1808–15. SAC '13.
Strace. Strace – useful diagnostic, instructional, and debugging tool. 2013 (accessed 19.03.14), http://freecode.com/projects/strace.
Vidas T, Christin N. Sweetening android lemon markets: measuring and combating malware in application marketplaces. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy; 2013. p. 197–208. CODASPY '13.
Vidas T, Tan J, Nahata J, Tan CL, Christin N, Tague P. A5: automated analysis of adversarial android applications. In: Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices; 2014. p. 39–50. SPSM '14.
VirusShare. VirusShare.com – Because sharing is caring. 2014 (accessed 19.03.14), http://virusshare.com/.
VirusTotal. VirusTotal – free online virus, malware and URL scanner. 2013 (accessed 19.03.14), https://www.virustotal.com/en/.
Wang Y, Zheng J, Sun C, Mukkamala S. Quantitative security risk assessment of android permissions and applications. In: Data and Applications Security and Privacy XXVII. Lecture Notes in Computer Science; 2013. p. 226–41.
Weichselbaum L, Neugschwandtner M, Lindorfer M, Fratantonio Y, van der Veen V, Platzer C. Andrubis: android malware under the magnifying glass. Vienna University of Technology; 2014. Tech. Rep. TRISECLAB-0414–001.
Yan LK, Yin H. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX Conference on Security Symposium; 2012. p. 29. Security'12.
Yang C, Yegneswaran V, Porras P, Gu G. Detecting money-stealing apps in alternative android markets. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security; 2012. p. 1034–6. CCS '12.
Yang W, Xiao X, Andow B, Li S, Xie T, Enck W. AppContext: differentiating malicious and Benign Mobile app behaviors using context. In: Proceedings of the International Conference on Software Engineering (ICSE); 2015.
Zheng M, Sun M, Lui JCS. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In: Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications; 2013. p. 163–71. TRUSTCOM '13.
Zhou Y, Jiang X. Dissecting android malware: characterization and evolution. In: Security and Privacy (SP), 2012 IEEE Symposium on; May 2012. p. 95–109.
Zhou Y, Wang Z, Zhou W, Jiang X. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: Proceedings of the 19th Annual Symposium on Network and Distributed System Security; 2012. NDSS '12.