Available online at www.sciencedirect.com

ScienceDirect

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

Computers & Security

Andro-Dumpsys: Anti-malware system based on the similarity of malware creator and malware 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 Service, Deloitt Anjin LLC, Republic of Korea
[c] Computer Science and Engineering Department, State University of New York at Buffalo (SUNY Buffalo), USA

A R T I C L E I N F O

Article history:
Received 2 July 2015
Received in revised form 12 December 2015
Accepted 28 December 2015
Available online 8 January 2016

Keywords:
Volatile memory acquisition
Similarity
Malware creator centric information
Mobile malware
Android

A B S T R A C T

With the fast growth in mobile technologies and the accompanied rise of the integration of such technologies into our everyday life, mobile security is viewed as one of the most prominent areas and is being addressed accordingly. For that, and especially to address the threat associated with malware, various malware-centric analysis methods are developed in the literature to identify, classify, and defend against mobile threats and malicious actors. However, along with this development, anti-malware analysis techniques, such as packing, dynamic loading, and dex encryption, have seen wide adoption, making existing malware-centric analysis methods less effective. In this paper, we propose a feature-rich hybrid anti-malware system, called Andro-Dumpsys, which leverages volatile memory acquisition for accurate malware detection and classification. Andro-Dumpsys is based on similarity matching of malware creator-centric and malware-centric information. Using Andro-Dumpsys, we detect and classify malware samples into similar behavior groups by exploiting their footprints, which are equivalent to unique behavior characteristics. Our experimental results demonstrate that Andro-Dumpsys is scalable, and performs well in detecting malware and classifying malware families with low false positives and false negatives, and is capable of responding zero-day threats.

1. Introduction

Despite the continuous detection and defense efforts of antivirus (AV) vendors, the number of mobile malware is rapidly increasing, and their capabilities are getting more sophisticated. According to a report by McAfee, the total number of mobile malware samples exceeded five million in the third quarter of 2014, increased by 16% from the previous quarter, and increased by 112% from that of the same quarter of the prior year (McAfee, 2014). To defend against malware, AV vendors analyze tens of thousands of malware samples daily, and prevent them from spreading. While successful in achieving their goal, AV vendors are always in an arms race with cyber criminals who utilize various sophisticated techniques to circumvent detection efforts. Such techniques are

* *Corresponding author.* Tel.: +82 2 3290 4898.
E-mail addresses: changkr@korea.ac.kr (J. 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.cose.2015.12.005
0167-4048/© 2016 Elsevier Ltd. All rights reserved.

very difficult to detect, especially given their potentially legit use.

Recently, code-packing techniques have been widely used for legitimate reasons: intellectual property protection; they are used to prevent competitors from analyzing codes. In particular, those techniques utilize anti-decompiling, anti-debugging, and anti-tampering, as well as compression and encryption of applications to reach their end goal. Malware creators, however, utilize the same techniques, making static malware analysis techniques ineffective. Especially, the number of malware families adopting anti-malware analysis techniques, such as packing (e.g., Apkprotect (Apkprotect, 2013) and Bangcle (Bangcle, 2014)), dynamic loading, and dex encryption (e.g., DES), has increased rapidly (Yu, 2014). Furthermore, the Android platform, a popular mobile operating system, supports dynamic loading methods for flexible memory management.

Dynamic loading methods of other binary files (such as `dex` or `jar`) provide flexible memory allocation and extend the dynamic functionality during runtime execution. The `dex` (encryption) technique is an anti-analysis method, used for intellectual property protection by encryption, and requires the application to be fetched in a memory section at the installation time. When utilized in malware as an embedding technique, it further complicates the analysis and makes the static analysis ineffective. In particular, many static analysis approaches in the literature failed to parse meaningful code patterns from the application embedding these anti-malware analysis techniques (Arp et al., 2014; Peng et al., 2012; Wang et al., 2013; Yang et al., 2014; Zhang et al., 2014). Dynamic analysis addresses obfuscation, packing, and encryption attempts, since all of those approaches are eliminated during the execution of malware (Enck et al., 2009; Jang et al., 2014; Pearce et al., 2012; Zhang et al., 2013). However, the dynamic analysis is only done on the part of the application that is actually executed, and the malicious behavior must be executed during the analysis for malware to be detected (Mohaisen et al., 2015). This leaves a lot of options for malware creators to game the uncertainty of malware analysts. This calls for utilizing other implicit and explicit threat signals and indicators that could be helpful in detecting malware and thwarting their authors' efforts.

For example, for the quick response, it is necessary for malware analysts to check the target of malware attack and its context, since it reflects the attack's intent as meant by the malware creator. Such intent could be a valuable piece of information to detect and analyze malware. In order to understand the attack's intent, we follow a detailed process that aims to answer the following questions. 1) What do malware creators want to obtain by launching an attack? 2) How do malware creators attack a victim? 3) What do malware creators need for launching the attack? By answering these questions, malware analysts can understand malware creator's attack patterns and use that pattern as a strong signal for analysis.

We rely on various artifacts to obtain such signal. For example, while verifying an application, the Android operating system (OS) requires each application to be digitally signed with a certificate. Each digital certificate is identified uniquely by the serial number, which is difficult to forge. Then, leveraging malware creator-centric attributes, such as the digital certificate and its serial number, could be essential and helpful to malware analyzers.

Our contribution combines malware-centric attributes with intent-based features for malware detection and classification. In particular, to overcome the drawbacks of previous malware-centric methods, we propose a novel and feature-rich hybrid anti-malware system, called Andro-Dumpsys. Despite the various anti-analysis methods heavily utilized by various malware families, all codes of malware are readable by the Android platform upon executing them; our system catches the moment the `odex` bytecode is loaded into the memory section (Kim et al., 2015), and utilizes runtime artifacts for malware characterizations. In particular, our system runs a target application on an emulator, extracts `odex` bytecode – which collects parts of an application that are optimized before booting (Khan, 2010) through volatile memory acquisition (dynamic analysis) in order to address the obfuscation, packing, and dynamic loading techniques. Then, our system parses meaningful and relevant code patterns from the `odex` file and creates a profile of each application. In particular, for capturing the intent of malware creator, we leverage footprints, including the serial number of a certificate, operation codes (opcodes) in the `odex` files, and meta-data in `AndroidManifest.xml` as feature vectors for malware characterization. By comparing the profiles, our system can detect and classify malware samples into related families. We also observe that: 1) malware samples have unique behavior patterns, 2) the malicious behavior is determined by operation codes and requires a particular permission set (e.g., READ_SMS, WRITE_SMS), and 3) such an operation code set influences the behavior of the malware.

1.1. *Contribution*

1. We propose an *integrated anti-malware analysis system* which considers both malware-centric information and malware creator-centric information for malware analysis. Using the malware creator-centric information simplifies the process of malware analysis.
2. We propose a hybrid malware detection and classification method coupled with volatile memory acquisition method. To the best of our knowledge, our approach is the first automated anti-malware system to deal with the practical issue related to sophisticated anti-malware analysis techniques such as packing and dynamic loading.
3. Our system enables AV vendors to react efficiently to many species of malware samples by conducting similarity matching. Andro-Dumpsys facilitates the detection of new malware including malware variants and zero-day malware. This is further highlighted by experiments using real-world up-to-date malware samples.

1.1.1. *Organization*

The rest of this paper is organized as follows. In section 2, we review the related work. In section 3, we present the data exploration to find meaningful features for the anti-malware system. In section 4, we present our anti-malware system, Andro-Dumpsys. In section 5, we provide the performance evaluation and results. In section 6, we discuss the limitation of our proposed method. Finally, we present concluding remarks in section 7.

2. Related work

2.1. Sandboxing approach

Yan and Yin (2012) proposed DroidScope, a fine-grained dynamic analysis framework built on top of QEMU. DroidScope leveraged low-level and high-level behavior characteristics, such as native/Dalvik instruction traces and a set of APIs, and reconstructed the behaviors of a malicious application. DroidScope performed fine-grained and coarse-grained analysis. Thus, it may suffer from high overhead of taint analysis. However, DroidScope was implemented based on hooking mechanism and needed to monitor a synchronized low-level and high-level events. Reina et al. (2013) introduced CopperDroid, a system-call centric framework for dynamic analysis. CopperDroid conducted an integrated analysis to reconstruct the behavior of a malicious application by leveraging OS-specific information, such as system call invocations and IPC/RPC interactions, and Android-specific information such as sensitive information leakage and SMS. Rastogi et al. (2013) proposed AppsPlayground by conducting dynamic analysis. AppsPlayground executed a malicious application on an emulator, and determined whether or not malicious activities are being carried out by tracking sensitive information leakage and monitoring sensitive API and system calls. However, AppsPlayground required a modified Android framework for malware analysis. In contrast with Andro-Dumpsys, these approaches can only analyze the part of the application actually executed, and the malicious behavior must appear during execution time for analyzing the malicious application. Andro-Dumpsys executes the malicious application by leveraging service and activity components in a manifest file. Since these components are the entry point of the application, Andro-Dumpsys can obtain bytecodes of the relevant execution phase.

Weichselbaum et al. (2014) introduced Andrubis, which is an extension to Anubis (Anubis, 2011) for analyzing Android malware. Andrubis was a fully automated analysis system coupled with static and dynamic analysis methods. In the static analysis step, Andrubis extracted information from `AndroidManifest.xml` and `dex` bytecode of an application. In the dynamic analysis step, Andrubis executed the application in an emulated environment as in TaintDroid and Droidbox. While executing applications, Andrubis monitored actions at both the Dalvik Virtual Machine and the system level. Vidas et al. (2014) presented A5, an automated anti-malware system based on static and dynamic analysis. In the static analysis phase, A5 extracted information from `AndroidManifest.xml` and dex bytecode by leveraging open project tools such as Androguard (Desnos, 2011) and Soot (Vallée-Rai et al., 1999). The output of the bytecode is used in the dynamic analysis phase. In the dynamic analysis phase, A5 conducted malware analysis in a sandboxed environment, which consisted of multiple physical devices and emulators. A5 monitored network threats presented on the Internet, executed the malicious application, and generated network intrusion detection signatures. Andrubis and A5 have implemented open source tools. That means these frameworks are influenced by the weakness of open source tools, and thus, these approaches fail to extract meaningful behavior or code patterns from the application embedding anti-malware analysis techniques (e.g., packing, dynamic loading, and `dex` encryption).

2.2. Android permission monitoring

Enck et al. (2009) proposed the Kirin security service, a lightweight certification service to mitigate malware at installation time. Kirin examined the requested permissions of applications in a manifest file, and determined whether or not malicious activities were executed by comparing them with self-defined rules. Pearce et al. (2012) introduced AdDroid, which separated advertising permissions for the Android platform from the rest of permissions. In AdDroid, the host application and the core advertising code are executed in an isolated environment where applications using AdDroid does not send private information to an advertisement server. However, AdDroid has a limitation to respond to information leakage unrelated to an advertisement, which applies to the majority of mobile malware. Peng et al. (2012) used probabilistic generative models for risk scoring, ranging from the simple Naïve Bayes to advanced mixture models. Their methods compute a quantitative risk score of applications based on the permissions in a manifest file, and discriminate between malware and benign applications. Wang et al. (2013) introduced DroidRisk, a framework based on quantitative risk assessment of permissions. By indicating the risk levels of applications, they presented that a reliable risk signal could be generated in order to warn potential malicious activities. Requested permission-based methods rely completely on the permissions in a manifest file. However, application developers tend to declare an excessive number of permissions in a manifest file, despite the application does not need them all. Thus, the capability of detecting and classifying malware with a high accuracy is limited, and requires the methods based on other criteria to achieve higher classification accuracy (Enck et al., 2009; Pearce et al., 2012; Peng et al., 2012; Wang et al., 2013). Zhang et al. (2013) proposed a VetDroid, which is a dynamic analysis platform for reconstructing sensitive behaviors in Android application. As leveraging API-related permission table, VetDroid completely identified all possible permission usage. However, permission-based detection methods are ineffective in identifying benign applications, since relevant rule sets only focus on detecting the malware, thus producing large false alarms.

2.3. Framework API monitoring

Arp et al. (2014) proposed DREBIN, which takes a hybrid approach and considers both permissions and sensitive APIs as features. DREBIN performed a broad static analysis to extract feature sets from both manifest file and `dex` bytecode. These features are embedded in a vector space that helps DREBIN identify malware automatically. Yang et al. (2014) proposed DroidMiner, which uses static analysis automatically to mine malicious behavior from a two-level behavioral graph representation. DroidMiner considered the frequency or names of APIs as well as the connections between multiple sensitive API functions. Zhang et al. (2014) proposed DroidSIFT, which classifies Android malware via weighted contextual API dependency graph. To defend against variants and zero-day malware, DroidSIFT leveraged graph similarity metrics for anomaly and signature detections. Zhou et al. (2012) proposed DroidRanger, which is malware detection method based on a permission-based scheme and a heuristic-based scheme. However, these approaches fail to parse meaningful code patterns from applications that embed anti-malware analysis techniques like packing, dynamic loading, and `dex` encryption.

2.4. *Memory acquisition*

AndroDump which is a part of Androguard (Androguard, 2011) enables dumping bytecode from a virtual machine. However, AndroDump has a limitation to solve the current issue since it just retrieves magic numbers of Java class files in the memory for memory acquisition. Yu (2014) has proposed a volatile memory acquisition method exploiting LiME (Linux Memory Extractor), a type of the Linux Kernel Module (LKM). Since their proposed method is dependent on the Android OS version, they need cross-compile procedure whenever changing target Android platform. Kim et al. (2015) proposed a novel technique to dump executable code from the memory section. They modified the Dalvik Virtual Machine, tracked the process identifier of an application, and dumped bytecode in the memory section. While they provide the basic framework, their proposed technique requires modifying the Android OS, and yields poor scalability in the malware analysis domain.

3. Data exploration

In order to extract the behavior pattern of malware, we adopt the following as feature vectors: the serial number of a certificate, suspicious API sequence, permission distribution (a critical permission set and its likelihood ratio), intent, and the usage of system commands for executing forged files. We review our previous works (Jang et al., 2015; Kang et al., 2015) and determine metrics for malware analysis through the data exploration. We used 906 malware samples and 1776 benign samples in our experiments.

3.1. *Serial number of a certificate*

When releasing an application to the GooglePlay, an application creator signs his application with the private key and a certificate. When the creator generates the relevant certificate, there are blanks for writing the creator's personal information. However, the application creator can submit that form with false information, since the process does not require verification. The certificate has a unique serial number according to the RFC 2459[1] (X.509) standard. We observe that a small set of serial numbers is used in many malware samples. To that end, we explore the serial number as a feature vector. Given the dataset of malware, we extracted the serial number of a certificate in each sample and studied the distribution of the serial numbers. With a total of 305 serial numbers observed in all the malware samples, we found only 22 unique serial numbers contributing 50% of the samples. This means that malware creators frequently use certain certificates with the same serial numbers. Among the 305 serial numbers, 19 numbers generated more than 2 malware families or 2 variants of each malware family. For primary screening, we made a blacklist of 18 serial numbers through empirical experiments by excluding an edge case. This is, we exclude "93:6e:ac:be:07:f2:01:df" from the serial numbers, since it is a standard test key for native applications built on a device or an emulator.

3.2. *On the uniqueness of serial numbers*

Ground truth of the serial number is not of any importance to our evaluation. Per the specifications of the X.509 digital certificate standard in RFC 2459, the serial number of each certificate issued by a CA is unique and cannot be reused. Furthermore, more important than the serial number is the key associated with the certificate, which is used for verifying the signed application: for using the serial number as a feature, we make sure that the public key in the certificate is a unique matching with the given serial number. The validity of the public key associated with the certificate is verified against the signature of the certificate, establishing an association between the serial number, the certificate, and application and the key in the certificate.

Furthermore, we explored the serial number distribution in benign samples. For benign samples, we crawled a variety of popular applications with high rankings (as of March 2015) from GooglePlay. Duplicate benign apps (sample) were excluded according to their hash digest and package name. We extracted the serial number of a certificate in each sample and studied the distribution of the serial numbers. With a total of 1776 serial numbers observed in all the benign samples, we found 1464 unique serial numbers contributing 82% of the benign dataset. Furthermore, we found 136 serial numbers only generated more than two kinds of benign applications; most of these applications are signed by Game/Application companies, or personal developers. In short, we conclude that the serial number distribution in benign samples is different from that of malware samples.

3.3. *Suspicious API sequence*

We explore the Application Programming Interface (API) of Android SDK, as a feature vector. The API is a set of functions provided conveniently to control the principal actions of Android platform. It is more efficient to consider certain APIs frequently used by malware than to consider all the APIs in Android SDK. Seo et al. (2014) analyzed a large number of malware samples and determined the suspicious APIs frequently used by malware in a statistical manner. They compared the frequency of their malware samples and benign applications, listing suspicious APIs. We updated their suspicious API list with additional APIs by examining all the APIs that might work in a similar way to suspicious APIs determined in Seo et al. (2014). These APIs are involved in gathering the user's

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

private information or the system information, accessing Web services, sending and deleting an SMS message, recording voice, and accessing and reading the content provider, among other actions. Although the malicious behavior is similar across multiple malware samples, the API patterns used in malware vary according to the malware creators. To resolve this problem, we convert the suspicious API into ASCII code. The transformed letter (ASCII code) sequence represents the behavior pattern of each malicious application. We name the ASCII code sequence the API sequence and use it as a feature vector. Therefore, we chose suspicious API sequence as a feature vector for malware analysis.

3.4. *Permission distribution*

We explore the permission as a feature vector. Peng et al. (2012) compared two datasets of benign and malicious applications, and analyzed the distribution of permissions requested by each dataset. They determined 26 risky permissions as the critical permission set from the perspective of security and privacy. In particular, they removed the `INTERNET` permission because this permission is necessarily required for network communication. Instead, `INSTALL_PACKAGES`, which is required in order to install additional packages, is included. The listing of the permissions is shown in Fig. 1. We used the aforementioned critical permission set in our system because they explored permissions from a large dataset and carefully determined that permission set. While requested permissions are notified to users at installation time, there are other methods of studying permission specification by analyzing API method graphs (Au et al., 2012). The requested permissions declared in a manifest file are not in fact necessary for the application functionality. Au et al. (2012) brought the current requested permission system into question and demonstrated that it was incomplete. Accordingly, they specified a list of permissions required for every API call and provided the permission mappings. In our system, we leveraged 26 critical permissions when applying PScout mapping that lists permissions required to use an API in (Au et al., 2012), along with the requested permission. PScout extracted the Android permission specifications of multiple Android versions (2.2 - 4.0) using static analysis. We named this feature the API-related permission. Benign and malware applications have different requests of permissions. Malware often requests more of the "critical permission set" for security and privacy. In our previous works (Jang et al., 2015; Kang et al., 2015), we analyzed the distribution of that permission set in benign and malware samples to calculate the likelihood of the critical permission set. The distribution of the permissions depends on the sample class (benign or malware).

By applying a Naïve Bayes classifier, we calculate the likelihood of the critical permission set for each class. Peng et al. (2012) used the critical permission set, by applying Naïve Bayes models to quantify the risk score of each application. The permissions should be relatively independent to multiply each probability of permission. Au et al. (2012) showed that most Android permissions have little correlation with other permissions except for 15 permission pairs in terms of API usage. Most of the critical permissions we used are not included in these pairs. Among the 15 permission pairs, only one pair (ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION) is highly correlated to each other. For reducing the complexity of computation, we assume that critical permissions are relatively independent in terms of the API usage.

Let n and m be the number of applications and the number of critical permissions respectively. A permission vector of application i is $\theta_i = (\theta_{i,1}, \theta_{i,2}, \dots, \theta_{i,m})$, where

$$\theta_{i,k} = \begin{cases} 1 & \text{if an application } i \text{ uses a critical permission } k \\ 0 & \text{otherwise} \end{cases},$$

which are independent variables, since all permissions are relatively independent. We put $c_i \in \{\text{benign, malware}\}$, indicating the category of an application i. Then,

$$P(c_i|\theta_i) = P(c_i|\theta_{i,1}, \theta_{i,2}, \dots, \theta_{i,m}) = \prod_{k=1}^{m} P(c_i|\theta_{i,k}).$$

Using Bayes' rule, the conditional probability of c_i given variable $\theta_{i,k}$, which informs about the use of the critical permissions, can be written as:

$$P(c_i|\theta_{i,k}) = \frac{P(\theta_{i,k}|c_i) \cdot P(c_i)}{P(\theta_{i,k})}.$$

Then, the ratio of probabilities is calculated as:

$$\frac{P(\text{malware}|\theta_{i,k})}{P(\text{benign}|\theta_{i,k})} = \frac{P(\theta_{i,k}|\text{malware}) \cdot P(\text{malware})}{P(\theta_{i,k}|\text{benign}) \cdot P(\text{benign})}.$$

ACCESS_COARSE_LOCATION	PROCESS_OUTGOING_CALLS	RECEIVE_WAP_PUSH
ACCESS_FINE_LOCATION	READ_CALENDAR	RECORD_AUDIO
BLUETOOTH	READ_CONTACTS	SEND_SMS
BLUETOOTH_ADMIN	READ_HISTORY_BOOKMARKS	WRITE_CALENDAR
CALL_PHONE	READ_LOGS	WRITE_CONTACTS
GET_ACCOUNTS	READ_PHONE_STATE	WRITE_EXTERNAL_STORAGE
INSTALL_PACKAGES	READ_SMS	WRITE_HISTORY_BOOKMARKS
MOUNT_UNMOUNT_FILESYSTEMS	RECEIVE_MMS	WRITE_SMS
NFC	RECEIVE_SMS	

Fig. 1 – Risky permissions.

We assume $P(c_i = \mathit{malware}) = P(c_i = \mathit{benign})$; there is no information of the category of an application, and thus, we suppose the application to select a variable of any category value with a uniform distribution. By multiplying the probabilities of permissions, the likelihood ratio (Λ) is:

$$\Lambda(\theta_i) = \frac{P(c_i = \text{malware}|\theta_i)}{P(c_i = \text{benign}|\theta_i)} = \prod_{k=1}^{m} \frac{P(\theta_{i,k}|c_i = \text{malware})}{P(\theta_{i,k}|c_i = \text{benign})}.$$

One of the conditional probabilities becomes zero, then the whole multiplication becomes zero. By calculating the conditional probabilities with the Laplace estimator (Leung, 2007), we avoid the worst case where a denominator is zero.

When the likelihood of the malware increases as compared with that of the benign application, the likelihood ratio (Λ) increases. Malware can be detected by comparing the likelihood ratio (Λ) of the critical permission set using some predefined threshold value T_{LR}. Therefore, we choose the likelihood ratio (Λ) as a feature vector for malware detection.

3.5. *Intent*

An Android application does not have the unique entry point that a program usually has in other operating systems. An Android application has four components: activity, service, broadcast receiver, and content provider. These four components work independently, and each component delivers an "intent" to other components to achieve the end goal of applications. The intent is delivered from one activity to other activities, including specific instructions about what the application has to do. We examined the intent-specific information for detecting malware that hides SMS notification. These malwares receive SMS messages with the highest priority, and prevent the delivery of the intent to other applications. This malicious behavior can be found by retrieving the intent filters in a manifest file, `AndroidManifest.xml`.

3.6. *Usage of system commands*

We study the usage of system commands. The system commands frequently used by malware are listed in Seo et al. (2014). We revised the list by excluding the system commands with low frequency in our dataset. We found system commands, such as " `chmod`", " `insmod`", " `su`", and " `bash`", to be frequently used by malware. Those system commands are executed after the malware obtains the admin privilege of the device. Further, we include `gingerbreak` and `rageaginstthecage`, which are types of root exploit codes.

3.7. *Existence of forged files*

In order to circumvent the detection methods of AV vendors, mobile malware hides the codes related to the execution of malicious behavior in normal-looking applications and executes the malicious behavior by loading those applications. However, some benign applications change the extension format of update files to another extension format for security reasons. In that case, utilizing file forging as a feature vector leads to high false positives. In order to reduce the false positives, we combine the two rules as a feature vector for malware detection: usage of system commands and the existence of forged files. Forged files containing malicious codes are usually hidden in `assets`, `lib`, and `res` folder, and system commands are needed for executing the malicious codes (Seo et al., 2014). Therefore, the combined rules are a good metric for detecting malware that overcomes the aforementioned shortcoming; we call the combined rules as "system commands for executing forged files" and use it as a feature vector.

4. System overview

4.1. *Overview*

As illustrated in the flow diagram in Fig. 2, we propose a hybrid anti-malware system. Our system consists of a client component that resides on a mobile device and a behavior profiling and analysis system that resides on a remote server. The client component collects the application's information and sends it to a remote server. The client sends only application-specific information such as the hash digest of the `apk` file and the package name. If the remote server fails to crawl that application, the client sends the application file to the remote server. The remote server analyzes the application information and determines whether it is malicious or not.

The remote server has three components: crawler, repository, and analyzer. A target application delivered from the client or the crawler is passed to the repository. The repository component retrieves analysis requests in its database when receiving them from the client. If the repository component fails to fulfill the client's request, it fetches the crawler component. The analyzer component analyzes a target application passed from the client or repository component. After completing the analysis, the analyzer component notifies the repository and client with the analysis results. The remote server provides the web interface to users. If a user needs to check whether a released application is malicious or not, the user uploads the application file to the remote server. Accordingly, the remote server performs the aforementioned processes.

The analyzer component consists of a footprints extraction and a decision processes, which are reviewed in the following.

4.1.1. *Footprints extraction process*

The footprints extraction process consists of a memory acquisition engine and a behavior-profiling engine.

(i) *Memory Acquisition Engine (MAE):* at the application level, the `apk` file is represented as a compressed archive file with meta-data, and the `classes.dex` inside the `apk` file is Dalvik Executable, which is stored without compression and padded from the archive file. Dalvik bytecode (`dex`) of an application is generally not optimized, since it is executed by a DVM which can run on different architectures. When the *system installer* performs installation of a target application, optimization is done at installation time, where the dex file is optimized for the underlying architecture, and an odex file is generated in

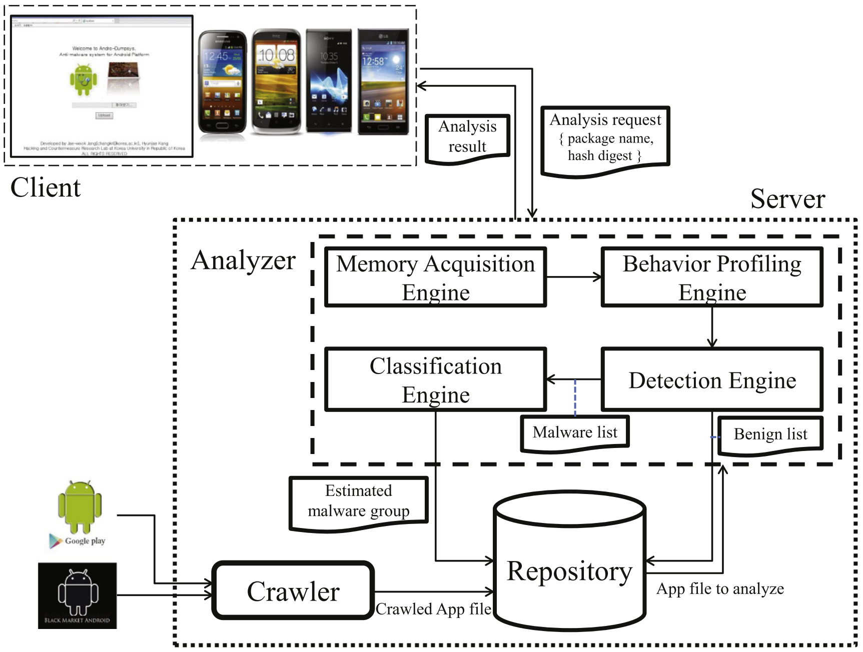


Fig. 2 – Overall procedure of Andro-Dumpsys.

the /data/dalvik-cache folder. In particular, if an application embeds a packing or a dynamic loading method, the odex file is allocated in a memory section after unpacking or dynamic loading is completed.

The MAE dumps meaningful volatile memory sections where a target application is allocated in an emulator. For that, we embed the strazzere's Native Development Kit code[2]. The strazzere's unpacker can unpack elaborate packing methods, such as Apkprotect, Bangcle, LIAPP, and Qihoo, without depending on gdb. The MAE also dumps volatile memory sections of applications embedding dynamic loading methods, including dex encryption. The memory acquisition of MAE proceeds as follows. First, the MAE retrieves a process ID (PID) of an application using the package name of an application. In order to avoid anti-debugging features using Ptrace, MAE "steals" the memory section of a cloned process ID (CID) which is never ptraced by recurring through the given PID, /proc/⟨PID⟩/task/ (step 1 in Fig. 3). Next, MAE retrieves the memory boundaries of a running CID using /proc/⟨CID⟩/maps, and checks whether the packing or dynamic loading method is adopted or not (step 2 in Fig. 3). Finally, MAE attaches that process to PTRACE and copies the memory layout whose signature is "dey" from /proc/⟨CID⟩/maps (step 3 in Fig. 3). After capturing the odex files, the MAE passes them to the profiling engine.

(ii) *Behavior Profiling Engine*: The Behavior Profiling Engine (BPE) extracts the serial number from a certificate in the META-INF folder. For efficiently searching and parsing relevant information, BPE searches only files and the folders with the same component name; our system sorts parsed components in ascending order. If a target application uses a dynamic loading method, BPE retrieves all smali files because malicious codes can be hidden in the dynamic loading files. The BPE retrieves the package name, requested permissions, component names and intents in a manifest file, AndroidManifest.xml, and extracts meaningful features from smali code, according to the aforementioned component name. Following the codes whose names are components, the system extracts suspicious APIs, system commands, and API-related permissions. Moreover, our system checks whether a forged file exists in appendix folders such as assets, res, and lib folders. The BPE captures the footprints of a target application, and then it creates the profile of a target application, and passes it to the *Detection Process*.

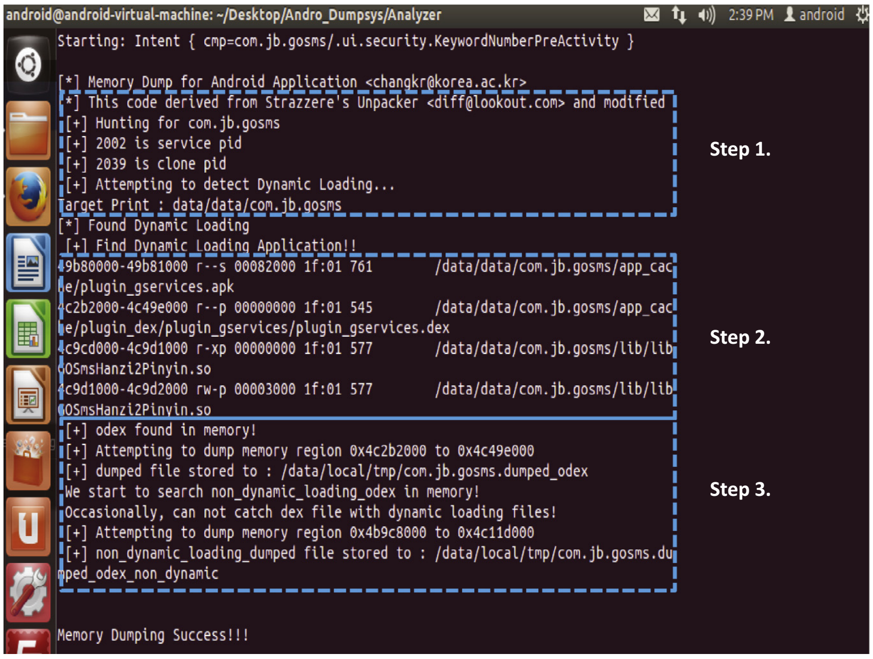
4.2. *The decision process*

4.2.1. *Detection engine*

Our detection engine (DE) determines whether a target application is malicious or not based on its behavior patterns. The DE contains detection rules, which consist of the serial number list, a rule for examining the usage of system commands to execute forged files, a rule for hiding SMS notifications, a rule for detecting smishing (SMS phishing) applications, the rule of checking leakage of sensitive information, and the likelihood ratio (Λ) of requested and API-related critical permission set. The detection algorithm starts by comparing the application's serial number with a blacklist for primary screening.

We only extract the serial numbers that generate families or variants of malware more than a threshold T_{SN}. In our dataset, there were some applications signed by the serial number in the blacklist but do not exploit any suspicious APIs explained in section 3. In that case, we discard these applications to avoid

[2] https://github.com/strazzere/android-unpacker



```
android@android-virtual-machine: ~/Desktop/Andro_Dumpsys/Analyzer
Starting: Intent { cmp=com.jb.gosms/.ui.security.KeywordNumberPreActivity }

[*] Memory Dump for Android Application <changkr@korea.ac.kr>
[*] This code derived from Strazzere's Unpacker <diff@lookout.com> and modified
[+] Hunting for com.jb.gosms
[+] 2002 is service pid
[+] 2039 is clone pid
[+] Attempting to detect Dynamic Loading...
Target Print : data/data/com.jb.gosms
[*] Found Dynamic Loading
 [+] Find Dynamic Loading Application!!
49b80000-49b81000 r--s 00082000 1f:01 761        /data/data/com.jb.gosms/app_cac
he/plugin_gservices.apk
4c2b2000-4c49e000 r--p 00000000 1f:01 545        /data/data/com.jb.gosms/app_cac
he/plugin_dex/plugin_gservices/plugin_gservices.dex
4c9cd000-4c9d1000 r-xp 00000000 1f:01 577        /data/data/com.jb.gosms/lib/lib
GOSmsHanzi2Pinyin.so
4c9d1000-4c9d2000 rw-p 00003000 1f:01 577        /data/data/com.jb.gosms/lib/lib
GOSmsHanzi2Pinyin.so
[+] odex found in memory!
[+] Attempting to dump memory region 0x4c2b2000 to 0x4c49e000
[+] dumped file stored to : /data/local/tmp/com.jb.gosms.dumped_odex
We start to search non_dynamic_loading_odex in memory!
Occasionally, can not catch dex file with dynamic loading files!
[+] Attempting to dump memory region 0x4b9c8000 to 0x4c11d000
[+] non_dynamic_loading_dumped file stored to : /data/local/tmp/com.jb.gosms.du
mped_odex_non_dynamic


Memory Dumping Success!!!
```

Fig. 3 – Memory acquisition procedure: Our system finds the process ID and cloned process ID in step 1, checks whether or not the packing or dynamic loading method is adopted in step 2, and dumps the bytecode in step 3.

over-fitting. Secondly, we check the usage of the system commands for executing forged files. The next detection step is to find malware that hides SMS notification. The purpose of hiding SMS notification is to subscribe to premium services, or is to receive SMS commands from a command and control (C&C) server. These applications use the intent filter and API methods related to received SMS handling, request the highest priority for SMS notification, and call `abortBroadcast()` to hide SMS notification to the users. This step checks whether an application leverages the aforementioned methods and intent filters for detecting malicious behavior.

Similar to premium-rate SMS, a smishing application receives SMS messages including C&C messages from a remote server, and sends hijacked sensitive information (e.g., call history, SMS content, location information, and digital certificate for financial transaction; mainly applicable to the context of South Korea). In this case, the smishing application hides the SMS notification while the malicious behavior is executed. In the final step, the algorithm calculates the likelihood ratio (Λ) of the critical permission set. Two likelihood ratios are obtained using the requested critical permission set and the API-related critical permission set. If the values are both greater than a threshold T_{LR}, then we consider the application to be malicious. To compensate for the limitations of the permission-based methods, we also check whether an application sends SMS messages or not, by calling `abortBroadcast()`.

4.2.2. *Classification engine*

The proper similarity metrics are applied to different types of behavior components using the aforementioned features. The classification engine (CE) calculates the similarity score between the profile of a target application and the representative profile of each malware group. The CE then assigns the malware to the most similar behavior group. The representative profile of each malware group has to depict the common behavioral patterns of each malware group. Then, the CE chooses one of the update methods for the representative profile as follows:

1. **Dumpsys-INT**: The representative profile for each malware group is updated by an intersection of the profiles of members in each group. In the update method of Dumpsys-INT, as the number of members of each group increases, the number of the representative profile decreases.
2. **Dumpsys-UNI**: The representative profile for each malware group is updated by a union of the profiles of the members in each group. In the update method of Dumpsys-UNI, as the number of members of each group increases, the number of the representative profile increases.
   The similarity score is defined as the weighted similarity sum of 3 behavior components. The similarity score between the profile of a target application and the representative profile for each group is given by:

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

where BCS_i and w_i are the similarity of behavior component i and weight of behavior component i, respectively. The behavior component similarity (BCS) is composed of three parts: similarity of suspicious API sequence, usage of system commands for executing forged files, and usage of critical permission set (requested and API-related critical permissions). We set the weight parameter (wi) at 1/3 (the arithmetic mean). Finally, we compute the similarity score for each behavior component as follows:

1. We calculate the similarity score for a suspicious API sequence. According to our pre-defined suspicious API

dictionary, we create a distinct API sequence of each malware. In order to compare the suspicious API sequence of a target with others, we convert the API method into ASCII code. The transformed letter sequence represents the behavior pattern of each malware. We adopted the Needleman–Wunsch algorithm (Needleman and Wunsch, 1970) for finding the optimal global alignment between two sequences. The value of the similarity score is in the interval [0, 1]. There is no clean form of mathematical expression for this similarity – using the Needleman–Wunsch algorithm, 1 is assigned for match, –1 assigned for mismatch, and –2 is assigned for gaps; the similarity score is the sum of the scores normalized by the length of the larger of the two sequences.

2. We calculate the similarity score for the usage of malicious system commands for executing forged files by applying the Jaccard coefficient. The Jaccard coefficient of two sets is defined as the number of attributes in an intersection divided by the number of attributes in a union of the two individual sets. The order of the malicious command is insignificant. The value of the similarity score is in the interval [0, 1]. For two sets of malicious system command usage defined as A and B, the mathematical expression of the Jaccard coefficient is defined as:

$$J(A,B)=\frac{|A\cap B|}{|A\cup B|}. \tag{2}$$

3. We calculate the similarity score for the usage of the critical permission set by averaging the similarity for requested critical permissions and API-related critical permissions. We apply the Levenshtein distance to compute the similarity of the critical permissions; it calculates the minimum number of edits required to make two strings identical. The order of the critical permissions is insignificant; therefore, we applied the Levenshtein distance after sorting the critical permissions. A value of similarity is defined as the number of edits divided by the maximum length of strings, and it is in the interval [0, 1]. Mathematically, for two strings (corresponding to the critical permission usage), the Levenshtein distance is defined as:

$$\mathrm{lev}_{a,b}(i,j)=\begin{cases}\max(i,j) & \min(i,j)=0\\ \min\begin{cases}\mathrm{lev}_{a,b}(i-1,j)+1\\ \mathrm{lev}_{a,b}(i,j-1)+1\\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i\neq b_j)}\end{cases} & \text{otherwise}\end{cases} \tag{3}$$

where $1_{(a_i\neq b_j)}$ is the indicator function equal to 0 when $a_i = a_j$ and equal to 1 otherwise.

The CE classifies a malicious application into the behavior group with the highest similarity score, which is the predefined threshold of at least 0.75. We assume that 0.75 is a sufficiently high score to determine if two signatures are similar. Whenever a new malicious sample is queued into Andro-Dumpsys, the CE updates the representative profile according to the pre-chosen update method. The CE seamlessly updates the likelihood of the critical permission sets and the blacklist of serial numbers at every weekly interval.

5. Performance evaluation

5.1. Implementation

Our anti-malware system consists of a client device and a server. The client was installed on a mobile device (SM-N900K) running on Android 4.4.2, and the three components (crawler, repository, and analyzer) were installed on the remote server. The remote server had an Intel Xeon X5660 processor (running at 2.8GHz) and 8 GB of RAM. The remote server runs an Ubuntu 12.04 LTS (64-bit) operating system. We performed all experiments in a virtualization environment[3]. We implemented Andro-Dumpsys using the Python programming language (as scripts) and the PHP web programming language. The remote server was composed of the MAE, BPE, DE, and CE. Among these, the MAE was implemented as a python script coupled with an Android emulator. The emulator was run on the Android 4.1.2 (level 16). The MAE passed the `odex` bytecode of a target application to the BPE and restored the emulator to the initial state only for reducing noise.

5.2. Experimental setup

For performance evaluation, Korea Internet and Security Agency (KISA) offered 906 malware samples[4] representing 13 malware families; these malware samples include up-to-date smishing and spy applications reported by mobile telecommunication companies and the Ministry of National Defense in South Korea. For benign class of applications, we downloaded a variety of popular applications with high rankings (as of March 2015) from GooglePlay. To further sanitize these benign samples, we also excluded samples diagnosed by at least one AV vendor, included in the VirusTotal dataset (VirusTotal, 2004). As a result, 1776 benign samples were used for our experiment. Duplicate malware samples were excluded according to their hash digest, and duplicate benign samples were excluded according to their hash digest and package name. We used the textual description produced by F-Secure (F-Secure, 1988) as a name. We used 5-fold cross-validation for the evaluation of our work. In a k-fold cross-validation method, data are divided into k folds, where $k - 1$ folds are used for training, and the k-th fold is used for testing. We rerun the evaluation by alternating the testing fold among the k folds, and compute the evaluation metrics each time. Finally, we average our results across the k runs (for the k folds). Our system was configured with the threshold values $T_{LR} = 1$ and $T_{SN} = 1$. It is reasonable to set T_{LR} as 1, which implies that our system detects a target application as malicious if the likelihood of malware is higher than that of a benign application. It is also reasonable to set T_{SN} as 1, according to data exploration described in section 3. We make a blacklist of serial numbers and the likelihood of the critical permission sets based on the training dataset.

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

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

Table 1 – The detection results of packed malware.

Packing method	Quantity	Detect	Miss
Apkprotect	6	6	0
Bangcle	4	4	0
None (+Lite[a])	896 (130)	880 (130)	16 (0)
Sum	906 (130)	890 (130)	16 (0)

Lite[a]: Apkprotect Lite, which only adds non-existing opcode.

5.3. *Experimental results and analysis*

In the following, we evaluate the performance of our system based on the effectiveness and the efficiency of detecting and classifying malware families.

5.3.1. *Effectiveness of malware detection*

We demonstrate that Andro-Dumpsys is effective in distinguishing malware from benign samples using the data in §5.2. As a result of this experiment, eight benign samples, corresponding to 0.45% of all benign dataset (false positive), were detected as malicious, whereas 16 malware samples, corresponding to 1.77% of all malware dataset, were detected as benign (false negative).

We note that, and as shown in this experiment, the anti-malware system must distinguish malware from benign applications with small errors, measured by both small values of false positive and false negative. We note that such findings are in line with the state-of-the-art: in the prior work (Feng et al., 2014), 10% false negative and 0.14% for false positive are achieved.

We demonstrate that Andro-Dumpsys provides a high true positive by identifying packed and malicious application as malware. As shown in Table 1, our system enables us to detect all packed malware samples, including smishing applications. However, as shown in Table 2, we find that some benign samples were labeled as malware in our system, which we explore further. We found that our system considered the applications providing free messaging (or call) services as malware, due to similar behavioral pattern with smishing. Some AV applications had the intent filter related to received SMS handling with high priority and codes for sending SMS messages; we presumed AV applications did not need these codes: superfluous privilege. We found that our system misclassified other applications providing spam-blocking service as malware.

To this end, we conducted an in-depth analysis in order to understand the false negatives generated by Andro-Dumpsys. Most of the false negatives circumvented our detection rules. In particular, upon further examination, we found that missed malware samples replicate and send smishing messages to their friends found in contacts, without hiding the malicious behavior. Some malware samples only request victim's financial secret card number (sensitive information widely used in South Korea), and transfer that sensitive information to the creator's remote server. Others hijack sensitive information, such as locations, call recordings, and call history, as a normal application, and transfer them to the creator's remote server.

We note that there is a trade-off between false positives and false negatives. We adjusted loose rules for reducing false positives, which caused our system to produce more false negatives. As for false negatives, Andro-Dumpsys failed to find malicious behavior through parsed footprints. However, the false negative rate is low, about 1.77%, compared to 10% false negative in recent work such as Feng et al. (2014), as mentioned earlier.

Additionally, we evaluated our approach using the serial number only to demonstrate the effectiveness of serial number-based detection, as shown in Table 2. As a result, 515 malware samples, corresponding to 56.84% of malware in our dataset, were filtered as malicious, and all benign samples were filtered as benign. Our primary screening provided effective and efficient malware detection by retrieving only certificate information. We note that while such approach is not inclusive of all malware, it is an efficient first-line filter of malware.

Finally, we conduct an analysis to determine the relative importance of features: what features have more influence on detection rate. For that, we highlight the detection decline ratio of the 906 malware samples in §5.2. Table 3 shows the results. Our baseline (case 0) corresponds to the scenario where all features are used, and two other cases correspond to scenarios where some features are discarded. In all cases, features are sorted in a descending order based on their contribution to our system's performance. In the first case (case 1), the ratio is degraded by 24.06% compared with the base case, whereas in the second case (case 2), the decline was by 14.72%, which highlight the significant contribution of the intent and serial number features to the performance.

Table 2 – The effect of detection rules.

Detection rule	Malware	Benign
Blacklist of serial number	515	0
Hiding SMS notifications	295	6
Pattern of smishing	80	2
Likelihood ratio	0	0
Total detected malware	**890**	**8**

Table 3 – The change of detection and decay rate for each feature attribute set (e.g., malware).

Case	Feature attribute set	Detect	Decline rate (%)
0	SN[a], API[b], Perms[c], Intent, Commands[d]	890	–
1	SN, API, Perms, Commands	676	24.04
2	API, Perms, Intent, Commands	759	14.72

SN[a]: serial number of a certificate, API[b]: API sequence, Perm[c]: permission distribution, Commands[d]: intersection of the usage of system commands and the existence of forged files.

5.3.2. *Effectiveness of malware classification*

We demonstrate the effectiveness of Andro-Dumpsys in malware classification. For that, we study the false positives and false negatives of classifying 13 malware families and benign applications. Table 4 shows the results, under the same settings shown earlier. We found that Andro-Dumpsys performed well in classifying malware families, producing 12 and 13 false positives and false negatives on average respectively.

Table 4 – Malware samples and benign samples for experiments.

Category	Family	Dumpsys-INT		Dumpsys-UNI	
		FPs	FNs	FPs	FNs
Malware (906)	Smforw (130)	66	25	66	24
	FakeBank (141)	24	10	23	12
	FakeKRBank (123)	6	10	11	6
	WroBa (117)	8	**72**	**10**	**74**
	Fakeinst (88)	**31**	**19**	**31**	**18**
	SmsSpy (79)	7	16	1	18
	MisoSMS (52)	3	0	4	0
	None[a] (43)	N/A[b]	N/A	N/A	N/A
	Gidix (40)	14	0	14	0
	Recal (25)	0	0	0	0
	SmsSend (25)	0	**19**	**0**	**20**
	TelMan (21)	5	3	4	2
	Helir (12)	4	0	4	0
	Fakeguard (10)	0	0	0	0
Benign (1776)		6	8	6	8
Average		12	13	12	13

FPs, FNs refer to false positives, false negatives.
None[a]: F-Secure fails to produce textual description; false negatives of F-Secure.
N/A[b]: We rule out malware whose textual description is none, when calculating FPs and FNs.
Bold text indicates the number of misclassified samples in our experiments.

Malware families, such as FakeKRBank, MisoSMS, Gidix, Recal, Helir, TelMan, and Fakeguard, were classified with low false positives and false negatives. However, the performance of classifying malware families such as WroBa, Fakeinst, and SmsSend was relatively low.

Some factors may affect classification of those families. By further exploration, we found that WroBa disguises itself as a GooglePlay application. If a victim launches the fake application, it seems to replace legitimate banking application with a malicious one, steals sensitive information, and monitors victim's device in the background (Paganini, 2013). The SmsSend sends SMS messages to a premium-rate number, and Fakeinst is an application installer that sends SMS messages to premium rate numbers (F-Secure, 2014).

However, in our data context, these malware families have different characteristics. They replicate and send the same SMS messages to recipients found in contact provider, and they also capture the incoming SMS messages and steal sensitive information (e.g., call log, contact information) to send it to the creator's remote server. To this end, they use an SMS message as a Command & Control (C&C) channel to communicate with a C&C server. Our system classifies these malware into similar behavior groups according to more granular malicious behavior criteria, while F-Secure's descriptions of these malware fail to capture all malicious behavior of malware. Since the F-Secure textual description provides fragmentary and broad malicious behavior, we believe our classification results are more specific.

5.3.3. *Effectiveness of detecting zero-day malware*

We demonstrate the effectiveness of identifying zero-day malware. We define a zero-day malware *as an application that has suspicious behaviors but is not previously detected by AV vendors.* We leveraged 10 malware samples offered by the Korea Internet and Security Agency (KISA). We uploaded those samples to the VirusTotal and checked the scanning results of various AV vendors, such as F-Secure, Avast, TrendMicro, Symantec, Kaspersky, McAfee, ClamAV, Sophos, and nProtect, among others. Through this scan, we noted that none of those applications is detected by any AV vendor as malware. However, Andro-Dumpsys detected all of the zero-day malware samples, having 100% detection accuracy. Note that in evaluating the zero-day samples on Andro-Dumpsys, we did not use any features extracted from those samples in the training process, and discriminative features of those zero-day samples are obtained through the analysis of the dataset highlighted above, which pertains to mass-market malware (Mohaisen and Alrawi, 2014).

5.3.4. *Efficiency of malware classification*

We examine the efficiency of our system across multiple criteria. First, we performed experiments to evaluate overhead of Andro-Dumpsys. For that, we selected 10 benign samples and 10 malware samples. The size of the benign samples was in the 40–50 MB range, and the malware samples were in the 6–20 MB range. We measure the CPU and memory utilization using the `vmstat` command, and measured the network utilization using `ifstat` command. We define the CPU utilization as the percentage of total CPU time spent running instructions of an application; we exclude the booting time of the emulator. And we define the memory utilization as the ratio of the amount of memory used by an application to the total amount of memory, and the network utilization as the ratio of current network traffic to the maximum traffic that the port can handle respectively; we exclude network traffic transferring an application between a server and a client when calculating current network traffic. Table 5 shows the average CPU, memory, and network usages introduced by Andro-Dumpsys. We find that the system overhead is less than 13% in all evaluation criteria; i.e., our proposed system has a reasonable performance impact on the server side, as compared to analyzing benign applications. Furthermore, that system overhead is dependent on hardware specification in use. Given that this evaluation is on a single processor without optimization, we further theorize that an optimized and multi-threaded system will allow for both horizontal and vertical scaling of the system, and benefit from unused resources.

We found that our system takes 74.18 seconds per megabyte to detect and classify malware samples. The majority of this time is spent creating the profile: it takes only 0.04 seconds on average to detect and classify malware into similar behavior groups. The overall elapsed time for analysis can vary according to the size of the code of malware, and hardware specification of the analysis system. The analysis time of system

Table 5 – The overhead of Andro-Dumpsys.

Category	Benign	Malware
CPU	12.04%	12.12%
Memory	6.62% (0.5 GB)	13.64 % (1.1 GB)
Network	0.003%	0.003%

has a fluctuation with certain mean and variance according to the number of entry points. In particular, analysis time ranges from 2.97 seconds per megabyte to 1667.1 seconds per megabyte (maximum). Note that the actually analyzed contents in our system are small in size. This is, the Android applications contain bytecode (classes.dex) in the form of Dalvik Executable (dex) file. The DEX specification limits the total number of methods that can be referenced within 2^{16}, including framework and library methods, and self-defined methods. That means dex is limited in size, regardless of the size of the application itself, which could be multiple gigabytes of code and supporting media (images, video, etc). Our system retrieves bytecode files (`smali`) based on component names in a manifest file, and determines whether or not a given application is malicious (not with an exhaustive search in contents).

6. Limitation

Andro-Dumpsys has a few limitations, since it extracts `odex` bytecode through dynamic analysis and employs static analysis to capture malware's behavior. First, the volatile memory acquisition process depends on emulator-based execution, and so, it is limited in analyzing malware embedding more sophisticated packing with obfuscation methods. For that, our system cannot attach Ptrace to extract meaningful bytecode in memory, resulting in a less meaningful analysis. Second, it is difficult for our system to analyze malware that is executed only under certain given conditions (e.g., shared library). These drawbacks are common in dynamic analysis and are addressed in the previous work in various ways. Depending on the number of malware samples to be analyzed, we can adopt manual inspection and feedback to analyze malware samples.

Second, the forgery of a certificate may significantly degrade the results of our system. We note that would only affect the part corresponding to the primary screening, and other features utilized in our system can still be useful: based on our experiments, the performance of our system was degraded by only 15% as a result of manipulating certificate features. Furthermore, the forged signature does not affect low-level certificate-related features. While the malware creators change their certificates, they do not change the core patterns of malicious behavior. For similar reason, AV vendors continuously update the signature database to respond to a large number of malware samples daily.

Finally, our system's overall CPU workload and elapsed time can differ based on the hardware specification. In this study, no optimizations are taken into account for performance evaluation, and the evaluation is used as rather a demonstration of how reasonably quickly the function of our system can be performed. In our future study, we plan to conduct an optimization of our system and compare ours with other approaches in the literature. In particular, to reduce average CPU usage, we may need to optimize for the idle time in the engines (e.g., Memory Acquisition Engine and Behavior Profiling Engine). For example, we may retrieve candidate entry points in `AndroidManifest.xml`. However, these entry points are not guaranteed to obtain bytecodes of an application.

7. Conclusion

In this paper, we proposed Andro-Dumpsys, an anti-malware system exploiting footprints obtained from volatile memory acquisition. Andro-Dumpsys can distinguish benign and malicious applications and classify malicious applications into similar groups. Furthermore, Andro-Dumpsys enables us to detect zero-day malware, which are missed by all antivirus scanners. Our experiments demonstrated that Andro-Dumpsys performs well in detecting malware with an accuracy of over 99% and classifying each malware family with low false positives/negatives. Our system hence enables AV vendors and security practitioners to respond to malware by detecting and classifying effectively and efficiently.

There are several directions that we will pursue in the future. First, our system conducts the volatile memory acquisition process on Dalvik Virtual Machine, not the Android Runtime (ART). Despite that the market share of the Lollipop is less than 2%[5], we will consider the volatile memory acquisition method on ART in the future for completeness of our results. Furthermore, we will upgrade our system to analyze malware embedding elaborately evolving anti-malware analysis techniques.

Acknowledgments

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]. In addition, this research is also supported by a Korea University Grant.

REFERENCES

Androguard. Reverse engineering, Malware and goodware analysis of Android applications. <https://code.google.com/p/androguard/>; 2011. [accessed 30.06.15].

Anubis. Anubis – Malware Analysis for Unknown Binaries. <https://anubis.iseclab.org/>; 2011.

Apkprotect. <http://www.apkprotect.com/>; 2013 [accessed 30.06.15].

Arp D, Spreitzenbarth M, Hübner M, Gascon H, Rieck K, Siemens C. Drebin: Effective and explainable detection of android malware in your pocket. In: Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS '14); 2014.

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. CCS '12; 2012. p. 217–28.

Bangcle. <http://www.bangcle.com/>; 2014 [accessed 30.06.15].

Desnos A. Androguard. <https://code.google.com/archive/p/androguard/>; 2011.

Enck W, Ongtang M, McDaniel P. On Lightweight Mobile Phone Application Certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. CCS '09; 2009. p. 235–45.

[5] The statistic about market share of each Android OS version is available at http://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/

F-Secure. Protect your life on every device – Internet security for device. <https://www.f-secure.com/en/web/home_global/home>; 1988 [accessed 30.06.15].
F-Secure. Threat Report H1 2014. <https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2014.pdf>; 2014 [accessed 30.06.15].
Feng Y, Anand S, Dillig I, Aiken A. Apposcopy: Semantics-based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM; 2014. p. 576–87.
Jang J, Yun J, Woo J, Kim HK. Andro-profiler: Anti-malware System Based on Behavior Profiling of Mobile Malware. In: Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion. WWW Companion '14; 2014. p. 737–8.
Jang J, Kang H, Woo J, Mohaisen A, Kim HK. Andro-AutoPsy: anti-malware system based on similarity matching of malware and malware creator-centric information. Digit Invest 2015;14:17–35.
Kang H, Jang J, Mohaisen A, Kim HK. Detecting and classifying android malware using static analysis along with creator information. Int J Distrib Sens Netw 2015;2015:doi:10.1155/2015/479174. Article ID 479174.
Khan A. What Is Odex And Deodex In Android. <http://www.addictivetips.com/mobile/what-is-odex-and-deodex-in-android-complete-guide/>; 2010 [accessed 30.06.15].
Kim D, Kwak J, Cheol Ryou J. DWroidDump: executable code extraction from android applications for malware analysis. Int J Distrib Sens Netw 2015; http://dx.doi.org/10.1155/2015/379682.
Leung KM. Naive Bayesian Classifier. Polytechnic University Department of Computer Science/Finance and Risk Engineering; 2007.
McAfee. McAfee Labs Threats Report, November 2014. <http://www.mcafee.com/ca/resources/reports/rp-quarterly-threat-q3-2014.pdf>; 2014 [accessed 30.06.15].
Mohaisen A, Alrawi O. AV-Meter: An Evaluation of Antivirus Scans and Labels. In: Detection of Intrusions and Malware, and Vulnerability Assessment - 11th International Conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings; 2014. p. 112–31. http://dx.doi.org/10.1007/978-3-319-08509-8_7.
Mohaisen A, Alrawi O, Mohaisen M. AMAL: high-fidelity, behavior-based automated malware analysis and classification. Comput Secur 2015;doi:10.1016/j.cose.2015.04.001.
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.
Paganini. Android Wroba banking Trojan targeted Korean users. <http://securityaffairs.co/wordpress/19041/malware/android-wroba-trojan-korea-banks.html>; 2013 [accessed 30.06.15].
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. ASIACCS '12; 2012. p. 71–2.
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. CCS '12; 2012. p. 241–52.
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. ACM; 2013. p. 209–20.
Reina A, Fattori A, Cavallaro L. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: EuroSec; 2013.
Seo SH, 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.
Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot-a Java bytecode optimization framework. In: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research. IBM Press; 1999. p. 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. ACM; 2014. p. 39–50.
VirusTotal. VirusTotal – Free Online Virus, Malware and URL Scanner. <https://www.virustotal.com/en/>; 2004 [accessed 30.06.14].
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. Technical Report TRISECLAB-0414-001; Vienna University of Technology; 2014.
Yan LK, Yin H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In: USENIX security symposium; 2012. p. 569–84.
Yang C, Xu Z, Gu G, Yegneswaran V, Porras P. DroidMiner: automated mining and characterization of fine-grained malicious behaviors in android applications. In: Computer security – ESORICS 2014, vol. 8712 of Lecture Notes in Computer Science. Springer International Publishing; 2014. p. 163–82.
Yu R. Android Packer facing the challenges, building solutions. <https://www.virusbtn.com/pdf/conference_slides/2014/Yu-VB2014.pdf>; 2014 [accessed 30.06.15].
Zhang M, Duan Y, Yin H, Zhao Z. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. CCS '14; 2014. p. 1105–16.
Zhang Y, Yang M, Xu B, Yang Z, Gu G, Ning P, et al. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security. CCS '13; 2013. p. 611–22.
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 Network and Distributed System Security Symposium(NDSS '12); 2012.

Jae-wook Jang is taking a doctoral degree course in Graduate School of Information Security, Center for Information Security Technologies (CIST), in Korea University. His research interests include mobile malware analysis, network security, and intrusion detection. Contact him at changkr@korea.ac.kr

Hyunjae Kang received a master's degree course in Graduate School of Information Security, Center for Information Security Technologies (CIST), in Korea University. Her research interests include mobile malware analysis and computer security. Contact her at janetk1004@gmail.com.

Jiyoung Woo received her Ph.D in Industrial Engineering from Korean Advanced Institute of Science and Technology in 2006. Currently, she is a research professor in Graduate School of Information

Security, Center for Information Security Technologies (CIST), in Korea University. Her research interests include Social Media Analytics and Online Game Security. Contact her at jywoo@korea.ac.kr.

Aziz Mohaisen obtained his M.S. and Ph.D. degrees in Computer Science from the University of Minnesota, both in 2012. He is currently an assistant professor at the Computer Science and Engineering Department of the State University of New York at Buffalo. From 2012 to 2015, he was a senior research scientist at Verisign Labs. Before pursuing graduate studies at the University of Minnesota, he was a member of Engineering Staff at the Electronics and Telecommunication Research Institute, a large research and development institute in South Korea. His research interests are in the areas of networked systems, systems security, data privacy, and measurements. Dr. Mohaisen is a senior member of Institute of Electrical and Electronics Engineers (IEEE) and Association for Computing Machinery (ACM). Contact him at mohaisen@buffalo.edu.

Huy Kang Kim received his Ph.D. in Industrial and Systems Engineering from Korea Advanced Institute of Science and Technology (KAIST) in 2009. Currently, he is an associate professor in Graduate School of Information Security, Center for Information Security Technologies (CIST), in Korea University. His research interests include Botnet Detection, Intrusion Detection System, Network Forensics and Online Game Security. Contact him at cenda@korea.ac.kr