Hindawi Publishing Corporation
Mobile Information Systems
Volume 2016, Article ID 6707524, 11 pages
http://dx.doi.org/10.1155/2016/6707524

Research Article

Function-Oriented Mobile Malware Analysis as First Aid

Jae-wook Jang and Huy Kang Kim

Graduate School of Information Security, Korea University, Seoul 136-713, Republic of Korea

Correspondence should be addressed to Huy Kang Kim; cenda@korea.ac.kr

Received 3 November 2015; Accepted 2 February 2016

Academic Editor: Seung Yang

Recently, highly well-crafted mobile malware has arisen as mobile devices manage highly valuable and sensitive information. Currently, it is impossible to detect and prevent all malware because the amount of new malware continues to increase exponentially; malware detection methods need to improve in order to respond quickly and effectively to malware. For the quick response, revealing the main purpose or functions of captured malware is important; however, only few recent works have attempted to find malware's main purpose. Our approach is designed to help with efficient and effective incident responses or countermeasure development by analyzing the main functions of malicious behavior. In this paper, we propose a novel method for function-oriented malware analysis approach based on analysis of suspicious API call patterns. Instead of extracting API call patterns for malware in each family, we focus on extracting such patterns for certain malicious functionalities. Our proposed method dumps memory sections where an application is allocated and extracts suspicious API sequences from bytecode by comparing with predefined suspicious API lists. By matching API call patterns with our functionality database, our method determines whether they are malicious. The experiment results demonstrate that our method performs well in detecting malware with high accuracy.

1. Introduction

As mobile devices become widely spread, highly well-crafted mobile malware has become one of the most dangerous threats to the mobile computing environment. According to a report by McAfee [1], the total amount of mobile malware has continued its steady climb as it broke 6 million in the 4th quarter of 2014 and increased by 14% over the 3rd quarter of the same year. Moreover, approximately 0.8 million new malware families and variants were reported to appear in the same quarter. Antivirus (AV) vendors analyze a large amount of malware samples daily, and prevent them from spreading widely; in other words, offenders and defenders have been waging an endless battle.

Despite the efforts of AV vendors, detecting and preventing all malware become difficult because malware tends to evolve over time. In particular, the number of malware samples that embed antimalware analysis methods, such as Android packer (e.g., APK Protect [2] and Bangcle [3]), dynamic loading, and `dex` encryption, has increased steadily [4]. Android packers enable encrypting an original `dex` bytecode, decrypting the `dex` bytecode on memory section at runtime, and then executing the `dex` bytecode via `DexClassLoader`. Under these circumstances, it is impossible to detect and prevent all malware and its variants, especially malware that embeds antianalysis techniques.

As shown in Figure 1, based on their purpose or object, we categorize malware analysis approaches into three types: autopsy-oriented approach, function-oriented approach, and symptom-oriented approach. The autopsy-oriented approach aims to represent how malware conducts malicious behavior and consists of traditional static and dynamic analysis. Static analysis aims to examine the binary code in order to determine its characteristics without executing it. Many static analysis approaches in previous works have failed to parse meaningful footprints from malware samples that embed these antianalysis techniques [5–9]; the malware that embeds antianalysis techniques leads malware analysts to problems. On the other hand, dynamic analysis aims to provide methods for extracting the unique behavioral patterns of malware. Dynamic analysis addresses obfuscation, packing, and encryption methods because all methods are eliminated during execution of a malicious application [10–13]. However, dynamic analysis is performed only on a part of the executed

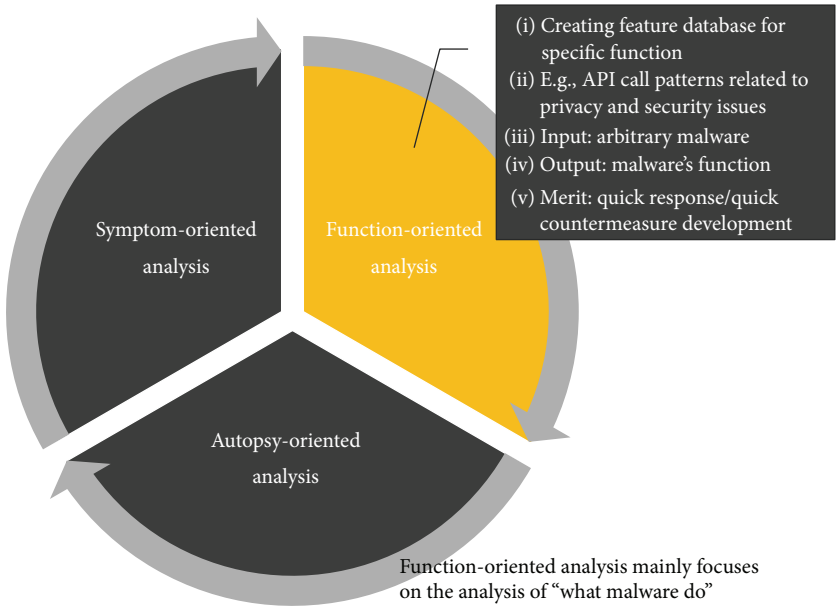


Figure 1: Categorization of malware analysis approaches.

application, and malicious behavior should be executed during analysis time; malware analysts might believe that malware conducts malicious behavior during analysis time. Most of the previous works have presented the traditional autopsy-oriented approach. However, the autopsy-oriented approach has a limitation in estimating malware's destructive power or influence on our lives because it is only dedicated to classifying malware samples into the most similar groups.

On the other hand, the function-oriented approach creates a feature database for specific functionalities. This approach allows efficient and effective incident responses or countermeasure development by analyzing the functionalities related to malicious behavior. Moreover, the function-oriented approach has evolved over time to become the symptom-oriented approach. The symptom-oriented approach, known as behavior analysis or reputation-based analysis, allows malware analysts to utilize the threat intelligence gathered by security communities. This approach prevents widespread dissemination of computer crime, although relevant signatures do not yet exist. When all static analysis and dynamic analysis methods fail, behavior observation can be useful for estimating a given application's maliciousness.

Tam et al. [14] manually inspected various malware samples from repositories such as contagion and Android Malware Genome Project. They classified malicious behaviors into six groups according to behavior patterns: calling, sending SMS, network access, retrieving personal information, altering filesystem, and executing external applications. We also manually examined malware samples we had and the extracted API call patterns for certain malicious functionality; since some behavior patterns are not found in our dataset, we leave them out. Few recent works focus on the fact that malware in different categories can have common API call sequences when the main purpose or functionalities are the same [15]. These approaches help detect malware efficiently by analyzing the core behavior functionalities of malware samples. Because modern malware has complex functionalities, traditional similarity matching-based malware classification can miss the malware's purpose if such malware has a critical functionality part that is much smaller than its noncritical functionalities. The critical functionality we define is the main target of malware attack, whereas the noncritical functionalities we define are additional malicious behaviors for concealing the critical functionality. For example, malware authors can mostly reuse known malware code and insert small pieces of code to cause intentionally analysts to lose focus. In addition, an automated analysis tool is based on similarity matching only reports that the given malware is similar to known malware. Therefore, in this case, the malware's critical functionality (e.g., deleting MBR) cannot be detected quickly. Without determining the malware authors' intent, AV vendors can fail to analyze significant malicious footprints or behavior patterns; they only concentrate on extracting fragmental signatures for malware detection or classification (Section 3). If excluding the noncritical functionalities purposely, AV vendors are limited to capture influentially malicious functionalities and they are prone to fail occasionally to detect malware.

To overcome the drawbacks of the traditional autopsy-oriented approach, we propose the function-oriented approach to more efficiently determine what a given malware wants to do. For this, malware analysts need to know what the malware authors want to obtain and how they trigger malicious behavior. Such intent of the malware authors can provide malware analysts with vital clues for identifying malware, and malware analysts can determine the malware authors' attack patterns from them and leverage these patterns as detection rules. To achieve the end goals, we introduce a hybrid antimalware system that combines static

analysis and dynamic analysis and focuses on extracting malicious functionalities related to the malware authors' intent. Our approach is not an exhaustive search for parsing meaningful footprints, but a fast primary screening.

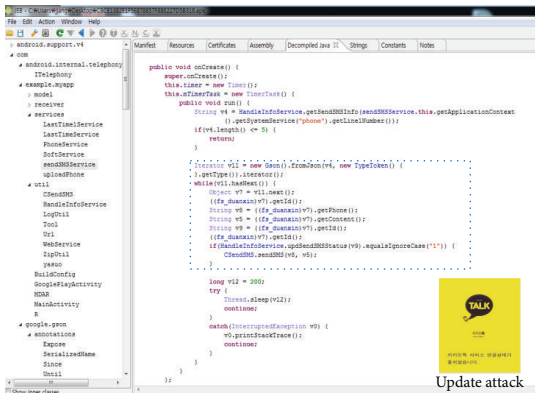
Despite malware that embeds various antianalysis methods, the ndroid platform decrypts (or unpacks) all malware on the memory section. Our system catches the moment when the `odex` bytecode is loaded onto the memory section [16] and leverages artifacts for malware analysis. In particular, our proposed system runs a malicious application on an emulator, dumps the meaningful volatile memory section (`odex` bytecode) where a target application is allocated, and extracts suspicious APIs from the `odex` bytecode in order to overcome challenging issues (i.e., addressing malware embedding antianalysis methods). Instead of extracting all API call patterns for malware in each family, as most previous works have done, we focus on extracting API call patterns for certain malicious functionality. As comparing these with the predefined suspicious API list in detection ruleset, our system enables the effective and efficient malware detection.

To this end, the contributions of this work are as follows. First, as first aid, our system can respond quickly to many species of malware samples by analyzing the functionalities related to malicious behavior. Second, we propose a hybrid malware detection method coupled with a memory acquisition method to enhance detection accuracy. Finally, with a recent real-world malware dataset, we empirically study whether the proposed approach generates superior results.

The rest of this paper is organized as follows. In Section 2, the related work is reviewed. In Section 3, we discuss the limitations of the autopsy-oriented approach. In Section 4, we present our methodology and experiment. In Section 5, we discuss the limitations of our proposed method, and we conclude with Section 6.

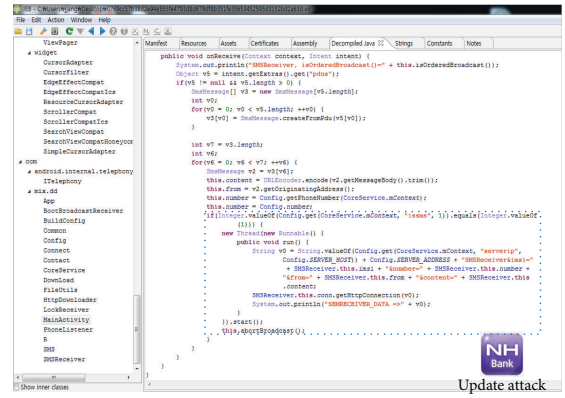
2. Related Work

Most of the previous works falls into two broad types: static and dynamic analysis. They have mainly presented the traditional autopsy-oriented approach. In static analysis, code patterns of malware samples through reverse engineering are used for a signature for malware detection. On the other hand, dynamic analysis aims to extract unique behavior patterns of each malware sample based on its behavior. Enck et al. [10] proposed the Kirin security service, a lightweight antimalware system, to mitigate malware at installation time. Kirin inspected the requested permissions in a manifest file and determined whether malicious activities were executed by comparing them with predefined rules. Pearce et al. [12] proposed a privacy information antileakage system, AdDroid, that separated permissions related to advertisement from other permissions. However, AdDroid was limited to responding to the majority of malware that does not spread via advertisements. Peng et al. [6] proposed probabilistic models to represent risk scoring, ranging from the simple Naïve Bayes to advanced mixture models. Their proposed system computed a risk score for each application based on the requested permissions in a manifest file and determined whether malicious activities were executed. Wang et al. [7] proposed DroidRisk, a quantitative risk assessment model of permissions. By quantifying the risk level of each application, they presented a reliable risk indicator that could be generated to warn of potential malicious activities. However, application developers tended to declare an unnecessary number of requested permissions in a manifest file, despite the application not requiring them at all; therefore, these methods were prone to high false positives. Recently, these methods have also considered the requested permission for using each API method in reality by analyzing an API call graph. However, these detection methods are not efficient for classifying benign applications as benign because the relevant rule sets merely focus on detecting malware. Zhang et al. [13] proposed VetDroid, a dynamic analysis approach for extracting the sensitive behavior of a given application. By leveraging the API-related permission table, VetDroid completely identified all possible API-related permission uses. However, their methods are not effective for identifying benign applications because the relevant rules only focus on detecting malware, thus causing large false alarms. Arp et al. [5] proposed DREBIN, which considered permissions and sensitive APIs as features. DREBIN extracted features from both a manifest file and bytecode and then identified malware automatically. Yang et al. [8] proposed DroidMiner, which automatically mined malicious behavior from a behavioral graph. DroidMiner considered the frequency of APIs and the dependencies between multiple sensitive API functions. Zhang et al. [9] proposed DroidSIFT for malware classification based on a weighted contextual API dependency graph. To prevent malware from spreading, DroidSIFT exploited graph similarity metrics for malware detection. Zhou et al. [17] proposed DroidRanger, a permission and heuristic-based method. Wu et al. [18] proposed DroidMat, a feature-based malware detection method. DroidMat chose the requested permissions, intent, and API calls as feature vectors and characterized the malicious behavior of each application. DroidMat used K-NN and K-means classifiers to determine whether a given application was malicious. Zheng et al. [19] introduced a signature-based analytic system, DroidAnalytics, to automatically collect malware, generate signatures for malicious application, and identify malicious logics to the opcode level. Moreover, DroidAnalytics easily discovered repackaged applications by estimating the similarity score. Deshotels et al. [20] proposed DroidLegacy, an automated method for extracting each malware signature. DroidLegacy identified repackaged malware by constructing signatures using API calls. DroidLegacy partitioned an APK into loosely coupled modules and compared the API calls of each module with the signature of each family. The API calls invoked by each malicious module were transformed into a signature for malware analysis. Lee et al. [21] proposed a malware detection method by leveraging signature entries or exactly matched binary patterns to estimate the similarity score. For detecting malware variants, they used the signature entries that consisted of class/method name, character string, and method body in the dex bytecode. However, these approaches have a limitation in parsing meaningful code patterns from an application that embeds antimalware analysis techniques (e.g., packing, dynamic loading, and bytecode encryption).



SHA256: d5928a3c9bf517daf7f96dd49024d2fe3c243551dc256ed4bb07763ce68bd806

(a) Fakeinst



SHA256: 0769cc57b38d2e94e5b5fe4793d8d679df8b351fe35653452595d3152bd2a63d

(b) SMSSend

Figure 2: In-depth analysis results of smishing samples. Fakeinst sequentially sends the same SMS message to recipients found in the contacts provider (marked by a dotted blue line box in (a)). SMSSend intercepts incoming SMS messages and sends that information to a remote server (marked by a dotted blue line box in (b)). However, F-Secure's descriptions fail to represent the critical behavior of the malware; these textual descriptions only indicate that the malware appears to be an application installer, but it sends SMS messages to premium rate numbers.

The previous works (autopsy-oriented approach) only take whole analysis time for detecting and classifying malware into the most similar groups by comparing a malware sample with the signatures of each malware group. These signatures do not represent malicious functionalities, and when circumventing the signatures of malware groups, malware easily achieves the goal of the malware author. In order to prevent missing vital clues when identifying malware, in this paper, we propose the function-oriented approach based on malware attack patterns that represent the malware attacker's intent. In the following section, we first study the aforementioned limitations of the traditional malware analysis and then review the proposed method.

3. Case Study: Limitations of Autopsy-Oriented Approach

Many previous works on API call sequence analysis have classified malware based on the textual description produced by AV vendors. However, such textual description often fails to capture all malicious behavior. Recently, we found elaborate malware of type Smishing (SMS phishing) and conducted in-depth analysis of the malicious application. As shown in Figure 2(a), the malware Fakeinst conducts an update attack: it disguises itself as the most popular chatting application (KaKaoTalk) in Korea. Fakeinst copies and sends the same SMS messages to recipients found in the contacts provider. In addition, Fakeinst captures incoming SMS messages and steals sensitive information, such as call history and contact information, in order to send that information to a remote server. As shown in Figure 2(b), the malware SMSSend conducts an update attack: it disguises itself as a well-known banking application in Korea. Similar to Fakeinst, SMSSend captures incoming SMS messages and steals sensitive information in order to send that information to a remote server. It also copies and sends the same SMS messages to recipients found in contacts provider.

However, the traditional malware analysis (autopsy-oriented) approach has a limitation in explaining the destructive power or influence on malware on our lives because its concern is on whether a suspicious application is malicious by leveraging known signatures; traditional malware analysis does not explain all the malicious behavioral patterns. Textual descriptions produced by F-Secure only indicate that Fakeinst and SMSSend appear to be application installers, but such malware sends SMS messages to premium rate numbers [22]. F-Secure's description fails to capture all the malicious behavior of the malware because it only explains fragmentary and broad malicious behavior. By excluding the premium-rate SMS function, the malware sample can cause confusion to AV vendors. Given that AV vendors do not focus on capturing influentially malicious functionalities, they are prone to occasionally fail to detect malware that circumvents their signature. In the aforementioned example, the malware author might evade AV vendors by hiding the function of sending premium-rate SMS. By analyzing the core behavior functionalities of malware samples, malware analysts can detect malware efficiently. In this case study, we only mention the F-Secure [23] case, but this is a common error of all mobile AV vendors.

4. Materials and Methods

4.1. Environment Setup. We performed all experiments in a virtualization environment (VMWare ESX_i; http://www.vmware.com/). As shown in Figure 3, our system consists of an analyzer and repository. The analyzer and repository are installed on the server, which runs an Ubuntu 12.04 LTS (64-bit) operating system. We implemented the analyzer using the Python programming language (as scripts) with an Android

Table 1: Malware and benign samples for experiments.

Category	Family	Number of samples	Family	Number of samples
Malware (906)	Smforw	130	None	43
	FakeBank	141	Gidix	40
	FakeKRBank	123	Recal	25
	WroBa	117	SmsSend	25
	Fakeinst	88	TelMan	21
	SmsSpy	79	Helir	12
	MisoSMS	52	Fakeguard	10
Benign		1,776		

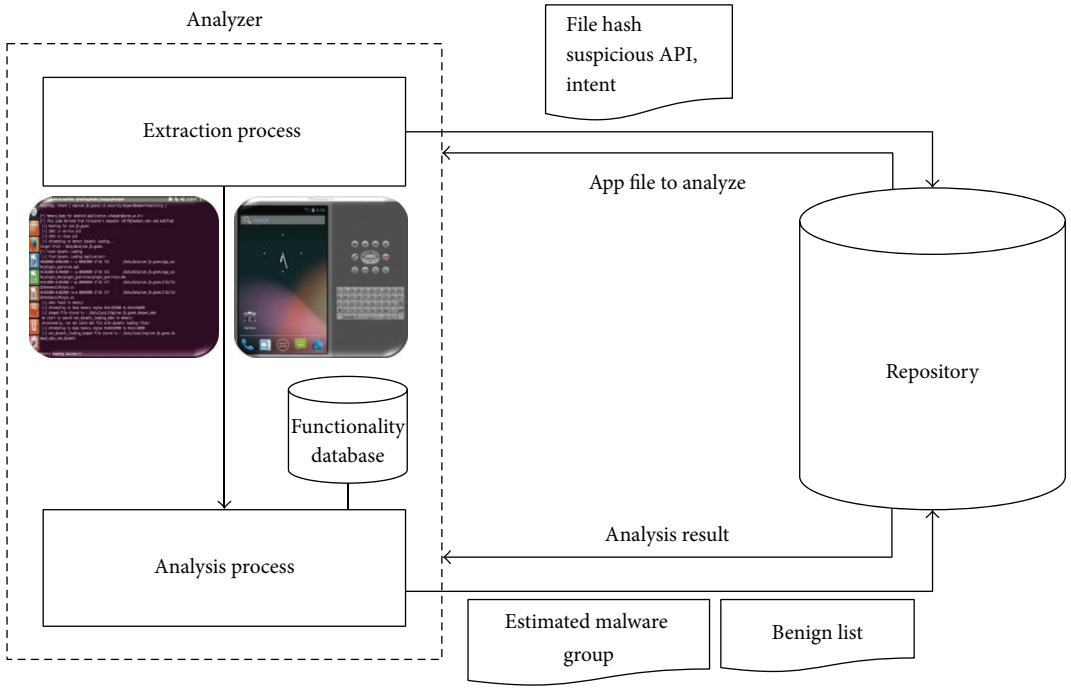


Figure 3: System architecture.

emulator that runs on Android 4.1.2 (level 16). The analyzer is composed of an extraction process for volatile memory acquisition and analysis process for malware detection. The analyzer restores the emulator to the initial state when a new application enters its queue.

4.2. Dataset Preparation. As shown in Table 1, we chose 906 malware samples composed of 13 malware families provided by the Korea Internet Security Agency (KISA) from March to December 2014. These malware samples are the latest smishing and spy applications reported by the Ministry of National Defense and mobile telecommunication companies in South Korea. To sanitize benign samples, we downloaded popular free applications with high rankings from Google Play in March 2015 and excluded the samples diagnosed by at least one AV vendor at VirusTotal; 1,776 benign samples were used for the experiments. Duplicate benign samples were excluded according to SHA256, and duplicate malware samples were excluded according to SHA256 and the package name. Our entire dataset is accessible at http://ocslab.hksecurity.net/sapimmds.

4.3. API Categorization. The Application Programming Interface (API) is a set of functions provided conveniently to control the main actions on a given platform, for example, the Android platform. For instance, we can easily implement the "sending an SMS message" functionality by calling some APIs, such as sendTextMessage() or sendMultimediaMessage(). Seo et al. [24] analyzed a variety of malware and benign samples and studied the distribution of APIs requested by each dataset, listing suspicious APIs.

Although various malicious behaviors seem to be similar to each other, the APIs found in malware vary based on the malware authors. To resolve this problem, we updated the suspicious API list by examining all the APIs that work similarly to those suspicious APIs listed by [24]. These APIs are involved in gathering private or system information, sending and deleting SMS messages, recording voice, and accessing

Table 2: Examples of malicious functionalities and their suspicious API call patterns.

Category	Suspicious API call pattern	Additional information
Hiding SMS notification	{getOriginatingAddress() ∨ getMessageBody() ∨ getDisplayMessageBody()} $\wedge^*$ abortBroadcast()	android.provider. Telephony.SMS RECEIVED with high priority
Hiding shortcut	setComponent EnabledSetting() $\wedge^*$ abortBroadcast()	
SMS message hijacking	{query() ∨ parse()} $\wedge^*$ getExternalStorageDirectory() ∧ getExternalStorageState() $\wedge^*$ Transmission APIs[a]	content://sms $\wedge^*$ {Hiding SMS notification ∨ Hiding shortcut}
Contacts content hijacking	getContentResolver() $\wedge^*$ query() $\wedge^*$ getLine1Number() $\wedge^*$ Transmission APIs	{Phone.CONTENT_URI ∨ Contacts.CONTENT_URI} $\wedge^*$ {Hiding SMS notification ∨ Hiding shortcut}
Bookmark hijacking	getContentResolver() $\wedge^*$ Transmission APIs	BOOKMARKS_URI $\wedge^*$ {Hiding SMS notification ∨ Hiding shortcut}
Location information content hijacking	getLastKnownLocation() $\wedge^*$ Transmission APIs	{Hiding SMS notification ∨ Hiding shortcut}
Hijacking certificate for financial transaction	{getExternalStorageDirectory() ∨ getExternalStorageState()} $\wedge^*$ FileOutputStream ∧ ZipOutputStream.close() $\wedge^*$ Transmission APIs	/npki $\wedge^*$ {Hiding SMS notification ∨ Hiding shortcut}

[a]The APIs of AQuery.ajax() or HttpClient() or DefaultHttpClient() or URLConnection() or HttpURLConnection() class.
A ∧ B denotes that malware calls functions A and B successively (other API calls cannot be executed between A and B).
A $\wedge^*$ B denotes that malware calls functions A and B, but not necessarily successively (other API calls can be executed between A and B).
A ∨ B denotes that malware calls function A or B.

the content provider and web services. Details of the suspicious APIs that we defined are listed in Appendix.

4.4. Suspicious API Call Patterns of Known Malicious Behavior. In order to extract suspicious API call patterns, we reviewed previous our work [25]. As a result of this analysis, we found that most malware hijacks sensitive information without the victim's consent; occasionally, the malware is controlled by the adversary's C&C server via SMS messages. In addition, we found that the malware causes the victim to subscribe to premium services without his/her notice by removing the SMS confirmation message. To achieve its goal, malware requests the highest priority for received SMS messages and calls `abortBroadcast()` to hide a notification of SMS messages. Similar to premium-rate SMS, smishing applications receive SMS messages for C&C from the remote server and send hijacked sensitive information (e.g., contacts, SMS content, call logs, and certificate for financial transactions in South Korea). In this case, the smishing applications hide the received SMS message after the malicious behavior is completed. Thus, we have concluded that malware has distinct malicious behavior patterns and characteristics, and the malicious behavior is influenced by the invoked API call patterns. Using a feature database for those API call patterns, we can identify more efficiently whether the application is malicious. To this end, we analyze the relationship between malicious behavior and API call patterns. Table 2 lists the samples of suspicious API call patterns that we found, and the following descriptions explain how suspicious API call patterns cause malicious behavior.

(1) Hiding SMS notification: with high priority, malware successively calls APIs related to incoming SMS message handling, and then such malware calls `abortBroadcast()` to cancel the broadcasting of SMS-related actions. This malicious behavior pattern is indispensable for conducting additional malicious behavior without the victim's consent.

(2) Hiding shortcut icon: because malware hides its shortcut icon, the user cannot be aware of installing such malware. This malicious behavior pattern is indispensable for conducting additional malicious behavior without the victim's awareness.

(3) SMS message hijacking: when given the URI for managing SMS messages and APIs for querying such URI, malware can access the SMS content provider. Moreover, when given the APIs for accessing external storage and APIs for transmitting data, malware can access that storage in order to save the victim's SMS messages and transmit them to a remote server. This type of malicious behavior always requires some APIs to communicate with the remote server. The functions

for hiding SMS or shortcut icons are necessary for this malicious behavior.

(4) Contacts hijacking: when given the URI to manage contact contents and the APIs to query that URI, malware can access the contacts provider. Moreover, when given the APIs to transmit data, malware can transmit contact contents to a remote server along with the device's identifier information. This malicious behavior always requires some APIs to communicate with the remote server. The functions for hiding SMS or shortcut icons are necessary for this malicious behavior.

(5) Bookmark or location information hijacking and hijacking certificates for financial transactions: similar to SMS message hijacking, malware transmits this information to a remote server. In the case of hijacking certificates for financial transactions, given that the unique string "npki" represents the folder name of a given financial certificate in Korea, malware can compress that folder into a zip archive in the external storage and then transmit this information to the remote server. This malicious behavior always requires some APIs to communicate with the remote server. The functions for hiding SMS or shortcut icons are necessary this for malicious behavior.

4.5. API Call Pattern Matching. In order to compare the API call patterns of the application with the feature database for suspicious API call patterns, we adopted the longest common subsequence (LCS) algorithm [26] using dynamic programming. The LCS algorithm finds the longest common subsequence of two strings, where a subsequence is the sequence that maintains the same relative order, although not necessarily continuous. Since problem finding the LCS is one of the most famous problems in computer science and the basis of data comparison, the LCS is leveraged in malware analysis domain [15, 27]. For example, the sequences "abcd" and "abbdccccbd" have an LCS of "abcd." If the result of LCS is found in the feature database for suspicious API call patterns, we determine that the application conducts malicious behavior.

4.6. Overall Process. In the extraction process, our method dumps the meaningful volatile memory section where a given application is allocated on an Android emulator, as shown in Figure 3. In order to dump that memory section, we modified Strazzere's Native Development Kit code (https://github.com/strazzere/android-unpacker) that allows unpacking elaborate packing methods without depending on `gdb`. We enhanced this tool to dump the volatile memory section of the application that embeds the dynamic loading method with bytecode encryption. Memory acquisition then proceeds in the following order. First, our system retrieves a process ID (PID) of the application and a cloned process ID (CID), according to the package name. Next, our method retrieves the memory boundaries of the running CID using `/proc/CID/maps` by checking whether the packing and dynamic loading methods are adopted. If the application adopts packing and dynamic loading methods, the `odex` bytecode is allocated in the memory section after unpacking and dynamic loading are completed. Finally, our method attaches that process to `PTRACE` and copies the memory section whose signature is "`dey`" from `/proc/CID/mem`. In the extraction process, our method dumps the meaningful volatile memory section where a given application is allocated on an Android emulator, as shown in Figure 3.

In the analysis process, our method extracts the package name as an identifier, component name, and intent-specific information in `AndroidManifest.xml` from the `apk` file and disassembles the `odex` bytecode into `smali` code. After sorting the parsed components in ascending order, our method searches only `smali` files and the folder with the same component name. In the case of the application that embeds the dynamic loading method, our method retrieves all `smali` files because we believe that the dynamic loaded code can hide malicious codes. Dynamic loading method provides flexible memory allocation and extends the dynamic functionality during runtime. An application developer leverages dynamic loaded code for intellectual property protection. However, when utilized in malware, it is difficult for malware analyst to analyze malware embedding dynamic loaded codes; the number of malware families adopting antianalysis techniques has increased rapidly [4]. Furthermore, our method extracts suspicious APIs from `smali` files and creates a suspicious API sequence by comparing with the predefined suspicious API list. Our method determines whether the target application is malicious by checking predefined malicious functions or records of the functionality database in the suspicious API sequence of that application. After completing the analysis, the process stores the result in the repository.

4.7. Accuracy Test. We demonstrate that our method provides an effective metric for detecting packed malicious application and identifying malicious application as malware. We demonstrate that our proposed method provides an effective metric for distinguishing malware samples from those benign samples. Table 3 is a confusion matrix that shows the performance of the detection algorithm. As a result, eight benign samples were detected as malware (false positives), and 147

Table 3: Confusion matrix for malware detection.

Category	Actual class	
	Malware	Benign
Estimated class		
Malware	759	8
Benign	147	1,768

Table 4: Detection results for packed malware.

Packing method	Number of samples	Detected	Missed
APK Protect[a]	6	6	0
Bangcle	4	4	0
None	896	749	147
Total	906	759	147

[a]We use APK Protect professional version.

Table 5: Suspicious API list that we defined.

Category	Class	Method
Retrieving system information	TelephonyManager	getDeviceId() getLine1Number() getNetworkOperator() getSimOperatorName() getSimSerialNumber() getSubscriberId() getCallState()
	UUID	toString()
	WifiInfo	getMacAddress()
	WifiManager	getConnectionInfo() getWifiState()
Retrieving personal information	LocationManager	getLastKnownLocation() requestLocationUpdates()
	ContentResolver	query(), delete()
	Audio.Media	getContentUriForPath()
	Images.Media	getContentUri()
	Video.Media	getContentUri()
	—	getContentResolver()
	Uri	parse()
Sending or receiving SMS	SmsManager	getDefault() sendTextMessage() createFromPdu() getDisplayMessageBody() getMessageBody() getOriginatingAddress() getUserData()
	gsm.SmsManager	sendTextMessage() createFromPdu() getDisplayMessageBody()
Calling	telephony.ITelephony	endCall()
Recoding	AudioRecord	startRecording()
	MediaRecorder	start(), stop()
Data transmission	HttpURLConnection	getOutputStream()
	URLConnection	getInputStream() getOutputStream()
	ssl.HttpsURLConnection	getOutputStream()
	client.HttpClient	execute()
Data transmission	client.DefaultHttpClient	execute()
	JSONObject	put()
	AQuery	ajax()
Device policy management	DevicePolicyManager	isAdminActive(), lockNow()
	DeviceAdminReceiver	—
Dynamic loading	AssetManager	getAssets()
	DexClassLoader	loadClass()
	SecureClassLoader	—
	URLClassLoader	—
	Runtime	exec(), getRuntime()
	VMRuntime	getRuntime()
	System	load(), loadLibrary()

Table 5: Continued.

Category	Class	Method
Encryption	crypto.Cipher	doFinal() getInstance()
	crypto.KeyGenerator	generateKey()
	SecretKeySpec	—
Reflection	Class	getDeclaredMethod()
	reflect.AccessibleObject	setAccessible()
ETC	PendingIntent	getBroadcast()
	—	abortBroadcast
	FileOutputStream	—
	ZipOutputStream	close()
	PackageManager	setComponentEnabledSetting()
	Environment	getExternalStorageDirectory() getExternalStorageState()
	String	equalsIgnoreCase(), split()
	ActivityManager	restartPackage()
	AudioManager	setVibrateSetting() setRingerMode()
	Context	getSystemService()

malware samples that correspond to 16.23% of all malware samples were detected as benign (false negatives). To summarize, precision (precision = true positive/(true positive + false positive)), recall (recall = true positive/(true positive + false negative)), accuracy (accuracy = (true positive + true negative)/(true positive + false positive + false negative + true negative)), and the F-Score (F-Score = 2TP/(2TP + FP + FN) where TP is true positive, TN is true negative, FP is false positive, and FN is false negative) are 0.990, 0.838, 0.942, and 0.907, respectively. This implies that our proposed malware detection method is highly reliable. Moreover, the proposed method allows detecting all packed malware samples as malware, as shown in Table 4.

We demonstrate that our method provides an effective metric for identifying zero-day malware. We define a zero-day malware as an application with malicious behavior undetected by AV vendors. We leveraged ten of the malware samples offered by KISA and then checked the scanning results of various AV vendors, such as F-Secure, Avast, TrendMicro, Symantec, Kaspersky, McAfee, ClamAV, Sophos, and nProtect. The results of our experiment show that our methods can detect all zero-day malware samples.

Comparing Different Methods. To the best of our knowledge, the closest approaches in the literature to our approach are Wu et al. [18], Zheng et al. [19], Deshotels et al. [20], and Lee et al.'s work [21]. Wu et al.'s method [18] distinguished malware from benign applications with 97.87% accuracy. Zheng et al.'s method [19] detected 2,494 malware samples and 342 zero-day malware samples. Deshotels et al.'s method [20] detected malware with 94% accuracy and approximately 95% precision/recall. Lee et al.'s method [21] had accuracy and recall rates of more than 97%. For completeness of our approach, we need to compare ours with these approaches. However, they are not available for public use. Moreover, these approaches need more improvement for addressing such antimalware analysis techniques as packing, dynamic loading, and bytecode encryption. In contrast with these, our approach allows analyzing malware that embeds antimalware analysis techniques. Thus, we expect to detect more malware that embeds these techniques.

5. Limitation & Future Works

Our prototype has not yet implemented a module for malware classification. For such classification, a similarity comparison between suspicious API sequences enables calculating through sequence alignment algorithms, such as the Needleman-Wunsch and Smith-Waterman algorithms. In order to compare suspicious API sequences with each other, we transform the API method into ASCII code. The converted letters, for example, the ASCII code sequence, represent the behavior patterns of each application. Moreover, our proposed method extracts `odex` bytecode through the volatile memory acquisition process and employs static analysis to capture malicious behavior footprints. Similar to other dynamic analysis-based methods, ours has a limitation when the given malware is obfuscated by more sophisticated packing methods. In this case, our method fails to attach `PTRACE` in order to dump the meaningful bytecode in the memory section, thus yielding a less meaningful analysis. However, this drawback is common in dynamic analysis. To overcome such limitation, we will study the deobfuscation method to enhance our system in future work.

6. Conclusion

Most malware detection systems focus on analysis based on the signature or similarity matching of malware families, and

they are not suitable for quick analysis intended to find the main purpose of such malware. For quick response against malware, we adopted the function-oriented approach based on suspicious API call patterns. Our proposed method dumps the meaningful volatile memory section where a target application is allocated and extracts suspicious APIs from `odex` bytecode by comparing with a self-defined suspicious API list. By matching API call patterns with our functionality database, our method can detect the functionalities implemented in a given mobile malware. Our experiments demonstrated that the proposed method performs well in detecting malware with high accuracy. We believe that our proposed method can be useful for responding to ever evolving malware.

Appendix

See Table 5.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was also supported by the ICT R&D Program of MSIP/IITP. [14-912-06-002 and the Development of Script-Based Cyber Attack Protection Technology]. In addition, this research is also supported by a Korea University Grant.

References

[1] McAfee, "McAfee Labs Threats Report," February 2015, http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2014.pdf.

[2] Apkprotect, http://www.apkprotect.com/.

[3] Bangcle, http://www.bangcle.com/.

[4] R. Yu, "Android Packer Facing the Challenges, Building Solutions," https://www.virusbtn.com/pdf/conference_slides/2014/Yu-VB2014.pdf.

[5] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "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)*, pp. 1–15, 2014.

[6] H. Peng, C. Gates, B. Sarma et al., "Using probabilistic generative models for ranking risks of Android apps," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*, pp. 241–252, Raleigh, NC, USA, October 2012.

[7] Y. Wang, J. Zheng, C. Sun, and S. Mukkamala, "Quantitative security risk assessment of android permissions and applications," in *Data and Applications Security and Privacy XXVII*, Lecture Notes in Computer Science, pp. 226–241, Springer, 2013.

[8] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "DroidMiner: automated mining and characterization of fine-grained malicious behaviors in android applications," in *Computer Security—ESORICS 2014 :19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7–11, 2014. Proceedings, Part I*, vol. 8712 of *Lecture Notes in Computer Science*, pp. 163–182, Springer, Berlin, Germany, 2014.

[9] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS '14)*, pp. 1105–1116, ACM, Scottsdale, Ariz, USA, November 2014.

[10] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pp. 235–245, ACM, November 2009.

[11] J.-W. Jang, J. Yun, J. Woo, and H. K. Kim, "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)*, pp. 737–738, 2014.

[12] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "AdDroid: privilege separation for applications and advertisers in Android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*, pp. 71–72, Seoul, Republic of Korea, May 2012.

[13] Y. Zhang, M. Yang, B. Xu et al., "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pp. 611–622, ACM, Berlin, Germany, November 2013.

[14] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: automatic reconstruction of Android malware behaviors," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS '15)*, San Diego, Calif, USA, February 2015.

[15] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on API call sequence analysis," *International Journal of Distributed Sensor Networks*, vol. 2015, Article ID 659101, 9 pages, 2015.

[16] D. Kim, J. Kwak, and J. Ryou, "DWroidDump: executable code extraction from android applications for malware analysis," *International Journal of Distributed Sensor Networks*, vol. 2015, Article ID 379682, 9 pages, 2015.

[17] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: detecting malicious apps in official and alternative android markets," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS '12)*, San Diego, Calif, USA, February 2012.

[18] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: android malware detection through manifest and API calls tracing," in *Proceedings of the Seventh Asia Joint Conference on Information Security (Asia JCIS '12)*, pp. 62–69, Tokyo, Japan, August 2012.

[19] M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware," in *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '13)*, pp. 163–171, IEEE, Melbourne, Australia, July 2013.

[20] L. Deshotels, V. Notani, and A. Lakhotia, "DroidLegacy: automated familial classification of Android malware," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop (PPREW '14)*, ACM, January 2014.

[21] J. Lee, S. Lee, and H. Lee, "Screening smartphone applications using malware family signatures," *Computers & Security*, 2015.

[22] F-Secure, "Threat Report H1 2014," https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2014.pdf.

[23] F-Secure, "F-Secure, 25 Years of the Best Protection in the World," 2013, http://www.fsecure.com/en/web/labs_global/.

[24] S.-H. Seo, A. Gupta, A. M. Sallam, E. Bertino, and K. Yim, "Detecting mobile malware threats to homeland security through static analysis," *Journal of Network and Computer Applications*, vol. 38, no. 1, pp. 43–53, 2014.

[25] J.-W. Jang, H. Kang, J. Woo, A. Mohaisen, and H. K. Kim, "Andro-AutoPsy: anti-malware system based on similarity matching of malware and malware creator-centric information," *Digital Investigation*, vol. 14, pp. 17–35, 2015.

[26] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings of the 7th International Symposium on String Processing and Information Retrieval (SPIRE '00)*, pp. 39–48, IEEE, A Coruña, Spain, 2000.

[27] Androguard, "Reverse Engineering, Malware and Goodware Analysis of Android Applications," 2014, https://code.google.com/p/androguard/.