Smartphone Dual Defense Protection Framework: Detecting Malicious Applications in Android Markets

X. Su*
CSE Dept
Lehigh University
Bethlehem, PA, USA
xis711@lehigh.edu

M. Chuah, G. Tan
CSE Dept
Lehigh University
Bethlehem, PA, USA
{chuah,gtan}@cse.lehigh.edu

Abstract— **In this paper, we present a smartphone dual defense protection framework that allows Official and Alternative Android Markets to detect malicious applications among those new applications that are submitted for public release. Our framework consists of servers running on clouds where developers who wish to release their new applications can upload their software for verification purpose. The verification server first uses system call statistics to identify potential malicious applications. After verification, if the software is clean, the application will then be released to the relevant markets. To mitigate against false negative cases, users who run new applications can invoke our network traffic monitoring (NTM) tool which triggers network traffic capture upon detecting some suspicious behaviors e.g. detecting sensitive data being sent to output stream of an open socket. The network traffic will be analyzed to see if it matches network characteristics observed from malware applications. If suspicious network traffic is observed, the relevant Android markets will be notified to remove the application from the repository. We trained our system call and network traffic classifiers using 32 families of known Android malware families and some typical normal applications. Later, we evaluated our framework using other malware and normal applications that used in the training set. Our experimental results using 120 test applications (which consist of 50 malware and 70 normal applications) indicate that we can achieve a 94.2% and 99.2% accuracy with J.48 and Random forest classifier respectively using our framework.**

Keywords-android security, data mining, system call monitor, netweork traffic monitor

I. INTRODUCTION

Today's most popular mobile devices are smartphones. A smartphone combines the functionality from both a mobile phone and a personal computer. A modern smartphone also serves as a multimedia player, a digital camera and a GPS unit. Smartphones have become ubiquitous with more than 700 million units all over the world [1]. With more smartphone usages, more feature rich smartphone applications have emerged in various market places. For example, by July 2012, there are over 470,000 applications in Google's official Android Market [2]. Apart from the official market places, there are also other third-party Android markets such as Amazon's Appstore.

As smartphones become prevalent, more and more hackers are targeting this platform. From July 2011 to November 2011, the number of malicious applications in the market increases by 472% [6]. Typically, such mobile malwares perform some malicious activities. For example, a Trojan called Zsone [7] sends SMS messages to premium numbers. To hide itself, it also removes billing notification messages from service providers. In Feb 2012, Rootsmart [8] was reported to create a botnet, consisting of thousands of Android devices. Once started, Rootsmart connects the infected device to a remote server; it then sends private information of the infected phone to the server and fetches a root exploit to obtain escalated privileges to the phone. To generate profits for the botmaster, the infected phone is configured to send premium SMS messages and use other premium telephony services. There are many other reports that revealed the presence of malware in smartphone applications (e.g., [3] [4] [5]).

Given the alarming growth of Android malware, it is urgently important that an effective mitigation scheme can be designed to defend against such attacks. Two possible defense mechanism can be designed, namely (i) detection modules which run on smartphones and produce alert messages when malwares are detected, and (ii) detection modules running on servers hosted by Android Markets which can analyze new applications (or subsets of new applications) to detect zero-day malwares. Examples of the first approach are the TaintDroid[12] and the AppFence[13] solutions while examples of the second approach are DroidRanger[14], and RiskRanker[17]. The first approach may incur too much CPU overhead on smartphones and hence we focus only on the second approach in this work. DroidRanger uses two heuristics to detect unknown malwares, namely dynamic loading of Java binary code from a remote untrusted website and dynamic loading of native code locally. The authors in [14] currently only monitor those system calls used by existing root exploits with root privilege, and hence new malwares which avoid calling such system calls with root permissions may avoid being detected. The authors in RiskRanker [17] use two order risk analysis to detect new malwares. They use signatures specific to existing root exploits to identify high risk applications. They also use complex code path analysis for identifying high risk applications. Second order analysis

X. Su was a visiting PhD student from Hunan University when he participated in this research at Lehigh University.

M. Chuah's effort is partially supported by National Science Foundation Grant DUE 0920271

focus on how specific patterns in an execution path e.g. assessing assets or res directory, using crypto related APIs, and calling Runtime.exec(). We are interested in exploring a more lightweight approach that can be used by Android markets to weed out new malware applications.

In this paper, we present a smartphone dual defense protection framework for detecting new malicious Android applications in the Android markets. Our solution consists of two main phases: (i) a verification service in Phase 1 which checks for potential malicious codes in new applications using system call statistics, (ii) a network monitoring tool in Phase 2 that allows users who install new applications to check for potential false negative cases generated from the verification process during Phase 1.

Every new application package that is uploaded to an Android market that supports our smartphone dual defense protection framework will undergo a verification process where the application is run several times on Android devices with our system call monitoring software running in the background. System call statistics are collected and sent to an Application Verification (AV) server in the cloud which determines if the system call statistics of the new application are similar to those seen in malware applications. If they are not, then the new application will appear in the Android market. Otherwise, it will not.

To deal with potential false negative cases from the verification process, the Android market also provides a network monitoring tool that allows network traffic of new applications to be captured, analyzed and sent to the AV server. The AV server will determine if the network traffic characteristics are similar to those observed in malware applications. If that is the case, that application will be immediately removed from the Android market. Human experts can then manually check this new application to confirm if new malware has been discovered.

We evaluated our approach by first training our system call and network traffic classifiers using statistics collected from selected malware and normal applications in our training set and then test the trained classifiers using statistics collected from different malware and normal applications which form the test set. Our training set consists of malware applications from 32 malware families and normal applications from 22 categories. Our test set consists of malwares from 22 families and normal applications from 11 categories. The evaluation results show that our approach is very effective. We can achieve an accuracy of 94.2%(99.2%) using J.48 (Random forest) classifier. Random forest classifier performs better.

In summary, our contributions in this paper are: (i) we present a dual-defense protection framework which uses system call and network traffic characteristics to detect malwares, (ii) we evaluate our approach using a training set which consists of malware applications from 32 malware families and normal applications from 22 categories.

II. RELATED WORK

Performing application analysis is an effective way of detecting malicious applications on smartphones. Application analysis can be performed statically before an application is running. Enck et al [9] used decompilation and static analysis techniques to study 1100 free applications from the official Android Market to understand a broad range of security related metrics associated with these applications. They discovered that sensitive information is widely leaked in applications. For instance, more than half of the applications include at least one advertisement libraries that collect and send private information, e.g. the location of the phone. ComDroid [10] uses static analysis to detect inter-application communication vulnerabilities. PiOS [11] uses binary static analysis to detect privacy leaks in iOS applications. In contrast to these systems, we collect system-call and network traces for analysis of malicious behavior.

Application analysis can also be performed dynamically by monitoring a running application. A representative example is TaintDroid [12]. It applies dynamic taint tracking and analysis on the usage of sensitive data on Android. Any information which comes from a trusted application is considered to be tainted. TaintDroid marks data coming from the "taint sources" and tracks the taint flow. If the data in the end are used by an untrusted application, TaintDroid reports it as a sensitive data leak. However, TaintDroid cannot print alert messages for many of the malware samples that we have evaluated. Building on top of TaintDroid, AppFence [13] implements two simple runtime mechanisms to protect users' privacy. The first one is data shadowing, a mechanism that returns fake or blank data when an untrusted application requests private data such as phone IDs and location information. The second idea is to block an application's communication from sending out private information at runtime. It prevents the exfiltration of sensitive data by intercepting calls to the network stacks to detect when such data is written to a socket. Such offending messages are dropped. These two approaches are phone-based solutions and can potentially add much CPU overhead. Furthermore, their implementations may not work when Android OS evolves.

Another line of defense is to provide automated tools that allow Android markets to screen new applications for potential malwares. In [14], the authors provide one such solution. They studied 204K applications to detect malwares. 211 malwares were found using their schemes: (i) using permission-based behavioral footprinting, (ii) using a heuristics-based filtering scheme.

The evaluation of any new defense design requires researchers to have access to many malware applications. In the past, research community has been constrained by the lack of a comprehensive mobile malware dataset to conduct their research. However, a group of researchers from NCSU has collected 1260 Android malware samples in 49 different malware families. These malware samples cover the majority of existing Android malware which appear from Aug 2010 to October 2011 [15]. They have performed a timeline analysis

of these malwares and studied their detailed behaviors. They discovered that 86% of these malware samples are repackaged versions of the legitimate applications with malicious payloads. They also observed that recent Android malware families adopt update attacks and drive-by downloads to infect users which make them stealthier and more difficult to detect. Furthermore, they discovered that (i) about one third of the collected malwares leverage root-level exploits to fully compromise the Android security, (ii) more than 90% turn the phones into a bonet controlled through network or short messages, and (iii) 28 of the 49 malware families have built-in support of sending out background short messages (to premium-rate numbers) or making phone calls without user awareness.

III. SMARTPHONE DUAL DEFENSE PROTECTION FRAMEWORK

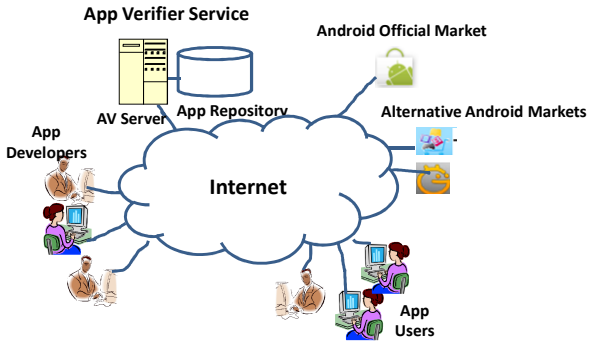


Fig 1. Smartphone Dual Defense Protection Framework

In our SDDP framework, we assume that an application verification service (AV) is provided by either the Android Official Market and/or Alternative Android Markets. The AV service will verify if any application is clean of any potential malwares. Such a service can be provided for free or at a low fee paid by the application developers. Android Markets can refuse to let any application developer upload a new application to their repositories unless that application received a verification approval by the application verification service. Every application developer who would like to release his new application to an Android market must first submit that new application for application verification. He uploads his new application to an Application Repository, and indicates which Android Markets he would like his new application to be uploaded to.

Periodically, the AV server runs those new applications which are submitted during a certain time period, and so that it can collect system call statistics using the system call monitoring tool which we develop. The system call monitoring tool generates a log of system calls generated by any new application. The AV server analyzes the log files of all new applications in that batch to extract relevant features that can be fed to a WEKA-based classification tool. Our SCM classification tool will determine if the new application that undergoes verification contains any malware characteristics. After Phase 1, the new applications will be labeled as "normal" or "malware". The AV server will then use trigger rules for normal or malware applications to decide if network monitoring tool will be triggered to collect network and analyze network traces in Phase 2. Phase 2 gives us an opportunity to correct some of the mislabeled applications. After the final labels for the applications are generated, the AV server will release all applications that are labeled as normal to the Android markets, and send an email to the application developer if his application fails the verification process.

Since the classifier may produce false negatives, the AV service provider also provides a network monitoring tool for each user to install. Upon running a new application, our network traffic monitoring tool can automatically check if the new application generates network traffic that is similar to malicious traffic. If that is the case, an alert message will be sent to the AV server which will then pull the application from the repository.

In subsequent sections, we elaborate more on the system call monitor and the network traffic monitor used in our system.

A. System Call Monitor

Fig 2 shows how our system call monitor (SCM) interacts with the application undergoing test (AUT) as well as the Android Operating System (AOS). Our SCM interacts with the AOS to obtain the process identifiers that are associated with the AUT. The AUT is run multiple times at the verification server, and relevant system call statistics for each session are recorded in a log file. A dataset of system call related statistics will be created for every application that undergoes evaluation. The more users use a particular category of applications, the more complete and accurate our system will be. The server will release the AUT to the relevant Android markets if the classification result indicates that it is very likely to be benign.

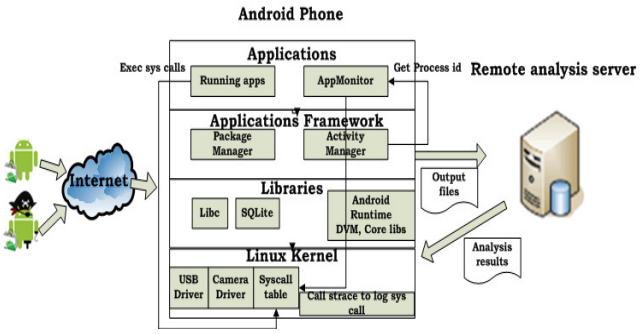


Fig 2. System Call Monitor

Android Operation System is based on linux kernel. In Linux, a system call is how a program requests a service from the operating system's kernel. System calls provide useful functions to application programs like network communication, file management or process related operations. When an application from user space makes a request to the Operation System, the request goes through

glibc library, system call interface, kernel and finally to the hardware. Functions like getpid(), open(), read(), chmod(), and socket() are some of the functions that glibc provides for applications to invoke a system call.

All requests made from upper layers pass through the kernel using the system call interface before they are executed in hardware. Thus, capturing and analyzing the system calls that pass through system call interface will provide accurate information about the behavior of an application. Our design makes use of a tool available in Linux called *strace* to collect the system calls. The aim of monitoring such system calls is to generate a log file with all events generated by that Android application under test. This file contains useful information e.g. counts of different system calls made by the application, clone processes, opened and accessed files.

To illustrate why system call statistics can be used to effectively identify potential malware applications, we collect system call statistics of two sets of applications, namely (a) Goldminer and (b) Bgserv-Chinamobile. In each set, we have a clean application from the official Android market, and another that contains malware from a 3rd party market. We plot the collected system call statistics for two runs of Goldminer and Chinamobile applications in Fig 3(a) and 3(b) respectively. The x-axis shows the system call identifiers while the y-axis shows the number of times that particular system call is activated. One can observe that the numbers of clone (callId=121) and mprotect (callId=126) calls in the malware version of Goldmine are significantly higher than its normal version while the numbers of sigprocmask (callId=127) and mmap (callId=193) system calls in the malware version of Chinamobile application are significantly higher than its normal version.

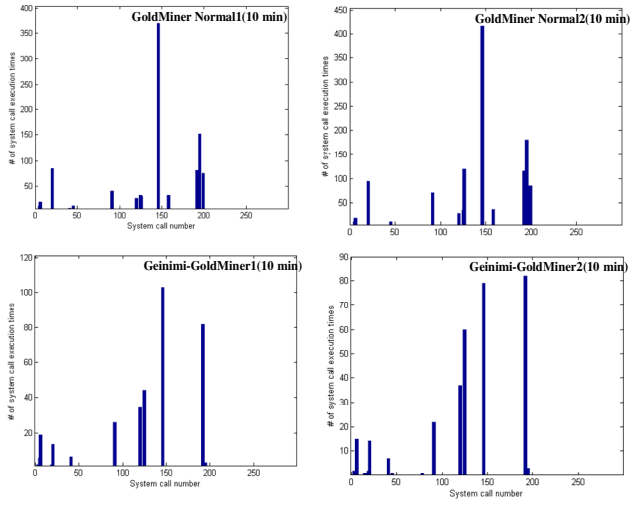


Fig 3(a) System Call Statistics of Normal and Malicious GoldMiner Application

There are 288 system calls but we merely focus on fifteen that are related to process, memory and I/O management since typically the statistics of such calls are different between normal and malicious applications. The fifteen system calls used in our SCM classifier are: sys_chmod, sys_chown, sys_mount, sys_access, sys_open, sys_clone, sys_getpriority, sys_mmap, sys_read, sys_exit, sys_kill, sys_brk, sys_execve, sys_kill, sys_times, and sys_nice.

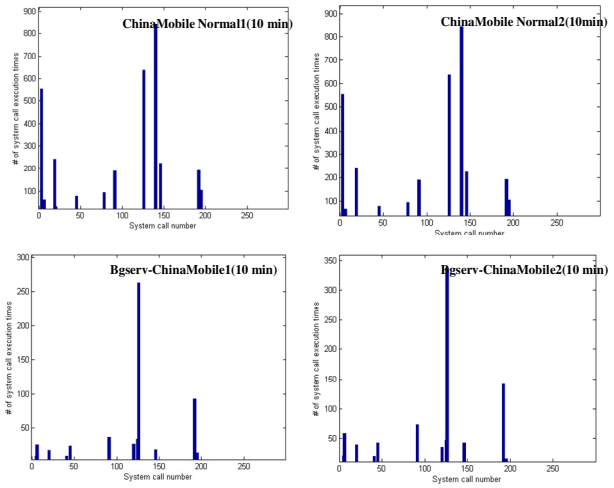


Fig 3(b) System Call Statistics of Normal and Malicious ChinaMobile Application

Next, we trained our system call classifier by using system call statistics collected by running 200 randomly selected normal, and 180 malware applications previously discovered (chosen from the android malware genome dataset [15]). The chosen malware families and the categories of normal applications are shown in Tables 1(a) & 1(b). We fed the collected system call statistics to the J.48 and Random forest classifiers in the WEKA tool and conducted a ten-fold cross validation experiment. The classification results were tabulated in Table 1(c). It shows that J48 classifier achieves a 91.6% accuracy rate while the Random forest achieves a 91.6% accuracy rate. The trained classifier identifies five of the system calls as important attributes that can be used to distinguish between normal and malware applications: sys_read, sys_mmap, sys_access, sys_brk, and sys_clone. For J.48 classifier, 15 malwares are classified as normal while 17 normal applications are classified as malwares. 7 of the 15 mislabeled malwares can be classified as malwares via the network monitoring tool which we will discuss in the next section. 3 of the malwares do not produce any network traffic.

To deal with false positive (i.e. normal applications being labeled as malwares), we suggest using the network monitoring tool in Phase 2 to evaluate those applications who are labeled as malwares, have native codes and only request INTERNET permission but do not request for either send or receive SMS permission to see if they generate malware-like traffic characteristics. If they do not, then, we can label them as normal. If we use this trigger rule (referred to as **Rule 1**), 9 out of the 17 normal applications will finally be labeled as "normal".

Table 1(a) Chosen Malware Families

Malware Family	#	MalwareFamily	#
adrd	7	Gone60	1
anserverbot	18	HippoSMS	2
asroot	3	Jifake	1
basebridge	21	jSMSHider	5
beanbot	3	NiceSpy	1
bgserv	2	Pjapps	12
coinpirate	1	Plankton	4
droiddream	13	RogueLemon	2
droiddreamlight	13	RogueSPPush	1
droidkungfu	26	SndApps	2
droidkungfusapp	1	Tapsnake	1
droidkungfuupdate	1	Walkinwat	1
endofday	1	YZHC	2
Geinimi	14	zHash	1
GGTracker	1	Zitmo	1
GoldDream	12	Zsone	6

Table 1(b) Chosen Normal Application Categories

App Category	# of Apps	App Category	# of Apps
Book	11	Photograph	9
Game	10	Business	8
Travel	11	Shopping	10
Social	9	Fincance	9
Tools	10	Lifestyle	16
Entertainments	10	Personalize	1
Communications	13	weather	3
Education	3	news	12
productivity	10	sports	10
Ccomic	10	medical	10
transportation	10	health	10

Table 1(c) System Call Classifier Results

Algorithm	Correctly classified	Incorrectly classified	Confusion matrix
J48	348/91.58%	32/8.42%	m n 165 15 m 17 183 n
Randomforest	348/91.58%	32/8.42%	m n 168 12 m 20 180 n

B. Network Traffic Monitor

Next, we describe the second component of our dual defense protection framework, namely the network traffic monitor. Any user that uses our dual protection framework will install a network traffic monitoring tool. Our traffic monitoring tool has a set of triggering rules that look for suspicious application behaviors e.g. high read and mmap system call statistics. In current implementation, our network traffic monitoring tool was built using the idea we borrowed from [13]. When an application writes to a socket's output stream, the buffer is sent to the sendStream() method within the OSNetworkSystem core library. We modified sendStream so that if the buffer is tainted by data that should not be sent to its intended destination, we trigger the network capture tool (tcpdump) available in the system.

When a user runs an application, and one of the triggering rules fires, then it will trigger a tcpdump capture action to log network traffic for a configurable period (e.g. 5 or 10 minutes). Once the logging completes, our program will parse the pcap files, extract TCP/IP flows, and analyze their statistics. The statistics will be recorded in an arff file. This arff file will be sent to a remote verification server which then invokes a J.48 based WEKA network classifier which has been previously trained to determine if the traffic characteristics are similar to those obtained from previously seen malwares. If the classifier output indicates that this application is a potential malware, then the verification server will remove this application and the user will be notified of the suspicious nature of this application. Our network traffic monitoring tool also maintains a whitelist such that all applications that have gone through such network testing do not undergo such testing again when they are run subsequently unless new versions of the applications are installed.

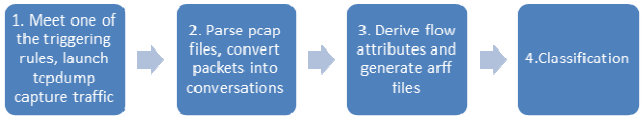


Fig 4 Network Traffic Monitoring Procedure

1) Training Our NTM Classifier

To train our NTM classifier, we installed 49 malware and 60 normal applications, and collected network traffic generated when each of these applications was run. The 49 malware applications were chosen from 22 malware families as shown in Table 2. Then, we extract TCP/IP conversations from these collected traces. Each of the TCP/IP conversation is referred to as an instance. We remove those conversations with known legitimate advertisement websites. We manually labelled each conversation as malware or normal instances based on whether the conversation is with a blacklisted website or carries sensitive information e.g. IMEI etc. There are 90 malware instances and 373 normal instances. Then, we extracted the following 9 statistics for each instance, namely the average and standard deviation of the number of sent/received packets, the average and the standard deviation of the number of bytes sent/received and the average TCP/IP session duration. We also randomly selected 60 normal applications e.g. Facebook, Twitter, Gmail, Tencent.qq, MSN, Google Play, Ebay, Wiki, Amazon etc, and analyzed the TCP/IP conversations (instances) generated by running these applications too. Then, we fed these statistics with the appropriate manually generated labels (malware or normal) to a J.48 and Random forest classifiers in WEKA tool.

Next, we randomly selected 50 malware applications and 70 normal applications that were not in the training set, collected their network traffic, remove the ad libraries network traffic, and extract the relevant statistics related to the TCP/IP conversations seen in these network traffic. The collected network traces consist of 571 TCP/IP conversations (instances), out of which 55 are malware instances (accessing blacklisted web sites or sending sensitive information) and 516 are normal instances. Then, we fed the statistics of each TCP/IP conversation as a test instance to our trained J.48 and Random forest network traffic classifiers.

Table 2 Malware Families Used in Traffic Classifier Training

Malware family	# of app	Malware family	# of app
adrd	3	golddream	3
anserverbot	3	gingermaster	1
asroot	1	geinimi	8
basebridge	4	Gone60	1
beanbot	1	hippoSMS	1
bgserv	2	jsmshider	2
ddlight	2	MMMarketPay	1
droidkungfu	4	pjapps	2
droidkungfuapp	1	plankton	3
droidcoupon	1	zhash	1
ggtracker	1	other	3

The test results showed that our trained J.48 and Random forest network traffic classifiers produce an accuracy rate of 91.6% and 96.7% respectively. 21 (11) malware instances are labeled as normal while 27 (8) normal instances are labeled as "malware" by J48 and Random forest classifiers respectively. Since a malware application may generate both "malware" and "normal" instances, we studied the results more carefully and observed that 33 (41) of the malware applications can be identified as "malware" if we use the rule that an application is a malware as long as 25% of the traffic instances that it generates are classified as "malware" by either the J.48 or the Random forest classifier. With such a rule, 33 (41) of our malware applications will be labeled as "malwares" and 69 (70) of normal applications will be labeled as "normal" by our J.48 (Random forest) classifier. Overall, Random forest classifier produces better results.

2) **Heuristic Rules for Triggering Network Monitoring Tool**

Now, we discuss how the two components, namely SCM and NTM are used simultaneously in our framework. Any new application will first be tested using the SCM classifier. If the SCM classifier deems this to be a normal application, the new application will be released to the Android Market for user downloads. Any application that is labeled as malware by SCM can be further verified using the NTM tool by the verification server. If the NTM tool indicates the new application is harmless, it will still be released.

When a user who installs our NTM tool first runs a new application, our NTM tool will also check the triggering rules to see if the NTM tool needs to be triggered. If so, the NTM tool collects network traffic statistics and sends them to the AV server. The NTM classifier will determine if this application is a potential malware. If it is, then the application will be removed from the Android markets.

Since we want to minimize the number of applications that will trigger the NTM tool, we define certain heuristic rules after we analyzed different categories of normal applications. Our heuristic rules focus on 4 system calls, namely, access, mmap, read and brk for we found that known malwares seem to have different statistics in these 4 types of system calls. In Table 3, we show the different categories which our 50 normal applications fall under, e.g. book, game, travel, social network, tools, entertainments etc and the chosen conditions. Our NTM will be triggered if((acesss>Th1)||((mmap>Th2) &&(read<Th3))||(brk>Th4) and internet permission is requested by the application. Currently, the thresholds were chosen by studying the statistics collected from limited number of normal applications. However, we find triggering using such rule not to be too effective. Thus, instead, we use the following rule (referred to as **Rule 2**) for applications labeled as "normal" to trigger network traffic monitoring: an application that requests for INTERNET permission and READ_PHONE_STATE and either (SEND_SMS or RECEIVE_SMS or WRITE_HD).

IV. EVALUATIONS

A. Accuracy of our SDDP Framework

In this section, we describe how we evaluate our smartphone dual defense protection framework. We developed our prototyped system call and traffic monitoring tools. Then, we randomly select 70 normal and 50 malware applications. The malware applications are chosen from the 16 of the 49 families of malware samples provided by the authors in [15] as well as 3 new ones which appear in July 2012. In particular, we included the four malware families, i.e., BaseBridge, DroidKungFuUpdate, AnserverBot, and Plankton, that adopt update attacks [15]. The 70 normal applications are chosen from 17 categories e.g. comics, sports,news,lifestyle, medical, communication, transportation, health, game, book, education, entertainment, etc.

We run each of these 120 applications, and collect system call statistics using our system call monitoring tool. Then, we test them using our trained J.48 & Random forest WEKA-based classifiers. The results we obtained are tabulated in Table 4. We can achieve an accuracy rate of 90.8%/96.7% for J.48 and Random forest classifier respectively using statistics from the 15 system calls that we describe in Section 3.1. Eight of the malwares are labeled as normal and three of the normal applications are labeled as malwares.

Next, we collect network traffic traces from applications that are labeled as "malwares" from Phase 1 which satisfy the triggering rule Rule 1 described in Section III.A, then we can identify these mislabeled normal applications as "normal". In addition, we collect network traffic from applications labeled as "normal" which satisfy Rule 2. 4 (3) of the misclassified malwares from the J.48/Random forest system call classifiers qualified, and the NTM classifier was able to infer that they are indeed malicious applications. Thus, our final accuracy rate is 94.2% (99.2%) are classified correctly after the 2 pronged approach) using J.48 and Random forest classifiers respectively.

B. CPU Overhead of Our Approach

Next, we investigate how much overhead our system call monitor adds to the CPU processing times. We use the OS Monitor tool available in Android Market [16]. In Fig 5(a) we show the CPU usage of major processes when a user uses Web Browser and in Fig 5(b) we show the CPU usage of major processes when a user uses GooglePlay with our system call monitoring program running in the background. The numbers show that our system call monitoring program does not add any significant overhead.

Next, we provide CPU usage of processes when the tcpdump tool is turned on to collect network traffic and when tcpdump is turned off in Figure 6(a) and 6(b) respectively. The CPU usage for tcpdump is included in the CPU usage reported under "System". The numbers show that the tcpdump tool does not incur too much CPU overhead. Recall that the tcpdump tool will only be invoked when the permission-based rule for triggering our network monitoring tool is invoked.

Table 3 Triggering Thresholds for various categories of normal applications

App category	access	mmap	read	brk
Book	>250	>1500	<2500	>300
Game	>250	>1500	<2500	>200
Travel	>250	>1500	<2500	>250
Social	>250	>1500	<2500	>500
Tools	>250	>1500	<2500	>100
Entertainment	>250	>1500	<2500	>200
Photograph	>250	>1500	<2500	>250
Business	>250	>1500	<2500	>200
Shopping	>250	>1500	<2500	>200
Finance	>250	>1500	<2500	>250
Lifestyle	>250	>1500	<2500	>350
Personalize	>250	>1500	<2500	>100
education	> 250	>1500	<2500	>200
comics	>250	>1500	<2500	>300
health	>250	>1500	<2500	>300
medical	>250	>1500	<2500	>400
sports	>900	>1500	<2500	>300

C. **Discussion**

As briefly described earlier, not every new application needs to be verified using either of the two tools. Based on what have been learnt so far about Android malware applications [14,15], one can reduce the number of new applications that needs to be verified using some simple rules e.g. all applications with native codes and request INTERNET permissions need to be tested, all applications without native codes but request for INTERNET, SEND_SMS, RECEIVE_SMS or permission-group.COST_MONEY permissions will also be tested etc.

Table 4 Results from Our System Call Classifier

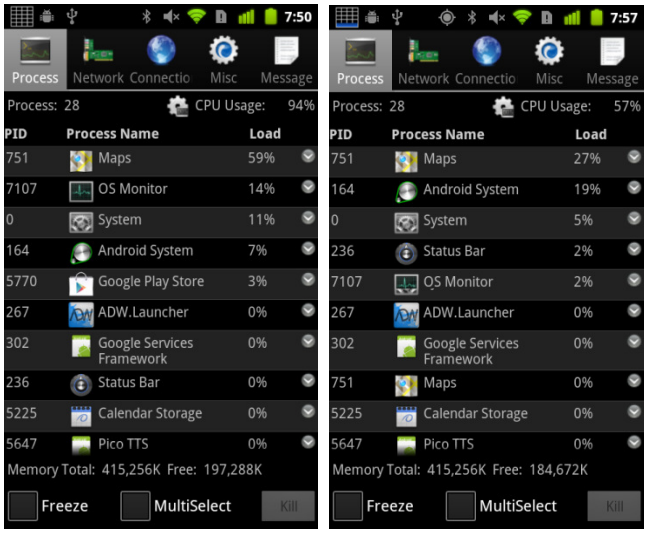
Classifier O	Manual Classificatio	Correctly Classified	Incorrectly Classified	Confusion Matrix	
J.48	50 malware	109/90.83	11/9.17%	m n	
	70 normal			42 8	m
				3 67	n
RandForest	50 malware	116/96.67	4/3.33%	m n	
	70 normal			46 4	m
				0 70	n

(a) CPU Usage (Google Web Surfing) (b) CPU Usage (Google Play)

Fig 5 CPU Usages of Various Processes

According to the study done in [17], the authors found that only 9.42% of the 104,874 distinct applications they inspect contain native codes. Among the 200 normal applications which we used for system call training, only 24.5% have native codes. Among the 180 malware applications which we used for system call training, 37% have native codes but only three (1.7%) of those without native codes do not request for either Internet, Phone State and Send/Receive SMS. Thus, using intelligent combinations of such rules will reduce the number of new applications that need to be checked. Our evaluation using 120 test applications also reveal that Rule 2 based on requested permissions for triggering network traffic monitoring tool is quite effective. In the near future, we hope to evaluate both Rules 1 & 2 using larger set of normal and

malware applications so that we can be sure that these two rules do not result in having too many new applications invoking network monitoring tools.



(a) tcpdump on **(b) tcpdump off**

Fig 6 CPU Usage with Network Monitoring Tool

V. CONCLUSIONS

In this paper, we have presented a smartphone dual defense protection framework that can be used to detect malware application in Android-based markets. Our framework consists of two major components, namely (a) system call monitoring tool which is used in Phase 1 to evaluate new application for potential malwares, (b) network monitoring tool which is used to identify new applications that are false negatives of the system call monitoring tool. Our preliminary results indicate that our framework indeed provide good detection accuracy. Random forest classifier produces better results than J.48 classifier. We intend to evaluate our framework using more malware samples from [15] and normal applications. We also intend to determine the appropriate rules which minimize the triggering of network traffic monitoring tool for normal applications. In addition, we are looking into whether adding Android system call behaviors of malware applications can further improve our system. Last but not least, we intend to deploy a cloud-based system that mimics a third-party Android market that provides the verification service described in our framework and advertise this in Android markets to gather more evidence about the effectiveness of our approach.

ACKNOWLEDGMENT

We thanked Y. Zhou, X. Jiang from NCSU for sharing their android malware genome dataset.

REFERENCES

[1] The mobile web in numbers. 2011. [Online]. Available: http://royal.pingdom.com/2011/12/07/the-mobile-web-in-numbers/.

[2] Appbrain, July, 2012. [Online]. Available: http://www.appbrain.com/stats/number-of-android-apps.

[3] "Security Alert: DroidDream Malware Found in Official Android Market," [Online]. Available: http://blog.mylookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/.

[4] "Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild," [Online]. Available: http://blog.mylookout.com/blog/2010/12/29/geinimi_trojan/.

[5] "Security Alert: DroidDreamLight, New Malware from the Developers of DroidDream," [Online]. Available: http://blog.mylookout.com/blog/2011/05/30/security-alert-droiddreamlight-new-malware-from-the-developers-of-droiddream/.

[6] T. Vennon, "Mobile malware development continues to rise, Android leads the way.," [Online]. Available: http://forums.juniper.net/t5/Security-Mobility-Now/Mobile-Malware-Development-Continues-To-Rise-Android-Leads-The/ba-p/132695.

[7] "Security Alert: Zsone Trojan found in Android Market," 2011. [Online]. Available: http://blog.mylookout.com/blog/2011/05/11/security-alert-zsone-trojan-found-in-android-market/.

[8] X. Jiang, "Security Alert: New RootSmart Android Malware Utilizes the GingerBreak Root Exploit," 2012. [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/RootSmart/.

[9] W. Enck, D. Octeau , P. McDaniel and S. Chaudhuri, "A study of Android application security," *20th Usenix Security Symposium,* 2011.

[10] E. Chin, A. P. Felt, K. Greenwood and D. Wagner, "Analyzing inter-application communication in Android," *Analyzing inter-application communication in android.,* pp. 239-252, 2011.

[11] M. Egele, C. Kruegel, E. Kirda and G. Vigna, "PiOS: Detecting privacy leaks in iOS applications," *In 18th Network & Distributed System Security Symposium,* 2011.

[12] W. Enck, P. Gilbert, B.-G. Chun, L. Cox, J. Jung, P. McDaniel and A. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *In 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI),* pp. 1-6, 2010.

[13] P. Hornyack, S. Han, J. Jung, S. Schechter and D. Wetherall, "These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from," *the 18th ACM conference on Computer and communications security (CCS),* pp. 639-652 , 2011.

[14] 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," *Proceedings of ACM NDSS,* 2012.

[15] Y. Zhou, X. Jiang, "Dissecting Android Malware: Characterizing and Evolution", IEEE Symposium on Security and Privacy, Aug, 2012.

[16] OS Monitor http://www.appbrain.com/app/com.eolwral.osmonitor

[17] M. Grace, Y. Zhou, Q. Zhang, S. Zou, X. Jiang, "RiskRanker: Scalable and Accurate Zero-Day Android Malware Detection", Proceedings of ACM Mobisys, June 2012.