Behavioral Malware Detection Approaches for Android

Mohammad Rakib Amin* md.rakib.amin@gmail.com
Mehedee Zaman* devmhd@gmail.com
Md. Shohrab Hossain* mshohrabhossain@cse.buet.ac.bd
Mohammed Atiquzzaman† atiq@ou.edu

*Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh
†School of Computer Science, University of Oklahoma, Norman, OK 73019

***Abstract*—Android, the fastest growing mobile operating system released in November 2007, boasts of a staggering 1.4 billion active users. Android users are susceptible to malicious applications that can hack into their personal data due to the lack of careful monitoring of their in-device security. There have been numerous works on devising malware detection methods. However, none of earlier works are conclusive enough for direct application and lack experimental validation. In this paper, we have investigated the natures and identities of malicious applications and devised two novel detection approaches for detection: network-based detection and system call based detection approaches. To evaluate our proposed approaches, we performed experiments on a subset of 1260 malwares, acquired from Android Malware Genome Project, a malware database created by Y. Zhou et al. [1] and 227 non-malware (benign) applications. Results show that our system call based approach is able to detect malwares with an accuracy of 87% which is quite significant in general malware detection context. Our proposed detection approaches along with the experimental results will provide security professionals with more precise and quantitative approaches in their investigations of mobile malwares on Android systems.**

***Index Terms*—Android, malware detection, netstat, pcap, ADB, Busybox**

I. Introduction

A malware is a program which disrupts computer operation, gathers sensitive information or gains access to private systems without users consent. With the ever increasing use of mobile devices, mobile malware pose a significant threat because these devices store contacts, bank account numbers, credit/debit numbers, private photos, messages and a lot of other sensitive information that can be leaked. According to a recent report published jointly by Kaspersky Labs and INTERPOL [2], 20% of devices that uses their software were attacked at least once by malware. Their study Financial Cyberthreats in 2014 reports that the number of financial malware attacks against Android users grew by 3.25 times in 2014. Given the tremendous growth of Android malware, there is a pressing need for effective malware detection methods.

There exists a few static and dynamic malware detection methods reported in the literature. While they convey valuable insight into malware research, the detection methods are not precise enough. Various surveys are made on defining detection procedures or categorizing them [4], [5]. These researches propose permission analysis which is obsolete as permissions cannot be faked in Android. There has been more focused approaches like network analysis by Iland et al. [6], aggregated system call events by Ham et al. [7]. The latter work suggests identifying specific "malicious" system call events which do not occur except in malicious applications. But such combination of system calls may also occur in legitimate applications, leading to false positive detection of malwares. Investigations on MRAs or Malicious Repackaged Application like system call patterns in repackaged applications [8] propose more fine-tuned approaches. Isohara et al. [9] also demonstrated a system-call logging based method.

The main *objective* of this work is to propose a *precise and quantitative* approach that detects presence of malicious behavior in a random android application by testing that app's behavior against our findings on 1260 malwares and 227 "good" applications, and classifies that app as "good" (non-malware) or "bad" (malware).

The *contributions* of the work are:

1) performing network traffic analysis by capturing the URLs of all remote locations that are contacted by applications for a specific period of time, then matching against a database of known malicious domains;
2) conducting system call analysis by logging all system calls made by applications from our dataset, then filtering and grouping the results by their frequencies in both malicious and non-malicious applications;
3) validating these findings on test applications to sort them in malware and non-malware categories mentioned after as "bad" and "good" app, respectively.

Our network analysis is a significant improvement on application behavior confirmation methods. The result shows successful capture and identification of malwares connecting to black listed domains. In the future attempts of finding such event, the illustrated method can be implemented quite easily.

Our System call based approach outlines a threshold for acceptability of an android app ("good" or "bad" as mentioned

978-1-4799-6664-6/16/$31.00 ©2016 IEEE

in the previous sections), which represents a *novel* approach, and can be used for experimentation related to malwares. No previous work reflects same quantitative definition we made in this approach.

The rest of the paper is organized as follows. In Section II, a brief idea of our detection approaches is introduced. Then, Section III describes our first approach to malware detection, the network analysis by packet capturing method, a full description of which can be found in [10]. Section IV discusses our second approach which uses system call traces of applications to predict malicious activity. In Section V, experimental results are presented to validate our model. Finally, Section VI provides concluding remarks.

II. Malware Detection Schemes

We concentrated our research on finding measures that can decisively classify any application as malicious or benign. To attain this, we proposed two independent approaches:

- Network based detection.
- System call based detection.

In network based detection, an app is classified as malware if it tried to connect to a malicious domain server (Section III). The system call based detection is based on identifying a malware by analyzing its similarity of system call frequencies to those of known malicious applications (Section IV).

III. Network based Malware detection

This malware detection method is divided into two steps. These steps are visualized in Fig. 1. At first, a log of Universal Resource Locators (URLs) that are contacted by applications for a specific period of time is created. Then we tried to match each entry (URL) of the log with a list of known malicious domains. If a match is found, the application that contacted the malicious domain is a malware itself or has been affected by one.

A. Creating the App-URL table

App-URL table is a history/log of all attempts made by all applications to communicate with remote servers over HTTP. The table consists of (url, app) entries. Each HTTP request maps to a single entry, where url is the URL which is contacted, and app is the one that originated the HTTP request.

This process is further subdivided into following tasks:

1) Packet dumping: We have recorded all incoming and outgoing network packets to/from the android device for specific duration of time. This creates a packet dump file that contains information of which port number (of the mobile device) is accessing which URL.

2) Netstat Logging: To relate port numbers with applications, we periodically executed *netstat* [11] command throughout the duration of packet dumping and saved the outputs. Netstat gives information of which port number is being used by which application when the command is executed.

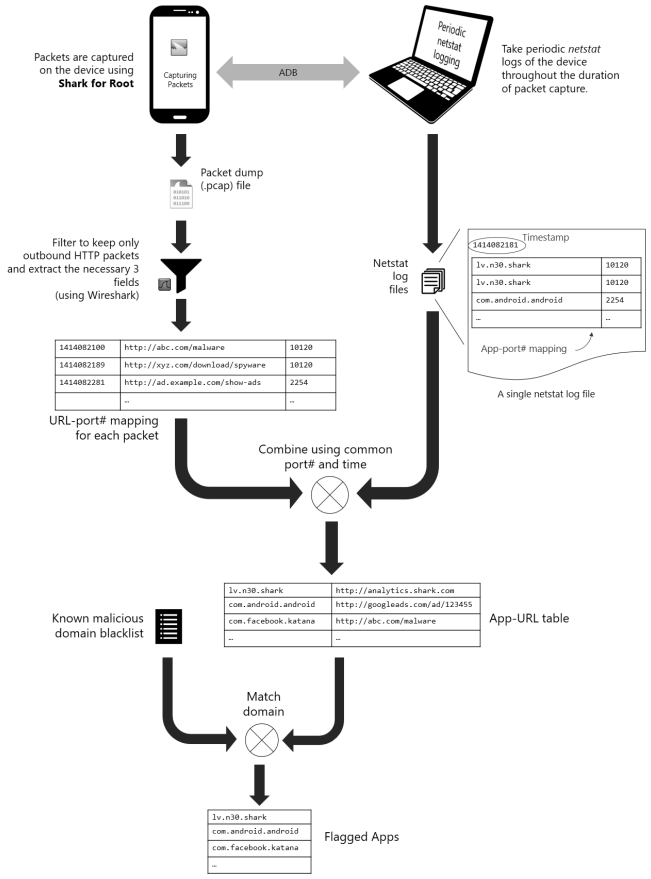


Fig. 1. Malware Detection by Network Analysis Workflow

3) Extracting necessary information from packet dump: We do not take all packets into consideration. We are only interested in HTTP packets (and only requests, not responses). We took only three fields from each packet: time, originating port and full request URI. This gives a time-sequenced log of port numbers and URIs that a port tried to connect to.

4) Aggregating packet dump and netstat logs: We have so far obtained two separate mappings: *application vs. port number* from netstat logs, and *port number vs. URL* from packet dump. We aggregate these two maps to create a time-sequenced log of applications and the URLs each application tried to contact (The App-URL table). Since netdump files contains *port-app* mappings for specific moments (1 second apart), a packet's time will not necessarily match exactly with any of these moments. To assign such a packet to an application, we have made following assumptions. Let t be the timestamp of a packet. Let $t_1, t_2, t_3, ..., t_{100}$ are the timestamps of the netstat outputs (they are stored in corresponding netdump files). Of course $t_1 < t_1 < t_3 < ... < t_{100}$. If $t < t_1$ or $t > t_{100}$, we discard the packet. We only consider packets with t such that $t_1 \leq t \leq t_{100}$. Now for each of these packets, there is an i such that $t_i \leq t$ and $t_{i+1} > t$. We assign a packet to an application using this rule: If the port

	Timestamp	Port#	App identifier	URL
1	1.414082194006204E9	52791	com.quora.android	http://www.quora.com/ajax/action_log_POST
2	1.414082195716379E9	42998	com.quora.android	http://www.quora.com/webnode2/server_call_POST
3	1.414082196603555E9	47619	com.quora.android	http://qph.is.quoracdn.net/main-thumb-9715372-5...
4	1.414082201279886E9	52225	com.quora.android	http://www.quora.com/webnode2/server_call_POST
5	1.414082240246866E9	45830	com.duolingo	http://api.duolingo.com/api/1/version_info
6	1.414082240286386E9	54574	com.duolingo	http://api.duolingo.com/api/1/store/get_invento...
7	1.414082255987588E9	45830	com.duolingo	http://api.duolingo.com/api/1/users/show?userna...
8	1.414082256455972E9	59259	com.duolingo	http://api.duolingo.com/api/1/store/get_invento...
9	1.414082269802286E9	39860	com.memrise.android	http://data.flurry.com/aap.do
10	...			

Fig. 2. Final output: App vs. URL table

was being used by application A at t_i and application B at t_{i+1} then, if $t - t_i \leq t_{i+1} - t$, application A originated the packet. Otherwise application B originated it. If no application was using the port at either t_i or t_{i+1}, We discard the packet. So for every packet (except the ones discarded), we know the app which originated it. And `filtered.txt` contains Full request URI of all packets. So we now know the URL specified in the packet was contacted by this application. We have logged these (Application, URL) entries for each packet and the App-URL table is ready. A sample table is shown in Fig. 2.

B. Matching the URLs with Domain-blacklists

We search the URLs in the App-URL table for known malicious domains. If an application tries to connect to a rogue domain (URL), we flag it as a malware.

IV. System call based malware detection

In this section, the overall strategy of our second approach for malware detection is discussed, which uses system call traces of applications to predict malicious activity. The results achieved in the experiment using this model is detailed and explained in section V.

There are two phases: training and classification. How the known malware and non-malware traces will be used to train a machine learning model/classifier is described in subsection IV-A. Subsection IV-B describes how the model will classify an unknown app using its syscall trace.

A. Training

We used a set of applications consisting both known malwares and known non-malwares as the training dataset. We collected system call traces of all applications of the training dataset (all of the applications are run for a specific amount of time). The system call trace of a single application is a list of system calls the application used during execution. For example: {recv, semget, msgget, ...}; where recv, semget, msgget are system calls.

After collecting system call traces, We aggregate this traces to create two binary relation matrices M_{mal} and M_{nmal}. M_{mal} shows relation between system calls and malware applications, Where M_{nmal} shows relationship between system calls and non-malwares. M_{mal} and M_{nmal} matrices are defined in equations 1 and 2:

$$M_{mal}(i,j) = \begin{cases} 1 & \text{if } i^{th} \text{ malware uses } j^{th} \text{ syscall} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

$$M_{nmal}(i,j) = \begin{cases} 1 & \text{if } i^{th} \text{ non-malware uses } j^{th} \text{ syscall} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Then we calculate the Goodness Rating of j^{th} syscall, γ_j as follows,

$$\gamma_j = \frac{1}{N_{nmal}} \sum_{i=1}^{N_{nmal}} M_{nmal}(i,j) - \frac{1}{N_{mal}} \sum_{i=1}^{N_{mal}} M_{mal}(i,j) \quad (3)$$

where N_{nmal} and N_{mal} are number of non-malware and malware samples.

B. Classification

To classify an unknown application as *malware* or *non-malware*, first, we execute the application for the same time duration duration we used with each training application. We collect the system call trace of that application during that execution, same as before. But this time, we also record the frequency of each syscall used by the application during execution. So now, the syscall trace of an application during classification phase can be expressed as a list of pairs of syscalls and their frequencies. For example: {(recv,1032), (semget, 143), ...} is a trace of an application which called the recv routine 1032 times, semget 143 times and so on.

Then we define the Goodness Rating of that application, γ_{app} as follows,

$$\gamma_{app} = \sum_{s \in S_{app}} \gamma_s \times f_s \quad (4)$$

Where, S_{app} is the set of system calls used by app, γ_s is the Goodness rating of syscall s and f_s is the frequency of syscall s in app.

Now for classification, we check if the goodness rating of the application under inspection exceeds some threshold. If so, we classify it as non-malware. Otherwise we flag it as malware.

$$\begin{cases} \text{app is not a malware} & \text{if } \gamma_{app} > \tau \\ \text{app is a malware} & \text{otherwise} \end{cases}$$

Where, τ is a **Threshold**. A Threshold (τ) defines a specific value of a metric beyond which a change is significant. Theoretically, the threshold should be zero. But practically, it depends on the experiment and the training data used. We used this approach to classify apps in our validation dataset and calculated accuracy (α), recall (ρ), specificity (σ), precision (ϕ) and f-measure (F). These metrics are defined in section V-A.

C. Preparing experiment

We collected system call traces of all applications using a single device. The reason behind this is we intended to provide identical environments for all applications to execute in. The device was reset to factory default configuration and the device needed to be *rooted*. We used standard linux utility *strace* to trace system call of applications. We also used *timeout* command to run every application for a fixed duration of time.

D. Experiment

We planned to collect system call traces of a total of 453 malwares and 227 non-malwares. The number of apps used in validation and training is shown in table I.

TABLE I
NUMBER OF APPS

	Training	Validation	Total
Malware	403	50	453
Non-malware	177	50	227

We wrote a batch script that automates the whole process. The script executes commands in the device using *Android Device Bridge (ADB)*. The workflow of the script is outlined in Algorithm 1.

After the execution of the above mentioned script, we are left with 453 malware trace files and 227 non-malware trace files. A sample single trace file is shown in Fig. 3.

E. Evaluating our model

1) Training: The program aggregates all the traces in the training dataset and produce two relation matrices M_{mal} and M_{nmal}. M_{mal} and M_{nmal} are defined in the previous section. A sample relation matrix is shown in Fig. 4.

The two relation matrices are used to calculate the **Goodness ratings** of all syscalls.

Algorithm 1 Syscall trace collect script

procedure COLLECT-ALL-SYSCALL-TRACE($directory$)
  for each apk file in $directory$ **do**
    $pckgname \leftarrow$ get package name from that apk using **aapt**
    Install the apk in the device.
    Launch the app
    $pid \leftarrow$ ps($pckgname$)
    $stracelogs[pckgname] \leftarrow$ output of **strace**(pid) with 20 seconds timeout
    Force close the app
    Uninstall the app
  end for
  return $stracelogs$
end procedure

% time	seconds	usecs/call	calls	errors	syscall
29.08	1.285126	2824	455		semget
26.44	1.168575	493	2371	6	recv
17.94	0.793062	793062	1		wait4
4.99	0.220610	1061	208		ioctl
4.51	0.199257	3558	56		fsync
4.23	0.186927	159	1176		msgget
3.36	0.148469	81	1830		mprotect
1.46	0.064444	198	325		write
1.14	0.050596	10119	5		nanosleep
1.14	0.050407	663	76	1	open
0.85	0.037481	487	77		close
0.67	0.029442	184	160		fstat64
0.60	0.026524	144	184		mmap2
0.50	0.021903	104	210		read
0.47	0.020971	142	148		sigprocmask

Fig. 3. System call trace of an application

	sigaltstack	mremap	rmdir	poll	pivot_root	...
apps.ignisamerica.cleaner.pro.	1	0	1	1	0	
ar.com.moula.zoomcamerapro.	1	0	1	0	0	
ccc71.at.	1	1	0	0	1	
com.a0soft.gphone.acc.pro.	1	0	1	0	1	
com.adobe.reader.	0	0	1	1	1	
com.agilebits.onepassword.	1	0	0	0	0	
com.alarmclock.xtreme.free.	1	1	1	1	1	
com.anydo.	1	0	0	1	0	
com.appspot.swisscodemonkeys.bald.	1	0	1	1	1	
com.apusapps.browser.	0	0	1	0	1	
com.bdjobs.app.	1	1	1	0	1	
com.bikroy.	1	0	0	1	1	
...						

Fig. 4. A sample relation matrix between syscalls and apps

2) Classification: After **Goodness ratings** of all apps have been calculated, our model is ready to classify an unlabeled app as malware or non-malware, given its system call trace. The same program calculates Goodness ratings of all applications in the validation dataset, using the equation given in section IV-B. If the Goodness rating of an app is lower than a Threshold (τ), our model/program flags the app as a malware, otherwise the app is considered to be non-malware.

V. RESULTS

The results achieved from our experiment described in section III and section IV is discussed in detail in this section.

A. Network Analysis Results

To validate this approach, we analyzed a total of 28 applications using this method (14 known malwares and 14 known

non-malwares). Among them, all of the 14 actual non-malware were correctly classified as non malwares (True negatives). None of the actual non-malware were mistaken as malware (False positives). However, 9 of the actual 14 malwares were correctly flagged as malwares (True positives), while the other 5 were mistakenly classified as non-malwares (False negatives). This result is shown in the form of a Confusion Matrix in table II.

TABLE II
CONFUSION MATRIX OBTAINED FROM EXPERIMENTAL DETECTION USING NETWORK ANALYSIS

9 True positives	0 False positives
5 False negatives	14 True negatives

We calculated the following metrics to evaluate the performance of our detection approach. All the respective values for True Positives (TP), False Positives (FP), True negatives (TN) and False Negatives (FN) are taken from table II.

1) **Accuracy:** The accuracy of prediction can be defined as follows: Accuracy is the proportion of true results (both true positives and true negatives) among the total number of cases examined. Therefore, Accuracy of our findings, Accuracy

$$\alpha = \frac{TP + TN}{TP + TN + FP + FN} = 82.14\%$$

2) **Recall:** Recall in information retrieval is the fraction of the documents that are relevant to the query that are successfully retrieved. Therefore, in our findings, Recall

$$\rho = \frac{TP}{TP + FN} = 60.8\%$$

3) **Specificity:** Specificity (also called the true negative rate) measures the proportion of negatives that are correctly identified. Therefore in our case, Specificity

$$\sigma = \frac{TN}{TN + FP} = 100\%$$

4) **Precision:** Precision or positive predictive value is defined as the proportion of the true positives against all the positive results (both true positives and false positives). Therefore, in our case, Precision

$$\phi = \frac{TP}{TP + FP} = 100\%$$

5) **F-measure:** The F-measure can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 100% and worst at 0. Therefore, in our case, F-measure

$$F = 2 \times \frac{\phi \times \rho}{\phi + \rho} = 75.6$$

where TP, TN, FP and FN are number of true positives, true negatives, false positives and false negatives respectively.

In Sub-subsection IV-E2, we introduced a Threshold (τ) and stated that the value of this threshold should be derived experimentally. It is dependent on the training dataset. Therefore, to find a reasonable value for τ, we tried different thresholds and for each threshold, we ran our classifier for all validation apps and calculated different metrics like accuracy (α), recall (ρ), specificity (σ), precision (ϕ) and F-measure (F). These metrics are defined in section V-A. We started from a threshold value of -200 and ended with 1500, with step 10. So the classifier was run a total of 171 times (each time with all validation apps). Now ideally, the **Threshold (τ)** should be zero, but our experiment showed better accuracy for other values. To be exact, the best accuracy (87%) is achieved when we use a threshold value between 300-340.

The Threshold (τ) vs Accuracy (α) graph is shown in Fig. 5. The effect of threshold on other performance metrics of the classifier is shown in Fig. 6 to 9. With increasing threshold, the classifier would tend to classify more apps as malware. So precision falls with increasing threshold. Threshold (τ) vs precision (ϕ) graph in Fig. 6 shows $\phi = 87.8\%$ for $\tau = 320$, which is more than acceptable.
Specificity (σ), also decreases as threshold increases. In Fig. 7, Threshold (τ) vs specificity (σ) graph shows $\sigma = 82.7\%$ for $\tau = 320$. So 320 is a good value for threshold considering specificity.

Detection rate of known malwares as malwares (recall) increases with threshold. Again, the Threshold (τ) vs recall (ρ) graph in Fig. 8 shows an excellent value of recall, $\rho = 90.1\%$ for $\tau = 320$.

And at last, Fig. 9 also shows an excellent F-measure of 0.889 for $\tau = 320$.

According to all these performance metrics, 320 seems to be a very plausible value as Threshold, (τ) for our model. The Confusion Matrix of our model resulted from using $\tau = 320$ is shown in table III.

TABLE III
CONFUSION MATRIX WITH $\tau = 320$

52 True positives	7 False positives
6 False negatives	35 True negatives

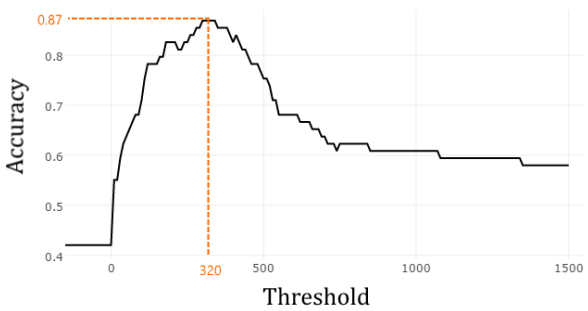


Fig. 5. Change in **Accuracy** (α) over **Threshold** (τ) graph

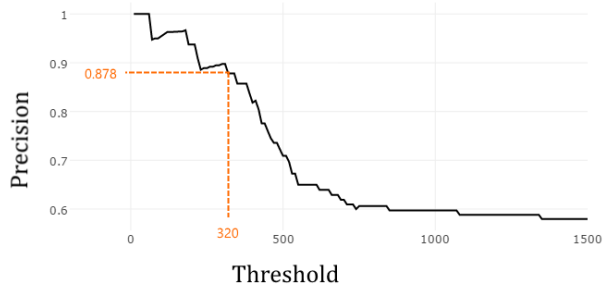


Fig. 6. Change in **Precision** (ϕ) over **Threshold** (τ) graph

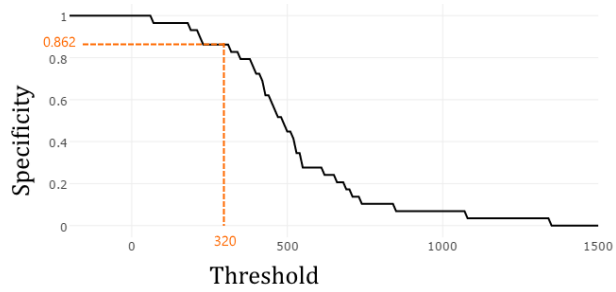


Fig. 7. Change in **Specificity** (σ) over **Threshold** (τ) graph

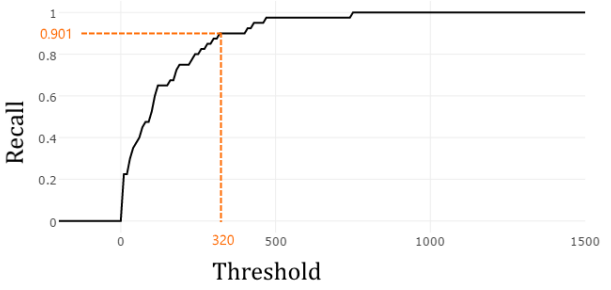


Fig. 8. Change in **Recall** (ρ) over **Threshold** (τ) graph

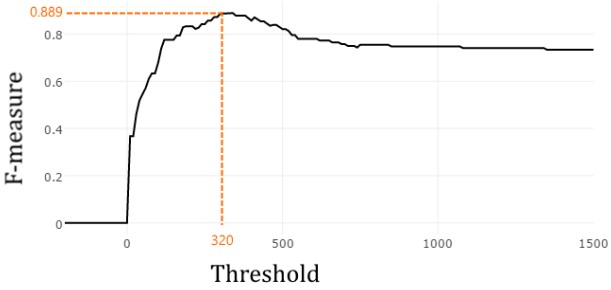


Fig. 9. Change in **F-measure** (F) over **Threshold** (τ) graph

Although we used widely variying types of malware and non-malware application samples in training and validation of our data, it is very difficult to amass a set of malware and non-malware apps that correctly emulates the distribution of all malwares in the wild and all non-malware apps in Google Play Store.

In this section, we showed our actual findings on network analysis after defining the performance metrics used in evaluation of our model. Then we have shown the visual representation of our findings in system-call based approach which shows our attempts of finding the perfect value for **Threshold**. After finding it our accuarcy reached 87% which is significant in general malware detection context. We have presented the result in a quantitative manner which to the best of our knowledge is not available in the previous works.

VI. Conclusion

In this paper, we proposed a definitive and novel approach of malware detection in Android devices. We have presented two independent methods of detection: in Network analysis by capturing the network packets, tracing their URLs, then cross-matching against a malicious server database, and in system call based analysis by measuring system call frequencies of an app, then doing a pattern-match with that of a malicious app. In the result section, we have shown that our methods have been validated by extensive experiments. We hope this will stimulate others to indulge into more focused, fine-tuned researches on malware detection.

VII. Acknowlegement

This research work has been conducted and supported by Bangladesh University of Engineering and Technology (BUET).

References

[1] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy*, San Francisco, CA, May 20-23, 2012, pp. 95–109.

[2] Kaspersky Lab and INTERPOL Survey Reports, http://media.kaspersky.com/pdf/Kaspersky-Lab-KSN-Report-mobile-cyberthreats-web.pdf , "Mobile cyber threats."

[3] The Number of Financial Attacks Against Android Users Tripled in 2014, http://www.kaspersky.com/about/news/virus/2015/The-Number-of-Financial-Attacks-Against-Android-Users-Tripled-in-2014 .

[4] S. Ramu, "Mobile malware evolution, detection and defense," in *EECE 571B, TERM SURVEY PAPER,*, April, 2012.

[5] M. Chandramohan and H. B. K. Tan, "Detection of mobile malware in the wild," *IEEE Computer*, vol. 45, no. 9, pp. 65–71, Sep 2012.

[6] T. S. D. Iland, A. Pucher, "Detecting android malware on network level," in *Technical representation UC Santa Barbara*, 2012.

[7] Y. J. Ham and H. Lee, "Detection of malicious android mobile applications based on aggregated system call events," *International Journal of Computer and Communication Engineering*, vol. 3, no. 2, pp. 340–350, March, 2014.

[8] C. C. H. T. Y. Lin, Y. Lai, "Identifying android malicious repackaged applications by thread-grained system call sequences," *Computers and Security*, vol. 39, pp. 340–350.

[9] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Internation Conference on Computational Intelligence and Security*.

[10] M. Zaman, T. Siddiqui, M. Amin, and M. Hossain, "Malware detection in android by network traffic analysis," in *Networking Systems and Security (NSysS), 2015 International Conference on*, Jan 2015, pp. 1–5.

[11] Netstat command, http://en.wikipedia.org/wiki/Netstat.