Two Phase Static Analysis Technique for Android Malware Detection

Priyadarshani M. Kate
Department of Computer Engineering
Defence Institute of Advanced Technology
Girinagar, Pune, India-411025
priyadarshani_mcse13@diat.ac.in

Sunita V. Dhavale
Department of Computer Engineering
Defence Institute of Advanced Technology
Girinagar, Pune, India-411025
sunitadhavale@diat.ac.in

ABSTRACT
The growing popularity of Android based smart phones has greatly fuelled the spread of android malware. Further, these malwares are evolving rapidly to escape from traditional signature-based detection methods. Hence, there is a serious need to develop effective android malware detection techniques. In this paper, we propose two phase static android malware analysis scheme using bloom filters. The *Phase I* involves two different bloom filters that classify a given sample into malware or benign class based on permission feature set only. The evaded malicious samples from *Phase I* are further analyzed by *Phase II* consisting Naïve Bayes Classifier using permission and code based mixed feature set. Inclusion of *Phase I* classification makes the technique computationally less intensive; while addition of the *Phase II* classification improves the overall accuracy of the proposed model. Experimental results indicate both detection accuracy and computational efficiency of the proposed technique.

Categories and Subject Descriptors
D.4.6 [Security and Protection]: Invasive software (e.g., viruses, worms, Trojan horses)

General Terms
Security, Experimentation

Keywords
Bloom Filter; Machine learning; Android malware detection

1. INTRODUCTION
Android based smart-phones have acquired nearly 85 percent of market in the second quarter of 2014, leaving behind all its rivals by huge margin [1]. Total number of Android apps available only on Google play store is 1,416,525 [2] and consumers have downloaded apps over 80 billion times [3]. Fortinet stated in its FortiGuard Quarterly lab reports of 2014 that the year 2013 had blast of mobile based malwares and detected over 1800 new distinct malware families with most of these focusing on Android [4]. Despite of the measures like Bouncer taken by Google, android malwares can evade using techniques like delayed attack; where malicious payload is included in the benign appearing app at the time of first or later updates [5], [6].

WCI '15, August 10-13, 2015, Kochi, India
© 2015 ACM. ISBN 978-1-4503-3361-0/15/08□$15.00
DOI: http://dx.doi.org/10.1145/2791405.2791558

Existing android malware detection techniques are broadly classified based on signature-based, static or dynamic detection. The signature based detection techniques requires a signature to be generated for each malware and can be easily evaded. Static detection technique is proactive and fast, flexible and can be easily automated. Also the malwares cannot modify its behavior during static analysis. In dynamic detection technique, the malware is executed in protected environment and run-time behavior of the malware is evaluated. Static analysis techniques can complement the signature based or dynamic methods for detection of unknown malwares. This paper proposes two phase static analysis for android malware detection. Bloom filter is used in *Phase I* of proposed technique for its quick pattern matching capabilities. As far as our knowledge, this is the first approach to utilize bloom filters for android malware detection. This in turn reduces computationally complexity. The unclassified samples from *Phase I* are again filtered using *Phase II* classifier. This improves overall accuracy of the system.

The rest of the paper is organized as follows: Section II explains the proposed android malware detection system. Section III contains the experimental results and findings and Section 4 presents the conclusion and future work.

2. PROPOSED WORK
The proposed model involves training cycle and testing cycle. Fig. 1 shows the training cycle of proposed scheme for *Phase I* and *Phase II* in detail.

2.1 *Phase I* Training Cycle
As shown in Fig. 1 (a), *Phase I* consists of Permission based Feature Extraction Module*, Bloom Filter (BFM)* trained with Malicious sample permission features and Bloom Filter *(BFB)* trained with Benign sample permission features respectively.

2.1.1 Permission based Feature Extraction Modules
To analyze any apk file it is first reverse engineered to extract the permissions. Every Android Application package contains AndroidManifest.xml (manifest file, all app components are declared in this file), classes.dex, resources and assets [10]. Before any application component is started by Android system,

it should know all the components present in the application, which it can read from AndroidManifest.xml file. The manifest file also contains the permissions that are required by the application, such as read access to user's camera or to user's messages. The aapt(Android Asset Packaging Tool) is used to extract the permissions which are stored in the Android manifest file [11].

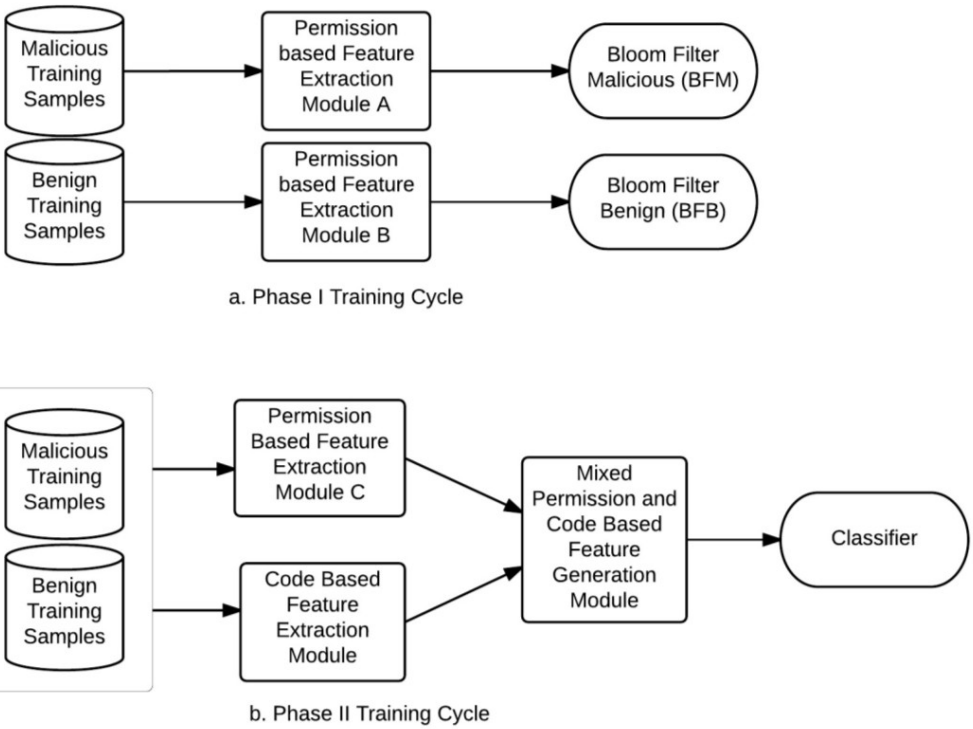

Figure 1. Training cycle for *Phase I* and *Phase II*

As stated in [16] top twenty requested permissions are extracted from malware samples (see Table 1) using Permission based Feature Extraction Module A to train *BFM*. Table 1 also shows top twenty requested permissions extracted from benign samples [16] using Permission based Feature Extraction Module B to train *BFB*.

Table 1. Top 20 Permissions Extracted

Bit Index	For training BFM	For training BFB
1	INTERNET	INTERNET
2	READ_PHONE_STATE	ACCESS_NETWORK_STATE
3	ACCESS_NETWORK_STATE	WRITE_EXTERNAL_STORAGE
4	WRITE_EXTERNAL_STORAGE	READ_PHONE_STATE
5	READ_SMS	VIBRATE
6	ACCESS_WIFI_STATE	ACCESS_COARSE_LOCATION
7	RECEIVE_BOOT_COMPLETED	WAKE_LOCK
8	WRITE_SMS	ACCESS_FINE_LOCATION
9	SEND_SMS	RECEIVE_BOOT_COMPLETED
10	RECEIVE_SMS	ACCESS_WIFI_STATE
11	VIBRATE	READ_CONTACTS
12	ACCESS_COARSE_LOCATION	WRITE_SETTINGS
13	READ_CONTACTS	GET_ACCOUNTS
14	CALL_PHONE	CAMERA
15	ACCESS_FINE_LOCATION	CALL_PHONE
16	WAKE_LOCK	WRITE_CONTACTS
17	WRITE_CONTACTS	GET_TASKS
18	CHANGE_WIFI_STATE	RECORD_AUDIO
19	WRITE_APN_SETTINGS	READ_HISTORY_BOOKMARKS
20	RESTART_PACKAGES	WRITE_HISTORY_BOOKMARKS

A 20 bit binary pattern is generated from extracted permissions, for each malware and benign training sample. A particular index in the pattern is set to 1 for corresponding permission present in the manifest file and set to 0 otherwise. For e.g., if bit pattern is

"10100000000000010000" means the sample is asking for INTERNET, ACCESS_NETWORK_STATE, WAKE_LOCK permissions. It has been found that malwares of same family mostly have same permission feature set because of which the pattern generated for them is usually same.

2.1.2 Bloom Filters BFM and BFB.

Bloom filter is simple, space efficient data structure defined in terms of a bit vector which answers the membership queries. Bloom filter has found its place in many applications like dictionaries, databases, distributed caching, IP trace back etc. [7] due to quick pattern matching capabilities e.g. Google Chrome uses bloom filter to identify malicious URLs [8], [9]. Bloom filter has also been used as an extension to ClamAV to reduce the time and memory usage [15].

Let A be the set of elements with which we want to initialize a bloom filter. A bloom filter is represented by Vector V of m bits, and k hashes are applied over every item in set A. The hashes provide an integer output between 1 and m and these outputs are used as indices in bloom filter bit vector. Indices which are obtained after applying k hashes to every item in set A are set to 1 in the bloom filter bit vector as shown in fig.2.

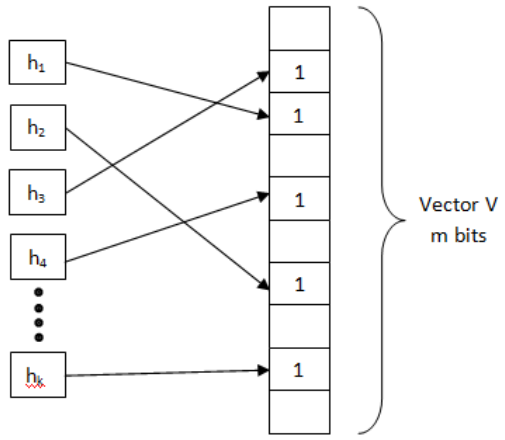

Figure 2. Bloom filter initialization

To check whether an item is present in bloom filter or not we first apply k hashes to that item. If the obtained indices are all set to 1 in the bit array, then that item is said to be the member of the Set A. If any of corresponding indices is set to 0, the item is not the member of Set A.

Bloom filter can thus be described by following parameters:

m : number of bits in the bloom filter

k : number of hash function applied to produce each index

n : number of items in Set A ($|A|$).

Hence, the probability P_{fp} of false positive can be given by Equation (1)

$$P_{fp} = \left(1 - e^{-\frac{kn}{m}}\right)^k \quad (1)$$

Therefore, there probability of false positives can be decreased by using $k = \ln 2 * \frac{n}{m}$ but that will increase the access time of bloom filter. Hence, k is chosen for acceptable value of false positive [7].

In *Phase I*, two different Bloom filters are used: Bloom Filter Malicious (*BFM*) and Bloom Filter Benign (*BFB*). *BFM* is initialized with permission features of malicious samples extracted in Permission based Feature Extraction Module A. Whereas, *BFB* is initialized with permission features of benign samples extracted in Permission based Feature Extraction Module B.

2.2 *Phase II* Training Cycle

As shown in Fig. 1 (b), *Phase II* consists of Permission based Feature Extraction Module C, Code based Feature Extraction Module, Mixed Code and Permission based Feature Generation Module and Classification Module.

The permissions are extracted using aapt Tool as stated earlier using Permission based Feature Extraction Module C. As stated in [13], the same top thirty ranked permissions are selected from both benign and malware samples to generate the permission based feature set.

Code based Feature Extraction Module is used to extract the code based features. Baksmali is used to disassemble the classes.dex file of corresponding APKs. The folders created by Baksmali contain multiple .smali files which are equivalent to corresponding java .class files [12]. The code based features are extracted by mining .smali file for each apk for some code based properties. These code based properties include, Java API calls, Linux System calls and some Android based commands. As stated in [13], the same top 25 ranked code based properties are selected from both benign and malware samples to generate the code based feature set. A 25 bit binary pattern is generated for each application based on selected top 25 ranked code based properties [13]. A particular index in the pattern is set to 1 for the corresponding code property present in the manifest file and set to 0 otherwise.

Further, a Mixed Permission and Code based Feature Generation Module is used to combine both permission and code based properties together to generate mixed feature set. The 30 bit permission and 25 bit code based pattern is merged together to form a 55 bits pattern. These 55 bit patterns are used to train Naïve Bayes classifier.

Naïve Bayes classifier is used in this phase as it suits well to our problem and is relatively faster than other classification techniques and is computationally less intensive. Naïve Bayes classifier is a simple probabilistic classifier based on Bayes' theorem with the assumption of independence between features.

2.3 Testing Cycle

During testing cycle, if any malicious sample gets escaped from *Phase I* detection, *Phase II* analysis will be executed. The Testing cycle uses trained bloom filters (*BFM* and *BFB*) as well as trained Naïve Bayes classifier for detection as shown in Fig. 3. The test samples are fed to *Phase I* of the proposed model. Permission based features are extracted from the test samples using both permission based feature extraction modules. The membership of test samples in checked for both bloom filters

BFM and *BFB*. The output given by *BFM* and *BFB* is fed to *Decision Module*

2.3.1 Decision Module

This module decides which samples should be sent to next phase on the basis of membership test result given by the bloom filters *BFM* and *BFB*. The samples which are classified as benign by both trained bloom filters are classified as benign. Samples classified as malicious by both bloom filters are classified as malicious. The samples that are ambiguously classified by *BFM*

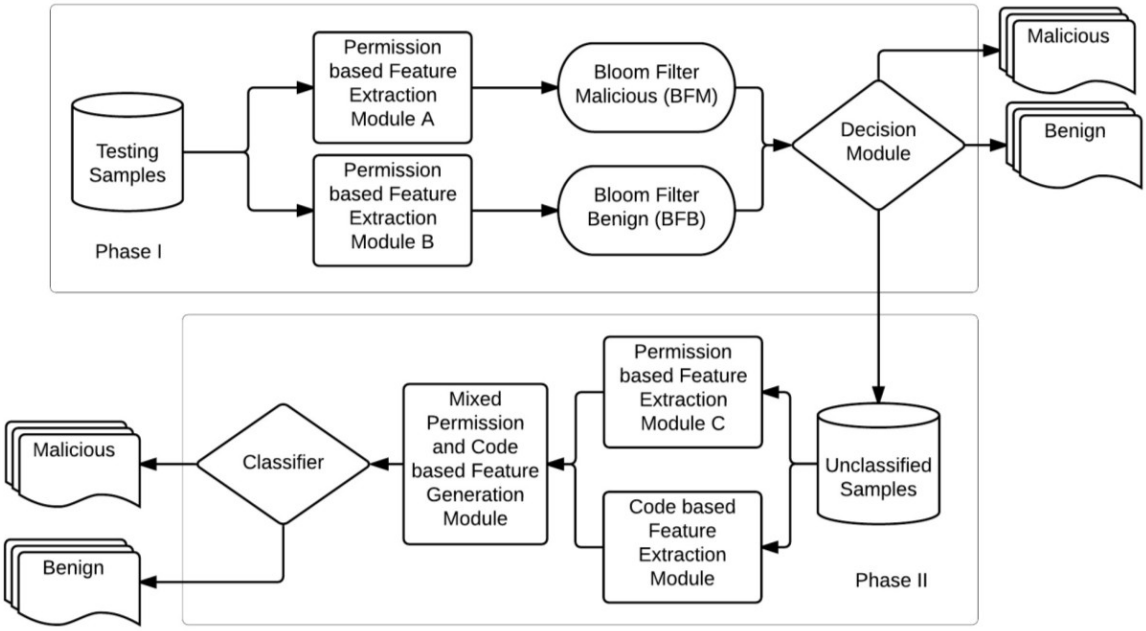

Figure 3. Testing Cycle

and *BFB* are then fed to the next phase. The decision logic of this module is given in table 2.

Table 2. Decision Module Logic

	BFM	BFB	Decision Module
Samples	Benign	Benign	Benign
	Benign	Malicious	Unclassified
	Malicious	Benign	Unclassified
	Malicious	Malicious	Malicious

The unclassified samples are then fed to *Phase II*. In *Phase II* testing cycle mixed features are generated from unclassified samples, by merging extracted permission and code based features. The trained Naïve Bayes classifier classifies these escaped samples based on mixed features.

3. EXPERIMENTATION

3.1 Experimental Setup

The experiments were carried out on machine with Windows 7 OS installed on 3.1GHz Intel core-i5 processor with 4GB RAM. The IDE used is Eclipse for Java. Aapt Tool is used to extract permissions from .apk file. Baksmali is used to disassemble the classes.dex file to get .smali files for each application.

3.2 Dataset Preparation

We have used 190 benign samples (acquired from Google Play Store and other sources), out of which 160 were used training samples and 30 as test samples and 190 malware samples(acquired from Contagio mobile mini dump), out of which 160 were used training samples and 30 as test samples.

3.3 Evaluation Measures

We used several different evaluation measures to check the performance of our model. Let $n_{ben \to ben}$ be the number of benign applications correctly classified as benign, $n_{ben \to mal}$ the number of misclassified benign applications, $n_{mal \to mal}$ the number of suspicious applications correctly identified as suspicious, while $n_{mal \to ben}$ represents the number of misclassified suspicious applications. Accuracy and error rate are, given by

$$Acc = \frac{n_{ben \to ben} + n_{mal \to mal}}{n_{ben \to ben} + n_{ben \to mal} + n_{mal \to ben} + n_{mal \to mal}} \quad (2)$$

$$Err = \frac{n_{ben \to mal} + n_{mal \to ben}}{n_{ben \to ben} + n_{ben \to mal} + n_{mal \to ben} + n_{mal \to mal}} \quad (3)$$

The accuracy measurement indicates the overall proportion of correctly classified instances, whether suspicious or benign, during the testing phase of the particular model. The error rate is the complementary measure to the accuracy, which can also be computed from $Err = 1 - Acc$. Fig. 4 shows the evaluation measures for proposed classification system. Here true positives are all the malicious samples correctly classified as malicious and given by Equation (4). True negatives (see Equation (5)) are all the benign samples correctly classified as benign. False positives are all the benign samples wrongly classified as malicious and false negatives are all the malicious samples wrongly classified as benign.

$$True\,Positive = \frac{n_{mal \to mal}}{n_{mal \to mal} + n_{mal \to ben}} \quad (4)$$

$$True\ Negative = \frac{n_{ben \to ben}}{n_{ben \to ben} + n_{ben \to mal}} \quad (5)$$

$$False\,Positive = \frac{n_{ben \to mal}}{n_{ben \to ben} + n_{ben \to mal}} \quad (6)$$

$$False\,Negative = \frac{n_{mal \to ben}}{n_{mal \to mal} + n_{mal \to ben}} \quad (7)$$

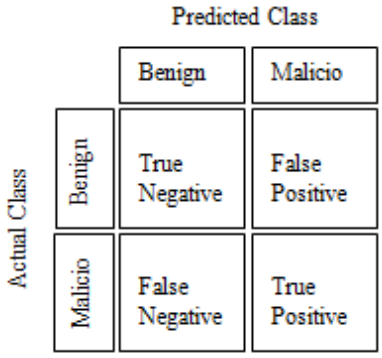

Figure 4. Evaluation Measures

3.4 Results

Table 3 and Table 4 show the results of *BFM* and *BFB* respectively. Further, the decision logic in *Phase I* applies the rules given in Table 2 to select unclassified samples for next phase classification. The samples which are classified as benign by both trained bloom filters are classified as benign. Samples classified as malicious by both bloom filters are classified as malicious. Table 5 shows the results for Decision Module. It can be seen from results shown in Table 3, 4 and 5 that, *Phase I* reduces the test dataset by nearly 50 percent, which reduces the need of Code based Feature Extraction Process for nearly half test set.

Total time taken to execute *Phase I* testing cycle is 952 milliseconds for 60 samples, where permission feature extraction process takes 944 milliseconds. Comparatively, total time taken to execute *Phase II* testing cycle for remaining unclassified 31 samples is found to be 36 minutes 52 seconds and 426 milliseconds for mixed feature extraction and 31 milliseconds for actual classification. This shows that, Code based Feature Extraction is computationally expensive compared to Permission based Feature Extraction process. Hence, inclusion of *Phase I* filtering make the proposed model computationally less expensive, thereby reducing the overall execution time.

Table 3. Results of BFM

		Predicted Results	
		Benign	Malicious
Actual Input	Benign (30 samples) To BFB	93% (28 samples)	7% (2 samples)
	Malicious (30 samples) To BFM	30% (9 samples)	70% (21 samples)

Table 4. Results of BFB

		Predicted Results	
		Benign	Malicious
Actual Input	Benign (30 samples) To BFB	23% (7 samples)	77% (23 samples)
	Malicious (30 samples) To BFM	10% (3 samples)	90% (27 samples)

Table 5. Results of *Phase I* Decision Module

		Predicted Results		
		Benign	Malicious	Unclassified
Actual Input	Benign (30 samples) To BFB	23% (7 samples)	7% (2 samples)	70% (21 samples)
	Malicious (30 samples)	3.33% (1 sample)	63.33% (19 samples)	33.33% (10 samples)

	To BFM			

Table 6 shows results obtained by *Phase II* classification. It can be seen that the FP and FN are reduced to lower values after *Phase II* classification. Experimental results show that the accuracy of the proposed model is 87% and error rate is 13%.

Table 6. Results after *Phase II*

		Predicted Results	
		Benign	Malicious
Actual Input	Benign (30 samples)	93% (28 samples)	7% (2 samples)
	Malicious (30 samples)	20% (6 samples)	80% (24 samples)

The proposed model is also compared with the code and permission mixed feature based model proposed in [13] using the same dataset and the results are stated in table 7. From results it can be seen that our approach provides low computational complexity and provides good tradeoff between time complexity and classification accuracy.

Table 7. Results after comparison with [13]

Model	Timing	Accuracy
Code and Permission mixed feature based model	62 minutes, 32 seconds and 213 milliseconds	89%
Proposed model	36 minutes, 53 seconds and 370 milliseconds	87%

4. CONCLUSION

In this paper, a two phase static analysis for android malware detection is proposed. Application of bloom filters is studied for *Phase I* detection to reduce computational complexity. As given in literatures, though Bloom Filter doesn't have any false negatives; it suffers from many false positives. To overcome this limitation, two separate bloom filters, trained with benign and malware datasets respectively are used. Naïve Bayes classifier is used in *Phase II* of classification for rigorous analysis of malwares. It can be seen from experimental results that, use of bloom filter substantially reduces the evaluation time of the proposed model. Also the overall accuracy of detection achieved is 87%. Our future work involves testing the scalability of our solution for larger dataset and increase the accuracy rate further.

5. REFERENCES

[1] Bagchi, S. 'With 85% Market Share, Android Set To Crush Rivals - Cxotoday.Com'. *Cxotoday.com*. Web. 16 August 2014.

[2] Appbrain.com,. 'Android Operating System Statistics - Appbrain'. Web. 26 Nov. 2014.

[3] Yale, Brad. 'The Fight For The Mobile App Market: Android Vs. Ios | Informit'. *Informit.com*. Web. 29 June 2014.

[4] Fortinet.com,. 'Fortinet'S Fortiguard Labs Reports 96.5% Of All Mobile Malware Tracked Is Android Based, Symbian Is Distant Second At 3.45%; Ios, Blackberry, Palmos, And Windows Together Represent Less Than 1%Fortinet | Network Security, Enterprise And Data-Center Firewall'. N.p., 2014. Web. 10 July 2014.

[5] Hou, Oliva. 'A Look At Google Bouncer | Malware Blog | Trend Micro'. *Blog.trendmicro.com*. N.p., 2012. Web. 28 July 2014.

[6] Filiol, E., Jacob, G., & Le Liard, M. 2007. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology*, *3*(1), 23-37.

[7] Broder, A., & Mitzenmacher, M. 2004. Network applications of bloom filters: A survey. *Internet mathematics*, *1*(4), 485-509.

[8] Yakunin, Alex. 'Alex Yakunin's Blog: Nice Bloom Filter Application'. *Blog.alexyakunin.com*. N.p., 2010. Web. 21 June 2014.

[9] Chromiumcodereview.appspot.com,. ' Issue 10896048: Transition safe browsing from bloom filter to prefix set.'. Web. 3 July 2014.

[10] Fundamentals, Application. 'Application Fundamentals | Android Developers'. *Developer.android.com*. Web. Feb. 2015.

[11] Elinux.org,. 'Android Aapt - Elinux.Org'. Web. 24 June 2014.

[12] Code.google.com,. 'Smali - An Assembler/Disassembler For Android's Dex Format - Google Project Hosting'. Web. 5 July 2014.

[13] Yerima, S. Y., Sezer, S., & McWilliams, G. 2014. Analysis of Bayesian classification-based approaches for Android malware detection. *IET Information Security*, 8(1), 25-36.

[14] Contagiominidump.blogspot.in,. 'Contagio Mobile'. Web. 7 Aug. 2014.

[15] Cha, S. K., Moraru, I., Jang, J., Truelove, J., Brumley, D., & Andersen, D. G. 2011. SplitScreen: Enabling efficient, distributed malware detection.*Communications and Networks*, Journal of, 13(2), 187-200.

[16] Zhou, Y., & Jiang, X. (2012, May). Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium* on (pp. 95-109). IEEE.