IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 17, Issue 4, Ver. IV (July – Aug. 2015), PP 52-60
www.iosrjournals.org

An Improved Feature Vector storage metric for fast Android Malware Detection Framework

Mohit Sharma[1], Meenu Chawla[2], Vinesh Jain[3], Jyoti Gajrani[4]
[1, 2](*Dept. of CSE, Maulana Azad National Institute of Technology, Bhopal, India*)
[3, 4](*Dept. of CSE, Government Engineering College, Ajmer, India*)

Abstract: *Android based devices are rapidly flourishing day-by-day, due to its ease of use and popularity. As a result, the number of malware attacks on Android is also increasing. This paper is based on the Text Mining approach for analyzing Android malware families. The proposed methodology is motivated by the method introduced by Guillermo Suarez-Tangil which aims to automate malware analysis process based on DENDROID. The main issue in this regard is the storage of Family Feature Vectors (FFV) which is stored as sparse matrix. Therefore, this work presents a novel concept of Compressed Row Storage (CRS) to store the statistical features intellectually. By implementing this methodology, the FFV of Malware families are stored in an efficient manner. The experimental result proves that the large reduction (79%) in space needed to store FFV which incorporates only the non-zero elements is observed. This eventually leads to the reduction in the Feature Vector generation time and the Total process time. The proposed methodology will reduce the dimensionality and hence the time searching for a particular malware family signature.*
Keywords: *Android Malware, Text Mining, Statistical Features, Family Feature Vectors, Sparse Matrix, Compressed Row Storage.*

I. Introduction

Smartphone sales growth is rapidly growing across the globe. The worldwide market of Smartphone has risen up to 27.2% from 2013 to 2014 says International Data Corporation (IDC) [1]. According to its report, the Smartphone shipments reached 334 million units till the 1st quarter of 2015 with Android Operating System (OS) taking the most part (≈ 78%) of the share. Android users, from the past few years, have been increasing very rapidly due to its open source nature [2]. But the counterpart of this fact is that there has been a continuous proliferation of Android Malware also.

The Cybercriminals are continuously exploring the vulnerabilities and have now become more creative in camouflaging their work. According to the Intelligence report published by Symantec Corporation in 2015 [3], There was an average of 39 Android malware variants per family discovered in May, 2015. These include Malware that steals information to Malware that are Adware. Continuous research work has been proposed to thwart such attacks. The discipline of analysis and detection of Android malware can be broadly classified into two categories, namely, **Static** and **Dynamic**. The Static analysis aims to detect the anomalous behavior by converting the executable code back into source code and then covering each and every path in the code. There are many methodologies pertaining to Static Analysis. One such methodology is the Text Mining. Text Mining gathers some meaningful information from the source code of app under consideration. Using Text Mining, the unique signatures and in turn, the behavior of Android Application Package (APK) file can be stored. The existing methodology used this approach for storing the FFV. This paper proposes an improvement regarding storage of the Feature Vector. The rest of the paper is summarized as follows. Section II elaborates the background of the field. Section III covers related work regarding Static Analysis. Further, Section IV presents the proposed methodology. Next, Section V discusses the implementation results. Lastly, Section VI draws the conclusion with proposed futuristic work.

II. Background

The background of this paper focuses on the existing methodology that was applied for malware detection. The technique proposed in [4] focuses on the Text Mining approach. In this context, it is referred to as retrieval of quality or "intelligent" information about the patterns appearing in the source code of APK file. The Android Malware Genome Project dataset APK files was used for this purpose which consists of 49 Malware families with a total of 1,254 APK files in it. The paper applied reverse engineering on the dataset to obtain the source code using the Google's Androguard open source tool [5]. The Androguard is based on Context Free Grammar (CFG) [6] through which the signature of Malware family is generated. The signatures are made up of "Code Chunks" (CC) which are formed by applying the rules of CFG. As Android Applications are written in Java language, each CC represents a method present in the class. After this, the mathematical analysis was performed on the signature files to evaluate the following terms:

CC(a) – The set of all different CCs found in the app a.
Redundancy R(a) – Measures the fraction of repeated CCs present in an app.
Family Code Chunks FCC_{Fi} – Total CCs of a particular family.
Common Code Chunks CCC_{Fi} – The set of Common CCs within a particular family.
Fully Discriminant Code Chunks ($FDCC_{Fi}$) – The set of Common CCs of family F_i that are not found in other families.

The FDCC value of Family signature reveals that it can be used as a component to detect a particular Malware family. The authors contradicted that as FDCC is fragile and is dependent on CCC, it can't be used for detection. To solve this problem, Vector Space Modeling (VSM) is applied. In the context of the methodology, VSM measures the relevance of a particular CC 'c' in an app 'a'. The Code Chunk Frequency (CCF) measures the frequency of a code chunk 'c' in a Family F_i. The Inverse Family Frequency (IFF) of a CC 'c' measures how frequently a CC appears in a Family F_i and not in all other Family F_j, where $i \neq j$. The CCF * IFF factor is evaluated and is thus stored as FFV.

The authors built a Java implementation of VSM and trained the system using a dataset of 621 malware instances from the Android Malware Genome Project (AMGP) dataset. A total of 84,854 unique CCs were found across all malware families. The system was tested using another dataset of 610 malware instances using the popular 1- Nearest Neighbor (1-NN) Malware classifier for predicting the family to which it belongs. The experiment was conducted and 5.74% false positives were found. They also inspected that DroidKungFu was the major malware and with time its many variants were emerged. The authors also did evolutionary analysis of Malware Families. As mentioned above, the feature vectors of each family are stored with a dimension of 84,854 making it very sparse. Therefore, the paper proposes a solution to this problem through efficient storage of FFV.

III. Related Work

A number of automated tools are available for Android Malware analysis. In this regard, the authors have proposed several research works. A comprehensive study has been done in [7] by reviewing about 100 papers regarding feature selection and categorized the features that can be extracted from the APK file for malware detection. This includes Static (permissions, Java code, Intent filter, network address, etc.), Dynamic (system calls, network traffic, User Interface, etc.), Hybrid (includes a combination of static and dynamic features), and Application metadata (metadata, APK category, rating & description, etc.). In [8], focus is made on permission based analysis and applied various feature selection methods and classification algorithms. The key idea they adopted was to consider the permissions requested by an APK and then selecting the features that best represents the features in the dataset and then apply the suitable classification algorithm. In [9], emphasis is made on mining the Android permission patterns, by extracting the required permissions of both benign and malicious applications' dataset. The technique also gained knowledge by analyzing the "used" permissions of various apps. The difference of these permissions identified the anomalies. They used Contrast Permission Pattern Mining (CPPM) algorithm for contrast detection and malicious app detection. In [10], the signatures are generated by extracting improbable byte features which are resistant obfuscation and repackaging. The method generated the entropy features on a byte block window and the normalized most popular features were extracted. Further, they employed a similarity digest hashing scheme on byte stream based on robust statistical malicious features. The signatures in [11] are generated by considering the source code as N-gram signatures. The N-gram is a probabilistic machine learning algorithm that predicts the next item in the sequence with given datasets in order (N-1) as in Markov Model. After this, a Common Vulnerability Scoring System (CVSS) was employed to define the level of vulnerability of each feature generated. The behavior of android applications is well governed by two things: Manifest file and class file was stated in [12]. So, they extracted the features based on both criteria. The manifest based features include the number of activities, the number of services, the number of receivers, the list of permissions and broadcasts listening to. The class based features are extracted from opcode frequencies, 2-gram opcode frequencies and application Programming Interface (API) calls. These parameters were called as sensitive parameters and the sensitive index was obtained taking these parameters. In [13], the function call graphs of the APK file is extracted and then employed an explicit mapping to efficiently map call graphs to an explicit feature space. Then, Support Vector Machine (SVM) was trained to distinguish benign and malicious applications. In [14], flexible and robust obfuscation and repackaging resistant family signatures is constructed by composing multiple entries, consisting of four types of binary patterns viz., the class name, the method name, the character string and the method (bytecode) body. The signatures were stored in hashed format with associated weight, thereby reducing the signature comparison process time. The unknown sample was compared for the hash patterns and a similarity detection metric was used for calculating the similarity between the sample's signature and trained dataset's signatures. Malicious behavior patterns or models was mined in [15], which they called modalities (programming logic segments corresponding to known suspicious behavior) by drawing a two-tiered behavioral graph (Component Dependency and Component Behavior Graph). Within

this behavior graph, it automatically identifies modalities and defines a modality vector. For unknown target app, its modality was constructed and compared with the modality vector for malware detection.

IV. Methodology

For implementing the concept introduced in Section II, the same AMGP dataset was taken [16,17]. The dataset consists of 33 Malware families with 1,249 samples under them. Another dataset that has considered is from the contagio [18]. These datasets are referred to as Dataset - 1 and Dataset - 2 respectively in the remaining context of this paper. The distribution of family wise apps for both the datasets is as follows:

Table 1: The AMGP dataset

Malware Family	Sample Count	Malware Family	Sample Count
ADRD	21	**GingerMaster**	4
AnserverBot	187	**GoldDream**	47
Asroot	8	**Gone60**	9
BaseBridge	122	**GPSSMSSpy**	6
BeanBot	8	**HippoSMS**	4
Bgserv	9	**Jitfake**	1
CoinPirate	1	**jSMSHider**	16
CruseWin	2	**KMin**	52
DogWars	1	**LoveTrap**	1
DroidCoupon	1	**NickyBot**	1
DroidDeluxe	1	**NickySpy**	2
DroidDream	14	**Pjapps**	58
DroidDreamLight	46	**Plankton**	11
DroidKungFu1	34	**RogueLemon**	2
DroidKungFu2	30	**RogueSPPush**	9
DroidKungFu3	309	**SMSReplicator**	1
DroidKungFu4	96	**SndApps**	10
DroidKungFuSapp	3	**Spitmo**	1
DroidKungFuUpdate	1	**Tapsnake**	2
EndOfDay	1	**Walkinwat**	1
FakeNetFlix	1	**YZHC**	22
FakePlayer	5	**zHash**	11
GamblerSMS	1	**Zitmo**	1
Geinimi	67	**Zsone**	12
GGTracker	1		

Table 2: The contagio dataset

Malware Family	Sample Count	Malware Family	Sample Count
Airpush	1	**LoveTrap**	1
AnserverBot	2	**NickySpy**	3
Bgserv	3	**OBada**	2
CruiseWin	2	**Pincer**	4
DogWars	2	**PjApps**	45
DroidDeuxe	1	**Plankton**	35
DroidDream	8	**Ransom**	8
Fakebanker	11	**RogueSPush**	10
FakeNetflix	2	**SMSTrojan**	14
FakePlayer	3	**SndApps**	10
Geinimi	7	**Spitmo**	2
Gone60	6	**TapSnake**	3
HippoSMS	2	**Titan**	2
jSMSHider	17	**Zitmo**	9
KungFu	311		

4.1 Introduction to CRS format

For the objective of storing the $M \, x \, N$ feature matrix in an efficient form, the concept of CRS format [19] for storing sparse matrices is introduced hereby. This Algorithm makes no assumptions about the sparsity structure of the matrix. It stores only necessary elements. It can be applied to any $M \, x \, N$ sparse matrix. This format puts the subsequent non-zero values of the matrix rows in contiguous memory locations. The $M \, x \, N$ matrix is stored and handled through three vectors as:

Value (val): This is a 1 x C vector which contains all the non-zero elements of the matrix in contiguous format when the elements are traversed in ROW fashion.

Column Index (col_idx): This is also a 1-Dimensional vector which contains the column indexes of the elements in the Val vector.

Row Pointer (row_ptr): A Single Dimensional vector whose size is (|M|+1), where M is the number of Rows. The first element of this vector is always initialized with zero. To understand the above format, consider the following $6 \, x \, 5$ sparse matrixes A having the elements as:

$$A = \begin{bmatrix} 12 & 0 & 0 & 0 & 0 & -1 \\ 5 & 0 & 0 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 23 \\ 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The val vector store all the non-zero elements of the matrix. The col_idx vector stores the column indexes of all the non-zero elements. The row_ptr stores the number of non-zero elements per row. Thus, the CRS format of the above matrix is depicted by the three vectors as:

- **val** = [12, -1, 5, 3, 1, 23, 6]
- **col_idx** = [1, 5, 1, 4, 2, 6, 1]
- **row_ptr** = [0, 2, 2, 2, 1, 0]

4.2 Discussions

A condition may come when this concept of storing sparse matrices may fail. Let, for example, the size of each of the three, 1-dimensional vector mentioned above, be 'K', the rows and columns of the sparse matrix be 'M' and 'N', then if the condition mentioned in the equation (1) below satisfies,

$$\boldsymbol{3K + 1 \geq M \times N} \tag{1}$$

then, CRS methodology fails to store sparse matrices. This condition comes when; the matrix is not much sparser. The dataset analyzed contains 33 Malware Families and these malware families possess different code structures. Thus, the signatures generated by different families, will be mostly mutually exclusive. The Fig. 1 depicts this situation:

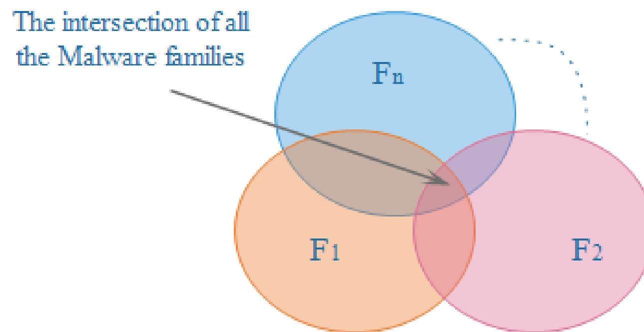

Figure 1: The mutually exclusive condition for the intersection of various families

To understand this mathematically, consider the equation (2) given below for evaluating the intersection of all the CCCs from all the families:

$$\begin{aligned} P(F_1 \cap\ F_2 \ldots\ldots \cap F_{33}) \\ &= \big(P(F_2 \cap\ F_3 \ldots \cap F_{33}) * \ P(F_3 \cap\ F_4 \ldots \cap F_{33}) \ldots\ldots P(F_{32} \cap\ F_{33}) * \ P(F_{33})\big) \\ &* \big(P(F_1 |\ F_2 \ldots \cap F_{33}) * \ P(F_2 |\ F_3 \ldots \cap F_{33}) * \ P(F_3 |\ F_4 \ldots \cap F_{33}) * \ P(F_{32} |\ F_{33})\big) \end{aligned} \tag{2}$$

Here, $F_1, F_2, \ldots\ldots F_{33}$ represents the CCCs present in their respective family. Consider two malware families CCCs. Their intersection will give the CCCs that are occurring in both the families represented as $I(CCC_{1,2})$. Now, computation of the probability of occurrence of CCC of third family with respect to this intersected value, i.e. $I(CCC_{1,2\,|\,1})$ is required which is represented as $I(CCC_{1,2,3\,|\,1,2})$. On generalizing this, the equation (3) forms as mentioned below:

$$I(CCC_{1,2,\ldots\ldots n|1,2,\ldots n-1}) \tag{3}$$

The above equation can be alternatively considered as the conditional probability of the CCCs occurring among all the families. In the dataset analyzed, 84,854 unique FFV elements pertaining to a Family were found. There are a total of 33 Malware families. Thus the total number of elements of the matrix is approximately 28 Lakhs. Out of these elements, only 1.6 Lakhs elements were found to be non-zero which computes to 5.7%.

The Equation (2) is applied on the CCC values of the dataset and considered for evaluation of the proposed methodology. The result as per the evaluation was 4.67%. This value can be considered as those important and concerned CCs relative to all the Malware families. In turn, this value depicts the percentage of non-zero elements. Our analysis is closer to actual value calculated above. However, it can be assumed that the upper bound of this value can be greater than 10%. On the basis of the above analysis, it can be concluded that the condition stated in Equation (1) is likely to be never achieved. Hence, the matrix will always contain sparsity of at least 90% as per the analysis of dataset. Another issue is the application of the Malware classifier algorithm to the thus reduced feature vector file. For this, a set of training and testing dataset can be prepared with almost equal amount of test APK files. The system can be trained with the proposed methodology and the feature

vector file is stored in CRS format. The testing dataset's feature vectors are generated in the same manner. Then, a matching algorithm can be employed that matches its corresponding values of column index and non-zero of the values with that of training dataset. A suitable threshold match value can be set that classifies the training set instances to their predicted malware families with a very less false positive rate. The algorithm of the CRS format for storing sparse matrix is as follows:

Input: M x N values that probably generate Sparse Matrix.
Output: Three 1-D vectors namely: **val, col_idx, row_ptr**
Procedure: 1

```
foreach row, i ε [1..M] do –
        Initialize countRow to 0
        foreach column, j ε [1..N] do –
                If ijth element is Non-Zero then do
                Add element to val vector
                Store its column value i.e. j in col_idx
                Update countRow
                End if
        end for
Add countRow to row_ptr
end for
```

The overall working of the proposed methodology is shown in Fig. (2) as shown below:

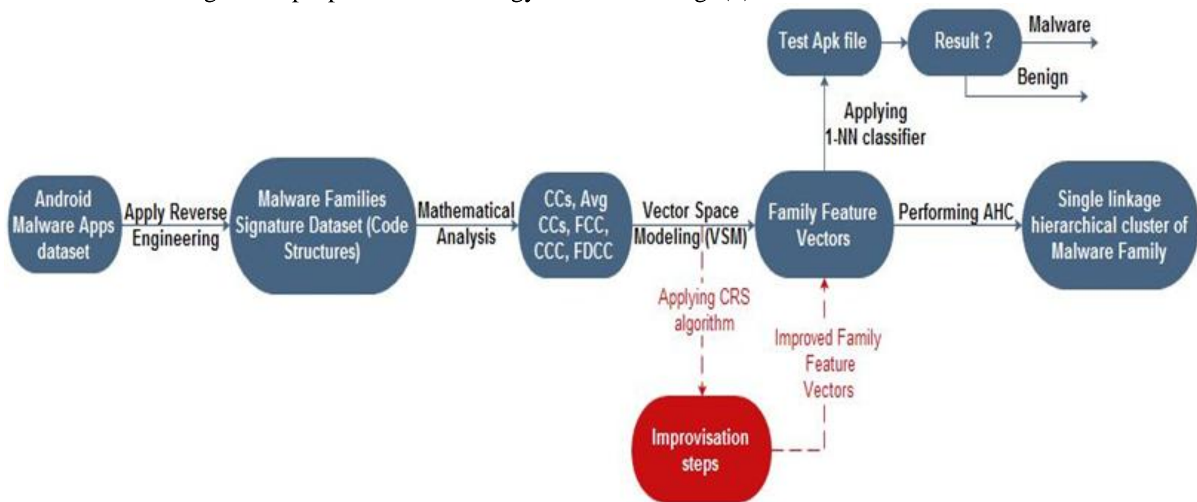

Figure 2: Overall working of existing approach and highlighted proposed approach

V. Evaluation

The original as well as the modified method used the datasets mentioned in Table 1 and 2 and were executed on a benchmark machine (Intel Core-i5, 4GB RAM) and the experimental results are figured as below. The size of the Feature Vector file that had been generated using the previous methodology showed a very great reduction from 11983 KB (11.70 MB) to 2492 KB (2.43 MB). Fig. 3 shows the comparison of the disk space taken in the storage of FFV using both the methodologies on Dataset - 1.

In addition to above improvement, it was also observed that in the FFV generation phase also, a speedup has been achieved. For Dataset - 1, there were in total, 15 runs that were performed on the benchmark machine, to calculate the difference in feature generation time and total process time when both the methodologies were applied. Among that runs, the minimum and the maximum time taken by the existing methodology for feature generation time was between 0.8 seconds and 1.7 seconds respectively, while that of the proposed methodology, it was computed to be 0.42 seconds and 1.7 seconds respectively. The two boundary outlier values were discarded while calculating the results. The five data values with a variation of 13% are plotted in which the existing methodology took 1.17 to 1.42 seconds while that of proposed methodology took 0.78 to 1.09 seconds. The results are shown in Fig. 4. For the whole process time, the five values with the existing methodology stood between 49.92 to 51.87 seconds while that of the proposed methodology between 48.73 to 49.81 seconds on Dataset - 1 only as shown in Fig. 5.

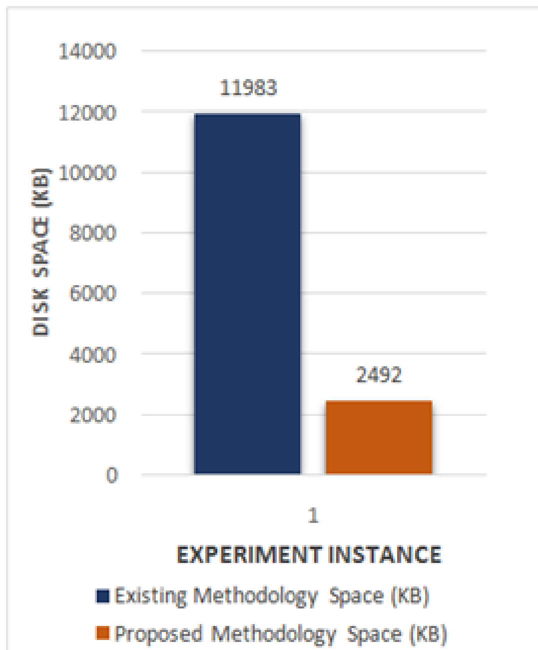

Figure 3: Disk space comparison between existing and proposed approach on Dataset - 1

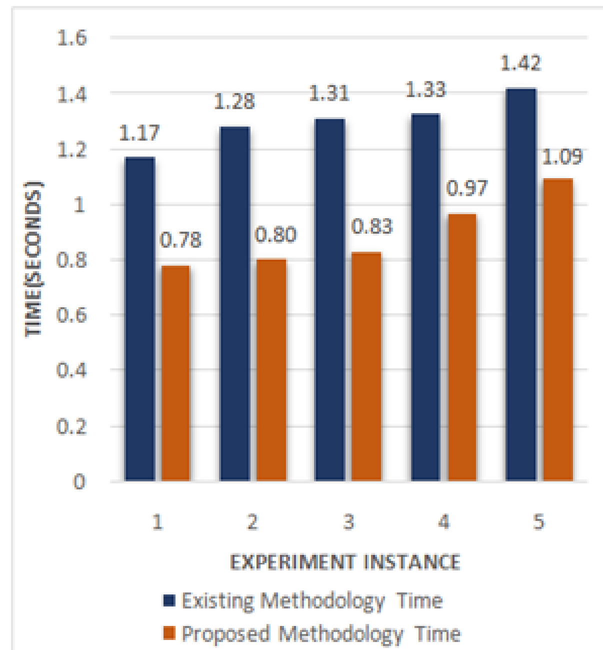

Figure 4: Feature vector generation time comparison between existing and proposed approach on Dataset -

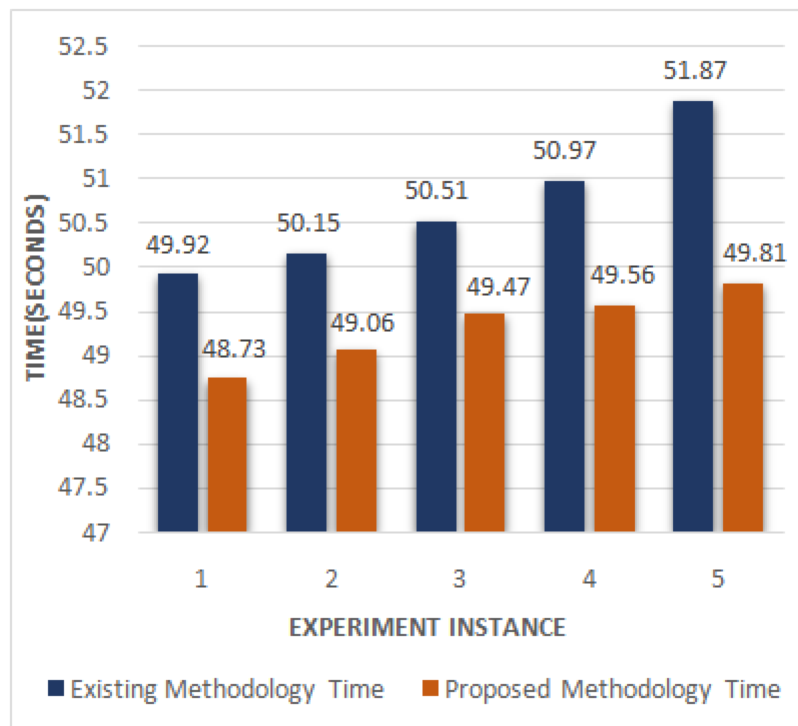

Figure 5: Total process time comparison between existing and proposed approach on Dataset - 1

For confirming the efficiency of the proposed methodology explained above, the methodology is applied to a new training dataset, collected from Contagio Mobile malware mini dump [19]. This dataset is named as **Dataset - 2**. The entire password protected APKs were extracted using a predefined password and

then arranged Malware family wise. The Dataset - 2 consist of 29 Malware families with 526 apps. The same methodology proposed above was applied to this dataset. The various results, expected, are explained as follows. The size of the feature file that had been generated using the previous methodology showed a reduction from 6,850 KB (6.69 MB) to 1,382 KB (1.35 MB) as shown in Fig. 6. The same amounts of runs, i.e. 15 runs, were performed with this dataset also. The five data values with the existing methodology took 0.69 to 0.75 seconds while that of proposed methodology took 0.36 to 0.55 seconds of the time required for feature vector generation phase as shown in Fig. 7. Next, the total time for the whole process of with the existing Methodology took around 51.56 to 52.99 seconds while that of the proposed methodology took 49.02 to 50.69 seconds only as shown in Fig. 8:

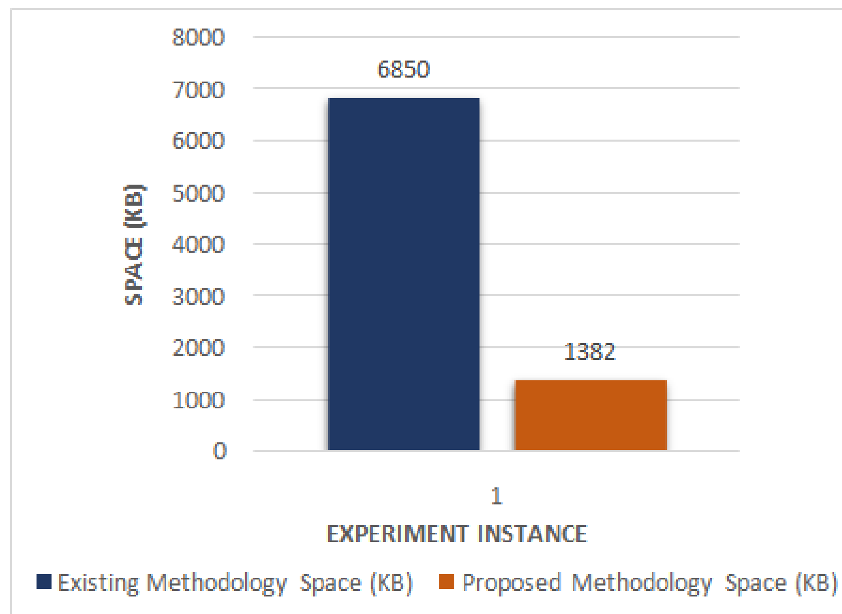

Figure 6: Disk space comparison between existing and proposed approach on Dataset - 2

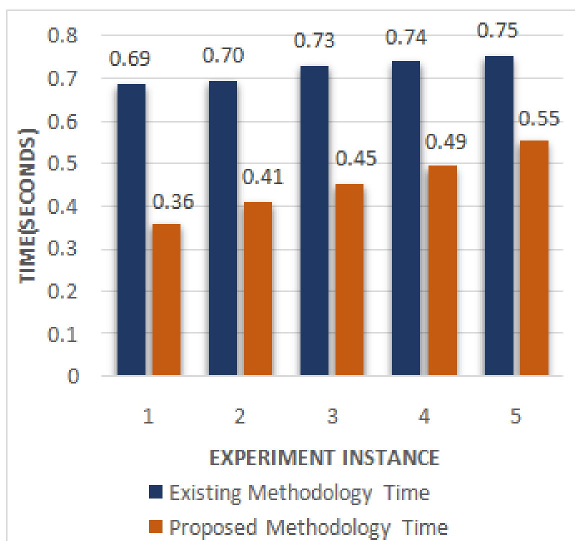

Figure 7: Feature vector generation time comparison between existing and proposed approach on Dataset -

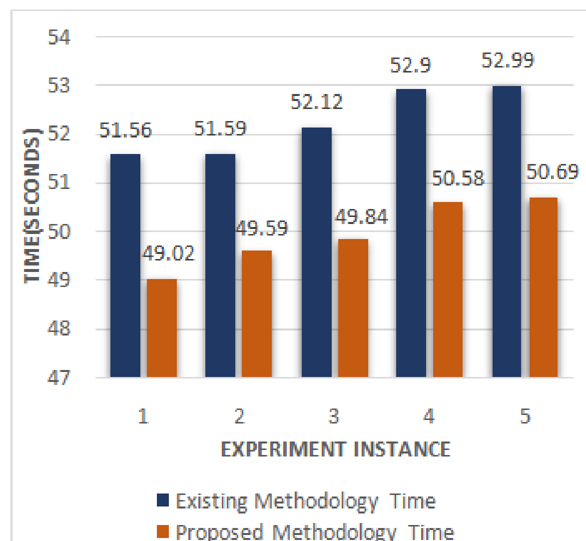

Figure 8: Total process time comparison between existing and proposed approach on Dataset - 2

Table 3: Summarized statistics of improved results of proposed methodology over existing methodology

Dataset considered	Disk Space	FFV generation time	Total process time
Dataset - 1	79.20%	31.54%	2.66%
Dataset - 2	79.82%	37.50%	4.38%

Table 3 illustrates the summarized statistics of improved results of proposed methodology over existing methodology on both the datasets. It is clear from the above results that by implementing the proposed methodology, reduction in the disk space as well as speedup in terms of time has been achieved. The removal of sparsity in the feature vector matrix considerably reduces the size of FFV file. The family feature generation time is comparatively reduced. Although the total process time does not showed much reduction, but still some speedup has been achieved. The total process time needs much improvement.

VI. Conclusions And Future Scope

In this paper, a solution to the storage of family feature vectors is presented in an efficient manner as compared to the previous methodology. The two-dimensional sparse matrix is now represented by three 1-D vectors incorporating only non-zero elements. The above proposed technique reduces the space in which feature vectors can be stored as well as the time in which feature vectors are generated. It is clear from the above results that by implementing the proposed methodology, reduction in the disk space as well as speedup in terms of time has been achieved. The removal of sparsity in the feature vector matrix considerably reduces the size of FFV file. The family feature generation time is comparatively reduced. Although the total process time does not showed much reduction, but still some speedup has been achieved. The whole process time needs much improvement. The future work comprises of the following aspects:

- The proposed methodology can be extended for classification of an unknown sample. The test dataset's FFV can be generated in this manner and then the predicted family can be decided on the basis of non-zero values and column pointers respectively.
- Measures to defeat obfuscation are not implemented. Some attackers use this art that consists of junk code insertions, string encryption so that the signature of the malware gets changed and it gets undetected by anti-malware.
- With the advent of new malware being detected, the signatures and feature vector gets changed. Thus, the database should be in synchronization with the new malware signatures. This can be achieved through communicating with various malware repositories regularly.
- Support Vector Machines (SVM) can be used for detection of unknown sample instead of the 1-NN classifier algorithm being used in the existing methodology. Once SVM classifier is trained on an imbalanced dataset, it has the capability to produce suboptimal models which are biased towards the majority class and have low performance on the minority class [20].
- Lastly, the whole methodology is not available online. A web version of this technique can be developed so that it can be accessed online where anyone can upload the sample APK to be analyzed and can get the result quickly.

References

[1] "IDC", "Smartphone OS Market Share, Q1 2015", http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] Khalid Alfalqi, Rubayyi Alghamdi and Mofareh Waqdan: Android Platform Malware Analysis, (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 6, No. 1, 2015.

[3] Android Introduction: Platform Overview, Mihail L. Sichitiu, 2011 SECURITY RESPONSE: Mobile Adware and Malware Analysis: Symantec 2013.

[4] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez and Jorge BlascoAlis: Dendroid: A Text Mining Approach to Analyzing and Classifying Code Structures in Android Malware Families. Expert Systems with Applications, Elsevier, July 2013.

[5] androguard, Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja!), https://code.google.com/p/androguard/.

[6] Cesare, S. and Xiang, Y.: Classification of malware using structured control flow. Proceedings of the eighth Australasian symposium on parallel and distributed computing (Vol. 107, pp. 61–70). Australian Computer Society, Inc, 2010.

[7] Ali Feizollah et al.: A review on feature selection in mobile malware detection, Volume 13, Elsevier, June 2015.

[8] Veelasha Moonsamy, Jia Rong and Shaowu Liu: Mining permission patterns for contrasting clean and malicious android applications, Future Generation for Computer Systems, Elsevier, 2013.

[9] Michael Spreitzenbarth, Florian Echtler and Johannes Hoffmann: Mobile-Sandbox: Having a Deeper Look into Android Applications, In SAC'13, Coimbra Portugal, ACM, 2013.

[10] Iker Burguera, Urko Zurutuza and Simin Nadjm-Tehrani: Crowdroid: Behavior-Based Malware Detection System for Android In SPSM'11, Chicago, Illinois, USA, ACM, 2011.

[11] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, M. S. Gaur and Ammar Bharmal: AndroSimilar: Robust Statistical Feature Signature for Android Malware Detection, In SIN '13 Aksaray Turkey, ACM, 2013.

[12] R.Dhaya and M. Poongodi: Detecting Software vulnerabilities using Static Analysis, IEEE ICACCCT, 2014.

[13] Samaneh Hosseini Moghaddam, Maghsood Abbaspour: Sensitivity Analysis of Static Features for Android Malware Detection, In 22nd Iranian Conference on Electrical Engineering (ICEE 2014), May 20-22, 2014

[14] Hugo Gascon, Fibian Yamaguchi, Daniel Arp and Konard Rieck: Structural Detection of Android Malware using Embedded Call Graphs, In AlSec'13. ACM, 2013
[15] Yajin Zhou and Xuxian Jiang: Dissecting Android Malware: Characterization and Evolution, 2012 IEEE Symposium on Security and Privacy.
[16] X. Jiang and Y. Zhou: Chapter 2 A Survey of Android Malware: Android Malware, Springer Briefs in Computer Science.
[17] "contagio mobile", "mobile malware mini dump", http://contagiominidump.blogspot.in/
[18] Shahadat Hossain: Data Structure for efficient storage of Sparse Matrices.
[19] Rukshan Batuwita and Vasile Palade: CLASS IMBALANCE LEARNING METHODS FOR SUPPORT VECTOR MACHINES, Copyright 2012 John Wiley & Sons, Inc.
[20] Jehyun Lee, Suyeon Lee, Heejo Lee: Screening Smartphone applications using malware family signatures, Elsevier, 2015
[21] Ugur PEHLIVAN, Nuray BALTACI, Cengiz ACARTÜRK, Nazife BAYKAL: The Analysis of Feature Selection Methods and Classification Algorithms in Permission Based Android Malware Detection, In Computational Intelligence in Cyber Security (CICS), 2014 IEEE Symposium, Dec 2014
[22] Hieu Le Thanh: Analysis of Malware Families on Android Mobiles: Detection Characteristics Recognizable by Ordinary Phone Users and How to Fix It: Journal of Information Security, 2013, 4, 213-224, October 2013.
[23] Zarni Aung and Win Zaw: Permission based Malware analysis, In INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH VOLUME 2, ISSUE 3, MARCH 2013.
[24] Egele, M., Scholte, T., Kirda, E., and Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. 44, 2, Article 6th February 2012.
[25] Parvez Faruki et al.: Android Security: A Survey of Issues, Malware Penetration and Defenses, IEEE COMMUNICATIONS SURVEYS AND TUTORIALS, VOL. 00, NO. 0, JANUARY 2015.
[26] Timothy Vidas & Nicolas Christin: Evading Android Runtime Analysis via Sandbox Detection, ASIA CCS'14, June 4–6, 2014, Kyoto, Japan.
[27] Minakshi Ramteke, Prof. Praveen Sen and Suchit Sapate: Comparative Study and a Survey on Malware Analysis Approaches for Android Devices, International Journal of Advanced Research in Computer Science and Software Engineering, Volume 4, 3rd March 2014.
[28] Chao Yang et al.: DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications, 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I, Copyright Springer
[29] Guillermo Suarez-Tangil et al.: Thwarting Obfuscated Malware via Differential Fault Analysis, Volume: 47, Issue: 6, 2014 IEEE.
[30] Muazzam Siddiqui, Morgan C. Wang and Joohan Lee: A Survey of Data Mining Techniques for Malware Detection using File Features, In ACM-SE '08, March 28-29, 2008.