Android Malware Classification Using Machine Learning and Bio-Inspired Optimisation Algorithms

Jack Pye, Biju Issac, Husnain Rafiq, Nauman Aslam
Computer and Information Sciences
Northmbria University
Newcastle upon Tyne, UK
{jack.pye, biju.issac, husnain.rafiq, nauman.aslam}@northumbria.ac.uk

***Abstract*—In recent years the number and sophistication of Android malware have increased dramatically. A prototype framework which uses static analysis methods for classification is proposed which employs two feature sets to classify Android malware, permissions declared in the *AndroidManifest.xml* and Android classes used from the *Classes.dex* file. The extracted features were then used to train a variety of machine learning algorithms including Random Forest, SGD, SVM and Neural networks. Each machine learning algorithm was subsequently optimised using optimisation algorithms, including the use of bio-inspired optimisation algorithms such as Particle Swarm Optimisation, Artificial Bee Colony optimisation (ABC), Firefly optimisation and Genetic algorithm. The prototype framework was tested and evaluated using three datasets. It achieved a good accuracy of 95.7 percent by using SVM and ABC optimisation for the CICAndMal2019 dataset, 94.9 percent accuracy (with f1-score of 96.7 percent) using Neural network for the KuafuDet dataset and 99.6 percent accuracy using an SGD classifier for the Andro-Dump dataset. The accuracy could be further improved through better feature selection.**

***Index Terms*—Android Malware Detection; Machine Learning; Optimisation; Bio-inspired optimisation;**

I. Introduction

The Android operating system remains one the most popular operating systems for malware. This is due to many factors; the Android operating system has the highest market share as compared to any other mobile operating system with a total market share of 65.7 percent as of September 2019 [1]. Moreover, unlike any other mobile operating systems such as iOS, Android also allows users to install unverified applications from third-party app stores and websites. Consequently, Android devices become more vulnerable to malicious attacks as permission-based security architecture employed by Android platform has already been proven to be ineffective for the security of the average user [2]. While Third-party app stores remain a source of malware, it has been observed that Google app store is also vulnerable to malware being by passed. In many cases malware applications have evaded *Bouncer*, an application verification tool for the Google Play Market[3]. Due to these contributing factors, the quantity and sophistication of malware has prevailed in recent years with a two-fold increase in malicious samples in last two years [4].

In this paper, a machine learning-based anti-malware solution is proposed using permissions and classes as features. This work will contribute towards ongoing research into the effectiveness of machine learning-based anti-malware software for Android operating system through the use of bio-inspired optimisation algorithms.

In summary, the main contributions of this work are: We propose a novel Android malware detection technique which employs permissions and android API based features. The proposed technique is lightweight and detects malware with high accuracy. We use multiple bio-inspired algorithms to optimise the hyper-parameters of various ML based classifiers to archive optimal classification results. We also perform comparison of different optimisation algorithms and select the best possible choice to built our framework. We use use multiple Android malware datasets to evaluate our framework. We also compare our classification results with existing state of the art solutions towards Android malware detection.

The rest of the paper is organised as follows: Related work is discussed in Section 2. Datasets and feature extraction process is explained in Section 3. The design of the malware classification framework is presented in Section 4. In section 5, we present the evaluation results and comparison with related techniques. Finally, we conclude the paper in Section 6.

II. Related Work and technology

A significant number of studies [5–11] have used the *manifest.xml* from Andord apps to extract meaningful features and further train ML-based algorthms on these features to classify between malicious and benign apps. Android *manifest.xml* contains the information about permissions requested by an app, hardware components and intents. We use permission based features from *manifest.xml* file and features from disassembled source code from an app to train our framework. Android uses a permission-based model for its security [2] and most of the naive users are unaware about the potential harm that can be caused by an app that requests irrelevant permissions [2]. [8] and [9] proposed a techniques to rank potentially harmful permissions requested by an app. DREBIN [7] also uses permission in addition to other features from *manifest.xml* to detect Android malware. Moreover, DroidMat [6] uses features from *manifest.xml* and achieve an accuracy of 97.87. MAMA [5] used permissions and hardware based features and achieved 94.83 percent. An Android application

can be reverse engineered to obtain Java source code [10]. The extracted source code can be further utilized to obtain meaningful features for classification. RiskRanker [12] used control-flow graphs and semantic features extracted from disassembled Java code and successfully detected 312 zero-day malware.

Static analysis is a process by which the program is analysed without executing any code. An Android app is stored as an APK file which a zip file with different components in it. The Applicationmanifest.xml contains information on the what the application does, how it does it, what hardware it uses and what permissions the app needs. Android uses a permission-based model for its security. When an application requires permission to perform a task, Android will allow the user to grant or deny 'dangerous' permissions during run time. An Intent allows an application component to interact with another application component in a different application, or two components within the same application using 'actions'. For a hardware feature of the phone to be used by an application, it needs to be declared in advance by the application in the AndroidManifest.xml file. Android applications are run using the Dalvik Virtual Machine which accepts Java class file and converters them into one large classes.dex.

III. Datasets and Feature extraction

In this section we provide the information of datasets being used, information about extracted features, features preprocessing and feature reduction techniques carried out in this research.

1) Datasets: In this study we use following three datasets as benchmark:

- CICInvesAndMal2019 [13]
- KuafuDet [14]
- Andro-Dumpsys [15]

Table I presents an overview of malicious and benign samples composed in each dataset. All the datasets listed in Table I have samples in form of executable Android applications (APKs). We use process of reverse engineering to extract *manifest.xml* and Java source code from the APK by using Androguard tool. Androguard is an open source python library that is capable of extracting different kinds information out of the individual components of an APK file. We use Androguard to extract permission based features from *manifest.xml* file and APIs from disassembled Java souce code. The process of feature extraction using static analysis is explained with the help of pseudo-code in Algorithm. 1.

2) Feature selection: Choosing the right features is the foundation of getting good results when training machine learning based algorithms. In this study we extract following feature classes from Android applications

- Permissions declared in the application manifest file.
- The Android classes used from the classes.dex file.

The total number of feature extracted from each application is 4830, 324 permissions and 4,506 classes, respectively. One set of features is from the application manifest file, and the other is taken from the classes.dex file. This feature-set ensures that features are taken from different components of an application,

TABLE I
Total malware and Benign samples in each dataset

Dataset	Malware	Benign	Total
CICInvesAndMal	425	1168	1593
KuafuDet	1260	360	1620
Andro-Dumpsys	907	1776	2683

3) Permissions: Declared permissions from the manifest.xml file have been extensively studied and used as features in the field of Android malware because they are highly influential in classification and can be used to produce high detection rate while keeping static analysis time to a minimum. Both classes and permissions are intrinsically linked through the permission system that the Android employs to restrict the use of dangerous Android APIs, for example, the Android permission `Android.permission.Write_SMS'` is needed for an application to use the API `'sendTextMessage()` from the Android class, `SMS_Manager`. In many applications, permissions are declared that are not even needed for the actual functionality of the application. These permission are declared in *manifest.xml* file but are not used in actual execution of the application. [16] analysed 795 applications and found that 32.7 percent of the applications were over-privileged by at least one permission. In this study, we consider declared permissions as a feature to classify malware.

Algorithm 1 Feature Extraction

Input: Android Applications Data-set D
Output: A CSV file containing binary encoded feature vectors for each App in the data-set
for $(all\ f \in D)$ **do**
  $APK_{File} \leftarrow Open(f)$
  $manifest_{File}$, $java_{Files} \leftarrow APK\_Tool(APK_{File})$
  $permissions_{(list)} \leftarrow Get\_Distinct\_Permission(manifest_{File})$
  $Classes_{(list)} \leftarrow Get\_Distinct\_Classes(java_{Files})$
for $(all\ f \in D)$ **do**
  $APK_{File} \leftarrow Open(f)$
  $manifest_{File}$, $java_{Files} \leftarrow APK\_Tool(APK_{File})$
  permissions$\leftarrow Get_Permission(manifest_{File})$
  Classes$\leftarrow Get_Classes(java_{Files})$
  for $(each\ permission \in permissions)$ **do**
    if $(permission \in Permission_{(list)})$ **then**
      $Vector_{(Permission)} \leftarrow 1$
    else
      $Vector_{(Permission)} \leftarrow 0$
  for $(each\ class \in Classes)$ **do**
    if $(class \in Classes_{(list)})$ **then**
      $Vector_{(Class)} \leftarrow 1$
    else
      $Vector_{(Class)} \leftarrow 0$
  $CSV_{(file)} \leftarrow Append(CSV_{(file)}, Concat(Vector_{(Class)}, Vector_{(Permission)}))$
return $CSV_{(file)}$

4) Android Classes: In addition to permission based features, we use features from `classes.dex` file which is

obtained after reverse engineering an APK. We extract the information about the APIs calls within the Java classes and use them as features for each APK file. APIs are set of rules that are followed while performing specific tasks e.g. sending a text message or accessing users location. Previous research has shown that malicious apps follow a specific pattern of API calls to perform a particular task and hence can be used as a strong feature to detect malware.

5) Vector space mapping: To classify Android malware, relationships between the features need to be extracted and formulated to result in a Boolean expression. However, inferring Boolean expressions from real-world applications is a hard problem, and is difficult to solve efficiently [11]. By using machine learning principles, this problem can be remedied by converting these features into a numeric vector space for machine learning algorithms to learn from. To convert the feature dependencies to vector space, binary expressions are used to indicate whether a feature was present in an application. A vector V is constructed by mapping each feature, a_i from application A to a dimension $v_i \in V$. Vector V can be represented as follows:

$$V = \{v_1, v_2, v_3, ..., v_n\} \tag{1}$$

$$v_i = \begin{cases} 1 if a_i \in A_i \\ 0 if a_i \notin A_i \end{cases} \tag{2}$$

Vector V in real terms will be represented as so, $V = \{1, 0, 0, 0, 1, 0...\}$ each 1 or 0 represents if a feature is present or not. A downside of this is that each vector is sparse of features due to only a small amount of total feature being present in each application vs the total amount of features analysed. To compensate for this [11] transform each vector again so that each dimension of the vector only represents present features, i.e. $V = \{0, 1, 0, 1, 1, 0...\}$ would be transformed into, $V = \{2, 4, 5, ...\}$ This saves a great amount of memory due to missing features not having to be represented in the vector. Due to the number of features that have been collected in this paper, this has not needed to be performed. Vector space reduction could be a consideration for future work that would potentially involve collecting a great number of features.

6) Feature reduction: Feature reduction is essential in classification because it can drastically reduce the time taken to run a classifier and produce a more general classifier by removing unwanted features. Three reduction algorithms were tested, Mutual Information, Chi2, Variance Threshold with feature Variance Threshold providing to best results. In Variance Threshold, a value from 0 to 1 is specified, 0 being deleting a feature that is the same in all samples to 1.0 deleting a feature that is different in all samples. A value to 0.1 proved to perform the best so all models were trained and optimised with with a Variance Threshold value of 0.1.

IV. Design of Malware classification Framework

In this section, we explain the main methodology employed to design our malware classification model and information about Bio-inspired optimisation algorithms used to tune the hyper-parameters of ML-based classifiers. During the Weka testing phase developed at the University of Waikato, 16 different machine learning algorithms were tested with the permissions + classes feature set. The dataset used for this initial test was the CICInvesAndMal2019 that has been made publicly available by the University of New Brunswick. The top five algorithms are: Logistic Model Trees, Simple Logistic, SGD, SMO and Random Forest.

A. Architecture of the Malware Classification Framework

We propose a prototype framework to detect Android malware. The framework is composed of four main modules as shown in Figure 1. The first step is to reverse engineer an APK and extract permissions from *manifest.xml* file and APIs from Java classes. After feature extraction, the second step is to embed the extracted feature vectors in to a feature vector space. After applying step 2, each APK in the dataset is represented as a binary feature vector where 0 means presence of a specific feature and 1 means absence of a particular feature in an application. Each app is then assigned a label which identifies it as a malware or benign sample. In third step, we apply ML algorithms based on supervised learning to train the model based on large datasets of malicious and benign applications. Finally, we apply bio-inspired optimisation algorithms to tune hyper-parameters of the machine learning-based models to increase classification accuracy.

B. Optimisation algorithms

Machine learning algorithms have hyperparameters. Hyperparameters are parameters that are set before training process is carried out that configure different aspects of the machine learning algorithm being used. Hyperparameters help to tune the level of complexity of the model if the model is too complex the model will be over-fitted and will perform very well on data that it has seen before but will perform poorly on new data. If the model's complexity is low, it fail to capture all relevant information in the data and eventually will be under-fitted. Hyperparameters serve to control the trade-off between these two aspects. Instead of using brute-force to optimise the hyperparameters, which would be a very time consuming or next to impossible task, optimisation algorithms are employed to tune hyperparameters automatically to obtain optimal results. Machine learning libraries such as Scikit-learn have default settings for hyperparameters for each instance if not provided by custom settings. However, these default hyperparameters may not be the best case for the given classification task. Random search and grid search are the easiest to be implemented. First random search is used to optimise the hyperparameters. Random search does not test all combinations of hyperparameters. Though it is not as computationally expensive it will not yield good optimal results. The hyperparameter values chosen by random search are iterated upon again using grid search. Using a smaller more refined search space around the values chosen by random search for the grid search helps to cut down the computational

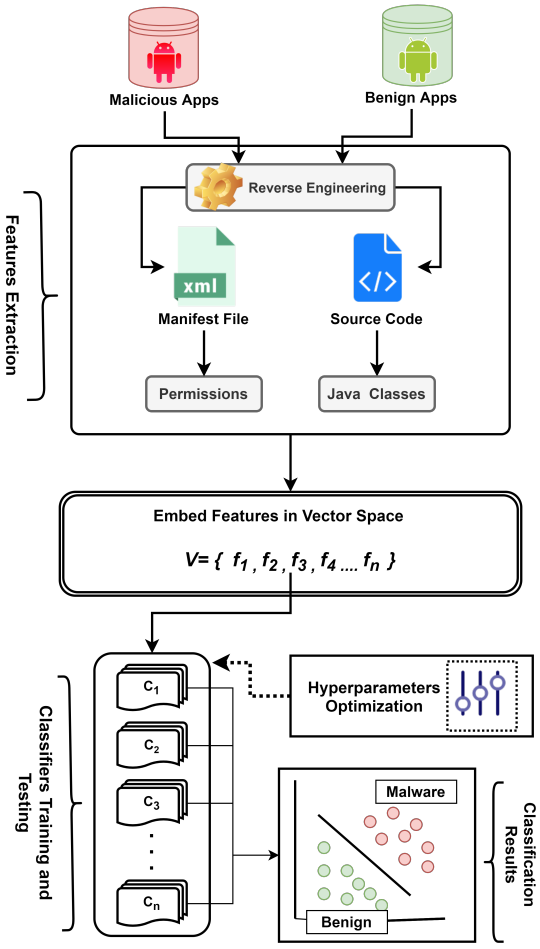

Fig. 1. Android Malware classification Framework

complexity needed to perform grid search. If grid search was used without using random search first, grid search would be too computationally expensive with a large search space, as with each value added to the search space the number of searches goes up exponentially. On each test of different hyperparameters the classifier is cross validated using a k-fold of 10 for statistical significance.

In this study, multiple optimisation algorithms are tested to find optimal settings. Bio-inspired algorithms have become quite popular for hyperparameters optimisation and in this study, we use four bio-inspired algorithms: Particle Swarm Optimization, Firefly Optimisation, Artificial Bee Colony and Genetic Algorithm. Further in this study, we perform experiments to compare the results obtained by using these optimisation algorithms against standard hyperparameter settings of ML-based algorithms. In case of Neural networks, several different optimisation techniques, Hyberband, random search and Bayesian search are used. Each optimisation algorithm tried to find the optimal value for number of layers, neurons on each layer, dropout value between each layer, learning algorithm and its respective learning rate. The baseline for evaluating the performance of the optimisation algorithms was a 4-layer ANN with 250 nodes in each hidden layer. ANNs have been chosen to be tested due to new prevalence of these types of machine learning methods being used for malware classification problems in the scientific community that boast a high classification accuracy.

The optimisation of Neural networks is done through the Python library Keras Tuner. Keras Tuner implements three optimisation algorithms, Random Search, HyperBand, and Bayesian Search. Optimisation of ANN's is looked at as a whole instead of individually optimising both 3-layer ANN's and 3-layer+ ANN's. The search space for the neural network is set up so that each optimisation algorithm can find an optimal model from a 3-9 layer search space if a deep learning approach (4+ layers) yields better results, then the optimisation algorithm can choose the optimal model appropriately. Each layer within neural network apart from the output layer has a range of nodes that can be search as well from 32 to a maximum of 1024 in each layer. In addition dropout has also been considered, each optimisation algorithm can choose to include a dropout rate between each hidden node if it yields better results.

V. Evaluation

In this section we evaluate our model using multiple ML-based algorithms with optimized hyperparameter tuning. We also perform a comparison of our model with related state of the art.

A. Metrics used

In this study we use following five metrics to evaluate our results:

1) Accuracy: Accuracy is the number of correct predictions divided by the number of total predictions.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

2) Recall: Recall measures the rate of correct classification of positive examples

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

3) Precision: Precision measures the rate of obtained results which are actually relevant.

$$Precision = \frac{TP}{TP + FP} \tag{5}$$

4) F1-Score: The F1-Score or F-Measure is a calculation of balance between precision and the recall.

$$F1 - Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{6}$$

5) AUROC: The Area Under the Receiver Operating Characteristic (AUROC) or AUC measure each classifier regardless of threshold boundaries in classification models by summarising model performance overall threshold values, a higher score indicates that the classifier is a better model even if predictive accuracy is similar. AUROC is the primary metric when using machine learning in the medical field.

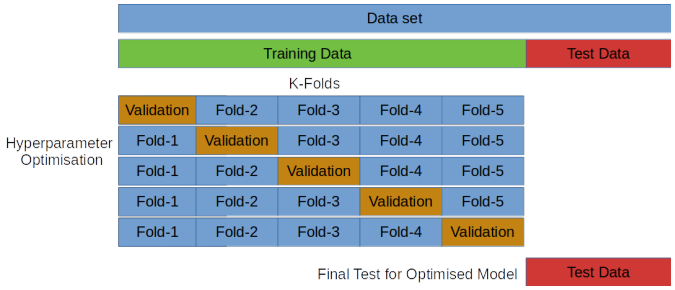

Fig. 2. Optimisation testing methodology

B. Testing validation

Each classifier without optimisation was tested using a k-fold of ten. In 10-fold cross-validation the dataset is split up into ten sections with 9 of the sections used for training and the remaining section to be used for testing. This is repeated ten times each time a new section is used for testing. Each result is then averaged to produce a final estimation, by using ten k-fold it reduces the variance of the result. This method was repeated using a variety of different feature selection methods at various thresholds.

For each classifier with optimisation, each dataset was split into a training-set and test-set. The training-set used a k-fold of 10 to tune hyperparameters, then the test-set was used to test the optimal model found by the optimisation algorithm. This was done so that each optimised model could be tested on the same test set of unseen samples for direct comparison. Fig. 2 details the testing method used for optimising each machine learning algorithm but with a k-fold of 5 for visual simplicity.

C. Detection Performance

In this section, we evaluate the performance of proposed framework. Table II shows the results of all optimisation algorithms for each machine learning algorithm on three datasets. Random search, Grid search and Bayesian search provide consistent accuracy with some increase or decrease close to the initial base metric performance. The results using Genetic optimisation, ABC, FA and PSO vary far greater than the others either resulting in improvement or decrease in performance. Our results suggest that for Andro-Dumpsys dataset, classifiers with default setting of hyperparameters performed better than the classifiers with tuned hyperparameters. The potential reason for this decrease in performance is due to a higher number of iterations needed for an optimisation algorithm to find a better model. The number of iterations that each optimisation algorithm underwent was 500. For the CICAndMal2019 dataset, the Bio-inspired algorithm ABC improved accuracy by 3.3 percent in comparison to default settings whereas 1.3 percent increase in detection was recorded in comparison to other optimisation algorithms. Our results demonstrate the potential of bio-inspired algorithms to improve the accuracy of classification problems.

D. Comparison to other frameworks

In this section, we compare our results to other relevant works. Our results suggest that the proposed technique perform as good as other state of the art appraoches such as DREBIN [7] and MAMA [5]. Moreover, our technique out performs DREBIN whilst also cutting down on features analysed for by 88.6 percent displaying that classes and permissions can give a good indication of whether an application is malware whilst keeping static analysis time to a minimum. Our best performing model trained on the Andro-dumpsys dataset achieved a remarkable accuracy of 99.6 percent. Our models trained on the CICAndMal2019 and KuafuDet dataset also achieved a high accuracy of 95.7 percent and 96.25 percent respectively. Furthermore, in this section, we compare our detection results with techniques which employ the same datasets as used in this study.

1) CICAndMal2019: As compared to our work, [13] have achieved a precision of 95.3 percent whereas in our work as in table III, SVM with ABC optimisation provides the best results with an accuracy of 95.7 percent. The features that this paper uses are similar to [13] which uses Permissions and Intents. This project has used Permissions and Classes. However, [13] get all their features from the ApplicationManifest.xml file while the features used in this project are from both the Classes.dex and the ApplicationManifest.xml.

2) KuafuDet: The technique in [14] employs Random Forest to produces the best accuracy rate at 96.25 percent, which is 1.35 percent higher than our work with the best classification accuracy of 94.9 percent (but with F1 score of 96.7 percent) through a Neural network approach. KuafuDet does, however, collect many more features than our approach. KuafuDet framework collects both syntax features and semantic features. The syntax features collected are: Permissions, Intents, Hardware, and API call. These additional features, compared to those extracted in this work could help to explain the accuracy gap however, increased analysis, more processing power required and higher memory consumption is required in KuafuDet's case .

3) Andro-Dumpsys: The technique in [15] claims to have an accuracy of over 99 percent. As compared to our results, said accuracy is almost the same as of ours. Our results suggest that all of the machine learning algorithms scored over 99 percent on the data with SGD with base hyperparameters scoring the best with 99.6 percent accuracy and 99.5 percent F1-Score as shown in table III.

VI. Conclusion and Future Work

In this study, we extract permissions and classes based features from Android application and apply ML-based algorithms to classify between malicious and benign Android applications. Moreover, we use bio-inspired optimisation algorithms to tune hyperparameter of ML-algorithms to extract optimal classification results. We evaluate our framework using three recent Android malware datasets and proved the effectiveness of our approach by conducting experiments by using multiple ML-based algorithms. We achieved 94.9

TABLE II
OPTIMISED MACHINE LEARNING ALGORITHMS ACCURACY FOR EACH DATASET

Dataset	Algorithm	Base	Random Search	Grid Search	Bayesian-1	Bayesian-2	GA	PSO	ABC	FA	Hyper -Band
CICAndMal2019	Random Forest	93.7	93.7	92.1	92.2	92.9	86.8	89.4	89.8	90.9	N/A
	SVM	92.4	94.4	92.5	94.4	91.1	94.4	94.4	95.7	95.7	N/A
	SGD	93.7	94.2	94.2	93.5	94.8	94.6	95.2	94.4	94.2	N/A
	Neural Network	94.2	95.5	N/A	95.5	N/A	N/A	N/A	N/A	N/A	94.6
KuafuDet	Random Forest	93.1	92.9	89.3	92.0	92.9	90.8	76.8	92.2	91.5	N/A
	SVM	92.6	94.0	91.6	93.3	92.4	93.5	94.0	92.0	94.0	N/A
	SGD	92.4	93.5	93.1	92.6	93.5	81.6	83.9	86.5	85.8	N/A
	Neural Network	94.9	93.8	N/A	93.1	N/A	N/A	N/A	N/A	N/A	94.2
Android-Dumpsys	Random Forest	99.2	99.3	99.5	99.1	99.1	98.3	97.4	97.9	96.6	N/A
	SVM	99.3	99.5	99.4	99.5	98.8	99.3	99.1	99.1	99.1	N/A
	SGD	99.6	98.7	98.8	98.8	99.1	98.8	98.8	92.4	87.5	N/A
	Neural Network	99.5	99.2	N/A	99.3	N/A	N/A	N/A	N/A	N/A	99.3

TABLE III
OUR RESULTS COMPARED TO OTHER PAPERS

Paper	Dataset	ML algorithm	Accuracy	F1-Score	AUROC	Recall	Precision
Taheri et al. [13]	CICAndMal2019	Random Forest	N/A	N/A	N/A	95.3	95.3
Chen et al. [14]	KuafuDet	Random Forest	96.35	N/A	N/A	N/A	N/A
Jang et al. [15]	Andro-Dumpsys	Levenshtein distanance	'over 99'	N/A	N/A	N/A	N/A
DREBIN [7]	DREBIN	SVM	93	N/A	N/A	N/A	N/A
MAMA [5]	Sanz et al.	Random Forest	94.83	N/A	0.98	N/A	N/A
Our work	CICAndMal2019	SVM + ABC Optimisation	95.7	92.0	94.5	92.0	92.0
Our work	KuafuDet	Neural Network + No Optimisation	94.9	96.7	90.3	98.8	94.7
Our work	Andro-Dumpsys	SGD + No Optimisation	99.6	99.4	99.5	99.2	99.6

percent accuracy (with F1-score of 96.7 percent) for the KuafuDet dataset by using Neural networks, 95.7 percent using SVM and ABC Optimisation for the CICAndMal2019 dataset and 99.6 percent using SGD for the Andro-Dumpsys dataset. Finally, we compared our classification results with other papers which use the same datasets. Our future plan is to scale our work by employing more feature classes and to built a adversarial evasion attacks resilient framework. We plan to incorporate Generative Adversarial Networks (GANs) with optimised hyperparameters to built a more stronger framework.

REFERENCES

[1] Kantar, "Quarterly market share held by the leading smartphone operating systems in great britain from 2013 to 2019," Statista, 2019.

[2] K. Benton, L. J. Camp, and V. Garg, "Studying the effectiveness of android application permissions requests," in *2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*.

[3] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[4] AV-TEST. (2018) Security report 2017/2018. [Online]. Available: https://www.avtest.org/fileadmin/pdf/security _report/AV-TEST_Security_Report_2017-2018.pdf

[5] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P. G. Bringas, and G. Álvarez Marañón, "Mama: Manifest analysis for malware detection in android," vol. 44, no. 6–7, 2013.

[6] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing." USA: IEEE Computer Society, 2012.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *NDSS*, 2014.

[8] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[9] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," vol. 9, no. 11, 2014.

[10] "Machine learning aided android malware classification," *Computers & Electrical Engineering*, vol. 61, pp. 266 – 274, 2017.

[11] G. Wang, D. Zhang, X. Su, and W. Li, "Mlifdect: Android malware detection based on parallel machine learning and information fusion," *Security and Communication Networks*, vol. 2017, pp. 1–14, 08 2017.

[12] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day android malware detection." New York, NY, USA: Association for Computing Machinery, 2012.

[13] L. Taheri, A. F. A. Kadir, and A. H. Lashkari, "Extensible android malware detection and family classification using network-flows and api-calls," in *2019 International Carnahan Conference on Security Technology (ICCST)*, 2019, pp. 1–8.

[14] "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," *Computers & Security*, vol. 73, pp. 326 – 344, 2018.

[15] J.-w. Jang, H. Kang, J. Woo, A. Mohaisen, and H. K. Kim, "Andro-dumpsys," vol. 58, no. C, 2016.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified." New York, NY, USA: Association for Computing Machinery, 2011.