Android Malware Family Classification: What Works – API Calls, Permissions or API Packages?

Saurabh Kumar, Debadatta Mishra, Sandeep Kumar Shukla
Indian Institute of Technology Kanpur, Kanpur, India
{skmtr,deba,sandeeps}@cse.iitk.ac.in

***Abstract*—With the increased popularity and wide adoption of Android as a mobile OS platform, it has been a major target for malware authors. Due to unprecedented rapid growth in the number, variants, and diversity of malware, detecting malware on the Android platform has become challenging. Beyond the detection of a malware, classifying the family the malware belongs to, helps security analysts to reuse malware removal techniques that is known to work for that family of malware. It takes manual analysis if a malware belongs to an unknown family. Therefore, classifying malware into exact family is important. This paper presents a technique and tool named MAPFam that applies machine learning on static features from the Manifest file and API packages to classify an Android malware into its family. This work is premised on a starting hypothesis that features extracted from API packages rather than on API calls lead to more precise classification. Our experiments indeed shows that API package based models provides ∼1.63X more accurate classification compared to an API call based method. Our machine learning based malware family classification system uses API packages, requested permissions, and other features from the Manifest files. The proposed family classification system achieves accuracy and average precision above 97% for the top 60 malware families by using only 81 features with 97.55% of model reliability rate (Kappa score). The experimental results also shows that MAPFam can perfectly identity 36 malware families.**

***Index Terms*—Android, static analysis, malware family, security, machine learning**

I. Introduction

Android dominates the smartphone market as a choice of an operating system on mobiles. Among the various reasons for its dominance, its open-source nature and the extensive support of application (App), cheaper price point are major reasons. Recently, a report shared by the IDC Corporate USA for smartphone OSes shows that in the 2nd quarter of year 2021, the share of Android was 83.8% and predicts that it will acquire 84.9% of market share by 2025 [1]. As a consequence of such large-scale adoption of Android, it has drawn the attention of Android malware authors for various cyber crimes – including account takeover, key logger or other persistent threat implantation, crypto jacking, privacy invasion etc.

A study published by the AV-TEST shows that around 3.12 million new Android malware were counted in the year 2020 which indicates that more than 8.5K new Android malware were developed each day during the year [2]. Many of these malware are slight variations of others, and belong to the same family of malware. With the rapid growth in the number, variants, and diversity of malware, it is challenging to manually analyse the malware to obtain signatures and prepare defense tactics (e.g., employ antivirus techniques) against the new malware. Automated classification of malware into different families can address the scale and dynamic nature of malware growth.

The process of auto-classification of malware into families can expedite the process of flagging the malware and keep the anti-virus defense mechanisms up to date by obtaining applicable signatures. On the other hand, if a malware can not be reliably mapped to a family, human expertise is warranted which may lead to creation of a new malware family. Existing mapping techniques use several application features to accurately classify malware into different families. In this paper, *we analyze the efficacy* of mapping techniques based on different App features—API calls, permissions and API packages (see Section IV-A). Moreover, *we propose MAPFam*, a malware family classification solution using combination of application features for better accuracy of malware to family mapping.

In the past, many [3]–[16] machine learning based Android malware *detection* systems have been developed. There are few [3], [5], [8], [10], [12], [15], [17]–[20] that classifies an Android malware into malware families. Mostly, the machine learning based Android malware detection tools utilize either static features ([3], [4], [7], [12], [19]) or dynamic features ([7]–[9]). Some machine learning based malware analysis tool (like EC2 [17]) use both types of features i.e., static and dynamic. Generally, dynamic features are obtained by executing a malware sample on an emulated device. An emulator based dynamic analysis platform generally fails to capture malicious behavior if the malware is designed to detect emulators [21]. Hence, our study mainly focuses on static analysis based malware detectors, specifically in the domain of malware family classification.

Most static malware analysis tool [3], [10], [19] extracts features from either manifest file, Dex code or both. In the past, requested permissions from the manifest file and API call information from the Dex code are widely used for Android malware detection and family classification. Requested permissions provide an overview of the capabilities that an App has whereas, API call information is used to capture actual malicious behavior.

This work is accepted at 14th International Conference on Security of Information and Networks (SIN-2021).

978-1-7281-9266-6/21/$31.00 ©2021 IEEE

In this work, we start with a hypothesis we formulated based on our experience with Android malware detection. We experimentally test the hypothesis and based on the evidences obtained – we established that the hypothesis holds. This leads to creation of a tool MAPFam – for mapping malware into their families.

The Hypothesis: The performance of API call information-based malware detectors or family classification models may be negatively impacted due to code obfuscation (use of java reflection) or obsolete API. Also, the use of API call information unnecessarily increases the size of the feature set while not contributing significantly towards malware family classification (see Section IV-A). Alternative to the API call information, system API packages can be used for malware detection and classification. An API package is a representative for a group of API calls. Whenever an API gets invoked, the corresponding API package is always referred. Furthermore, a system API package is free from the obfuscation attack, as its implementation details is not present in an APK. Therefore, using API package information in place of API calls will reduce the size of features and provides a better classification accuracy.

Testing the Hypothesis: In this paper, first we comparatively analyze the effectiveness of features like API package usage, API call information and requested permissions in classifying malware to their respective families (section IV-A). Subsequently, we design MAPFam, an Android malware family classification system that uses features from the *Manifest file* and *API packages used* to identify the malware family. In MAPFam, we first extract static features from the Android Manifest file and Dex code (API Packages), and encode them to use with machine learning algorithm. The extracted features are then passed to a feature selection module to reduce the feature set size to obtain optimal features that are most relevant to malware family identification. We evaluate MAPFam using AMD dataset [22] to show that MAPFam provides increased accuracy compared to techniques using only API calls (for restricted APIs) or requested permissions.

Overall, our contributions are as follows:

- We design MAPFam, an Android malware family classification model that uses static features from the manifest file and API packages (Section III). To design a family classification model, we develop a feature extraction and encoding module to extract features from an Android App (Section III-B). The API package information has not been evaluated/used heretofore, for android malware family classification, to the best of our knowledge.
- First, we show the effectiveness of the API package feature set against the requested permissions and API calls. API package feature set is $\sim$1.63X and $\sim$1.04X accurate compared to models that only use either APIs or requested permissions. We compute the model reliability of API-packages and it comes out to 95.83% (Section IV-A).
- Finally, we evaluate MAPFam against the known malware families with multiple classifier algorithms. MAPFam model can classify malware families with an accuracy above 97% and 97.55% of model reliability rate (Section IV-B). We also evaluate the effectiveness of MAPFam for the identification of individual malware families. The evaluation results show that MAPFam can *perfectly* identify 36 malware families out of 60 with an average precision rate of more than 97% (Section IV-C).

TABLE I
DISTRIBUTION OF MALWARE FAMILY IN THE DATASET.

ID	Family	#Samples	ID	Family	#Samples
0	airpush	7843	30	andup	44
1	dowgin	3384	31	boxer	44
2	fakeinst	2172	32	ksapp	36
3	mecor	1820	33	gorpo	32
4	youmi	1300	34	stealer	25
5	fusob	1270	35	updtkiller	24
6	kuguo	1199	36	zitmo	24
7	jisut	558	37	vidro	23
8	droidkungfu	546	38	aples	21
9	bankbot	460	39	fakedoc	21
10	rumms	402	40	fakeplayer	21
11	lotoor	329	41	ztorg	20
12	mseg	235	42	winge	19
13	boqx	215	43	penetho	18
14	minimob	203	44	cova	17
15	triada	197	45	mobiletx	17
16	kyview	175	46	fjcon	16
17	slembunk	174	47	kemoge	15
18	simplelocker	172	48	spambot	15
19	smskey	165	49	mmarketpay	14
20	gumen	145	50	svpeng	13
21	gingermaster	128	51	vmvol	13
22	leech	109	52	faketimer	12
23	nandrobox	76	53	steek	12
24	bankun	70	54	utchi	12
25	koler	69	55	fakeangry	10
26	mtk	67	56	opfake	10
27	golddream	53	57	spybubble	10
28	androrat	46	58	univert	10
29	erop	46	59	finspy	9

II. ANDROID MALWARE DATASET (AMD)

We have used the real-world malware samples from AMD [22]. AMD is the largest public dataset that contains 24,553 unique labelled malware distributed among 70 different families. Malware samples in the dataset were collected between the year 2012 to 2016. Note that, the recent labelled malware is not publicly available. In the AMD dataset, the `airpush` malware family size is the largest with 7843 unique malware whereas, the smallest size of the family contains only one sample (`roop` malware family). A sufficient amount of representative samples are needed to train and test a model for a machine learning algorithm. However, in AMD dataset, some malware family does not have enough samples. To overcome this issue, we utilize the top 60 malware families of the AMD dataset (see Table I), which accounts for 24,205 unique malware samples and have at least 9 or 10 unique malware.

III. DESIGN

This section presents an overview of MAPFam: a **M**anifest file and **A**PI **P**ackage based Android malware **Fam**ily classi-

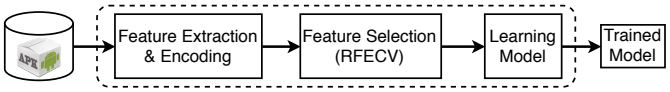


Fig. 1. Architecture of learning malware family classification model.

fication model, and elaborates the working of its core components.

A. An Overview

The main aim of this work is to develop a system that can label Android malware into its respective families without human expertise. Figure 1 shows the process of learning the Android malware family classification model. As shown in Figure 1, there are three main components of the proposed work — *(i)* feature extraction & encoding, *(ii)* feature selection, and *(iii)* learning model. In the feature extraction & encoding module, static features are extracted from an Android App. The extracted features are then encoded to get feature vectors that are given to the feature selection module. The feature selection module selects the best features that directly contributes to the family classification work. Eventually, the Learning Model module trains the final classification model on the reduced feature set as selected by the feature selection module. We describe details of each of these component in the following subsection.

B. Feature Extraction and Encoding

This module is the heart of family classification model. We extract features from two locations, i.e., *(i)* Android manifest file [23] and *(ii)* Dex code similar to Drebin [3]. However, the features extracted for this work are slightly different from Drebin [3]. In Drebin, all the features are of binary type where the value of a feature is set to one if it is present in an App otherwise zero. Whereas, in our work we use two type of features—*(i)* Numeric and *(ii)* Binary (see Table II).

Furthermore, Drebin extracts used permissions, URLs, restricted APIs and sensitive APIs from the Dex code of an App, whereas we only extract the information about the API packages used by an App. The main reason of using package information is—
(i) Generally, a package contains multiple classes of the same type. Similarly, a class is a group of methods (APIs) and related data. Therefore, package information reduces the number of features for a malware family classification model compared to the APIs based mechanism. For example, In Android API level 30, there are 4833 unique system-defined classes. At the same time, the count of system-defined unique packages is 226, which is 21.38X lesser than classes. Suppose a class contains an average of 4 methods (generally it is more than 4). Then there are in total of 19332 unique APIs available as features, which is $\sim$85.54X larger than system package based features.

Therefore, we extract API package information from the Dex code. Features extracted from the manifest file are—

```
Activities
Services
Broadcast Receivers (receivers)
Content Providers (providers)
Intent Filters (intents)
Requested Permissions
```

The above mentioned features from manifest file and Dex code have been extracted using the *Androguard* [24] and represented as strings.

1) Feature Encoding: The primary task of this sub-module is to encode the extracted features and generate the feature vector that can be passed to machine learning classifier for training and testing. To generate feature vector we opt following strategy:

- Initially, all the features in each category of feature sets are of binary type (features are represented as strings). Taking all the features in the feature set as binary will increase the feature set size because some of the features belong to a user-defined name like activities, services, etc., which can be changed from one sample to another. We can transform such features into one feature by taking their count. Such transformation reduces the feature set size drastically.
- Secondly, the features that have a predefined name like API packages and system defined requested permissions are considered to be used as a binary feature.
- Lastly, we also consider the number of API packages, total requested permissions, and custom permissions count as features. A custom permission is the permission which is defined by an App. As custom permission can take any name, hence we use number of custom permission as feature instead of binary.

Using the strategy mentioned above, this module produces two types of features, i.e., numerical and binary features. In Table II, column *Feature Type* describes the name of a feature. In the feature name, suffix C denotes that the feature is of Numeric type, whereas suffix B denotes binary feature. Similarly, sub-column *Original* shows the total number of unique features in the feature vector after the feature encoding step. Finally, we are left with the 428 unique features, which we pass to the feature selection module (see Section III-C) to find optimal features.

C. Feature Selection (RFECV):

After feature encoding, the resulting features are utilized to identify optimal features that directly contribute to the Android malware family classification task. To determine optimal features, we used RFECV (recursive feature elimination with cross validation) [25]. RFECV uses a feature ranking method and selects the best feature that contributes more in solving the desired problem. It takes a classifier C (RandomForest), a ranking function F (accuracy) and the number of features N (set to 1) to eliminate in each steps. As a result, RFECV provides a grid of score and the set of optimal features that gives highest accuracy. The grid of score provided by the

TABLE II
ENCODING SCHEME OF STATIC FEATURES EXTRACTED FROM MANIFEST FILE AND DEX CODE.

	Feature Type	#Features		Source	Encoding
		Original	Selected		
Numeric (Count)	$Activities_C$	1	1	Manifest	Number of activities
	$Services_C$	1	1	Manifest	Number of services
	$Receivers_C$	1	1	Manifest	Number of Receivers
	$Providers_C$	1	0	Manifest	Number of Providers
	$Intents_C$	1	1	Manifest	Number of intent filters
	$ReqPerm_C$	1	1	Manifest	Number of requested
	$CstPerm_C$	1	1	Manifest	Number of user defined permissions
	$Packages_C$	1	1	Dex code	Number of system API packages used
Binary	$ReqPerm_B$	261	33	Manifest	1 if a system defined permission is declared in the Android manifest file otherwise 0
	$Packages_B$	159	41	Dex code	1 if a system API package is used by an App otherwise 0
Total Features		**428**	**81**	–	–

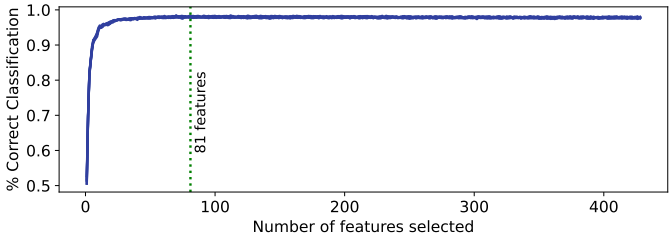


Fig. 2. Feature selection using RFECV.

RFECV is shown in Figure 2 as accuracy vs. #features graph, and the optimal number of features obtained is enlisted in sub-column *Selected* of Table II (total 81 unique features).

D. Learning Model

The final features obtained from Section III-C contains optimal features that directly contibutes to the malware family classification work. For modeling the final malware family classifier, we use ExtraTree classifiers. ExtraTree is an ensemble learning based classifier which internally uses multiple Decision Tree classifier. ExtraTree classifier takes the number of estimator for training a machine learning model which we set to 18. The final model has been trained on 70% of samples of our dataset, and the remaining 30% for evaluating it. We show the evaluation of MAPFam in Section IV.

IV. EVALUATION

To evaluate the effectiveness of MAPFam in terms of labeling a malware with its appropriate family, we have separated 30% samples (referred to as evaluation set) from the dataset (see Section II) and assumed them to be unknown malware family. The rest of the 70% malware sample (referred to as training set) are used for training in every experiment including the feature selection step (see Section III-C). This section answers the following research question to evaluate the proposed malware family labeling (classification) system effectively:

***(i)* Effectiveness of API Packages** (Section IV-A): What is the effectiveness of API packages against the requested permissions and restricted APIs?

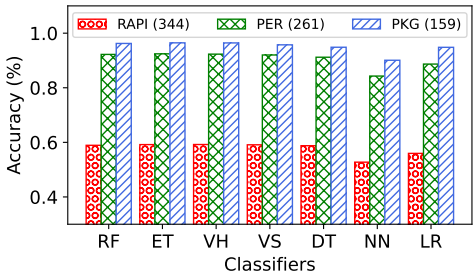
***(ii)* Performance with different classifiers against unknown malware family** (Section IV-B): How well MAPFam model (original and reduced feature set) performs to label unknown malware families by training a model with different classifiers?

***(iii)* Correctness of labeling of individual malware family** (Section IV-C): How well final model of MAPFam classifies a malware into respective family?

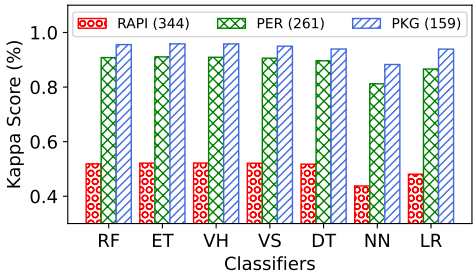
A. Performance Comparison of Features

In this experiment, we extract three categories of features from Android manifest file and Dex code—*(i)* restricted APIs (RAPI), *(ii)* requested permissions (PER), and *(iii)* API package (PKG). To compare the performance of these feature sets, we train seven different classifiers on the training set, namely *(i)* RandomForest (RF), *(ii)* ExtraTree (ET), *(iii)* Voting classifier in hard mode (VH), *(iv)* Voting classifier in soft mode, *(v)* Decision Tree, *(vi)* Neural Network, and *(vii)* Logistic Regression, and evaluates against the evaluation set. We have used three tree-based classifiers for the voting classifier (in hard and soft voting mode), namely—RandomForest, ExtraTree, and Decision Tree. Figure 3 shows the performance comparison result of different features set (i.e., RAPI, PER, and PKG) with two evaluation metrics—*(i)* Accuracy and *(ii)* Cohen's Kappa score. The accuracy represents the number of samples (in percentage) correctly classified by a classifier, whereas the Kappa score is a quantitative measure of reliability for two observers for labeling the samples. In our case, the first observer is labeled dataset (AMD) which provides actual label for malware samples, and the second observer is a classifier. In other words, the Kappa score is used to measure the reliability of a machine learning model (classifier).

Suppose we observe accuracy (see Figure 3a) for all the features set in every classifier, in that case, the API package features are more accurate in terms of labeling the malware family. API package-based ExtraTree classifier can correctly label 96.46% malware samples into the respective family, which is ∼1.63X and ∼1.04X accurate from restricted APIs

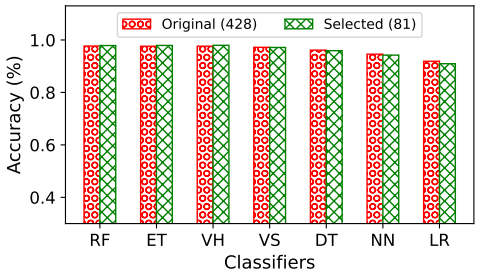


(a) Accuracy of family classification models.

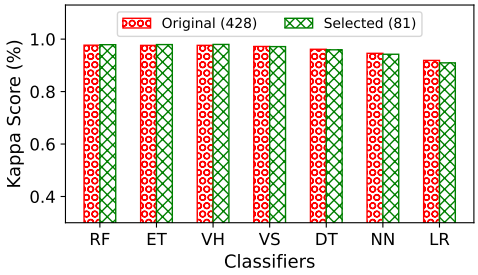


(b) Cohen's Kappa score.

Fig. 3. Accuracy and Cohen's Kappa score for the model build on different features (i) restricted APIs (RAPI), (ii) requested permission (PER), and (iii) API package (PKG).



(a) Accuracy of MAPFam model.



(b) Cohen's Kappa score.

Fig. 4. Evaluation of final model against unknown malware family with different classifiers.

and requested permissions based classifier, respectively. Similarly, when we measure the reliability of a model, in that case, the API package-based model is much more reliable with a 95.83% reliability rate, whereas the restricted APIs and requested permissions based classifier are 52.14% and 91.06% reliable, respectively (see Figure 3b). Therefore, API package-based feature set is more suitable as compared to APIs and permissions. Similarly, permissions are more reliable than APIs. Hence we select the requested permissions and API Package feature set for our family classification model.

Summary: In general, API packages based feature set is more effective and reliable as compare to API call and requested permissions.

B. Evaluation Against Unknown Malware Family with Different Classifiers

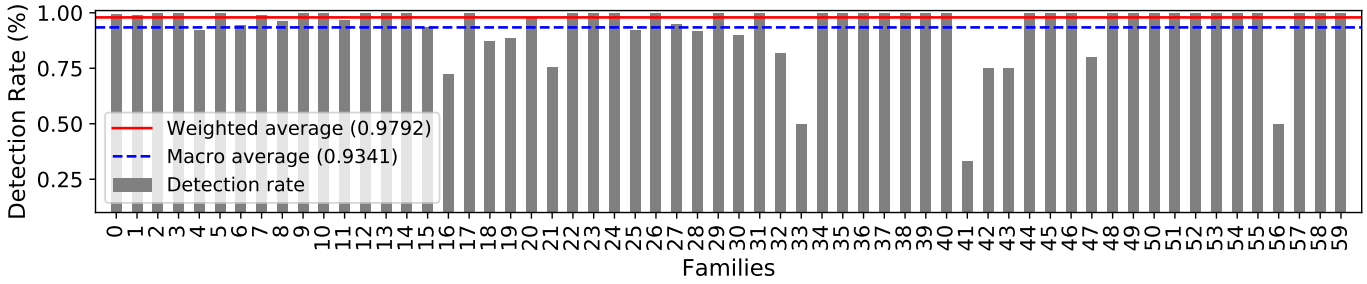
To show the effectiveness of MAPFam before and after feature selection to predict unknown malware family, we use evaluation set. For the training, we use the same seven classifiers as used in Section IV-A and train them on the training set. Figure 4 shows the accuracy and Kappa score for the identification of unknown malware family by the MAPFam. When the model is evaluated without feature selection, it correctly labels more than 91% unknown malware family (see Figure 4a). The lowest accuracy achieved is 91.89% when the classifier choice is Logistic Regression, whereas the highest accuracy achieved by MAPFam is 97.62% when classifier choice is ExtraTree. However, when the feature selection is applied on the MAPFam, it correctly classifies more than 90% sample with 90.95% lowest accuracy for Logistic Regression and 97.92% highest accuracy when the classifier is ExtraTree. From the Figure 4a, it is clear that MAPFam performance increases after feature selection when ExtraTree classifier is used for final model whereas performance decreases if the choice of classifier is Logistic Regression. The primary reason for this behavior is due to the unbalanced dataset and feature set size. ExtraTree can efficiently handle the unbalanced dataset with less number of features, whereas Logistic Regression performs well when the feature set size is large.

Similarly, when we measure the reliability of MAPFam (see kappa score in Figure 4b), best and worst reliable classifiers are ExtraTree and Logistic Regression, respectively. For the ExtraTree classifier, reliability rate goes up from 97.19% to 97.55% when feature selection is applied. However, in the case of Logistic Regression, it goes down from 90.45% to 89.33%.

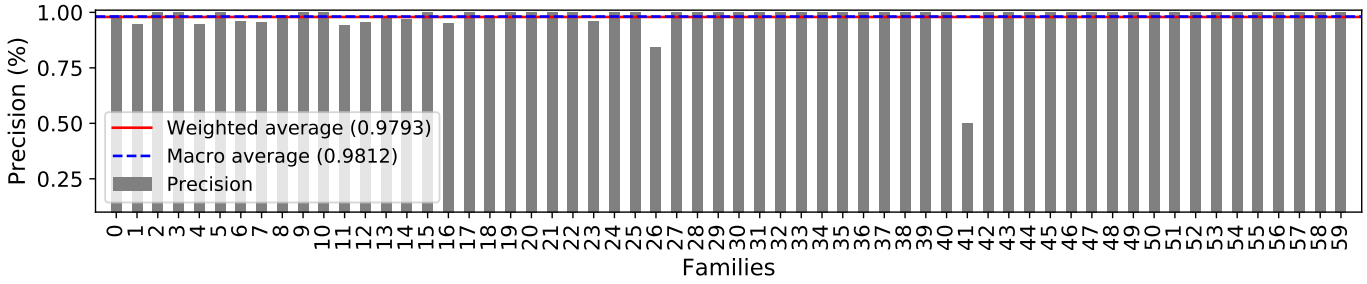
Summary: In general, MAPFam can correctly classify 97.92% of unknown malware into their respective families with 97.55% of reliability rate.

C. Detection of an Individual Malware Family

In this experiment, we analyze MAPFam performance for the classification of an individual malware family. To evaluate the model, we train ExtraTree classifier on the MAPFam feature set after performing feature selection (81 features), and the evaluation results are shown in Figure 5. Figure 5a shows the detection rate for an individual malware family along with the macro and weighted average detection. Here, the detection rate represents the percentage of samples a model can correctly classify for a single family. Evaluation results show that MAPFam can reliably detect most families with a macro and weighted average detection rate of 93.41% and 97.92%, respectively. There are only three malware families for which the detection rate is $\leq$50%, and these malware families are `gorpo` (33 with 50% detection), `ztorg` (41 with



(a) Detection rate of individual malware family.



(b) Precision for detecting individual malware family.

Fig. 5. Performance of final model for detecting individual Android malware family.

33.33% detection), and `opfake` (56 with 50% detection). Otherwise, all other malware families are reliably detected with more than 75% of detection rate. Out of 60 malware families, 36 are perfectly labeled by the MAPFam with 100% detection rate and 12 families with a detection rate of more than 90% but not 100%. Apart from the detection rate, we also measure the precision for each malware family shown in Figure 5b. Results show that MAPFam is able to detect an individual malware family precisely with macro and weighted average precision of 98.12% and 97.93%, respectively. There are only two malware families for which the precision rate is less than 85%. One is `ztorg` (41) with a precision of 50%, and the other is `mtk` (26) with 84.21% precision, rest of the malware families are precisely classified with more than 94% precision rate.

D. Discussion and Limitations

As MAPFam makes use of API-package information for Android malware family classification, which correctly classifies 97.92% of unknown malware families. However, we do not know its runtime performance when the same technique is used for family classification on a real smartphone. In future, we would like to analyze it and develop an on-device Android malware family classification system. Furthermore, we would like to evaluate MAPFam with latest labelled dataset whenever it is publicly available or by creating our own latest dataset. Even though MAPFam provides good classification results, it also has the following limitations:

(i) It cannot identify the family of malware that is encrypted.

(ii) If malware tries to load a code from an external source dynamically, MAPFam cannot predict its family. However, if code is present inside the APK, MAPFam can predict the family for such malware because we check all the Dex files bundled with an APK.

(iii) If a combination of two or more Apps are performing a malicious activity, and one App alone is provided to identify its family, in that case, MAPFam cannot predict its family. However, if the combination of Apps involved in malicious activities is provided, then MAPFam can identify its family by constructing a combined feature vector from the Apps.

V. Related Work

With the rapid growth in the numbers, variants, and diversity of Android malware, various defense system has evolved to detect and categorize malware. The existing solutions [3]–[16] uses static, dynamic, or combination of both (hybrid) techniques for malware detection. The main aim of malware detection work is to predict whether a new sample is malicious or benign. However, it does not provide information about a family to which malware belongs. If a malware family is known, then the same removal techniques can be reused that has been known to work for that family of malware, and analysts can give their attention to the new samples of unknown malware family. With this goal, several efforts ([3], [5], [8], [10], [12], [15], [17]–[20]) have been made to group similar malware into families automatically. This work is also focusing on malware family identification. Therefore, we restrict our discussion to malware family classification or characterization in the context of static and dynamic/hybrid analysis.

Static Analysis based Classification: Static analysis-based solutions extract information from an App without executing it. Existing solutions [3], [10], [12], [19], [26], [27] extract static information from the manifest file, Dex code, and

sometimes additional information from other resources like certificate, developer information, create time, and others. Alswaina et al. [26] extract information about the permission and reduce the feature set size by excluding the least important features (with zero importance) by using ExtraTree and achieving an accuracy of 95.97% for 28 malware families. Drebin [3] extracts more than 0.5 million binary features from the Dex code and manifest file, including permissions, API calls, URLs, components, and many more. Drebin took the top 20 malware families for the classification task and achieved an average accuracy of 93%.

Fan et al. [27] make use of frequent subgraphs to understand malware behavior with the help of a function call graph of sensitive API calls inside the Dex code. DroidLegacy [19] first partitions an APK into loosely coupled modules to identify piggybacked malware. After that, it compares the API call made by each module to the signature of each family. DroidLegacy has used 14 unique malware families for the evaluation where the size of family rise between 12 to 309 and achieves an accuracy of 94.03%.

AndMFC [10] utilizes requested permissions and API call information for classifying the Android malware family. In AndMFC, the feature importance method has been used to reduce the size of features. It selects the top 1000 important features while achieving more than 96% accuracy with a precision rate of 95.02%. XU et al. [12] extracts CFGs (control flow graph) and DFGs (data flow graph) and then generate a weighted graph by abstracting both of them. The combined weighted graph is then used to identify a malware family. They used top 20 malware families where family size ranges from 57 to 2753 unique malware samples and achieved an accuracy of 94.71%.

All the above work utilizes API call information in some form, whereas we use API package information to identify Android malware families. R-PackDroid [7] is the most closely related work that utilizes API package information to characterize and detect Mobile Ransomware. However, in later work [28], authors have again shifted their focus on API call information. Also, most of the work utilizes outdated dataset or operates on fewer families that contain a sufficient number of unique samples, whereas we evaluate our solution on a large number of families.

Dynamic/Hybrid Solutions: In dynamic analysis-based solutions, the behavior of an App is obtained by executing it either on an emulated platform or on an actual device. Whereas, a hybrid solution utilizes information from both the method, i.e., static and dynamic. Several efforts [5], [8], [9], [11], [17], [18] have also been made to identify malware families using dynamic/hybrid methods. Most work [5], [9], [11], [17], [18] utilizes an emulator-driven analysis framework to capture dynamic information.

EC2 [17] uses both supervised and unsupervised machine learning techniques to predict Android malware families with the ability to identify singleton families. Wang et al. [11] use all static features except for network address and restricted API used in Drebin [6] and the dynamic information captured by executing a malware sample on CuckooDroid [29]. It utilizes the top 20 malware families containing samples until 2014 and achieves an average positive rate of 98.94% (accuracy).

Similarly, Andro-Simnet [5] also uses the hybrid method and collects dynamic logs by executing a sample on emulated device. Andro-Simnet uses a malware similarity graph (a social network analysis technique) and achieves an accuracy of 97% to classify eight malware families.

All the method provides good classification results as dynamic analysis can capture actual behavior of an App. However, these techniques use an outdated dataset that does not represent the state of toady's malware which senses the execution platform. A platform-sensing malware did not show its actual behavior on finding that an execution environment is an emulated platform [21]. Recently, a work [30] has come to hide an emulated platform from a platform-sensing malware.

VI. Conclusion

In this paper, we have shown that the API package-based malware family classification model is $\sim$1.63X more accurate than the API call-based method. Later, we have developed MAPFam, a manifest file and API package-based family classification system that is capable of predicting a malware family precisely and accurately. We have performed several experiments to show the effectiveness of MAPFam to identify unknown malware families. The evaluation results showed that the MAPFam classification system can accurately classify malware families with an average precision and accuracy of more than 97% for the top 60 malware family. The MAPFam model is 97.55% reliable. We have also shown the effectiveness of MAPFam in identifying individual malware families and found that it can perfectly identify 36 malware families out of 60.

Acknowledgment

This work is partially supported by Visvesvaraya Ph.D. Fellowship grant MEITY-PHD-999, SERB and DST through C3i center and C3i hub projects at IIT Kanpur.

References

[1] IDC - smartphone market share - market share. Accessed: 30 Sep 2021. [Online]. Available: https://www.idc.com/promo/smartphone-market-share

[2] Malware statistics & trends report — av-test. Accessed: 1 October 2021. [Online]. Available: https://www.av-test.org/en/statistics/malware/

[3] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *Symposium on Network and Distributed System Security (NDSS)*. NDSS, 02 2014.

[4] A. Fatima, S. Kumar, and M. K. Dutta, "Host-server-based malware detection system for android platforms using machine learning," in *Advances in Computational Intelligence and Communication Technology*, X.-Z. Gao, S. Tiwari, M. C. Trivedi, and K. K. Mishra, Eds. Singapore: Springer Singapore, 2021, pp. 195–205.

[5] H. M. Kim, H. M. Song, J. W. Seo, and H. K. Kim, "Andro-Simnet: android malware family classification using social network analysis," in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, 2018, pp. 1–8.

[6] Li et al., "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, July 2018.

[7] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli, "R-PackDroid: api package-based characterization and detection of mobile ransomware," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1718–1723. [Online]. Available: https://doi.org/10.1145/3019612.3019793

[8] L. Massarelli, L. Aniello, C. Ciccotelli, L. Querzoni, D. Ucci, and R. Baldoni, "Android malware family classification based on resource consumption over time," in *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, 2017, pp. 31–38.

[9] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, Jan 2018.

[10] S. Türker and A. B. Can, "AndMFC: android malware family classification framework," in *2019 IEEE 30th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC Workshops)*, 2019, pp. 1–6.

[11] X. Wang, Y. Yang, and Y. Zeng, "Accurate mobile malware detection and classification in the cloud," *SpringerPlus*, vol. 4, 12 2015.

[12] Z. XU, K. Ren, and F. Song, "Android malware family classification and characterization using CFG and DFG," in *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2019, pp. 49–56.

[13] K. Xu, Y. Li, R. H. Deng, and K. Chen, "DeepRefiner: multi-layer android malware detection system applying deep neural networks," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, April 2018, pp. 473–487.

[14] S. Y. Yerima, S. Sezer, and I. Muttik, "Android malware detection using parallel machine learning classifiers," in *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, 2014, pp. 37–42.

[15] Z. Yuan, Y. Lu, and Y. Xue, "DroidDetector: Android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[16] K. Zhao, D. Zhang, X. Su, and W. Li, "Fest: A feature extraction and selection tool for android malware detection," in *2015 IEEE Symposium on Computers and Communication (ISCC)*, 2015, pp. 714–720.

[17] T. Chakraborty, F. Pierazzi, and V. S. Subrahmanian, "EC2: Ensemble clustering and classification for predicting android malware families," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 2, p. 262–277, 3 2020. [Online]. Available: https://doi.org/10.1109/TDSC.2017.2739145

[18] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, "DroidScribe: classifying android malware based on runtime behavior," in *2016 IEEE Security and Privacy Workshops (SPW)*, 2016, pp. 252–261.

[19] L. Deshotels, V. Notani, and A. Lakhotia, "DroidLegacy: automated familial classification of android malware," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, ser. PPREW'14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2556464.2556467

[20] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "DroidMiner: automated mining and characterization of fine-grained malicious behaviors in android applications," in *Computer Security - ESORICS 2014*, M. Kutyłowski and J. Vaidya, Eds. Cham: Springer International Publishing, 2014, pp. 163–182.

[21] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 447–458. [Online]. Available: https://doi.org/10.1145/2590296.2590325

[22] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, pp. 252–276.

[23] S. Kumar and S. K. Shukla, *The State of Android Security*. Singapore: Springer Singapore, 2020, pp. 17–22. [Online]. Available: https://doi.org/10.1007/978-981-15-1675-7_2

[24] S. B. Anthony Desnos, Geoffroy Gueguen. (2019, Feb) Welcome to Androguard's documentation! - androguard 3.3.5 documentation. Accessed: 10 Dec 2020. [Online]. Available: https://androguard.readthedocs.io/en/latest/

[25] (2019) SKLEARN: RFECV. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html

[26] F. Alswaina and K. Elleithy, "Android malware permission-based multi-class classification using extremely randomized trees," *IEEE Access*, vol. 6, pp. 76 217–76 227, 2018.

[27] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, 2018.

[28] M. Scalas, D. Maiorca, F. Mercaldo, C. A. Visaggio, F. Martinelli, and G. Giacinto, "On the effectiveness of system API-related information for android ransomware detection," *Computers & Security*, vol. 86, pp. 168–182, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404819301178

[29] C. S. Technologies. (2014) CuckooDroid book. Accessed: 10 Sep 2021. [Online]. Available: https://cuckoo-droid.readthedocs.io/en/latest/

[30] S. Kumar, D. Mishra, B. Panda, and S. K. Shukla, "STDNeut: neutralizing sensor, telephony system and device state information on emulated android environments," in *Cryptology and Network Security*, S. Krenn, H. Shulman, and S. Vaudenay, Eds. Cham: Springer International Publishing, 2020, pp. 85–106.