IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received November 2, 2019, accepted November 28, 2019, date of publication December 12, 2019,
date of current version December 27, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2958927

Android Malware Detection Based on Factorization Machine

CHENGLIN LI[1], KEITH MILLS[1], DI NIU[1], RUI ZHU[1], HONGWEN ZHANG[2], (Member, IEEE), AND HUSAM KINAWI[2], (Member, IEEE)
[1]Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 2R3, Canada
[2]Wedge Networks, Calgary, AB T3J 5K1, Canada

Corresponding authors: Chenglin Li (ch11@ualberta.ca) and Keith Mills (kgmills@ualberta.ca)

This work was supported by Wedge Networks Inc., MITACS, and the NSERC of Canada.

ABSTRACT As the popularity of Android smart phones has increased in recent years, so too has the number of malicious applications. Due to the potential for data theft that mobile phone users face, the detection of malware on Android devices has become an increasingly important issue for the field of cyber security. Traditional methods like signature-based routines are unable to protect users from the ever-increasing sophistication and rapid behavior changes in new types of Android malware. Therefore, a great deal of effort has been made recently to use machine learning models and methods to characterize and generalize the malicious behavior patterns of mobile apps for malware detection. In this paper, we propose a novel and highly reliable classifier for Android Malware detection based on a Factorization Machine architecture and the extraction of Android app features from manifest files and source code. Our results indicate that the numerical feature representation of an app typically results in a long and highly sparse vector and that the interactions among different features are critical to revealing malicious behavior patterns. After performing an extensive performance evaluation, our proposed method achieved a test result of 100.00% precision score on the DREBIN dataset and 99.22% precision score with only 1.10% false positive rate on the AMD dataset. These metrics match the performance of state-of-the-art machine-learning-based Android malware detection methods and several commercial antivirus engines with the benefit of training up to 50 times faster.

INDEX TERMS Android Malware detection, factorization machine, sparse representation.

I. INTRODUCTION

Smartphone usage is prevalent in our daily lives. According to surveys on global OS market shares [7], [32], Android is the visibly dominant mobile OS with a solid hold of around 75% market share across *all* mobile devices and 85.1% dominance for smartphones specifically in 2018. The rapid growth of mobile device usage, coupled with the majority market share that the Android OS enjoys have not only brought about opportunities for Android app development, but also serve to emphasize the challenge involved in defending devices from malware. According to Kaspersky's Mobile Malware Evolution Reports for 2016 through 2018 [4], [33], [34], the number of malicious installation packages amounted to 8,526,221, 5,730,916 and 5,321,142, respectively. While these numbers indicate a downward trend, one should not be fooled as the number of new Trojans targeting financial information was 128,886, 94,368, and 151,359 each year, respectively, indicating that these classes of malicious, theft-enabling software constitute a larger proportion of Android malware each year - from 1.51% in 2016 to 1.65% in 2017 to an alarming 2.84% in 2018.

To win the battle and protect mobile phone users, a number of anti-virus companies, like McAfee and Symantec, provide software products as a major defense against these kinds of threats. These products typically use a signature-based method [35] to recognize threats. Signature-based methods involve the generation of a unique signature for each previously known malware, while detection involves scanning an app to match existing signatures in a malware database. On the other hand, the heuristic-based method, introduced in the late 1990s, relies upon explicit expert rules to distinguish malware, giving rise to errors induced by human bias. In fact, both methods will be less effective if the development of the malware database or expert rules cannot keep pace with the speed at which new malware emerges and evolves.

To overcome the aforementioned problems, an alternative emerging approach is to develop intelligent malware detection techniques based on Machine Learning (ML), whose generalization capabilities include discovering unintuitive

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Esposito.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

patterns in previously undetected malware samples. One major type of machine learning-based malware detection method is called static analysis [2], [37], which can make decisions about an app without executing it in a sandbox, thus incurring a low overhead for execution. Static analysis has two phases: feature extraction and classification. In the first phase, various features such as API calls and binary strings are extracted from an original file. In the second phase, an ML model learns to automatically categorize the file sample into malware or benign-ware based on a vectorized representation of the file. For example, DroidMat [37] performs static analysis on the manifest file and the source code of an Android app to extract multiple features, including permissions, hardware resources, and API calls. It then uses k-means clustering and k nearest neighbor (k-NN) classification to detect malware. DREBIN [2] extracts similar features from the manifest file and source code of an app and uses a Support Vector Machine (SVM) for malware classification based on one-hot encoded feature vectors.

However, existing machine learning techniques for malware detection have yielded limited accuracy with high false positive rates, mainly due to the use of first-order models or linear classifiers, such as SVM [2]. These are insufficient to discover all malicious patterns. A natural idea to introduce non-linearity into malware detection is to consider the interaction between features, or in other words, *feature crossing* or *basis expansion*. For example, an app concurrently requesting both GPS and `SEND_SMS` permissions may be attempting to execute a location leakage, while the presence of either one of such requests alone does not point to any malicious behavior. However, ML models involving feature crossing are not scalable to long feature vectors.

For example, a total number of 545, 000 features are used by DREBIN [2], the SVM-based detector, which means that more than 297 billion interactions need to be considered if feature crossing was used. One could expect this number to be even larger in a more recent dataset; the Android Malware Dataset (AMD) [36], which contains more file samples thus exposing more features. Moreover, although the total number of features is large, the number of features activated by each file sample is usually much smaller, leading to a sparse vectorized representation for each individual app. This will further lead to even sparser interaction terms (the crossed terms), posing significant challenges to model training—there are not enough non-zero entries in the dataset to train the coefficient of each crossed term.

In this paper we aim to accurately model features interactions and efficiently handle long and sparse feature representations. To this end we propose a novel *Factorization Machine* (FM) model for Android malware detection. In contrast to feature crossing or basis expansion, which suffers from the model size issue and the sparsity issue mentioned above, Factorization Machines [27] aim to learn the coefficient of each interaction as the inner product of two latent vectors, thus effectively reducing the number of parameters to n - the length of the feature vector. Therefore, we hypothesize that such a factorization-based approach would not only able to accurately predict whether a Malware sample is benign or harmful as accurately as a conventional approach like a Multi-Layer Perceptron or those utilized by [2], but that it would also learn to do so much quicker, due to having far fewer synaptic parameters.

We evaluated our model on two Android malware datasets: DREBIN [2] and AMD [36], which contain 5,560 and 24,553 samples, respectively. In order to gauge performance, the metrics we utilized consisted of accuracy, false positive rate (FPR), precision, recall, F1 and last but not least, training time.

In addition, we also evaluated the performance when identifying specific families of malware, which is especially important given the growing share of banking Trojans on the internet. With respect to this task, we focused primarily on accuracy and false positive metrics.

The remainder of this paper is organized as follows: Section II reviews the background of the Android system and our feature extraction technique, while Section III describes the mathematics behind a Factorization Machine. In Section IV we elaborate on the two datasets used in this experiment, formally describe the metrics in play before stating our test results in Section IV-B1 and interpreting them in Section IV-B2. Next, we compare our test results to several several popular antivirus engines and gauge our model's detection rating with respect to specific malware families in Sections IV-C and IV-C2. Finally, we discuss future work and list a few related research projects in Sections V and VI, respectively, before concluding in Section VII.

II. ANDROID FEATURE EXTRACTION

Android applications are written in Java and executed within a custom Java Virtual Machine (JVM). Each application package is contained in a jar file with the extension of `apk`. Android applications consist of many components of various types. Each component has an entry point through which the system or a user can enter the application. In addition, there are four fundamental building blocks of an Android app: *Activities*, *Services*, *Broadcast Receivers* and *Content Providers*. All components must be declared in the application manifest file in order to be used. Communication between these components is achieved by using intents and intent filters. Intents are messaging objects that can be used to request actions from other application components while intent filters are expressions declared in the application manifest file that specify the intent type that a component will receive. Since application components interact via the intent method, it is critical to analyze both the components themselves, as well as their communication intents, for security concerns.

Before classification on any model can be done, raw data must be processed. The feature engineering section of our malware detection system consists of three parts: Unpacking and Decompiling, Feature Extraction, and Encoding – all shown below in Fig 1.

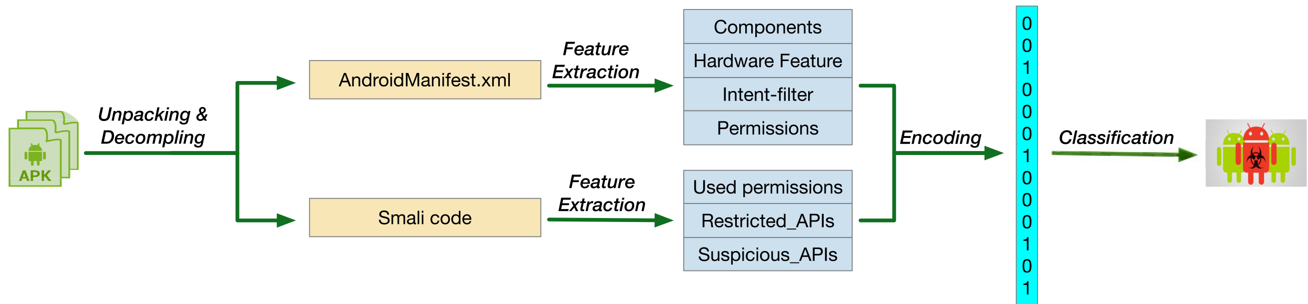


FIGURE 1. System architecture of our Malware detection model.

A. UNPACKING AND DECOMPILING

Each apk file is actually a specialized zipped file that consists of the application source code, resources, assets, and manifest file. The source code is encoded as dex files (i.e., Dalvik Executable Files) that can be interpreted by the Dalvik VM. The manifest file consists of a number of declarations and specifications. Finally, other resources may contain images, HTML files, etc.. Since the dex files are compiled, binary executable code, and therefore not meant to be read or interpreted, features cannot be readily extracted from them directly. Therefore, they must be decompiled into other formats that can be read and interpreted, such as Smali code or even Java code. Smali code is an intermediate form that is decompiled from the dex files; it is essentially the assembly code format of an application. Only after the apk files have been decompiled can we continue onto our next step.

B. FEATURE EXTRACTION

Feature extraction is one of the the most important aspects involved in the training of a machine learning model. The upper bound of a given model's performance directly depends on the nature of the features used. After performing a study of the Android system and comparing it to previous work experience in its field, we chose to extract 7 kind of features from both the source code and manifest file, of which, the following four types are extracted from a given app's manifest file:

1) **App components**: The components declared in the manifest file define the different interfaces that exist between app and end-user as well as the app and the larger Android OS as a whole. The names of these components are collected to help identify variants of well-known malware, for example the DroidKungFu family share the name of several particular services [2].
2) **Hardware features**: If an application wants to request access to the hardware components of the device, such as its camera, GPS or sensors, then those features must be declared in the manifest file. Requesting certain hardware components or pairs of components may have security implications. For example, requesting usage of the GPS and network modules may be a sign of location leakage.
3) **Permissions**: Android uses a permission mechanism to protect the privacy of users. An app must request permission to access sensitive data (e.g. SMS), system features (e.g. camera) and restricted APIs. Malware usually tends to request a specific set of permissions. In this respect, this is similar to how we handle hardware features.
4) **Intent filter**: Intent filters declared within the declaration of components in the manifest file are important tools for inter-component and inter-application communication. Intent filters define a special entry point for a component as well as the application. Intent filters can be used for eavesdropping specific intents. Malware is sensitive to a special set of system events. Thus, intent filters can serve as vital features.

Furthermore, we also extract another three types of features from the decompiled application source code (e.g., Smali code):

1) **Restricted APIs**: In the Android system, some special APIs related to sensitive data access are protected by permissions. If an app calls these APIs without requesting corresponding permissions, it may be a sign of root exploits.
2) **Suspicious APIs**: We should be aware of a special set of APIs that can lead to malicious behavior without requesting permissions. For example, cryptography functions in the Java library and some math functions need no permission to be used. However, these functions can be used by malware for code obfuscation. Thus, attention should be paid to the unusual usage of these functions. We mark these types of functions as *suspicious APIs*.
3) **Used permissions**: We first extract all API calls from the app source code, and use this to build a set of permissions that are actually used in the app by looking up a predefined dictionary that links an API to its required permission(s).

C. ENCODING

Next, we encode our extracted features into a common format that can be fed into any generic classifier. Our method uses an N-dimensional indicator to encode each application into a feature representation, where N is the feature dimension.

To be specific, suppose all the extracted features form a feature set S with size $|S|$, then we will represent each `apk` file as a binary vector of length $|S|$, whose entries are 1 only if a given feature is used by the app.

For example, suppose we have two Android applications, A and B, which each request three permissions as illustrated in Fig. 2.[1] As there are five unique permissions requested by A and B, we can then create a vector $\mathbf{x}_A, \mathbf{x}_B \in \{0, 1\}^5$ such that each entry represents exactly one permission, e.g., the first entry as a blue block represents the permission `SEND_MSG` and the second entry represents the permission `BIND_ADMIN`. As a result, we can write $\mathbf{x}_A = (1, 1, 1, 0, 0)$ and $\mathbf{x}_B = (1, 0, 0, 1, 1)$. It is straightforward to extend this idea to all kinds of extracted features as discussed in Sec. 2. The formal name for this scheme in literature is *one-hot* encoding.

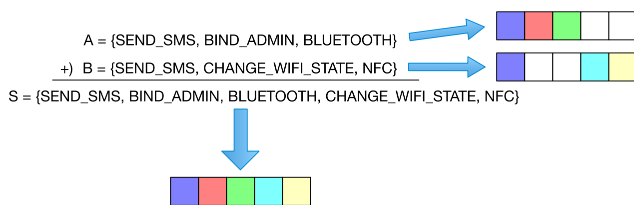


FIGURE 2. One-hot encoding for string features.

To visualize the effectiveness of our feature extraction and representation technique in distinguishing between malware and clean files, we applied a t-SNE [20] algorithm on 2,000 already encoded samples, 1,000 of which came from the DREBIN [2] dataset and while the other 1,000 were clean. The result is shown in Fig 3. Through this representation we can clearly see that the malware and clean files have formed several visibly identifiable, yet overlapping clusters, which implies the need for a non-linear classifier in order to accurately discriminate each class.

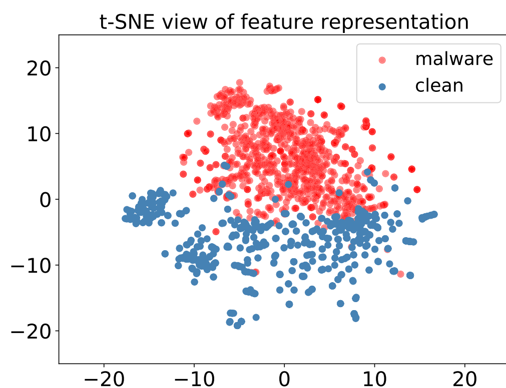


FIGURE 3. t-SNE view of 2000 samples.

[1] Here we use different color blocks to represent different feature values in the permission set, although in practice active features are represented by a '1'. White blocks mean the feature is not used (set to '0').

III. FACTORIZATION MACHINE FOR MALWARE DETECTION

Generally, a classification problem in machine learning is to infer a function $h : \mathbb{R}^n \to \mathbb{R}$ for all possible $\mathbf{x} \in \mathbb{R}^n$ to predict how much it belongs to a class. To find such a function, we are given a set of samples, each of which has been marked as a "malicious" or "benign".

In an abstract sense, at the core of this paper is the goal of achieving a high accuracy on a binary classification problem using a classifier that is not very well-known when compared to the peers its performance is contrasted against, and is quick to learn relative to said peers. Specifically, we have chosen to use a Factorization Machine [27] to meet this objective, for reasons to be explained below.

A. LIMITATIONS OF FIRST-ORDER CLASSIFIERS

Given a data sample $\vec{x}$, a typical machine learning algorithm will attempt to determine its class, $\hat{y}(\vec{x})$,[2] by learning a set of weights, $\vec{w}$, such that,

$$\hat{y}(\vec{x}) = h(\vec{x}; \vec{w}) \tag{1}$$

where h is sometimes known as the *transfer function* of the algorithm. For example, in the case of Support Vector Machines, used by DREBIN [2], h can be written as,

$$h(\vec{x}; \vec{w}) = \vec{x}^T \vec{w} + w_0 \tag{2}$$

where $\vec{w}_0$ is the intercept coefficient, technically a part of $\vec{w}$ but always multiplied by 1. Most of the basic, well-known classifiers, including Support Vector Machines, Naive Bayes (NB), and Logistic Regression, operate in a similar manner, where the input sample is compared to the learned weights allowing a class decision to then be made.

These models are not suitable for Android malware detection for two reasons: *First*, as Figure 2 implies, the feature vectors from one-hot encoding consist of ones and zeroes and are likely to be highly sparse. For example, samples in the benchmark dataset DREBIN [2] will be encoded into vectors with 93,324 entries, and on average only 73 features are non-zero, which makes weight training difficult, because a weight whose value is 55 is just as useful as one whose value is −249, in all situations, if both weights are always multiplied by 0.

Secondly, these models only exploit the First-Order information found within the features – interactions between features and weights. They do not take interactions *among* the features themselves into account. For example, going back to Figure 2 again, if a specific class of Malware can be reliably detected by checking if it requests a certain set of features (e.g. `BLUETOOTH` and `CHANGE_WIFI_STATE`) together - meaning it requests all of them not just a few - then a good starting point for a reliable classifier is one that can detect if that specific set of features are active. This is a second-order interaction - first-order classifiers such as SVMs and Naive Bayes cannot handle these automatically unless an interaction

[2] This is usually a probability (binary) or vector of class probabilities (multiclass) that are processed later.

term between these features was added previously in the feature engineering stage. The inclusion of these interaction terms requires *a priori* knowledge regarding the malware and also serve to expand the number of features.

B. SECOND-ORDER FEATURE CROSSING AND FACTORIZATION MACHINES

Typically Multi-Layer Perceptrons (MLP) and Deep Neural Networks (DNN) are the go-to solution for solving hard classification problems due to the properties they possess as universal function approximators [28], [30]. However, the number of parameters involved in training these models for a given task involve the tuning of very complex model with a large number of synaptic weights, even before feature interactions are introduced. With that in mind, consider a natural method for learning interactions of different features is through basis expansion or feature-crossing:

$$h(\mathbf{x}) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} W_{ij} x_i x_j. \tag{3}$$

By assigning a weight $W_{i,j}$ for each pair of x_i and x_j, we have the easiest way to capture pairwise interactions. However, it is not efficient here due to the large number of parameters: this model has $n(n-1)/2$ free parameters. As stated in Section III-A, the input vector for the DREBIN [2] dataset has a length of 93, 324 but the number of nonzero entries is about 73 on average. In this case, full feature crossing like W would necessitate roughly four billion weights! This brings heavy burdens on the training process since the model becomes so complicated it requires large computational resources and is very time-consuming. This problem is further compounded by the sparse nature of the data in this problem, which makes the task of applying meaningful updates to a large number of weights very difficult. Hence, MLPs are not an optimal solution to this problem.

Popular techniques to overcome the issues mentioned above and in Section III-A are low-rank or dimension reduction methods. In particular, we have chosen to use a classifier that implements feature interactions in the *learning stage* - the Factorization Machine [27] (FM), described by Figure 4.[3] More specifically, FM assumes that W is with the largest rank of k and therefore, we can decompose $W = VV^{\mathsf{T}}$.

If we denote $\mathbf{v}_i$ as the i-th row of V, FM will train a hidden vector $\mathbf{v}_i$ for each x_i and models the pairwise interaction weight w_{ij} as the inner product of the corresponding hidden vectors of entries x_i and x_j:

$$h(\vec{x}) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j, \tag{4}$$

where $\langle \cdot, \cdot \rangle$ denotes the dot product of two vectors of length k:

$$\langle \mathbf{v}_i, \mathbf{v}_j \rangle := \sum_{f=1}^{k} v_{i,f} v_{j,f}, \tag{5}$$

[3]The dark gray node stands for the inner product operator.

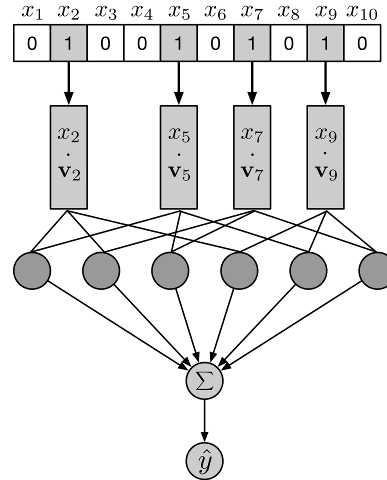


FIGURE 4. The architecture of Factorization Machine model.

The parameters that are learned during the training stage – w_0, w_i, v_i and v_j – can be updated using generic stochastic methods such as gradient descent [15], [27] with augmentations such as Adam [17]. In practice, the hyperparameter k is much smaller than the feature dimension n ($k \ll n$). Thus, the number of parameters to be estimated reduces from $O(n^2)$ to $O(nk)$.

We can further improve the performance of FM by using more sophisticated feature engineering schemes for cross terms. For example, by using "partial FM", which only involves interactions between selected features, e.g., between `Used permissions` and `Permissions`, thus ignoring crossed terms that are not relevant to malicious behavior discovery.

Let us take some toy examples to see how the relationship between two features can facilitate prediction. If an application requests the GPS hardware feature as well as the network modules permission, it is likely that this application may attempt to send geo-location information to a command & control server, therefore it is more prone to perform malicious behaviors. Another example is that some malware samples like `BaseBridge` can collect personal/device information and send it elsewhere via SMS messages. They will request two permissions, `READ_PHONE_STATE` and `SEND_SMS`, and this behavior is hardly seen in benign examples. Regardless, by encoding these permission requests as input data, a second-order classifier can learn which pair-wise interactions are indicative of malicious activity without requiring $n(n-1)/2$ free parameters.

This method of classification is naturally resistant to some of the methods used by malicious programmers to disguise their malware's purpose, such as the insertion of lines of code that are either unreachable or never called and permissions that may be requested only to never be used. This is because such methods relay on adding unnecessary information on top of what already exists, however the crucial interaction terms that are responsible for executing malicious activity still exist - those permissions are not hidden and the weight

TABLE 1. Performance metrics of Android malware detection.

Metrics	Description
TP	# of malicious apps correctly detected
TN	# of benign apps correctly classified
FP	# of false prediction as malicious
FN	# of false prediction as clean
ACC	$(TP+TN)/(TP+TN+FP+FN)$
PR	$TP/(TP+FP)$
RC	$TP/(TP+FN)$
$F1$	$2*PR*RC/(PR+RC)$
FPR	$FP/(FP+TN)$

of their interaction can still be computed separately. In fact, the addition of unused permissions serves to emphasize the use of a classifier that handles sparse inputs, such as a factorization machine, for while it may be likely that hackers would insert them into their malware code, it is unlikely that all hackers will utilize the same unused permissions, thus leading to an even sparser permission request vector.

IV. EXPERIMENTS

In this section, we evaluate the performance of our Factorization Machine-based Android malware detection system. We apply our system to malware detection tasks and malware family identification tasks, based on two public benchmark datasets: DREBIN [2] and AMD [36]. We also check our FM model against popular antivirus engines and state the logistics of our decompilation and feature extraction procedure.

A. ANDROID APK DATA

To perform this experiment, we used two public benchmark datasets: DREBIN [2], which has been mentioned previously, and AMD [36]:

- **DREBIN**: Contains 5, 560 malware files collected from August 2010 to October 2012. All malware samples are labeled as 1 of 179 malware families. This is one of the most popular benchmark datasets for Android malware detection.
- **AMD**: Known as the Android Malware Dataset, it contains 24, 553 samples that are categorized in 135 varieties among 71 malware families. This dataset consists of samples collected from 2010 to 2016. This is one of the largest, public datasets. AMD provides more recent evolution trends for Android malware when compared to DREBIN.

Further details regarding these two datasets are shown in Table 2. When doing experiments on the AMD dataset, we evaluated all 16, 753 clean files. When evaluating on the DREBIN dataset, we randomly sampled 5, 600 clean files to match the number of malware samples in this dataset. To simplify our terminologies, the DREBIN dataset (or the AMD dataset) consists of both clean samples and malware samples in the subsequent to this section. Also, from Table 2

TABLE 2. Datasets for detection performance evaluation.

Dataset	Malware	Clean files	Total	Features
DREBIN	5,560	5,600	11,160	93,324
AMD	24,553	16,753	41,306	294,019

we see that the overall feature set size grows from 93,324 to 294,019 as the dataset size grows from 11,160 to 41,306.

We also collected a number of real-world Android applications from the internet. Resources of these files include `apkpure` [1] with 5,400 samples, 700 samples from `360.com` and 13K commercial applications from the HKUST Wake Lock Misuse Detection Project [19]. In total, we have 19, 100 real-world applications. Then, we uploaded all these files to VirusTotal, a public anti-virus service with 78 popular engines, and inspected scanning reports for each file as a check to ensure that they were truly clean files. Each engine in VirusTotal would show one of three detection results: `True` for "malicious", `False` for "clean", and `NK` for "not known", respectively. If an application had more than one `True` result, we labeled it as `malware`; otherwise, we considered it as `clean`. Thus, only 16, 753 out of 19K collected samples are labeled as `clean`, and we will only use these samples in further experiments.

In our system, we used APKtool [6] to decompile the source code into Smali code and extract information from the `AndroidManifest.xml` file. We found this procedure to be quite time-consuming. However, for different applications this would often take a fixed processing time due to the fixed feature space size. Therefore, we focused on evaluating the processing time for unpacking, decompiling and feature extraction, then give out an average processing time for all applications on the encoding and prediction phase.

B. CLASSIFIER EVALUATION

We first evaluated our proposed FM-based method and compared it with other existing baseline algorithms, including SVM with a Logistic Kernel, which is used in DREBIN [2] to achieve an accuracy (ACC) score of 93.9%, classical machine learning algorithms such as Naive Bayes using Gaussian, Bernoulli and Multinomial Kernels [37] and shallow, one-hidden layer neural networks [28].[4] In addition, we also sent all samples, including malware samples, to the VirusTotal service and compared it with commercial anti-virus engines.

The dataset was randomly split into training (80%) and testing (20%) sets for both experiments in accordance with the *pareto principle*. All models were trained using stratified 5-fold cross validation for hyper parameter tuning and then tested for performance evaluation. The hyperparameters turned for baseline algorithms and our proposed method were

[4] Throughout this paper, we use some abbreviations to denote these baseline algorithms for figures and tables. In particular, the name of Algorithm "NB-Bernoulli" (NB-B) refers to the Naive Bayes classifier using Bernoulli kernel, and the same for "NB-Multinomial" (NB-M) and "NB-Gaussian" (NB-G).

TABLE 3. DREBIN test results.

ALG	SVM	NB-G	NB-B	NB-M	MLP	FM
ACC	95.65	97.72	94.71	94.71	**99.73**	99.46
PR	96.34	96.25	90.25	90.86	99.91	**100.00**
RC	94.87	99.28	**99.82**	99.37	99.55	98.92
F1	95.60	97.74	94.95	94.93	**99.73**	99.46
FPR	3.57	3.84	10.35	9.90	0.09	**0.00**

TABLE 4. AMD test results.

ALG	SVM	NB-G	NB-B	NB-M	MLP	FM
ACC	92.69	93.53	90.42	86.24	**99.05**	**99.05**
PR	93.87	90.39	86.35	81.35	**99.24**	99.22
RC	93.65	99.56	**99.60**	99.32	99.14	99.16
F1	93.76	94.75	92.51	89.44	**99.19**	**99.19**
FPR	8.69	15.04	22.36	32.36	**1.07**	1.10

trained and tested in the same manner. Finally, we recorded the time it took to re-train the best model chosen by cross-validation over the entire training set. With exception to our FM, all models were trained using Sci-kit Learn's [25] API while we used Polylearn [22] to train our FM. All experiments were done on a computer equipped with a stock Intel Core i9-9900X processor with 128GB of RAM running Ubuntu 18.04 LTS.

Moreover, the metrics we utilized for performance evaluation - all expressed as percentages - are listed in Table 1. Specifically, we focused on *precision* (PR), *recall* (RC), *F1* and *False Positive Rate* (FPR).[5] The Factorization Machine, Multi-Layer Perceptron and Naive Bayes models all produce *probabilities* that a given sample is malware. If this probability was greater than a certain threshold, 0.5 in this experiment unless otherwise stated, it was classified as malware for the purposes of cross-validation and out-of-sample test results.

1) HYPERPARAMETER SPECIFICATIONS

The best hyperparameters for each algorithm we used – except for Naive Bayes which had no hyperparameters for cross validation to tune – are as follows: SVM with a Logistic Kernel preferred a penalty of 1 and kernel coefficient, γ of 5e^{-5} when the malware and benign classes were weighted equally. The best MLP performance was found by updating the weights in accordance with the Adam [17] learning rule, the ReLU [21] activation function, with hidden layers that consisted of 150 and 200 neurons for the DREBIN and AMD sets, respectively with a batch size of 200. For our FM models we found a value of $k = 10$ to be the most optimal. Finally, both the MLP and FM models were trained for 200 epochs.

[5]Note that in the literature, recall and false positive rate corresponds to malware detection rate and false alarm rate for the detection system.

2) INTERPRETATION

Next, Tables 3, 4 and 5 illustrate the effectiveness of our FM classifier with respect to SVM with a Logistic Kernel and Gaussian Naive Bayes and especially an MLP. From the first two figures it is clear that the most relevant algorithms are the Mutli-Layer Perceptron and the Factorization Machine. Both classifiers are neck-and-neck with each other; on the DREBIN set the MLP edges out the FM in terms of accuracy with a score of 99.73% to 99.46%, however the FM achieved perfect precision and false-positive scores of 100% and 0%, respectively. The situation is even tighter when looking at the AMD set, with both classifiers achieving matching accuracy and F1 scores of 99.05% and 99.19%, again, respectively. ROC Curves, Figures 5 and 6 show that FMs pulled ahead of MLPs slightly in terms of area-under-curve (AUC).

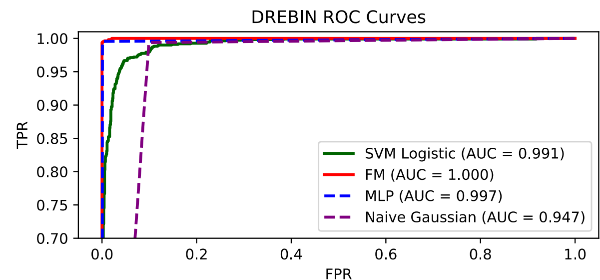


FIGURE 5. ROC curves for high-performing algorithms on the DREBIN set.

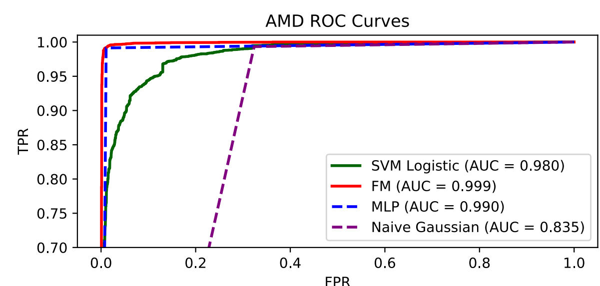


FIGURE 6. ROC curves for high-performing algorithms on the AMD set.

Our Factorization Machine pulled ahead of Support Vector Machines with a Logistic Kernel, verifying our assertion that interaction terms are important for revealing malicious behaviour patterns. However, they also cast doubt on our additional assertion that the large number of adjustable, synaptic weights used by an MLP would make it hard to train

TABLE 5. Out-of-sample training times; H:MM:SS format.

ALG	SVM	NB-G	NB-B	NB-M	MLP	FM
DREBIN	0:01:40	0:00:13	12ms	9ms	0:09:01	0:00:35
AMD	0:21:41	0:02:38	54ms	34ms	2:13:42	0:02:40

given the sparse data involved. As a universal approximator [13], [30], MLPs used the same datasets to produce top-notch results at the *cost* of taking much more time to train a classifier.

Taking a look at Table 5, we can see that our Factorization Machines trained much faster than our single hidden-layer MLPs. Specifically, the FM trained **15** times faster than the MLP on DREBIN and **50** times faster on AMD. This advantage cannot be understated, especially when we consider how much quicker the FM was to train on the AMD set, which was more difficult across the board to get an accuracy score above 99%. This emphasis is compounded by the fact that AMD is newer, larger, more recent and therefore relevant than the DREBIN data. Also, note that the MLPs used very basic – wide and shallow – and that their training time would only be increased with the additional parameters added by stacking more layers. In short, when compared to Multi-Layer Perceptrons, Factorization Machines trade universal approximation for second-order interactions while striking a balance between time, accuracy, and complexity.

C. COMMERCIAL ENGINES AND FAMILY DETECTION

We also compared the performance of our malware detection algorithm with existing commercial Anti-Virus engines on VirusTotal [31]. The critical point to mention is that all of the *truly* clean files used in our experiments are actually labeled by these AV engines using the rule described in subsection IV-A. Therefore, AV engines are supposed to have a better false positive rate than their normal performance. Tables 6 and 7 summarize the scanning results of the best performing and popular commercial AV engines on `VirusTotal`, such as `Kaspersky`, `Cylance` and `McAfee` on the DREBIN [2] and AMD [36] sets.

TABLE 6. Test performance of VirusTotal – DREBIN.

Scanner	PR	RC	F1	FPR
McAfee	**99.91**	98.74	99.32	**0.089**
CAT-QuickHeal	99.64	99.46	99.55	0.357
Symantec	**99.91**	99.28	**99.59**	**0.089**
Kaspersky	99.63	97.21	98.41	0.357
Cylance	50.09	**99.91**	66.73	98.66
Qihoo-360	97.78	94.96	96.35	2.141

TABLE 7. Test performance of VirusTotal – AMD.

Scanner	PR	RC	F1	FPR
McAfee	99.73	93.82	96.69	0.358
CAT-QuickHeal	99.70	98.84	**99.27**	0.418
Symantec	99.57	67.26	80.29	0.42
Kaspersky	**99.84**	53.35	69.54	**0.119**
Cylance	58.86	**99.64**	74.00	98.96
Qihoo-360	97.50	68.92	80.76	2.507

To show our model's capacity to distinguish one malware family from other families as well as clean files, we determined whether each input sample belonged to a specific malware family. Here we regard clean files as a special family named "clean". We further evaluated our Factorization Machine model for this task on the AMD dataset. Specifically, we used all samples from the 7 largest malware families in the AMD dataset as well as 1, 500 clean samples.

1) COMPARISON WITH COMMERCIAL ENGINES

We can compare our test results in Tables 3 and 4 with existing commercial Anti-Virus engines available on VirusTotal [31] in Tables 6 and 7. Our results are competitive with the most popular engines listed in the latter two tables, however our most accurate classifiers did fall a little short of the best classifiers available – `McAfee`, `Symantec` and `Cylance` for DREBIN, `Kasperskey` and `Cylance` for AMD. Although we did not record the accuracy of these AV engines, we can see that precision, a metric that measures how many times a classifier accurately deemed a sample to be malware, recall, a measure of how many malware samples the classifier detected in total, and F1, the harmonic mean of both, only fell below 99% for FM on the DREBIN set – all other times the FM and MLP scores were over 99%. The critical point to mention is that all of the *truly* clean files used in our experiments are actually labeled by these AV engines using the rule described in Subsection IV-A. Therefore, AV engines are supposed to have a better false positive rate than their normal performance.

2) DETECTION OF SPECIFIC MALWARE FAMILIES

Finally, using Table 8, we can see that the easiest malware to detect was the *Mecor* family, which is Trojan Spyware [36], while the hardest to detect was *Youmi* – Adware. The brand of malware that was detected with the smallest FPR was *FakeInstaller*, which is a Trojan that wrecks havoc on the device's SMS services. Out of the families listed in in Table 8, the one that is potentially very dangerous yet did not receive one of the highest scores was *Fusob* [36], ransomware which can lock down the device until certain conditions, which usually involve monetary payment to the hacker in question, are met. However, our FM design scored over 99% in the fields of

TABLE 8. Malware family classification by FM – AMD.

Family	Samples	PR	RC	F1	FPR
Airpush	7606	99.54	99.72	99.63	0.17
Youmi	1256	*97.53*	*98.75*	*98.14*	0.09
Mecor	1762	**99.77**	**99.89**	**99.83**	0.11
FakeInstaller	2129	99.57	99.57	99.57	**0.05**
Fusob	1238	99.68	99.52	99.60	*0.48*
Kuguo	1122	99.64	99.82	99.73	0.18
Dowgin	3298	98.52	99.63	99.07	0.07
Clean	1500	95.58	97.30	96.43	0.22
Average	—	98.73	99.27	99.00	0.17

precision, recall and F1 on this scarce family of 1, 238 entries. Ironically, our classifier had a harder time deeming apk files to be clean than it did detecting any brand of malware, but as the saying goes better safe than sorry.

D. FEATURE PROCESSING OVERHEAD

Now we evaluate pre-processing time which consists of decompiling the apk files to Smali code and then extracting the features listed in Section II. All work relating to this subsection was done on a virtual machine hosted on ESXi. The VM was running Ubuntu 16.04 with a memory of 4G and 2 CPUs. For this task we randomly sampled 3, 794 AMD samples, 6, 120 clean files and all 5, 560 DREBIN samples. Histograms for dex code size and processing time for all 15, 474 samples are given in Figures 7 and 8, respectively. The results of processing time vs. size-on-disk are shown in Figure 9; the three figures in the first row show the relation between dex source code size and processing time. The figures in the second row show the relation between apk file size and processing time.

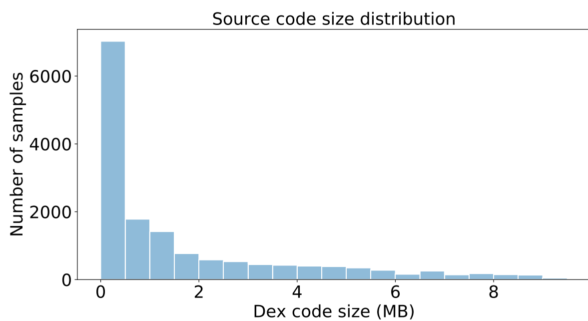


FIGURE 7. Dex source code size distribution.

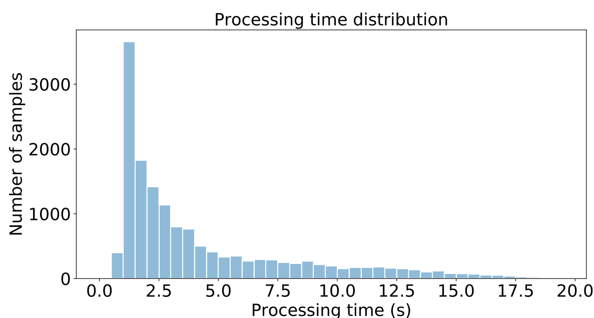


FIGURE 8. Processing time distribution.

1) DISCUSSION OF RESULTS

With respect to apk file decompiling and feature extraction, Figures 7, 8 show us that over 78% of samples have a dex code size of less than 3*MB* and over 70.6% of samples have a processing time of less than 5 seconds. On the same samples we also measured the mean time for encoding and prediction. Figure 9 tells us that the relationship between processing time and dex code size is almost linear and that for the samples in all three datasets the slopes are in the neighborhood of 0.4. Conversely, the three top graphs of Figure 9 indicate no fixed relation between apk file size and processing time, but rather that the relationship between apk size and processing time has a rough upper and lower bound. It is likely that this decoupling of processing time between apk file size and dex code size is because in addition to the dex code and manifest file, an apk file also contains other resource files like HTML, figures, which can vary wildly from app-to-app.

Compared with DREBIN [2], it seems that our system does not have much of an advantage in processing time. However, this is not the case. To begin with, the test is done on a system that is not fully integrated, the output of Smalisca is first written into a json file and then reload into RAM for further processing. The I/O between RAM and flash storage would often take a long time. Secondly, the feature sets used in our system are simpler and smaller than sets used in DREBIN, so under same condition our system should take less processing time than DREBIN.

V. LIMITATIONS AND FUTURE WORK

While machine learning techniques such as ours provide a powerful tool for automatically inferring models, they require a representative dataset for training. That is, the quality of the detection model depends on the availability, quality and quantity of both malware and benign applications. While it is straightforward to collect benign applications, gathering recent malware samples is a non-trivial task that requires some technical effort. Fortunately, offline analysis methods, e.g. RiskRanker [12], can help to acquire malware and provide the samples for updating and maintaining a representative dataset in order to continuously update our model.

Outside of model training times, the major limitation of our architecture is the decompilation and feature extraction process. We plan to integrate our system into Wedge Networks' in-line, real-time security solution which only allows us to have millisecond-scale processing time. For encoding and prediction our system takes about 4.8ms on average, however, decompiling and feature extraction is on the order of seconds. Fortunately, methods exist that will allow us to improve our system's time efficiency. Such techniques include reducing I/O and finishing all work at once on a computer with a large amount of main memory (RAM), or even using Application-Specific Integrated Circuits (ASIC) such as FPGAs for speed up. In addition, we noted that decompiling apk files can fail when using some existing tools.

Finally, in our experiments, we observed failures for some files, more with malware samples than clean files. This is expected as malware samples may use some additional techniques such as code obfuscation [5] that may lead to decompiling failures. While this limits the effectiveness of Android malware detection schemes that extract features from apk files, it is a fault of the decompiling tools available, is tangential to the main topic of this paper, but may also serve as an avenue for future research.

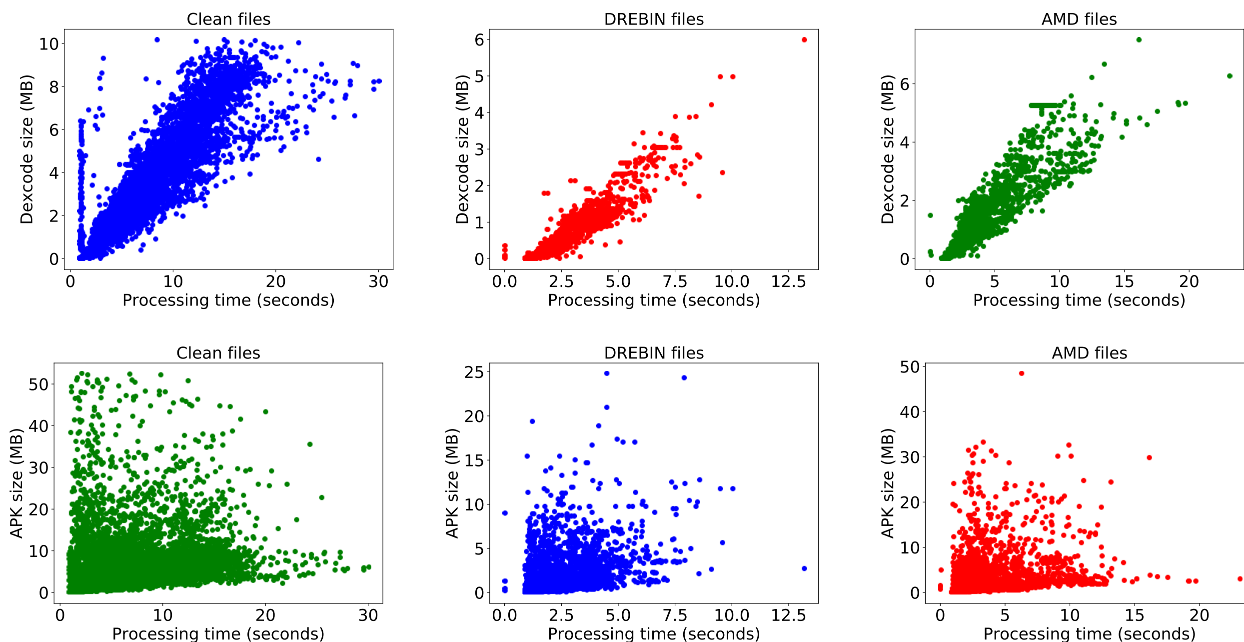


FIGURE 9. **Scatter plot of processing time vs. file size.**

VI. RELATED WORK

Many recent papers have tried to find malicious behavior patterns through control flow graphs or call graphs, although these can be obfuscated by "method overloading" [5]. AppContext [38] classifies applications using machine learning based on the contexts that trigger security-sensitive behaviors. It builds a call graph from an application source code and extracts the context factors through information flow analysis. It is then able to obtain the features for the machine learning algorithms from the extracted context. In that paper, 633 benign applications from the Google Play store and 202 malicious samples were analyzed. AppContext correctly identified 192 of the malware applications with an accuracy of 87.7%. [11] also utilized call graphs to detect malware. After extraction of call graphs from Android applications, a linear-time graph kernel was applied in order to map call graphs to features. These features were then given as input to SVMs to distinguish between benign and malicious applications. They conducted experiments on 135, 792 benign and 12, 158 malware applications, detecting 89% of the malware with an FPR of 1%. This kind of method relies heavily on the accuracy of call graph extraction. However, current works like FlowDroid [3] and IC3 [23] cannot fully solve the construction of Inter-component control flow graphs (ICFG), especially the inter-component links involving intents and intent filters.

Other works focus on the detection of specific malicious behavior such as privacy breaches and over privilege usage. For example, [16] goes through the source code with predefined sources and sinks to find potential privacy breaches. [9] further examines all the URL addresses to see if the app is trying to steal users' private information. Fuchs *et al.* [10] uses data flow analysis for security certification. However, static taint-analysis and over privilege are prone to false positives.

Studies closer to the one performed in this paper, such as [14], [26] try to directly classify an application as malicious or benign through permission request analysis for application installation [8], or control flow analysis [18]. These works take different approaches in both the feature extraction and the classification phase. [26] used permissions and API calls as features for SVM and Decision Tree Ensemble classifiers. Hindroid [14] built a structured heterogeneous information network (HIN) with an Android application and related system APIs as nodes and their rich relationships as links, and then used meta-paths for malware detection. DREBIN [2] extracted features from manifest files and source code, including permissions, hardware, system API calls and even all the URLs, and then used an SVM as the final classifier for malware detection. However, DREBIN only achieved a test detection rate of 93.90% on their full dataset. Sahs and Khan [29] used a one-class SVM with kernels and as general classifier, but only used 2,081 benign and 91 malicious applications. Next, [39] used Deep Neural Networks for malware detection, but they restricted themselves to only 500 `apk` files and achieved an test accuracy of 96.5%. Finally, in an extensive study that tested the ability of past malware's effectiveness at training classifiers to deal with new malware, [24] used a mix of call graphs and Markov Chains for prediction. However, their metrics focused on the F-measure[6] and their results greatly varied depending on across different sets of data. By contrast the accuracy scores of our single hidden-layer MLP and FM models, were each above 99% on both datasets.

[6]Generalization of the F1 score.

We believe one of the reasons behind these high scores is that we differentiated ourselves from existing works and instead of only focusing on feature engineering and ignoring the importance of choosing a suitable algorithm, after acquiring the feature representations of apps, we first made two critical observation regarding the interaction between features and the sparsity of the feature vectors. Then, the optimum machine learning algorithm designed to appropriately handle our problem was chosen for malware detection based on those observations and assumptions.

VII. CONCLUSION

In this paper, we raised the issue of considering interaction terms across features for the discovery of malicious behavior patterns in Android applications. The features used to represent an `apk` file consisted of app components, hardware features, permissions and intent filters from the manifest file, as well as restricted APIs, suspicious APIs and used permissions from source code. Based on the extracted features, a highly sparse vector representation was constructed for each application using one-hot encoding. We then proposed the use of a Factorization Machine-based malware detection system to handle the high sparsity of vector representation and model interaction terms at the same time.

To the best of our knowledge, this is a first for using FM models for malware detection. A comprehensive experimental study on two real sample malware collections, the DREBIN and AMD datasets, alongside clean applications collected from online app stores were performed to show the effectiveness of our system on malware detection and malware family identification tasks. Promising experimental results with accuracy, precision, recall and F1 scores of around or above 99% demonstrated that our method matches the performance of commercial antivirus engines and holds steady against the incredible results produced by Multi-Layer Perceptrons with the benefit of taking up to 50 times less time to train.

REFERENCES

[1] (2018). *Apkpure*. Accessed: May 9, 2018. [Online]. Available: https://apkpure.com/
[2] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.
[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[4] R. Unuchek. (2017). *Mobile Malware Evolution 2016*. [Online]. Available: https://securelist.com/mobile-malware-evolution-2016/77681/
[5] M. Chua and V. Balachandran, "Effectiveness of Android obfuscation on evading anti-malware," in *Proc. 8th ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, 2018, pp. 143–145.
[6] C. Tumbleson and R. Wiśniewski. (2018). *Apktool*. Accessed: May 9, 2018. [Online]. Available: https://ibotpeaches.github.io/Apktool/
[7] International Data Corporation. (2018). *Smartphone Market Share*. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os
[8] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proc. 8th Symp. Usable Privacy Secur.*, 2012, p. 3.
[9] H. Fu, Z. Zheng, S. Bose, M. Bishop, and P. Mohapatra, "LeakSemantic: Identifying abnormal sensitive network transmissions in mobile applications," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2017, pp. 1–9.
[10] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated security certification of Android," Dept. Comput. Sci., Univ. Maryland, College Park, MD, USA, Tech. Rep. CS-TR-4991, Nov. 2009.
[11] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proc. ACM Workshop Artif. Intell. Secur.*, 2013, pp. 45–54.
[12] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Services*, 2012, pp. 281–294.
[13] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Netw.*, vol. 2, no. 5, pp. 359–366, 1989.
[14] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "HinDroid: An intelligent Android Malware detection system based on structured heterogeneous information network," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2017, pp. 1507–1515.
[15] Y. Hu, D. Niu, and J. Yang, "A fast linear computational framework for user action prediction in tencent MyApp," in *Proc. 27th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, 2018, pp. 2047–2055. [Online]. Available: http://doi.acm.org/10.1145/3269206.3272015
[16] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "ScanDal: Static analyzer for detecting privacy leaks in Android applications," *MoST*, vol. 12, no. 1, pp. 1–10, 2012.
[17] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent.*, Dec. 2014, pp. 1–15.
[18] S. Liang, W. Sun, and M. Might, "Fast flow analysis with godel hashes," in *Proc. IEEE 14th Int. Working Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2014, pp. 225–234.
[19] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for Android applications," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 396–409.
[20] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, pp. 2579–2605, Nov. 2008.
[21] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, 2010, pp. 807–814. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104322.3104425
[22] V. Niculae. *A Library for Factorization Machines and Polynomial Networks for Classification and Regression in Python*. Accessed: May 13, 2019. [Online]. Available: https://github.com/scikit-learn-contrib/polylearn
[23] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proc. 37th Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 77–88.
[24] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android malware by building Markov chains of behavioral models (extended version)," *ACM Trans. Privacy Secur.*, vol. 22, no. 2, pp. 14:1–14:34, Apr. 2019.
[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.
[26] N. Peiravian and X. Zhu, "Machine learning for Android malware detection using permission and API calls," in *Proc. IEEE 25th Int. Conf. Tools Artif. Intell. (ICTAI)*, Nov. 2013, pp. 300–305.
[27] S. Rendle, "Factorization machines," in *Proc. IEEE 10th Int. Conf. Data Mining (ICDM)*, Dec. 2010, pp. 995–1000.
[28] D. W. Ruck, S. K. Rogers, M. Kabrisky, M. E. Oxley, and B. W. Suter, "The multilayer perceptron as an approximation to a Bayes optimal discriminant function," *IEEE Trans. Neural Netw.*, vol. 1, no. 4, pp. 296–298, Dec. 1990.
[29] J. Sahs and L. Khan, "A machine learning approach to Android malware detection," in *Proc. European Intell. Secur. Inform. Conf. (EISIC)*, Aug. 2012, pp. 141–147.
[30] F. Scarselli and A. C. Tsoi, "Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results," *Neural Netw.*, vol. 11, no. 1, pp. 15–37, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S089360809700097X

[31] H. Sistemas. *VirusTotal: Analyze Suspicious Files and URLs to Detect Types of Malware, Automatically Share Them With the Security Community*. Accessed: May 22, 2019. [Online]. Available: https://www.virustotal.com

[32] StatCounter. (2018). *Mobile Operating System Market Share Worldwide*. [Online]. Available: http://gs.statcounter.com/os-market-share/mobile/worldwide/2018

[33] StatCounter. *Mobile Malware Evolution 2017*. 2018. [Online]. Available: https://securelist.com/mobile-malware-review-2017/84139/

[34] V. Chebyshev. (2019). *Mobile Malware Evolution 2018*. [Online]. Available: https://securelist.com/mobile-malware-evolution-2018/89689/

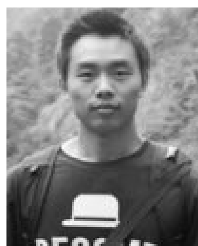
[35] D. Venugopal and G. Hu, "Efficient signature based malware detection on mobile devices," *Mobile Inf. Syst.*, vol. 4, no. 1, pp. 33–49, 2008.

[36] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current Android malware," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*. Bonn, Germany: Springer, 2017, pp. 252–276.

[37] 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," in *Proc. 7th Asia Joint Conf. Inf. Secur. (Asia JCIS)*, Aug. 2012, pp. 62–69.

[38] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "AppContext: Differentiating malicious and benign mobile app behaviors using context," in *Proc. IEEE 37th Int. Conf. Softw. Eng. (ICSE)*, vol. 1, May 2015, pp. 303–313.

[39] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-Sec: Deep learning in Android malware detection," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 371–372, 2014.



CHENGLIN LI received the B.Eng. degree from the University of Science and Technology of China (USTC), in July 2016, and the M.Sc. degree in computer engineering from the University of Alberta, in 2018, where he is currently pursuing the Ph.D. degree. His current researches focus on mobile sensing and distributed machine learning.

KEITH MILLS received the B.Sc. degree (Hons.) in computer engineering from the University of Alberta, in 2018, where he is currently pursuing the M.Sc. degree. His current research interests include, but are not limited to, the optimization and innovation of specialized actor and critic neural networks used in deep reinforcement learning tasks as well as neural network distillation through neural architecture search and evolutionary algorithms.

DI NIU received the B.Eng. degree from Sun Yat-sen University, in 2005, and the M.Sc. and Ph.D. degrees from the University of Toronto, in 2009 and 2013, respectively. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of Alberta, specialized in the interdisciplinary areas of distributed systems, data mining, machine learning, text mining, and optimization algorithms. He was a recipient of the Extraordinary Award of the CCF-Tencent Rhino Bird Open Grant 2016 for his research on natural language processing and machine learning for web document understanding at scale.

RUI ZHU received the master's degree from Xidian University, in 2014. He was a Ph.D. student, from September 2014 to December 2018. He enjoys understanding some elegant mathematical theories, utilizing them to enhance system performance, as well as implementation. In particular, he worked on adapting tools from optimization, graph theory, and information theory to design algorithms for large-scale distributed machine learning and statistical machine learning systems.

HONGWEN ZHANG received the M.Sc. degree in computer engineering, and the Ph.D. degree in computer science. He previously co-founded the 24C Group Inc., which pioneered the first digital receipts infrastructure for secure electronic commerce, and was a principal of Servidium Inc., a global leader in agile development methodology. Throughout his 25+ years career and leadership in the enterprise software industry, he has been instrumental in launching several commercially successful cyber security and safety products into the global market. This has resulted in large customer adoptions. He has served as the Chair for the Metro Ethernet Forum's (MEF) Security-as-a-Service Working Group, which defined the practices of Managed Security Service Providers (MSSPs) for many of the largest telecom service providers in the world. He is a speaker and writer in the areas of security and cloud computing. As a co-founder of Wedge Networks, he has led the design, implementation, and launch of the firm's patented, award-winning Deep Content Inspection and Security Services Orchestration platform.

HUSAM KINAWI received the Ph.D. and M.Sc. degrees in computer science from the Universities of Calgary, Canada, and London, U.K. Within his role at Wedge Networks as the President and Chief Scientist, he works with Wedge's customers, R&D, and engineering teams to cater to customer requirements and demands, while ensuring that the company's product and IP portfolio is continually updated and relevant within the fast moving security industry. Prior to Wedge Networks, he also co-founded M power Technologies Inc., a wireless telecommunications software company, in 1997, and ActiveIq.com (NASDAQ: AIQT), a Boston-based e-Business applications firm, in 1999. He is a co-founder of Wedge Networks. He is an innovator with more than 17 years of research and development experience working with industry leaders such as Newbridge (Alcatel), Siemens, United Technologies, and Apple in the areas of distributed information systems, embedded applications, and wireless Internet solutions.