Malytics: A Malware Detection Scheme

MAHMOOD YOUSEFI-AZAR[1,2], **LEONARD G. C. HAMEY**[1],
VIJAY VARADHARAJAN[3], **AND SHIPING CHEN**[2]

[1]Department of Computing, Faculty of Science and Engineering, Macquarie University, Sydney, NSW 2109, Australia
[2]Commonwealth Scientific and Industrial Research Organisation, Data61, Marsfield, NSW 2122, Australia
[3]Faculty of Engineering and Built Environment, The University of Newcastle, Callaghan, NSW 2308, Australia

Corresponding author: Mahmood Yousefi-Azar (mahmood.yousefiazar@hdr.mq.edu.au)

ABSTRACT An important problem of cyber-security is malware analysis. Besides good precision and recognition rate, ideally, a malware detection scheme needs to be able to generalize well for novel malware families (a.k.a zero-day attacks). It is important that the system does not require excessive computation particularly for deployment on the mobile devices. In this paper, we propose a novel scheme to detect malware which we call Malytics. It is not dependent on any particular tool or operating system. It extracts static features of any given binary file to distinguish malware from benign. Malytics consists of three stages: feature extraction, similarity measurement, and classification. The three phases are implemented by a neural network with two hidden layers and an output layer. We show feature extraction, which is performed by *tf*-simhashing, is equivalent to the first layer of a particular neural network. We evaluate Malytics performance on both Android and Windows platforms. Malytics outperforms a wide range of learning-based techniques and also individual state-of-the-art models on both platforms. We also show Malytics is resilient and robust in addressing zero-day malware samples. The F1-score of Malytics is 97.21% and 99.45% on Android dex file and Windows PE files, respectively, in the applied datasets. The speed and efficiency of Malytics are also evaluated.

I. INTRODUCTION

Malware detection is of paramount importance to our digital era and thus the daily life. Only in 2016/2017 [1], a vast numbers of metamorphic variants of malware for Windows and Android platforms were developed. In addition to the volume of malware generated, novel families make the detection task overwhelming.

Malware detection is mostly based on static or/and dynamic analysis of samples [2]–[4]. Static analysis uses a binary file and/or disassembled code without running it. It is quite efficient, in most cases, but has problems with heavy obfuscation. Dynamic analysis is a better solution for obfuscated samples because it relies on the run-time behaviour, but it is computationally expensive, and the analysis might not see malicious behaviour during testing. Given features extracted, a classic method to detect malicious codes is to generate a signature for every malware sample. The signature-based methods are only good for detecting known malware.

In particular, it is not difficult to create many polymorphic/metamorphic variants of a given malware sample. The new variants easily evade signature-based defence systems.

However, the different variants of the same malware typically exhibit similar malicious patterns. Learning the patterns is the given task of most modern malware detection schemes [5], [6].

Deep Convolutional Neural Network (CNN) and other deep learning models have been developed to address a wide range of our daily life phenomenons such as vision, speech and NLP [7], [8]. The motivations behind them are quite intuitive for the given task and make them state-of-the-art for most cases; however, the proposed scheme of this paper outperforms a wide range of such models. This might be because the scheme is particularly developed for our given task.

This paper presents a novel learning-based scheme that shows robust ability to detect malware compared with existing state-of-the-art learning-based models and other baselines. The proposed scheme which we call *Malytics* is resilient to zero-day samples.

We named our model Malytics because the intuition behind the scheme is an analytic solution to detect malware. That is, the learning algorithm comes from a top-down theory
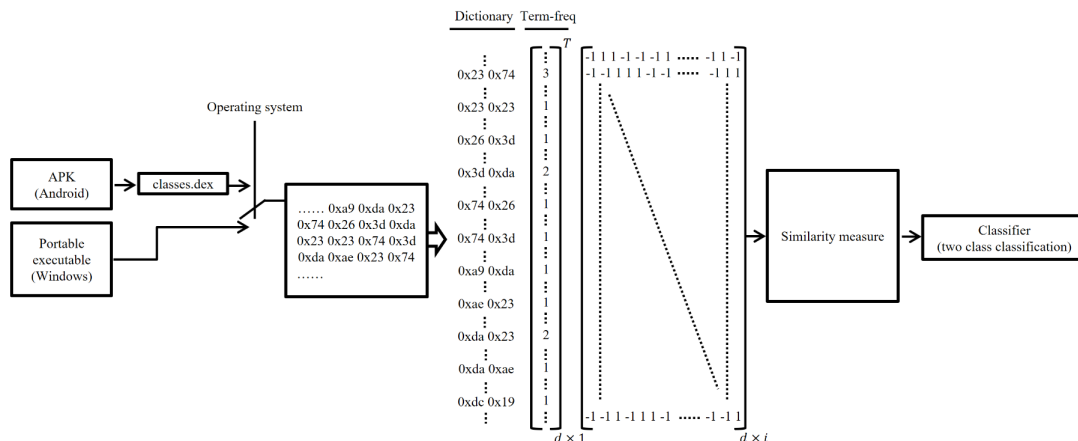
FIGURE 1. Detailed schematic of the proposed solution for malware detection. We used 2-gram (see section II) in this particular example.

with a direct solution rather than learning through samples in an iteration fashion. A wide range of learning algorithms have been developed to learn from input samples [9]. We do not argue the capability of learning models that initiate a hypothesis space (i.e. a model) and adapt this hypothesis space into the training samples. However, we propose to use a learning algorithm that is theoretically related to our proposed feature representation.

The model is an integrated system in which static features are extracted from a binary file and classified by a neural network. Although deep learning models can be this neural network, it is computationally very expensive to use back-propagation to learn a very large feature space. A common solution for this situation is to use random projection techniques [10]. The projected feature space is then fed to the deep neural network. Random projection with our training algorithm shows quite strong results, supported by a theoretical justification.

Figure 1 present a high level concept of the Malytics. Inspired from Natural Language Processing (NLP) (see section II), the term-frequency (*tf*) of the given binary file is multiplied by the random projection matrix including 1 and -1. The result is called *tf*-simhashing. This process is linear.

The representation is fed to the next stage/layer where the similarity indices are obtained as the input for classification. To improve classification, generic non-linear features (e.g. the Gaussian kernel) can be used [11]. This can cause poor generalization [12]. A motivation that this paper uses Extreme Learning Machine (ELM) (see section III) as the supervised classifier is to address the generalization.

We collected different datasets for our experiments. Because the samples were collected in the wild, they could be a malware file or malicious code that was imaged into another file. This setting helps test the model for real world application. We cannot directly compare our model with the other work because we do not have access to specific state-of-the-art work datasets, except one Android dataset; however,

we think that in many cases similar datasets have been used [13]–[16]. Our ground-truth for malware samples is a collection of 19 well-known AV vendors.

To evaluate Malytics a wide range of experiments on Android and Windows samples is conducted. For Android, We propose to use the Dex (Dalvik Executable) file rather than the raw APK. Our experiments show *tf*-simhashing of a dex file of an APK carries more efficient information rather than the APK itself. Dex files are also smaller than APKs. The results on Windows Portable Executable (PE) files show that the model is not dependent on a particular operating system.

Raff *et al.* [17] showed for n-grams with $n > 2$ (e.g 4-gram and 6-gram), the byte-level representation is informative while the information contained in n-grams stem from string features. However, we think 2-gram can carry more information related to instructions and also preserve the string pattern in the frequency.

A. PROBLEM
Malware must be distinguished from benign samples. The system needs to be fast and efficient. Novel Malware families must be detected.

B. SOLUTION
Malytics is a resilient solution for the problem. The byte representation contains important information such as APIs, op-codes. The model learns the pattern of bytes. The *tf*-simhashing static feature representation is a fast solution to embed the byte patterns into a short size vector. Malytics generalizes the patterns well even for novel samples.

C. CONTRIBUTION
- We propose a single and integrated model for malware detection. The model has no dependency on particular tools. Malytics places no restriction on the operating system. Evaluation shows it outperforms other single (non-ensemble) state-of-the-art models for both Android and Windows static analysis.

- We bridge the gap between simhashing and a type of neural network. In particular, we show that simhashing has a close relation to the first layer of ELM. This paper theoretically and empirically show this neural network where the first layer is not trained, has a strong capability for malware detection.

- We show least squares regression in the form of the ELM with a non-linear kernel can provide a template to fully enhance the feature space rather than implicit feature selection of the regressor used in [17]. The Malytics generalization performance for unseen data also shows the effectiveness of the applied regularization technique.

- A further empirical evaluation of Malytics shows that it can successfully detect new malware families and zero-day samples in the wild. This paper also shows Malytics can be tailored for large scale data application while still remains competitive.

The feature extraction method is presented in section 2, and providing the detail of the proposed scheme in section 3. In section 4, we present a comprehensive evaluation on the performance of Malytics for both Android and Windows platforms, and also discuss the results. In section 5, we provide the limitations of Malytics and future direction of research. Section 5 and 6 present related work to our work and conclusion respectively.

II. THE FEATURE EXTRACTION

Hashing is a computation which maps arbitrary size data into data of a fixed size. Hashing algorithms have been widely used in the security application domain [18]–[20]. Locality Sensitive Hashing (LSH) is one of the main categories of hashing methods. It hashes input data so that similar data maps to the same "buckets" with high probability, maximizing the probability of a "collision" for similar inputs. Simhashing is one of the most widely used LSH algorithms, adopted to find similar strings [21], [22]. Simhashing is an LSH that is designed to approximate the cosine similarity between inputs. The main concept of simhashing comes from Sign Random Projections (SRP) [21], [23], [24]. Given an input vector V, SRP utilizes a random Gaussian unit vector (a random hyper-plane) I with each component generated from a Gaussian unit (i.e., $I_i \sim N(0, 1)$ where i is the number of component) and only stores the sign of the projected data as:

$$hash(V) = sign(V \cdot I) \qquad (1)$$

where $\cdot$ is the dot product. Depending on which side of the hyper-plane V lies, $hashing(V) = \pm 1$. A family of the hash function with the mentioned characteristics provides a setting where for two inputs vectors V and U:

$$Pr[hash(V) = hash(U)] = 1 - \frac{\theta(V, U)}{\pi}$$
$$\theta = \arccos\left(\frac{|V \cap U|}{\sqrt{|V| \cdot |U|}}\right) \qquad (2)$$

Where $\theta(V, U)$ is closely related to $cosine(V, U)$ for the two vectors.

If $V \cdot I \geq 0$ then $hash(V) = 1$ and otherwise $hash(V) = 0$, the hamming distance is related to the similarity and it provides a good space to solve the nearest neighbour problem; however, this is not the problem we look to solve. The hash function family generates a real value vector if $hash(V) = V \cdot I$ and equation 2 is still guaranteed. We use simhashing that produce real values.

Simhashing has wide-ranging applications from detecting duplicates in texts (e.g. websites) to different security and to malware analysis, specifically with the Hamming distance similarity measure [25]–[27]. Inspired from NLP application domain, a n-gram is a contiguous sequence of n items (here, a byte pair) from a given sequence of the binary file. The n-gram feature representation is a specific type of bag-of-words representation in which only the number of occurrences of the items is decisive and the location of the items in the binary file is neglected. The theory behind simhashing allows us to weight the byte n-gram [21] with the number of occurrences rather than only representing presence (i.e. zero and one) of the byte n-gram in the file.

The proposed feature representation generates a fix size vector from an arbitrary size binary file. Given a binary file, each n-gram is first hashed to a single fix size vector. To speed up this process, first a dictionary of n-grams is provided and then, this vocabulary hashed to binary values. Having *tf* of the vocabulary stored, each hash bit with value 1 or −1 is weighted with *tf* of the n-gram. Thus, *tf* is inserted into the representation [28]. In the next step, all the vectors sum up bit-wise, thereby providing a final fix size vector. With this process, we embed the distribution of the n-grams of bytes into the vector. This representation provides the two vectors that are close to each other when two files have many common n-grams and different when the files have many different n-grams.

Bit-wise summing up all the real-value hash vectors of a file (i.e. $hash(V_i) = V_i \cdot I_i$ where i is the number of components of hash vector, for example 1024) results in a vector with high variance that needs to be reduced for feeding to an learning algorithm. Support Vector Machines (SVMs) as a competitor to Malytics requires normalizing each feature independently. This transformation reduces the performance in real word application.

Because we want to map the representation into the space where dot product of vectors directly depends on the angle between vectors, each vector needs to be linearly transformed to have zero mean and unit variance. This transformation is different than normalizing each feature independently, because it normalizes each input vector. The other option is to normalize each vector to its Euclidean length (a.k.a L_2 norm). In this case, the dot product of two vectors is directly Cosine similarity. But, since the representation will further map to an infinite-dimension space using a Euclidean-distance-based similarity measure, we do not use L_2 normalization but linearly transform ($Z = \frac{X - \mu}{\sigma}$ to have zero mean and
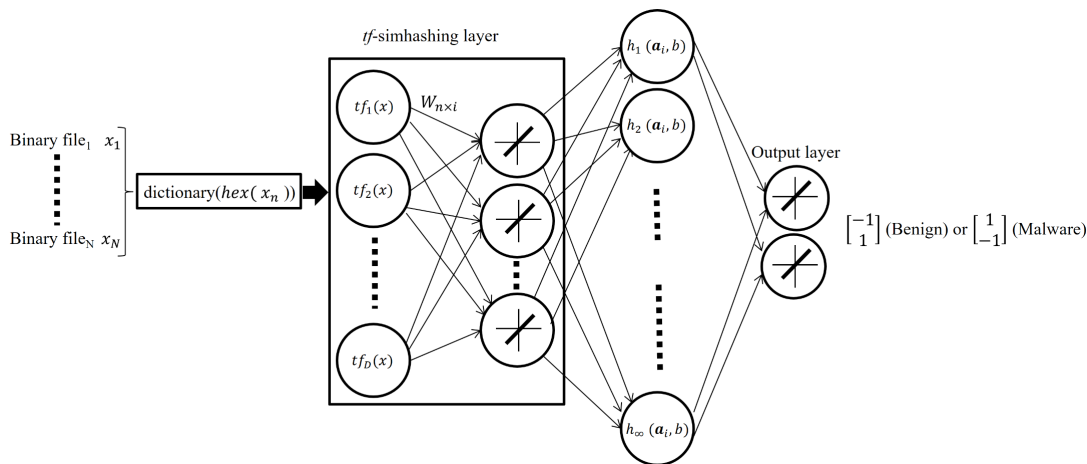
FIGURE 2. The proposed scheme. *tf*-simhashing algorithm has been considered as the first hidden layer.

Algorithm 1 *tf*-simhashing

1: **procedure** tfSimhash(Dataset, ngram, *i*)
2:     $dictionarySize \leftarrow 2^{8*ngram}$
3:     $componentsSize \leftarrow i$
4:     $\mathbf{I}_{dictionarySize \times componentsSize} \leftarrow \mathrm{I}_i \sim \mathcal{N}(0, 1)$
5:     Where $i_i \geq 0$ set to 1 and $i_i < 0$ set to -1
6:     *repeat*
7:         *for each binary file*:
8:             $HexStr \leftarrow Hex(BinaryFile)$
9:             $TF_{1 \times dictionarySize} \leftarrow dic(HexStr, dictionarySize)$
10:
            $tf - simhash_{1 \times componentsSize} \leftarrow normalization(TF \times \mathbf{I})$

unit variance). We observed that this transformation provides better results. The pseudo-code of the proposed greedy-wise algorithm is:

The aforementioned algorithm is how *tf*-simhashing can be implemented; however, in the context of neural network, algorithm 1 is equivalent to the whole process of feeding *tf* representation of byte n-grams to a layer with weights randomly set to 1 or -1; thus, no training is required for this layer (see section III for more theoretical elaboration). The output of the hidden layer is exactly our *tf*-simhashing. Indeed, the proposed algorithm bridges the gap between simashing and a neural network in which the first hidden layer has random weights.

We already know that the similarity of the output of the hidden nodes in algorithm 1 closely depends on the Cosine angle between two samples (i.e. *V* and *U*). In algorithm 1, the vector size *i* corresponds to the number of hidden nodes in the neural network.

In the next section, we call the *tf*-hashing phase as the first layer of our neural network.

III. THE PROPOSED SCHEME
Because the latent representation generated in the output of the first hidden layer is based on the similarity of the

original space, the second hidden layer of the proposed model can provide a similarity measure. Indeed, we need a task-specific similarity over pairs of data points to facilitate the prior knowledge (i.e. training samples in the first hidden layer). This similarity measure followed by a linear predictor also yields a convex optimization problem [29]. Kernel methods can play this role. The relation between kernel machines and the neural network has been widely investigated [29]–[31]. Because kernel layer is data-dependent but unlabeled, the kernel layer training could be seen as unsupervised.

Figure 2 presents the proposed scheme. The output layer weights are analytically obtained using the linear least squares technique. The output layer is the ELM. The whole scheme has more than one hidden layer; thus, it is a deep neural network. However, because the training does not use the back-propagation algorithm for training, the scheme differs from the deep learning that is a well-known term in the machine learning community.

The kernel layer is a non-parametric and nonlinear model to match the input to the templates that are obtained from the training samples. The Radial Basis Function (RBF) kernel is well known for providing an infinite-dimensional kernel space and is commonly used with the kernel trick [31], [32]. As we show later, our model supports the kernel trick, so the RBF kernel is a logical choice.

$$K(x, x_i) = exp^{\left(\frac{-d(x-x_i)^2}{2\gamma^2}\right)} \tag{3}$$

where *d* is the Euclidean distance and γ is the spread parameter. The function is symmetric $K : \mathbf{X} \times \mathbf{X} \rightarrow \mathbb{R}$, a positive-definite matrix and always a real-valued square matrix. This function projects *tf*-simhashing vectors into an infinite dimensional space. The output layer weight can be trained to predict both classes (i.e. 1 and -1).

In detail, although approximations to RBF can also provide good results [32], our malware detection task alongside with the first hidden layer topology give a good reason not to use

any approximation but to use the kernel trick. The kernel trick implicitly maps the feature space to an infinite-dimension feature space. The trick makes the mapping limited by the number of data. We empirically evaluate the effect of the kernel dimension (see the subsection IV-F2).

To obtain the output layer parameters, we use the ELM. Let $\{(x_i, t_i)|x_i \in \mathbb{R}^d, t_i \in \{-1, 1\}\}_{i=1}^N$, where N is the number of training samples, d is the dictionary size and m is the number of output nodes. The ELM model is f as follows:

$$f(x) = \sum_{i=1}^{L} \beta_i h(x, a_i, b_i) = h(x)\beta \quad (4)$$

Where L is the number of hidden nodes, $\beta = [\beta_i, \ldots, \beta_L]^T$ is the output weights and a and b, in our model, are the kernel parameters (i.e $exp^{(-b||x-a||^2)}$. b are constant when ELM is being trained. $h(\cdot)$ is the RBF kernel. The ELM objective function is to minimize:

$$\text{Minimize}_\beta : \frac{1}{2}||\beta||^2 + C\frac{1}{2}\sum_{i=1}^{N}||\xi||^2$$

$$\text{Subject to} : h(x_i)\beta = t_i^T - \xi_i^T, \quad i = 1, \ldots, N \quad (5)$$

Where C is the trade-off parameter, $\xi = [\xi_{i,1}, \ldots, \xi_{i,m}]^T$ is the error between the desired target (e.g. $[-1,1]$ for benign and $[1,-1]$ for malware) and the labels predicted by the models. There are different techniques to obtain the output layer weights β including orthogonal projection method, iterative methods, and singular value decomposition (SVD) [33]–[35]. To minimize the least squares norm, the methods are based on the calculation of the Mooreâ Ă"Penrose pseudo-inverse matrix [35], [36] as follows:

$$\beta = H^\dagger T \quad (6)$$

Where $H^\dagger$ is the matrix. For the sake of feasibility, Kozik [37] used the SVD matrix factorization technique for malware activity detection; however, the proposed malware detection scheme of this paper allows us to use the closed form solution [38]. With Karush-Kuhn-Tucker conditions, the Lagrangian dual problem is defined:

$$L_{\text{Dual}_{\text{ELM}}} : \frac{1}{2}||\beta||^2 + C\frac{1}{2}\sum_{i=1}^{N}||\xi||^2$$

$$- \sum_{i=1}^{N}\sum_{j=1}^{m}\alpha_{i,j}(h(x_i)\beta_j - t_{i,j} + \xi_{i,j}) \quad (7)$$

Where β_j is the out put layer weight/vectors and in our case $m = 2$. The dual problem can be optimized (see Appendix VII) and provides the direct solution as follows:

$$\beta = H^T\left(\frac{I}{C} + HH^T\right)^{-1}T \quad (8)$$

The ELM function is:

$$f(x) = h(x)\beta = h(x)H^T\left(\frac{I}{C} + HH^T\right)^{-1}T \quad (9)$$

Algorithm 2 The Proposed Scheme Algorithm

Input : given N training samples as $\{(x_i, t_i)|x_i \in \mathbb{R}^d, t_i \in \{-1, 1\}\}_{i=1}^N$, given V testing samples as $\{(x_i, t_i)|x_i \in \mathbb{R}^d, \}_{i=1}^V$, n-gram, i, l

Output : predicted labels (Benign or Malware)

Training:

$tf\text{-}simhash_{N \times i} \leftarrow tfSimhash(Dataset, ngram, i)$

$KernelMatrix_{N,N} \leftarrow RBF(tf\text{-}simhash_{N \times i})$

if $l \neq N$ **then**

$RandK \leftarrow Random(1 : l)$

return $KernelMatrix_{l,l}(RandK, RandK)$;

end

$\beta \leftarrow \left(\frac{I}{C} + KernelMatrix\right)^{-1}T$

Testing :

$tf\text{-}simhash_{V \times i} \leftarrow tfSimhash(binaryfile, ngram, i)$

$Predictions \leftarrow \beta \times RBF(tf\text{-}simhash_{V \times i})$

Where $h(\cdot)$ can be unknown and an implicit function satisfies the task. A kernel matrix Ω using a kernel function K can be used as follows:

$$\Omega = HH^T : \Omega_{i,j} = h(x_i) \cdot h(x_j)$$

$$= K(x_i, x_j), \quad i, j = 1, \ldots, N \quad (10)$$

The output function as follows:

$$f(x) = \begin{bmatrix} K(x, x_1) \\ \vdots \\ K(x, x_N) \end{bmatrix}\left(\frac{I}{C} + \Omega\right)^{-1}T \quad (11)$$

The applied kernel is RBF. The method is similar to RBF kernel in SVM. Indeed, SVMs are the particular case of ELM. That is, in ELM all the inputs construct support vectors [38]. Based on the ELM universal approximation capability that is:

$$\lim_{L \to \infty} ||\sum_{i=1}^{L}\beta_i h_i(x) - f(x)|| = 0 \quad (12)$$

As long as $h(\cdot)$ is a strict positive definite kernel [39], [40], a sufficient number of hidden nodes still satisfies 12. In our model, the first hidden layer (*tf*-simashing algorithm) is equivalent to the random nodes; additionally, we also show that we can choose a random subset of the support vectors, from kernel matrix, to reduce the computational over-head and required memory for big data in the cost of the model's performance, that is, $||K_{l \times l}\beta - T|| < \epsilon$ where $l < L$.

The proposed scheme is summarized in Algorithm 2

IV. EVALUATION

To evaluate our scheme, we conducted a wide range of experiments on real datasets collected from the wild for both Android and Windows platforms. During all experiments, we kept the training data balanced (i.e. malware to benign

ratio (MBR) is 0.5) except especially to evaluate the capability of the scheme to deal with imbalanced data. Our benign samples were collected from Androzoo, a freely available APks repository [41]. Androzoo crawled several markets including the official Google Play. We randomly selected apps collected from Google Play. Our collected malware samples (from Virussahre.com)[1] includes a wide range of malware families for both Android and Microsoft Windows platforms. We double checked the status of all malware samples using VirusTotal.com API.[2] VirusTotal provides the results of analysis by about 55 anti-virus vendors. To avoid having a considerable inconsistency, we selected 19 of the most well-known vendors' results. The selected companies are Kaspersky, Symantec, ESET-NOD32, Avast, McAfee, AVG, Avira, Microsoft, BitDefender, Panda, F-Secure, Malwarebytes, TrendMicro, Comodo, VIPRE, AVware, Ad-Aware, Sophos and Qihoo-360. A malware that is detected by at least one of the vendors was picked to be included in our datasets. This procedure was used to establish the ground-truth in all our experiments. Because we collected the samples from the wild, they can be full malware or partially malicious files.

The first malware dataset is Drebin [42]. It consists of 5560 malware of which 5555 have a.dex file.

We also collected our own malware dataset from the two packages VirusShare_Android_20130506.zip and VirusShare_Android_20140324.zip, that we downloaded from VirusShare.com. Together these consist of 35397 malware that were collected before April 2014. Since some samples of the packages have been reversed engineered and were re-compressed, we focus on intact samples. Also, each malware must meet our ground-truth threshold. 20255 malware samples met our criterion. Figure 3 shows the number of samples detected by the 19 anti-virus vendors. We again randomly select 20255 benign from our repository. The statistics of the dataset (DexShare) is presented in the table 1.

For Microsoft Windows, 8912 PE (WinPE) benign has been collected from a fresh installed Windows 10 with 2016 updates. The other benign set, consisting of 11983 PE (WinAppPE), was collected by combining the Windows benign with 77 applications (e.g. firefox, Adobe Reader, etc), automatically install by Ninite.[3] For the malware set, we downloaded VirusShare_00271.zip a package containing 65,536 malware of which 11483 are PEs that also meet the threshold of the selected anti-virus vendors. The package was captured from 2016-11-01 to 2016-11-20 from the wild. To provide balance, we collected further 500 malware from the previous package in VirusShare and added this malware set to yield MalPE2016. Because one of our evaluation goals is zero-day detection, we also collected PEs of VirusShare_00298.zip (MalPE2017) that is a package collected about one year after PE2016. The table 1 shows the
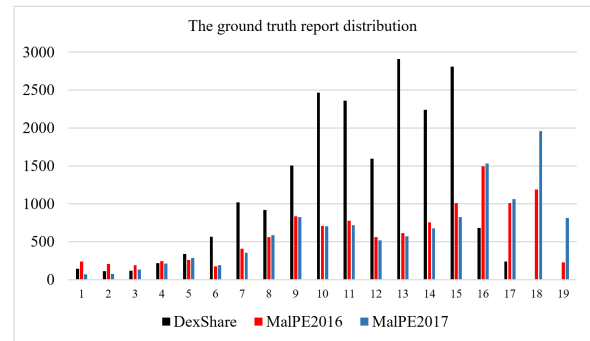
[1] https://Virusshare.com/
[2] https://www.virustotal.com/##/home/search
[3] https://ninite.com/

FIGURE 3. The number of samples detected by the 19 selected anti-virus vendors.

TABLE 1. The statistics of the datasets. Max, Min and Ave stand for maximum size (MB), minimum size (KB) and average size (MB) of the files respectively.

Dataset	Type	Qty	Max.	Min.	Ave.
Drebin	Malware	5555	06.0	2.3	0.36
	Benign	5555	13.5	1.4	3.00
DexShare	Malware	20255	09.2	1.0	0.65
	Benign	20255	10.5	1.9	1.56
PEShare	MalPE2016	11983	53.0	0.8	0.97
	MalPE2017	12127	54.1	1.5	1.26
	WinPE	8912	33.6	1.7	0.35
	WinAppPE	11983	80.3	1.5	0.50

statistics of the Windows dataset (PEShare). Figure 3 also shows the number of samples detected by 1 to 19 anti-virus vendors for both malware sets. When it comes to testing Malytics for WinPE set, we randomly select 8912 malware from MalPE2016.

We used different evaluation metrics to analyze the performance of the proposed scheme. The metric for our two class classification task is based on the confusion matrix:

		Predicted	
		Benign	Malware
Actual	Benign	True Negative (TN)	False Positive (FP)
	Malware	False Negative (FN)	True Positive (TP)

The common performance metrics are:

Recall (a.k.a hit/detection rate or sensitivity) $= \frac{TP}{TP+FN}$. And, False Negative Rate (FNR) $= \frac{FN}{TP+FN}$. where FNR is 1 - recall.

$$\text{Precision} = \frac{TP}{TP+FP}$$

$$\text{f1-score} = 2 * \frac{precision*recall}{precision+recall}$$

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

$$\text{False positive rate (FPR)} = \frac{FP}{TN+FP}$$

The mentioned metrics are enough to evaluate a model. We also used AUC (the area under receiver operating characteristic (ROC) curve) where we found other work with this index. AUC is the probability that a classifier will rank a randomly chosen positive sample higher than a randomly chosen negative sample.

A. EXPERIMENTAL SETUP

To practically show the scheme's efficiency and meeting the mentioned classifier characteristics, we used SVMs, Gradient Boost (XGBoost), Deep Neural Network (DNN), Random Forest (RF) and K-Nearest Neighbors (K-NN) as our baselines. The scikit-learn Python library was used to implement the baselines for SVM, XGBoost, RF, K-NN. We use Keras API, that runs on top of TensorFlow, to implement Deep Neural network. We used grid search to optimize the hyperparameters of the baselines. The results of this the grid search (range from 10^{-4} to 10^4 for both C and gamma) for SVM $C = 0.1$ and $\gamma = 100$ with RBF kernel. The grid search for K-NN ranged from 1 to 20 for n_neighbors and is either 'uniform' or 'distance' for weights. The Deep learning model has 3 hidden layer with 1024, 128 and 64 nodes and 1 node in the output layer. The activation functions are linear rectifier, linear rectifier, sigmoid. The first hidden layer is the proposed representation. The optimizer, batch size and the number of epochs are adam, 5, 100. The first two hidden layer adapt dropout regularization with 0.2 probability.

The two hyper-parameters of Malytics are trade-off parameter C and kernel parameter γ. The result of the grid search for hyper-parameters C (ranging from 10 to 500) is 200 and for γ (ranging from 0.5 to 1.5.) is 1.

Our machine specification is Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 32.0 GB RAM and hard disk drive (HDD).

The hashing algorithm was designed to generate the vector of size 1024, showing strong performance. We observe that larger vectors provide the slightly better results and smaller vectors reduce the performance. The vector of size 1204 is an optimum size while it still is computationally cheap. Over our experiments, we choose 2-gram and used 5-fold cross-validation. Malytics even with 2-gram outperforms the state-of-the-art models. We do not fix a threshold to ensure the highest precision and recall but rather the range of the capability of the model is also a goal of this paper.

B. RESULTS AND DISCUSSION

This part of the paper presents the results of the proposed model. To compare the scheme with other models, we feed *tf*-simhashing to different classifiers. Thus, the proposed feature extraction technique is also examined using different classifiers. Because the presented results on both Android and Windows are based on the 5-fold cross-validation, we present mean and standard deviation (Std) of all 5-folds in the tables. It is usual to present FPR as a fix value rather than mean ($\pm Std$). We also calculated FPR in this way.

C. ANDROID MALWARE DETECTION

This section presents the performance of Malytics on Android malware detection. First, we show the model's capability compared with different baselines on Drebin and DexShare datasets. Then, the model is evaluated in different settings.

Further analysis of Android malware detection is presented in the section IV-F.

Table 2 shows the performance of Malytics compared with the state-of-the-art models [3], [43], 5 baselines and ELM (without kernel layer) on the Drebin dataset. Grosse *et al.* [43] with MBR=0.5 can be compared with our experiments. Mariconti *et al.* [3] also provides a similar setting to ours. Malytics outperforms all other techniques when it comes to detecting malware, that is, *FNR* = 1.44%. This superior performance is seen for f-score and accuracy as well. Interestingly, DNN is the most precise model compared with all others with only 1% FPR while its FNR is the worst.

TABLE 2. The Mean and Std of Malytics and the baselines for Drebin Dataset.

Model	FNR	Precision	f1-score	Accuracy	FPR
SVM	04.81%	96.62%	95.90%	95.93%	3.33%
	($\pm$1.15%)	($\pm$0.40%)	($\pm$0.80%)	($\pm$0.77%)	
XGBoost	02.97%	93.59%	95.28%	95.19%	6.64%
	($\pm$0.20%)	($\pm$0.31%)	($\pm$0.23%)	($\pm$0.25%)	
DNN	13.50%	**98.77%**	92.22%	92.70%	**1.00%**
	($\pm$0.64%)	($\pm$0.62%)	($\pm$0.46%)	($\pm$0.42%)	
RF	04.18%	92.52%	94.14%	94.03%	7.76%
	($\pm$0.55%)	($\pm$0.98%)	($\pm$0.34%)	($\pm$0.38%)	
K-NN	02.80%	93.36%	95.25%	95.15%	6.91%
	($\pm$0.48%)	($\pm$0.70%)	($\pm$0.55%)	($\pm$0.57%)	
ELM	03.00%	94.51%	95.76%	95.70%	5.60%
	($\pm$0.30%)	($\pm$0.47%)	($\pm$0.31%)	($\pm$0.30%)	
Malytics	**01.44%**	96.45%	**97.36%**	**97.33%**	3.90%
	($\pm$0.33%)	($\pm$0.45%)	($\pm$0.29%)	($\pm$0.30%)	
[43]	06.37%	–	–	95.93%	3.96%
[3]	3.00%	95.00%	96.00%	–	–

We also tested the models on a more diverse dataset, DexShare, with more samples, collected in wider time windows. We used AVCLASS tool to label malware samples of both Drebin and DexShare [44]. The tool labels the malware sets based on VirusTotal reports. Because we use VirusTotal to double-check the collected malware set, reports were available to use AVCLASS for labeling. With the tool, Drebin has 180 malware families while DexShare has 309 families.

The performance of the models on DexShare is presented in table 3. It is to be expected that all models perform weaker on the dataset compared to Drebin, since the detaset is more complicated to deal with. Malytics again outperforms all models on DexShare. The results show that Malytics has the highest hit rate (a.k.a recall = 1 − FNR) to detect malware and the highest precision that corresponds to low FPR.

In addition to Malytics, most baselines also provide good performance compared with Zhu *et al.* [15]. This trend shows the *tf*-simhashing feature representation is rich and many classifiers can leverage it and provide good performance. It is true that Zhu *et al.* [15] did not exactly use the DexShare dataset, but they collected their dataset from Virusshare.com as we did. Virusshare.com has two packages for Android malware, and DexShare is a combination of both. So, the results can be compared. Additionally, Hui-Juan's [15] feature extraction is on the basis of static analysis. For example, *tf*-simhashing feeding to SVM yields 93.35% ($\pm$0.16%), 08.00% ($\pm$0.48%), 94.77% ($\pm$0.25%), 93.44% ($\pm$0.16%) for AUC, FNR, precision, and accuracy

TABLE 3. The Mean and Std of Malytics and the baselines for DexShare Dataset.

Model	FNR	Precision	f1-score	Accuracy	FPR
SVM	08.00% (±0.48%)	94.77% (±0.25%)	93.34% (±0.18%)	93.44% (±0.16%)	05.07%
XGBoost	10.12% (±0.55%)	90.74% (±0.39%)	90.30% (±0.36%)	90.35% (±0.35%)	09.17%
DNN	24.40% (±3.53%)	90.40% (±1.73%)	82.23% (±1.45%)	83.72% (±0.89%)	08.13%
RF	13.07% (±0.35%)	92.73% (±0.32%)	89.73% (±0.22%)	90.05% (±0.21%)	06.82%
K-NN	7.45% (±0.48%)	93.36% (±0.70%)	91.38% (±0.55%)	91.28% (±0.57%)	10.00%
ELM	16.50% (±0.29%)	92.25% (±0.67%)	87.66% (±0.30%)	88.24% (±0.36%)	07.00%
Malytics	**05.53%** (±0.46%)	**95.88%** (±0.40%)	**95.17%** (±0.20%)	**95.20%** (±0.20%)	**04.06%**
[15]	11.60% (±2.76%)	88.16% (±1.8%)	– –	88.26% (±1.73%)	–

respectively while Zhu *et al.* [15] reported 86.00% (±2.0%), 13.82% (±2.3%), 84.13% (±3.5%) and 84.93% (±1.8%) for AUC, FNR, precision, and accuracy respectively when they used SVM as the classifier.

Table 4 provides more inside into Malytics. The *tf*-simhashing feature extraction algorithm can be implemented on APK as well as only Dex file of respective APK. Yousefi-Azar *et al.* [28] is based on *tf*-simhashing of the APK. Table 4 shows hashing the dex file yields much better performance compared with hashing the whole APKs.

TABLE 4. The Mean and Std of Malytics for DexShare dataset on the APK, Dex. Also, the results when the dataset is imbalanced and for zero-day (novel families) detection.

Model	FNR	Precision	f1-score	Accuracy	FPR
Malytics (APK)	09.43% (±0.61%)	91.76% (±0.73%)	91.16% (±0.54%)	91.22% (±0.48%)	8.1%
Malytics (Dex, MBR=0.5)	05.33% (±0.46%)	95.88% (±0.40%)	95.17% (±0.20%)	95.20% (±0.20%)	4.1%
Malytics (Dex, MBR=0.2)	**05.27%** (±0.63%)	**98.45%** (±0.69%)	96.55% (±0.55%)	98.65% (±0.21%)	3.7%
Malytics (Zero-day)	**10.59%**	**96.31%**	**92.68%**	92.99%	3.4%
[3]	12.00%	86.00%	87.00%	–	–

A common test is to evaluate a model in an imbalanced setting to mimic the real world settings. For the test, MBR is typically 10%, 20%, 30%. To have enough malware to test the scheme and also to provide the imbalanced settings, we chose MBR = 0.2. That is, we randomly selected 5060 malware from the DexShare malware set while the total benign set was used. Table 4 shows that Malytics performs more precisely with imbalanced data. We expect this results because the model saw more benign sample in the training. FNR does not show a statistically significant change. It demonstrates that Malytics is robust to the imbalanced situations.

One of the most important tests of a malware detection system is to evaluate the system against zero-day malware. There are different evaluation methods to do a zero-day experiment. Mariconti *et al.* [3] use a time frame test. That is, they trained the model with samples of a given date, the model was tested on samples of one year and also two years later than the given date. In short, training on past sample and testing on new samples in time.

Although we could use the timestamp of the samples of DexShare, because the timestamp of a file is easily forged, both by malware and benign writers, we think timestamp is not a good index to partition our dataset into past and future samples. We also think that AVCLASS is not a very accurate technique to partition our dataset.

However, because we do not have any other concrete option, we again rely on AVCLASS. As mentioned earlier in this section, AVCLASS labels DexShare with 309 different families. From 309 families, about 20 families have more than about 150 samples in each family. We chose these 20 families for our novelty detection test. To do this test, we selected training and test sets and we do not use cross-validation. From malware set, out of 20 families, 4 families were chosen to be the test set and the rest of malware set were chosen to be training set. We did this test 5 time to test on all 20 families. To be clear, when 4 families were chosen to be the test set, the other 16 families plus the rest of the malware set are the training set. The benign set was randomly chosen from DexShare to keep the training and test sets balanced.

Table 4 shows the average FNR, precision, f1-score, accuracy and FPR of Malytics with our proposed feature representation. The main important index of the test is FNR as a measure that shows how well Malytics detected new families. Mariconti *et al.* [3] also provided a novelty detection setting based on detecting future malware. Our test is different with their test. But if we assume that our family exclusion test is at least as difficult as predicting future malware (e.g. test on one year in future), we can see that Malytics is quite competitive with the state-of-the-art in novelty detection. For further explanation see section IV-F1.

For real word application, we can increase the size of hash vector to improve the performance while Malytics still requires a light computation. Motivated from [45], we replaced the *tf*-simashing weights (i.e −1 and 1 values) with a sparse matrix including −1, 1 and 0 [46]. We set the sparsity to 1% and the size of *tf*-simashing vector is 3000. Thus, only 30 elements of the hashing matrix are non-zero but after summing over the entire vocabulary, the hash size is 3000. Then, this vector is used as the input to the kernel layer and then the output layer. This sparse setting helps reduce the complexity of the *tf*-simashing computation while increasing the hidden feature representation size. Table 5 presents the results of the experiment on both datasets. Malytics False positive improves slightly while hit-rate is very close to dense setting (see table 2). The size of hashing provides richer hidden representation for DexShare samples. In addition to being more precise, Malytics has better hit-rate (see table 3). The imbalanced setting shows Malytics performance for real word application. We set hash size 3000 which has slight impact on the LEM computation.

We conducted the last experiment to show Malytics can perform in different settings and its improvement capability. To have a comparable settings, other experiments of the paper are all based on dense matrix setting with the hash size 1024.

TABLE 5. The Mean and Std of Malytics with sparse *tf*-simashing for both Drebin and DexShare Datasets.

Dataset	FNR	Precision	f1-score	Accuracy	FPR
Drebin	01.53%	96.68%	97.56%	97.54%	3.38%
	(±0.50%)	(±0.45%)	(±0.40%)	(±0.30%)	
DexShare	04.72%	96.69%	95.96%	96.00%	3.30%
	(±0.50%)	(±0.26%)	(±0.38%)	(±0.35%)	
DexShare (MBR=0.2)	**04.42%**	98.91%	97.21%	98.90%	**2.60%**
	(±0.27%)	(±0.82%)	(±0.53%)	(±0.18%)	

D. WINDOWS MALWARE DETECTION

This section presents the capability of Malytics to detect Microsoft Windows malware. We show that the scheme is not restricted by any specific feature of the operating system.

Table 6 shows that Malytics and Wuechner *et al.* [14] outperform other models when it comes to distinguishing original Windows PE clean files from PE malware. Malytics is the most capable method in detecting malware with lowest the FPR compared with all methods, in particular, with our machine learning baselines. It has better FNR compared with Wuechner *et al.* [14] and both schemes have the same precision while Wuechner *et al.* [14] used an imbalanced dataset. In an imbalanced setting, trade-off indices are more reliable for concluding an analysis. F-score of the proposed model in Wuechner *et al.* [14] is more than Malytics but the difference is not statistically significant. AUC indices show that Malytics outperforms other models. The FPR as an important factor for Windows platform malware analyzer is well less than 1% that is critical for Windows.

TABLE 6. The Mean and Std of Malytics and the baselines for WinPE and Mal2016 of the PEShare Dataset.

Model	FNR	Precision	f1-score	Accuracy	AUC	FPR
SVM	1.30%	99.13%	98.91%	98.92%	98.92%	0.86%
	(±0.30%)	(±0.25%)	(±0.26%)	(±0.26%)	(±0.25%)	
XGBoost	1.30%	98.43%	98.56%	98.56%	98.57%	1.57%
	(±0.27%)	(±0.26%)	(±0.19%)	(±0.19%)	(±0.19%)	
DNN	2.51%	96.77%	97.11%	97.09%	97.09%	3.31%
	(±0.59%)	(±2.04%)	(±0.78%)	(±0.83%)	(±0.83%)	
RF	2.18%	98.44%	98.13%	98.14%	98.14%	1.55%
	(±0.35%)	(±0.32%)	(±0.22%)	(±0.21%)	(±0.25%)	
K-NN	1.56%	98.50%	98.47%	98.47%	98.47%	1.50%
	(±0.38%)	(±0.04%)	(±0.20%)	(±0.20%)	(±0.20%)	
ELM	1.00%	95.82%	97.38%	97.34%	97.79%	4.30%
	(±0.17%)	(±0.35%)	(±0.18%)	(±0.19%)	(±0.18%)	
Malytics	**0.55%**	99.20%	99.32%	**99.32%**	**99.96%**	**0.80%**
	(±0.23%)	(±0.27%)	(±0.12%)	(±0.11%)	(±0.19%)	
[14]	1.00%	**99.20%**	**99.70%**	—	99.30%	—
	(±0.00%)	(±0.00%)	(±0.5%)	—	(±0.1%)	
[13]	0.80%	—	99.10%	99.05%		1.10%

Table 7 shows that the proposed solution outperforms other models over all evaluation indices. It is to be expected that all models performance is reduced compared with table 6, mainly because the benign set of this experiment is a mix of Windows PEs and third-party PEs while the malware set is from the same source and only has more samples.

An interesting result of trying to distinguish Mal2016 malware set from WinAppPE benign set is in the comparison of DNN with Raff *et al.* [16]. Raff *et al.* [16] used deep CNN for detection. The results show deep learning models also can be competitive for malware application domain. Although the

TABLE 7. The Mean and Std of Malytics and the baselines for WinAppPE and Mal2016 of the PEShare Dataset.

Model	FNR	Precision	f1-score	Accuracy	AUC	FPR
SVM	2.78%	98.32%	97.76%	97.78%	97.78%	1.7%
	(±0.13%)	(±0.16%)	(±0.14%)	(±0.14%)	(±0.14%)	
XGBoost	2.24%	97.61%	97.68%	97.68%	97.69%	2.4%
	(±0.36%)	(±0.19%)	(±0.20%)	(±0.20%)	(±0.20%)	
DNN	4.43%	94.71%	95.09%	95.02%	95.02%	5.5%
	(±0.60%)	(±3.90%)	(±1.90%)	(±2.00%)	(±2.05%)	
RF	4.46%	97.85%	96.68%	96.72%	96.72%	2.1%
	(±0.58%)	(±0.26%)	(±0.29%)	(±0.27%)	(±0.28%)	
K-NN	2.30%	96.90%	97.30%	97.29%	97.29%	3.1%
	(±0.15%)	(±0.31%)	(±0.19%)	(±0.19%)	(±0.19%)	
ELM	2.18%	93.95%	95.84%	95.76%	97.42%	6.3%
	(±0.17%)	(±0.42%)	(±0.19%)	(±0.14%)	(±0.19%)	
Malytics	**1.32%**	**98.65%**	**98.66%**	**98.67%**	**99.81%**	**1.3%**
	(±0.06%)	(±0.04%)	(±0.20%)	(±0.19%)	(±0.22%)	
[16]	—	—	—	94.00%	98.10%	—

input of CNN is an image representation of the PE files and DNN's input is *tf*-simhashing, we think deep learning models can also be competitive if the feature representation has more theoretical justification in the deep learning models' training algorithm.

Figure 4 shows the detection rate of Malytics on Mal2017. The training sets are Mal2016 and WinAppPE while Mal2017 is the test set. To have a similar setting to the real world, the training benign set was WinAppPE rather than WinPE. This experiment is to evaluate how well the scheme can detect zero-day attacks. We assume that a one year interval between the malware set in training and the malware set for testing is an acceptable chronological gap.
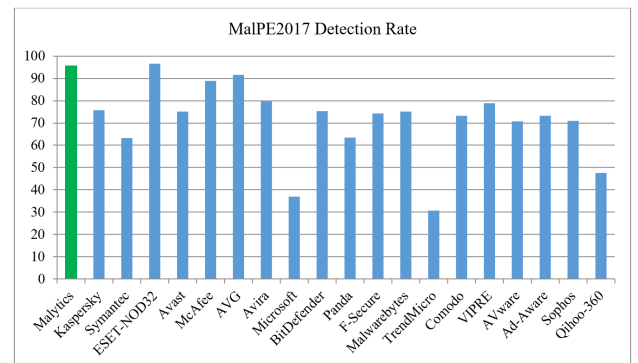

FIGURE 4. The detection rate of Malytics and the 19 malware vendor for Mal2017, trained on Mal2016 and WinAppPE.

Malytics could successfully detect 95.5% of the Mal2017 as zero-day samples. It is only one percent less than ESET-NOD32 detection rate. AVG with 91.5% is the third in the ranking. Our ground truth for the detection rate of AV vendors is VirusTotal real-time update report. VirusTotal always uses the latest update of AVs; thus, the detection rates it reports are considerably better than they would be if the virus detector data was one year old. After one year, Malytics performs competitively with the best AV vendor software fully up to date. In another experiment, we trained Malytics using Mal2016 and WinPE sets and tested on Mal2017. As it is to be expected, the detection rate increases to 98.1%.
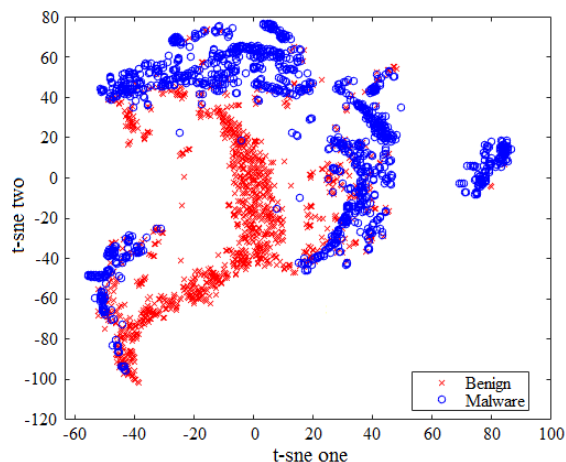
FIGURE 5. t-SNE of WinPE versus Mal2016. WE clustered each class with 1200 centroids for each class.

E. TF-*SIMHASHING VISUALIZATION*

To have more insight into the proposed latent representation, that is, *tf*-simhashing layer and RBF kernel, a visualization experiment was conducted on the feature space. We implemented the experiment on Windows malware dataset, WinPE and Mal2017 and used t-SNE [47] to visualize the space.

To this end, *tf*-simhashing values have been clustered using k-means clustering technique and then the centroids of the clusters fed to t-SNE. We think because in the k-means optimization algorithm the Euclidean distance is used as a metric, it can provide a similar ground with RBF kernel that is also based on Euclidean distance. But RBF kernel provides an infinite feature space that we cannot visualize easily. Also, the intention of the experiment is to show that similar vectors of *tf*-simhashing are quantified similarly and *tf*-simhashing is meaningful.

More precisely, *tf*-simhashing of the dataset (here $17824 = 2 \times 8912$) benign and malware samples) are clustered into 2400 clusters (1200 centroids per class). So, the input of the k-means function is a matrix of size 17824×1024 and the output is a matrix of size 2400×1024. Effectively, we use k-means as a vector quantisation algorithm, yielding on average, one codebook per about 7.4 vectors ($\frac{17824}{2400} = 7.43$). Our experiment shows the 2400-vector representation provides a good visual picture to understand the dataset. The matrix of 2400×1200 is fed to t-SNE to be mapped into two-dimensional space (2400×2) for visualization.

We obtain codebooks from malware and benign sets separately. That is, 1200 centroid per class. Figure 5 shows clustering the representation provides a meaningful result and the codebooks are basically distributed similarly. This means the vectors of *tf*-simhashing is meaningful and can settle in closed distance when we optimize k-mean with its Euclidean distance measure. We do not show representation of feeding *tf*-simhashing to t-SNE directly because it generates a mean-ingless distribution.

F. CASE STUDY AND FURTHER ANALYSIS OF THE SCHEME
1) ANDROID FAMILY DETECTION CASE

The scope of this paper is not a particular malware family but covers the range of malware disseminating all over the network. Our datasets were collected with this purpose. However, looking into some specific cases may provide better inside into the model. We used the DexShare dataset for detail study.

As described in section 4, we tested the model on 20 new families. In brief, we chose groups of 4 families for testing and the remaining 16 families plus all the other malware and benign sets for training. This routine was repeated 5 times.

Figure 7 presents the number of total samples in each family and false negative detection. We tested Malytics in the balanced and imbalanced group of families. Also, one of the groups consists of malware families (Fakeinst,[4] Adwo,[5] SMSreg,[6] Lotoor)[7] with 4 different functions/intention. The detection rate is similar for most families and the diversity of malware function did not prevent detection.

The Adwo family is the least challenging for detection by Malytics. This is to be expected since although Adwo is not in the training set, other adware variants are used in training. Fakeinst and Opfake were reported as similar families and Fakeinst had been continued to be detected while Opfake not.[8] Malytics could detect Opfake better than Fakeinst. It might because of the complexity of the Fakeint that our model could not detect it well, as it was also continued to disseminate over the net in the real world. But, it might because of the number of sample in training when another family is presented only in testing set.

The worst detection rate belongs to Plankton[9] ($\frac{80}{344} = 23.3\%$). This family sits silently, collecting information and sending it to a remote location. Its variants have a wide range of actions.[10] Calleja *et al.* [48] has particular analysed on Plankton. They showed that this family is very similar to the Nyleak and BaseBridge families. These two similar families to Plankton have only 28 and 2 samples in DexShare. We think that because, in training, there are few similar malware samples to the Plankton family, Malytics' detection rate is reduced for this family; however, the 76.7% hit rate is still very good for this family in this setting.

Figure 6 presents ROC and respective AUC of the novelty detection. With 2-3% FPR, the hit rate is more than 75% for all four families that seems promising.

---

[4]https://www.f-secure.com/v-descs/trojan_android_fakeinst.shtml

[5]https://www.sophos.com/en-us/threat-center/threat-analyses/adware-and-puas/Android%20Adwo/detailed-analysis.aspx

[6]https://home.mcafee.com/virusinfo/virusprofile.aspx?key=8503749

[7]https://www.symantec.com/security_response/writeup.jsp?docid=2012-091922-4449-99

[8]https://threatpost.com/opfake-fakeinst-android-malware-variants-continue-resist-detection-080712/76887/

[9]https://www.f-secure.com/v-descs/trojan_android_plankton.shtml

[10]https://www.avira.com/en/support-threats-summary/tid/8996/threat/ANDROID.Plankton.C.Gen
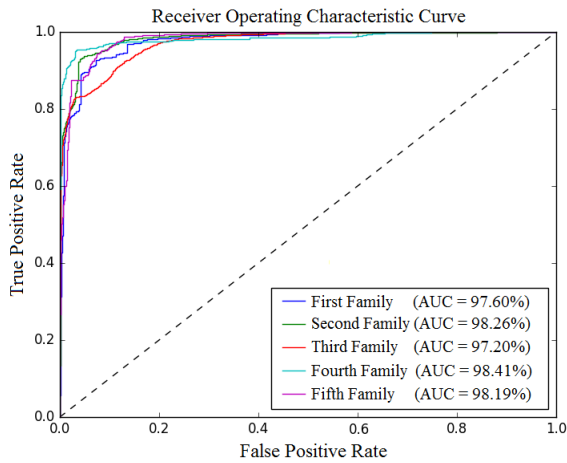
FIGURE 6. The ROC curve for novelty detection on DexShare. Each curve shows the ROC curve and AUC when groups 4 families have been fetched out from DexShare. First family is the first left 4 family in the figure 7 and so on.
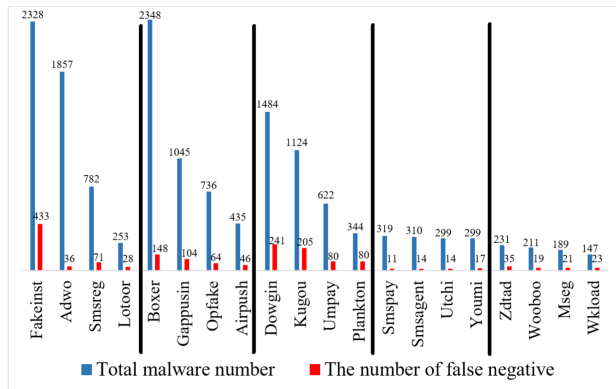

FIGURE 7. Total number of samples and False negative detection based on 4 families fetching out on DexShare. The black bar separates each group of 4 families.

2) RANDOM KERNEL

Malytics is based on the batch learning algorithm. That is, the entire training set is fed to Malytics for training in one batch. This process might become computationally very expensive and demand a large amount of memory for big data.

As mentioned in III, Malytics has the capability of being trained on a random subset of training data while still keeping the generalization performance. This random selection has a negative impact on the performance of Malytics. Figure 8 presents the f-score for both platforms when the kernel matrix size varies from 10% to 100% of the original kernel matrix. Similar to all previous evaluations, Android platform is more impacted than Windows. It is to be expected because Windows PEs are all from Microsoft while Android apps are developed by many different developers and therefore exhibits a greater variety.

Table 8 shows the run-time performance of Malytics with random kernel sampling from 10% to 100% of the kernel matrix. The training and testing time increases but
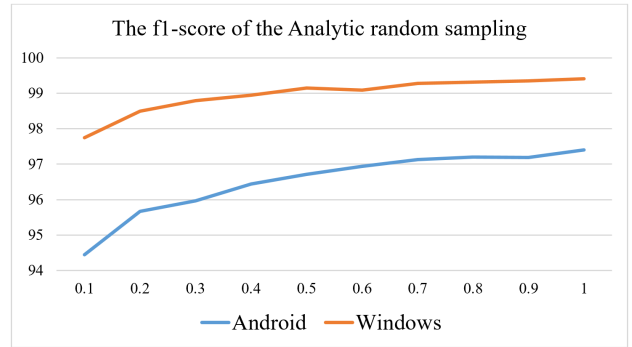

FIGURE 8. The f1-score of Malytics trained on a random subsets of the datasets. For Android, Drebin dataset was used and for Windows, WinPE versus Mal2016.

not sharply. In contrast, the f-score value increases sharply initially (see figure 8). This shows that choosing more than a threshold number of samples may provide the desired performance with optimum memory and computation requirements. This demonstrates Malytics's scalability.

3) RUN-TIME PERFORMANCE

Because Malytics is based on static analysis, we compare it with MAMADROID [3]. The run-time performance of Malytics is not dependent on the operating system while MAMADROID is proposed to detect Android malware.

The average execution time of MAMADROID for benign samples in family and package modes are 27.3s and 33.83s per sample respectively. MAMADROID says for 10000 apps that are being submitted to Google Play per day, the model requires less than one and a half hours to complete execution with 64 cores.

In our prototype of Malytics, on average, tf-simhashing algorithm speed is 560KB per second. This includes extracting a dex file out of the APK. Also, for the Drebin dataset with 11110 samples (on average $\frac{3+0.36}{2} = 1.68MB$), the training and testing time are 3.6s and 0.6s respectively, for each fold (from table 8). That is, $\frac{3.6s}{11110 \times 0.8} = 0.4ms$ per sample to train the model and $\frac{0.6s}{11110 \times 0.2} = 0.27ms$ per sample to test an App.

Based on the statistics of Drebin malware set and random benign set collected from Google Play, Malytics requires, approximately, $\frac{1.68MB}{0.560MB/Sec} = 3$ second for each sample to hash and 0.27ms to detect. For 10000 apps, 30003 seconds to complete execution with one core and 470 seconds with 64 cores. In short, Malytics needs less than 8 minutes to complete execution for all the Google Play new samples in a day. The speed of Malytics provides the possibility of frequently training with new samples.

V. LIMITATIONS AND FUTURE WORK

The main limitation of Malytics is the amount of the required memory. More precisely, the advantage of Malytics batch learning technique is speed and mostly convexity; nevertheless, the model needs to store all input samples as a batch in memory. The learning process also requires a considerable amount of memory to obtain the output layer weights,

TABLE 8. The run-time performance of Malytics with randomly selected kernel matrix *L* ranging from 10% to 100% of the kernel matrix *N*. The time is for each fold (i.e. 8888 training and 2222 test samples) of the 5-fold cross-validation.

Dataset	Phase	$L = \frac{N}{10}$	$L = \frac{N}{20}$	$L = \frac{N}{30}$	$L = \frac{N}{40}$	$L = \frac{N}{50}$	$L = \frac{N}{60}$	$L = \frac{N}{70}$	$L = \frac{N}{80}$	$L = \frac{N}{90}$	$L = N$
Drebin	Train	0.12s	0.15s	0.29s	0.58s	0.80s	1.20s	1.75s	2.35s	3.20s	3.60s
	Test	0.10s	0.16s	0.22s	0.27s	0.35s	0.40s	0.45s	0.54s	0.58s	0.6s
WinPE vs Mal2016	Train	0.09s	0.35s	0.78s	1.50s	2.40s	3.70s	5.30s	7.20s	9.60s	11.6s
	Test	0.19s	0.34s	0.52s	0.70s	0.80s	1.14s	1.20s	1.30s	1.50s	1.55s

of course, but only for big data. Also, for the test phase, the kernel layer needs to keep all the training samples. This intensifies the memory issue and also makes the test process directly dependent on the training set.

Malytics hashes the whole binary file into one vector. This may not be effective for binary files that are partly infected with malicious codes. To address this issue, hashing windows of a binary file may help and is a potential direction of future study.

Malytics relies on the static analysis of binary files and it does not have any knowledge of the behavior of the binary. Despite the computational expenses of dynamic analysis, it can cover deficiencies of the static features of malware in particular for obfuscated malware and more importantly for advanced persistent attacks.

tf-simhashing visualization experiment suggests that k-means might be used for feature learning the hashing dictionary rather than random generation from a distribution. Also, deep learning models have no theoretic underlying with a random initialization layer. This may contribute to deep learning models as well.

The dex file is the not the only source of information in Android apps. The manifest file also contains critical information. For Windows, PEs structure can be informative if we could embed this information into the hashing algorithm. Also, developing a feasible algorithm for larger n-grams may improve the performance. Finally, an ensemble of fast learning models is a potential direction of study for future work.

VI. RELATED WORK

In addition to malware as a general concept, there are novel systems to deal with particular types of malware [49], [50]. In both cases, learning-based systems show very promising results [51]–[54].

For Android malware detection, Mariconti *et al.* [3] proposed a static-feature extraction model that could provide a very good performance. A novelty of the work was the proposed random variable based on Markov chain. The output was fed to a feature extraction phase in which Principal Component Analysis (PCA) [55] was used. At the end, each sample file yeilds a vector of size 100,000 to be classified as either benign or malware. Applying PCA to such feature space requires a huge amount of memory to obtain the co-variance matrix. Zhu *et al.* [15] recently showed that rotational forest, as a classifier, has the capability of being

applying for Android malware detection. They also used static features.

Wuechner *et al.* [14] used a compression-based graph mining technique to detect Windows malware. They widely evaluated the effect of classifiers on their scheme and reported that all the applied classifiers provide similar results. Carlin *et al.* [13] used the run-time opcodes of every sample with significantly different approach compared with Wuechner *et al.* [14] and still presented competitive results. Both papers used dynamic analysis of Windows PEs. Dynamic analysis of malware is more computationally expensive than static analysis. Having said that, it is well-known to be used in many anti-malware production due to its reliable performance and capability to cope with obfuscated files.

Dynamic analysis of a malware also has the vulnerability of crafting an adversary. Very recently, Stokes *et al.* [10] proposed a detection system using dynamic analysis and they showed it is still vulnerability to crafted adversarial attacks.

The *tf*-simhashing representation is a promising feature representation [28]. Simhashing has also been used for malware detection and detecting similarity between data/files [56], [57]. In both [28] and [56], simhashing was used to represent each file as an image which was then fed to a naive classifier or a CNN. Malware detection on the basis of visualization is not restricted only to simhashing. Raff *et al.* [16] proposed a new feature representation to map any binary file into an image and used CNN as the classifier.

ELM shows very promising results for malware activity detection and identification of malicious packed executables. Kozik [37] presented a distributed ELM using NetFlow data structure alongside the Apache Spark[11] framework that provided good performance. Different types of ELM have been applied for malicious packed executable identification [58].

VII. CONCLUSION

In this paper, we proposed a learning-based malware detection model called "Malytics". This integrated model comprises two layers of latent feature representation and a layer for prediction. The first layer is a hashing algorithm (*tf*-simhashing) and we showed that it has a close relation to the first layer of the Extreme Learning Machine (ELM). ELM is the output layer of the proposed scheme. We showed that having a layer to measure the similarity of *tf*-simhashing

---

[11]https://spark.apache.org/

before output layer strongly improves the performance of the scheme. We used the RBF kernel for the similarity measure.

We conducted comprehensive evaluations on Drebin, DexShare, PEShare datasets and Malytics outperforms different baselines including non-ensemble state-of-the-art models. Drebin and DexShare are Android apps and PEShare is Windnows PEs. The dex file of Android apps is informative enough to compete with related work. We tested how well Malytics could perform on imbalanced datasets, for novel family detection. The Novelty detection was organized in two different ways: particular family detection and chronological novelty detection. We also evaluated the speed and scalability of Malytics. It shows promising results for large-scale data.

APPENDIX
OPTIMIZATION

The dual problem can be optimized as follows:

$$\frac{\partial L_{\text{Dual}_{\text{ELM}}}}{\partial \boldsymbol{\beta}_j} = 0 \rightarrow \beta_j = \sum_{i=1}^{N} \alpha_{i,j} h(x_i)^T \rightarrow \boldsymbol{\beta} = \boldsymbol{H}^T \boldsymbol{\alpha} \quad (13)$$

$$\frac{\partial L_{\text{Dual}_{\text{ELM}}}}{\partial \xi_j} = 0 \rightarrow \boldsymbol{\alpha}_i = C\xi_i, i = 1, \dots, N \quad (14)$$

$$\frac{\partial L_{\text{Dual}_{\text{ELM}}}}{\partial \boldsymbol{\alpha}_j} = 0 \rightarrow \boldsymbol{h}(x_i)\boldsymbol{\beta} - \boldsymbol{t}_i^T + \xi_i^T = 0, \quad i = 1, \dots, N \quad (15)$$

Where $\boldsymbol{\alpha}_i = [\alpha_{i,1}, \dots, \alpha_{i,m}]^T$ and $\boldsymbol{\alpha}_i = [\boldsymbol{\alpha}_1, \dots, \boldsymbol{\alpha}_N]^T$. With a bit of calculus, for $\boldsymbol{\beta}$:

$$\boldsymbol{\beta} = \boldsymbol{H}^T \left(\frac{\boldsymbol{I}}{C} + \boldsymbol{H}\boldsymbol{H}^T \right)^{-1} \boldsymbol{T} \quad (16)$$

REFERENCES

[1] *AV Test: Facts and Figures—Security Report 2016/2017*. Accessed: Aug. 17, 2018. [Online]. Available: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2016-2017.pdf

[2] M. Y. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of Android malware," in *Proc. NDSS*, vol. 16, 2016, pp. 21–24.

[3] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android malware by building Markov chains of behavioral models," in *ISOC Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, 2017, pp. 1–15.

[4] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 1043–1054.

[5] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Berlin, Germany: Springer, 2008, pp. 108–125.

[6] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proc. NDSS*, vol. 9, 2009, pp. 8–11.

[7] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 3859–3869.

[8] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[9] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Artificial Intelligence: A Modern Approach*, vol. 2. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.

[10] J. W. Stokes, D. Wang, M. Marinescu, M. Marino, and B. Bussone. (Dec. 2017). "Attack and defense of dynamic analysis-based, adversarial neural malware classification models." [Online]. Available: https://arxiv.org/abs/1712.05919

[11] B. Schölkopf and A. J. Smola, *Learning with Kernels—Support Vector Machines, Regularization, Optimization & Beyond*. Cambridge, MA, USA: MIT Press, 2002.

[12] Y. Bengio, O. Delalleau, and N. L. Roux, "The curse of highly variable functions for local kernel machines," in *Proc. Adv. Neural Inf. Process. Syst.*, 2006, pp. 107–114.

[13] D. Carlin, P. O'Kane, and S. Sezer, "Dynamic analysis of malware using run-time opcodes," in *Data Analytics and Decision Support for Cybersecurity*. Cham, Switzerland: Springer, 2017, pp. 99–125.

[14] T. Wuechner, A. Cislak, M. Ochoa, and A. Pretschner, "Leveraging compression-based graph mining for behavior-based malware detection," *IEEE Trans. Dependable Secure Comput.*, to be published, doi: 10.1109/TDSC.2017.2675881.

[15] H.-J. Zhu, Z.-H. You, Z.-X. Zhu, W.-L. Shi, X. Chen, and L. Cheng, "DroidDet: Effective and robust detection of Android malware using static analysis along with rotation forest model," *Neurocomputing*, vol. 272, pp. 638–646, Jan. 2018.

[16] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. (Oct. 2017). "Malware detection by eating a whole exe." [Online]. Available: https://arxiv.org/abs/1710.09435

[17] E. Raff *et al.*, "An investigation of byte n-Gram features for malware classification," *J. Comput. Virol. Hacking Techn.*, vol. 14, no. 1, pp. 1–20, 2018.

[18] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: Feature hashing malware for scalable triage and semantic analysis," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 309–320.

[19] S. Dharmapurikar and J. W. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *IEEE J. Sel. Areas Commun.*, vol. 24, no. 10, pp. 1781–1792, Oct. 2006.

[20] E. A. Manzoor, S. Momeni, V. N. Venkatakrishnan, and L. Akoglu. (Feb. 2016). "Fast memory-efficient anomaly detection in streaming heterogeneous graphs." [Online]. Available: https://arxiv.org/abs/1602.04844

[21] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proc. 34th Annu. ACM Symp. Theory Comput.*, 2002, pp. 380–388.

[22] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for Web crawling," in *Proc. ACM 16th Int. Conf. World Wide Web*, 2007, pp. 141–150.

[23] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," *VLDB*, vol. 99, no. 6, pp. 518–529, 1999.

[24] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Proc. 47th Annu. IEEE Symp. Found. Comput. Sci. (FOCS)*, Oct. 2006, pp. 459–468.

[25] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *Proc. IEEE 18th Work. Conf. Reverse Eng. (WCRE)*, Oct. 2011, pp. 13–22.

[26] P.-T. Ho, H.-S. Kim, and S.-R. Kim, "Application of sim-hash algorithm and big data analysis in spam email detection system," in *Proc. ACM Conf. Res. Adapt. Convergent Syst.*, 2014, pp. 242–246.

[27] K. Han, B. Kang, and E. G. Im, "Malware analysis using visualized image matrices," *Sci. World J.*, vol. 2014, Jul. 2014, Art. no. 132713.

[28] M. Yousefi-Azar, L. Hamey, V. Varadharajan, and M. D. McDonnell, "Fast, automatic and scalable learning to detect Android malware," in *Proc. Int. Conf. Neural Inf. Process.* Cham, Switzerland: Springer, 2017, pp. 848–857.

[29] Y. Bengio and Y. LeCun, "Scaling learning algorithms towards AI," *Large-Scale Kernel Mach.*, vol. 34, no. 5, pp. 1–41, 2007.

[30] Y. Cho and L. K. Saul, "Kernel methods for deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2009, pp. 342–350.

[31] A. G. Wilson, Z. Hu, R. Salakhutdinov, and E. P. Xing, "Deep kernel learning," in *Artificial Intelligence and Statistics*. 2016, pp. 370–378.

[32] A. Vedaldi and A. Zisserman, "Efficient additive kernels via explicit feature maps," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 3, pp. 480–492, Mar. 2012.

[33] G. Huang, G.-B. Huang, S. Song, and K. You, "Trends in extreme learning machines: A review," *Neural Netw.*, vol. 61, pp. 32–48, Jan. 2015.

[34] G. H. Golub and C. F. Van Loan, *Matrix Computations*, vol. 3. Baltimore, MD, USA: JHU Press, 2012.

[35] C. R. Rao and S. K. Mitra, *Generalized Inverse of Matrices and Its Applications*. New York, NY, USA: Wiley, 1971.

[36] R. Penrose, "A generalized inverse for matrices," *Math. Proc. Cambridge Philos. Soc.*, vol. 51, no. 3, pp. 406–413, 1955.

[37] R. Kozik, "Distributing extreme learning machines with Apache Spark for NetFlow-based malware activity detection," *Pattern Recognit. Lett.*, vol. 101, pp. 14–20, Jan. 2018.

[38] G.-B. Huang, "An insight into extreme learning machines: Random neurons, random features and kernels," *Cognit. Comput.*, vol. 6, no. 3, pp. 376–390, 2014.

[39] W.-Y. Deng, Y.-S. Ong, and Q.-H. Zheng, "A fast reduced kernel extreme learning machine," *Neural Netw.*, vol. 76, pp. 29–38, Apr. 2016.

[40] G.-B. Huang, L. Chen, and C.-K. Siew, "Universal approximation using incremental constructive feedforward networks with random hidden nodes," *IEEE Trans. Neural Netw.*, vol. 17, no. 4, pp. 879–892, Jul. 2006.

[41] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories (MSR)*, May 2016, pp. 468–471.

[42] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, 2014, pp. 1–15.

[43] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," in *Proc. Eur. Symp. Res. Comput. Secur.*, Oslo, Norway: Springer, 2017, pp. 62–79.

[44] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVclass: A tool for massive malware labeling," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Cham, Switzerland: Springer, 2016, pp. 230–253.

[45] C. Chen, C.-M. Vong, C.-M. Wong, W. Wang, and P.-K. Wong, "Efficient extreme learning machine via very sparse random projection," *Soft Comput.*, vol. 22, no. 11, pp. 3563–3574, 2018.

[46] P. Li, T. J. Hastie, and K. W. Church, "Very sparse random projections," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 287–296.

[47] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, pp. 2579–2605, Nov. 2008.

[48] A. Calleja, A. Martín, H. D. Menéndez, J. Tapiador, and D. Clark, "Picking on the family: Disrupting Android malware triage by forcing misclassification," *Expert Syst. Appl.*, vol. 95, pp. 113–126, Apr. 2018.

[49] A. Kharaz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, "UNVEIL: A large-scale, automated approach to detecting ransomware," in *Proc. USENIX Secur. Symp.*, 2016, pp. 757–772.

[50] I. Gasparis, Z. Qian, C. Song, and S. V. Krishnamurthy, "Detecting Android root exploits by learning from root providers," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)* Vancouver, BC, USA: USENIX Association, 2017, pp. 1129–1144. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gasparis

[51] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: Effective and efficient behavior-based Android malware detection and prevention," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 1, pp. 83–97, Jan./Feb. 2016.

[52] D. Ucci, L. Aniello, and R. Baldoni. (Oct. 2017). "Survey on the usage of machine learning techniques for malware analysis." [Online]. Available: https://arxiv.org/abs/1710.08189

[53] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, 2012, Art. no. 6.

[54] S. Wang, Q. Yan, Z. Chen, B. Yang, C. Zhao, and M. Conti, "Detecting Android malware leveraging text semantics of network flows," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 5, pp. 1096–1109, May 2018.

[55] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics Intell. Lab. Syst.*, vol. 2, nos. 1–3, pp. 37–52, 1987.

[56] K. Han, J. H. Lim, and E. G. Im, "Malware analysis method using visualization of binary files," in *Proc. ACM Res. Adapt. Convergent Syst.*, 2013, pp. 317–321.

[57] C. Sadowski and G. Levin, "Simhash: Hash-based similarity detection," Google, Tech. Rep., 2007.

[58] P. Xie, X. Liu, J. Yin, and Y. Wang, "Absent extreme learning machine algorithm with application to packed executable identification," *Neural Comput. Appl.*, vol. 27, no. 1, pp. 93–100, 2016.

MAHMOOD YOUSEFI-AZAR is currently pursuing the Ph.D. degree with the Computing Department, Macquarie University, Sydney, NSW, Australia, and is also a Student Researcher with Data61, Commonwealth Scientific and Industrial Research Organization. His research aims to develop novel learning-based algorithms for cyber security applications. His Ph.D. project is machine learning and malware analysis. He is a Reviewer for journals, including ACM TOIT, IET Signal/Image, and ESWA.

LEONARD G. C. (LEN) HAMEY is currently a Senior Lecturer and the Deputy Head of the Department with the Department of Computing, Macquarie University, Sydney, NSW, Australia. He works in both image analysis and machine learning, with applications ranging from industrial vision systems through to medical image analysis and malware detection. He has authored over 50 refereed papers and holds two patents.

VIJAY VARADHARAJAN is currently a Global Innovation Chair Professor with The University of Newcastle, Australia, and the Director of the Advanced Cyber Security Research Centre. He has authored over 400 papers in international journals and conferences and 10 books in information technology, security, networks and distributed systems. He holds three patents. He has been/is on the Editorial Board of several journals, including ACM TISSEC, IEEE TDSC, IEEE TIFS, and IEEE TCC.

SHIPING CHEN is currently a Principal Research Scientist at CSIRO, Data61. He also holds an Adjunct Associate Professor with The University of Sydney and the University of New South Wales through teaching and supervising Ph.D. students. He has been involving on distributed systems for over 20 years with a focus on performance, security, and trust. He has authored over 150 research papers in these areas. He is also actively involved in software engineering and service/cloud computing research communities through publications, journal editorships, and conference TPC services, including WWW, EDOC, ICSOC, and IEEE ICWS/SCC/CLOUD. His current research interests include data services, secure data sharing, and services for collaboration.

• • •