

A Multimodal Deep Learning Method for Android Malware Detection Using Various Features

TaeGuen Kim^{ID}, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im^{ID}

Abstract—With the widespread use of smartphones, the number of malware has been increasing exponentially. Among smart devices, android devices are the most targeted devices by malware because of their high popularity. This paper proposes a novel framework for android malware detection. Our framework uses various kinds of features to reflect the properties of android applications from various aspects, and the features are refined using our existence-based or similarity-based feature extraction method for effective feature representation on malware detection. Besides, a multimodal deep learning method is proposed to be used as a malware detection model. This paper is the first study of the multimodal deep learning to be used in the android malware detection. With our detection model, it was possible to maximize the benefits of encompassing multiple feature types. To evaluate the performance, we carried out various experiments with a total of 41260 samples. We compared the accuracy of our model with that of other deep neural network models. Furthermore, we evaluated our framework in various aspects including the efficiency in model updates, the usefulness of diverse features, and our feature representation method. In addition, we compared the performance of our framework with those of other existing methods including deep learning-based methods.

Index Terms—Android malware, malware detection, intrusion detection, machine learning, neural network.

I. INTRODUCTION

WITH the growing popularity of mobile devices such as smartphones or tablets, attacks on the mobile devices have been increasing. Mobile malware is one of the most dangerous threats which cause various security incidents as well as financial damages. According to the G DATA report [1] in 2017, security experts discovered about 750,000 new

Manuscript received October 18, 2017; revised January 29, 2018 and June 2, 2018; accepted July 22, 2018. Date of publication August 21, 2018; date of current version September 13, 2018. This work was supported in part by the MSIT (Ministry of Science, ICT), South Korea, under the ITRC (Information Technology Research Center) Support Program under Grant IITP-2018-2013-1-00881 supervised by the Institute for Information & Communication Technology Promotion (IITP), in part by the IITP Grant funded by the Korea Government (MSIT), Development of Defense Technologies against Ransomware, under Grant 2017-0-00388, and in part by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) under Grant NRF-2016R1A2B4015254. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Wei Yu. (Corresponding author: Eul Gyu Im.)

T. Kim is with the Department of Computer and Software, Hanyang University, Seoul 04763, South Korea (e-mail: claudio17@hanyang.ac.kr).

B. Kang is with the Centre for Secure Information Technologies, Queen's University of Belfast, Belfast BT7 1NN, U.K. (e-mail: B.Kang@qub.ac.uk).

M. Rho and E. G. Im are with the Department of Computer Science and Engineering, Hanyang University, Seoul 04763, South Korea (e-mail: minarho@hanyang.ac.kr; imeg@hanyang.ac.kr).

S. Sezer is with the Centre for Secure Information Technologies, Queen's University of Belfast, Belfast BT7 1NN, U.K. (e-mail: s.sezer@qub.ac.uk).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/TIFS.2018.2866319

Android malware during the first quarter of 2017. It is expected that a large number of mobile malware will keep developed and spread to commit various cybercrimes on mobile devices. Android is a mobile operating system that is most targeted by mobile malware because of the popularity of Android devices. In addition to the number of Android devices, there is another reason that leads malware authors to develop Android malware. The reason is that the Android operating system allows users to install applications downloaded from third-party markets and attackers can seduce or mislead Android users to download malicious or suspicious applications from attackers' servers.

To mitigate the attacks by Android malware, various research approaches have been proposed so far. The malware detection approaches can be classified into two categories; static analysis based detection [2]–[19] and dynamic analysis based detection [20]–[24]. The static analysis based methods use syntactic features that can be extracted without executing an application, whereas the dynamic analysis based methods use semantic features that can be monitored when an application is executed in a controlled environment. Static analysis has an advantage that it is unnecessary to set the execution environments, and the computational overheads for static analysis are relatively low. Dynamic analysis has an advantage that it is possible to handle malicious applications which use some obfuscation techniques such as code encryption or packing.

In this paper, we assume that obfuscated malware is processed by dynamic analysis based methods, and we focus on the development of a static analysis based method to distinguish between malware and benign applications. This paper proposes a novel malware detection framework based on various static features. Our framework is flexible to add a new type of features, so, it is possible to utilize dynamic features in the future.

There are many previous works that are related to Android malware detections, but most of the previous studies use only limited types of features to detect malware. Each type of feature can represent only a few properties of applications. On the other hand, we propose a framework to detect malware using many feature information to reflect various characteristics of applications in various aspects. Our proposed framework first extracts and processes multiple feature types, and refines them using our feature vector generation methods. Our feature vector generation method consists of an existence-based method and a similarity-based method, and these are very effective to distinguish between malware and benign applications even though malware has many similar

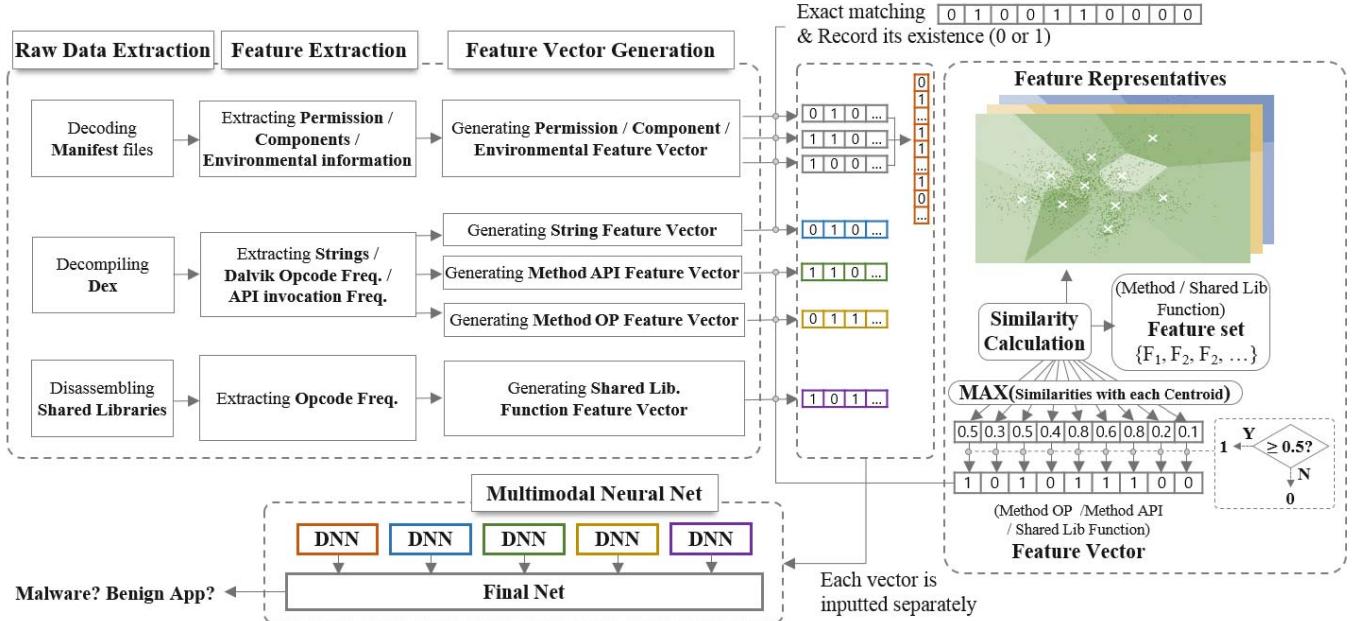


Fig. 1. The overall architecture of the proposed framework.

properties of benign applications. In addition, our framework uses a classification model that implies the degree of classification according to their importance. Among many useful classification algorithms, we concluded that the deep learning algorithm is the suitable classification algorithm for our framework that uses various types of feature.

We propose a multimodal deep neural network model to fit the features with different properties. The multimodal deep learning method is generally utilized to make the neural network to reflect the properties with different kinds of feature. For example, the multimodal deep learning method was used to recognize human speech using both voice information and mouth shape information [48]. The different types of the feature are inputted and processed in different initial neural networks separately, and each initial network is connected to a final neural network to produce the classification results. According to our survey, our research is the first application of the multimodal deep learning to the Android malware detection.

We conducted many experiments using our framework with a large dataset from VirusShare [38] and the well-known small dataset from the Malgenome project [37]. We measured and compared the performance of our model with that of the deep neural network model. In addition, we evaluated our framework in various aspects including efficiency in model updates, the usefulness of diverse features and effects of our feature representation method. According to the comparison results with other deep learning based methods, we argue that our framework has good performance on the malware detection.

Our contributions can be summarized as follows:

- We proposed a novel Android malware detection framework using diverse features that can reflect the characteristics of Android applications.
- We suggested feature vector generation methods that can represent malware characteristics effectively even when

malware shares many common properties with benign applications.

- We introduced how the multimodal neural network can be applied in malware detection system. Model learning strategies and an online update method for malware detection are proposed. To the best of our knowledge, this research is the first application of the multimodal deep learning to the Android malware detection.
- We provided various experimental results of our framework to evaluate the performance in various aspects. Total seven experiments were conducted in this paper.

The rest of the paper is organized as follows: Section II explains the overall architecture of our Android malware detection framework and describes how the framework works in detail, Section III presents the feature types that are used in our framework, and the multimodal neural network algorithm is explained in Section IV. Section V shows the experimental results to show the performance of our framework, and Section VI discusses related work, followed by Section VII that summarizes our research and provides future work of this ongoing research.

II. PROPOSED FRAMEWORK

Fig.1 shows the overall architecture of our framework, and our framework uses seven kinds of the feature; String feature, method opcode feature, method API feature, shared library function opcode feature, permission feature, component feature, and environmental feature. Using those features, the seven corresponding feature vectors are generated first, and then, among them, the permission/component/predefined setting feature vectors are merged into one feature vector. Finally, the five feature vectors are fed to the classification model for malware detection. The framework conducts four major processes for the detection; raw data extraction process, feature extraction process, feature vector generation process,

and detection process. These processes are explained in the next subsections.

A. Raw Data Extraction Process

The raw data extraction process is performed to make Android APK (Android Package Kit) files interpretable. To extract the raw data, an APK file is unzipped, and a manifest file, a dex file, and shared library files are extracted first. The manifest file and the dex file are decoded or disassembled by APKtool [32], and the shared library files (i.e. .so files) in the package can be disassembled by IDA Pro [33].

B. Feature Extraction Process

The feature extraction process is conducted to obtain the essential feature data from the raw data. The detailed definition of feature types is explained in Section III.

First, method opcode features and method API features are extracted from smali files which are the disassembled results of a dex file. The smali file is separated into the method blocks, and, by scanning Dalvik bytecodes, the Dalvik opcode frequency of each method is calculated. In addition, during the bytecode scanning, it is checked whether the invocation of the dangerous APIs exists in the method, and the dangerous API invocation frequency of each method is calculated. In case of string features, strings are simply collected from the whole smali files without considering the method separation.

Shared library function opcode features are extracted from the instruction sequences of the disassembled code of .so files. The instruction sequence of each function is scanned to extract the information of the assembly opcode frequency.

The permission features, the component features, and environmental features are extracted from the manifest XML file. While visiting the XML tree nodes, each node's tag is checked to confirm whether the node contains the information about permissions, application components, and so on.

C. Feature Vector Generation Process

The extracted features in the previous process are used to compose feature vectors. Seven kinds of the feature vector are generated from extracted features. The seven feature vectors are divided into two types according to their feature representations: existence-based feature vectors and similarity-based feature vectors. The existence-based feature vector is the feature vector whose elements only represent the existence of features in the malicious feature database, and examples of these are string, permission, component and environmental feature vectors. On the other hand, the similarity-based feature vector is the feature vectors whose elements are similar to the malware representatives in the malicious feature database, and method opcode, method API and shared library function feature vectors are the similarity-based feature vectors.

The malicious feature database herein is a repository that contains features and malware representatives of known malicious applications. The structure of the database is described in Fig. 5 in APPENDIX B, and each feature is explained in Section III. In addition, the malware representatives mean

the centroids of the clusters which are calculated using the K-means clustering algorithm [44].

Algorithms I and II explain in APPENDIX A the processing flows of the feature generation. First, as explained in Algorithm I, the existence-based feature generation process is simple. The feature values in the malicious feature database correspond to the elements of the feature vector, and every feature value is searched in the features extracted from input applications. If there is no certain feature value in the extracted features, its absence is represented as zero. Otherwise, the existence of the feature value is represented as one in the vector.

Second, the similarity-based feature vectors are generated as explained in Algorithm II. The method opcode feature, the method API feature, and the shared library function opcode feature used in this feature vector generation process are in the form of a list of frequencies. The frequency values can vary considerably, so the features of an input application are first normalized to fit the feature values in the range of [0, 1]. The min-max scaling method is used in the normalization [45]. Then, each malware representative (the centroid of the cluster) in the malicious feature database is compared with the features of the input application using the Euclidean distance measure. Among the distances of each malware representative, the minimum distance is selected to convert to the similarity, and the calculated similarity is recorded in the corresponding element of the feature vector. By recording the highest similarity values of the multiple malware representatives, the feature vector can contain similarities to multiple clusters' centroids which are computed with known malware applications. Therefore, the similarity-based feature vector can represent information whether the input application's features belong to clusters.

To improve the performance of our framework, we refined the feature vector with a predefined threshold value. The similarity values that exceed the predefined similarity threshold become one. Otherwise, it is set to zero. This refinement removes the features that are not close enough to a certain malware representative but have small similarity values, and it also simplifies the computation in the deep learning process.

D. Detection Process

After all the seven feature vectors are generated in the previous process, the detection process is conducted to determine whether the given application is malicious or not. Before examining the feature vectors with the detection model, the permission feature vector, the component feature vector, and the environmental feature vector are merged into a single feature vector. Therefore, our model gets the five feature vectors and performs mathematical operations at each layer. If all operations are conducted completely, the model produces the estimated label for the given input application.

III. THE DEFINITION OF FEATURES

Diverse features could be helpful to reflect the characteristics of an application. Even though some features such as environmental information are not directly related to malicious activities, these features may contribute to defining the application characteristics.

Our proposed framework uses the following features:

- String feature
- Method opcode feature
- Method API feature
- Shared library function opcode feature
- Permission feature
- Component feature
- Environmental feature

In our framework, the deep learning algorithm is used to classify the unknown samples into the malware class or the benign class. The deep learning algorithm generates a neural network model that can derive the best classification accuracy by updating the weight of each neuron input. The degree of influence of the feature on classification is determined according to the weight of the neurons affected by the feature. If there is an insignificant feature in the classification, the weight of the relevant neurons is reduced. Therefore, each feature can be used differently by their contributions.

The next subsections explain each feature type that is used in our framework. It is noted that the features are converted to the feature vectors to apply them to the neural network.

A. String Feature

The string feature is extracted from a set of string values in smali files. The feature extraction module collects all operand values with the types of `const-string` and `const-string/jumbo`. There are also the Dalvik opcodes that move a reference to a string into a specific register. The number of strings in an application spans a wide range. If the number of applications increases, then the number of strings from those applications will increase explosively. Therefore, strings are hashed, and the hashed values of strings are applied to the modular operation. The hash function used in the framework is the SHA512 hash function.

B. Method Opcode and API Feature

Dalvik opcode frequency and API invocation frequency of methods may imply application behaviors and coding habits of the developer. For this reason, Dalvik opcode frequency and API invocation frequency of methods are used to define the method features. The method opcode frequency can be calculated by scanning the bytecode in each method. In the case of the API invocation frequency, the bytecodes for API invocation are checked to count the API invocations in each method. To capture malicious behaviors, invocations of only selected APIs are counted. The APIs that might be used in malicious activities are investigated manually using the Android Developer reference pages [50]. Additionally, the APIs that were introduced in [35] are also added to the selected API list. According to [35], those selected APIs are useful to distinguish malware and benign applications.

C. Shared Library Function Opcode Feature

Android provides the Java Native Interface (JNI) and allows applications to incorporate native libraries. It is well known that native code defeats Android security mechanisms because

native code is not covered by the security model. For example, shared library files can be used to hide malicious behaviors or to avoid countermeasure against attacks. That is the reason why many malicious applications use the native code to attack the Android system.

To prevent malware with native code from hiding its behaviors, our framework defines and uses the shared library function features in the detection. Similar to the method feature extraction, ARM opcode frequency and system call invocation frequency are extracted from native code. While scanning the disassembled code of each function, the opcodes and system call invocations in each function are counted.

D. Permission Feature

Android is a privilege-separated operating system, and an application runs with a unique system identifier. Android provides a permission-based access control mechanism to restrict the operations that a process can perform. In addition, per-URI permissions are used to grant access to specific data. To perform a certain behavior, an application should request necessary permissions to Android, and this means that permissions defined in an application can indicate the behaviors of an application.

The manifest file in the application includes various information related to permissions. First, the permissions to be requested when the application is installed are defined in the manifest file. Second, security permission that can be used to limit accesses to specific components is also defined to protect the application. The permission-related information can be collected by parsing the `<uses-permission>` tag and the `<permission>` tag in the manifest file. The request permissions' names are collected from the `<uses-permission>` tag, and the security permissions' names, permission groups and protection levels are collected from the `<permission>` tag. The extracted request permissions and security permissions (the tuples of name, permission group, and protection level) are used as permission features.

E. Component Feature

Application components are the essential building blocks of an Android application. There are four components in an Android application; Activity, service, broadcast receiver, and content provider. The basic role of each component is explained as follows. The activity component presents a visual user interface that an application developer defines. The service component does not have any visual interface but performs background processing. The content provider component provides database interfaces to share data with other applications. The broadcast receiver component provides a channel to receive messages from other applications.

Every component of an application is able to register and to receive messages, called intents. The intent can be used to start the component or to deliver some important data to the component. Whenever the intent is transmitted to the component, the predefined callback function is executed to handle the intent. The intent can present how the components communicate each other. The component-intent pairs are

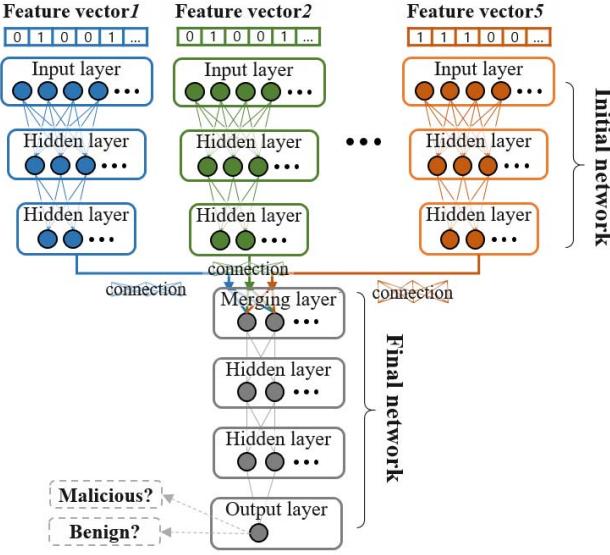


Fig. 2. Multimodal deep neural network.

extracted as the component feature to analyze the relationship between the components. In the case of the content provider, we also extract the URI path information that specifies which data subsets of the parent content provider permission can be granted.

The manifest file of the application is parsed to record the declared components and their specified intents. First, the component name is extracted from `<activity>`, `<service>`, `<provider>`, and `<receiver>` tags, and the names of intents are extracted from `<intent-filter>` tags. The URI paths are collected from the `<grant-uri-permission>` tag and are paired with the name of the content provider and the intent.

F. Environmental Feature

In the manifest file, there is some information about the environment settings to execute the application. Requirements or usage of hardware and software, list of shared libraries that the application requires, and SDK(Software Development Kit) version for execution are extracted as environmental features. `<uses-feature>`, `<uses-library>`, and `<uses-sdk>` tags are parsed to collect the names of requested hardware/software, the names of linked libraries, and the SDK version respectively.

IV. MULTIMODAL NEURAL NETWORK

Fig. 2 shows the architecture of the multimodal deep neural network for malware detection in our framework. Our proposed neural network model uses five feature vectors, and each vector is inputted separately to the initial networks which consist of five DNNs (Deep Neural Network). The initial networks are not connected to each other, and the last layers of the initial networks are connected to the merging layer that is the first layer of the final network. The final network is a DNN, and it produces the classification results. Each DNN of the initial networks consists of an input layer and two hidden layers, and each layer only receives connections from the previous layer. Each layer is fully-connected, and the

TABLE I
THE PARAMETERS OF THE MULTIMODAL NEURAL NETWORK

Network	Layer	Parameter setting
Initial DNNs	Input	5000 neurons
	Hidden	2500 neurons
	Hidden	1000 neurons
Final DNNs	Merging	500 neurons
	Hidden	100 neurons
	Hidden	10 neurons
	Output	1 neurons
		Sigmoid

activation functions used in the DNNs are the rectified linear units (ReLU) activation function [51]. The ReLU activation function is utilized to prevent the vanishing gradient problem in training, and it also makes our model computationally efficient.

The final network is a similar shape of the DNNs to the initial network except for the first and the last layers. The first layer, i.e. merging layer, is connected with the last layers of the DNNs of the initial networks. The last layer of the final network, i.e. output layer, produces the classification results. In the output layer, there is only one neuron that uses the sigmoid function to label an input application as malware or a benign application. The parameters used in each layer in the multimodal neural network are summarized in Table I.

A. Formal Description

In this subsection, the formal description of our multimodal neural network model is explained. Let $t \in \{1, 2, 3, 4, 5\}$ indicates each initial network, $l^t = \{1, 2, 3\}$ be a layer in t initial network, x^t the input vector corresponding t initial network, $z^{(l^t)}$ the incoming vector into layer l^t , $y^{(l^t)}$ the output vector of the layer l^t , $W^{(l^t)}$ the weights of the layer l^t , $b^{(l^t)}$ the biases of the layer l^t , and f the activation function (ReLU). The equation for the feed-forward operation of the initial network is as follows:

$$y^{(0^t)} = x \quad (1)$$

$$z^{(l^t+1)} = W^{(l^t+1)}y^{(l^t)} + b^{(l^t+1)} \quad (2)$$

$$y^{(l^t+1)} = f(z^{(l^t+1)}) = \max(0, z^{(l^t+1)}) \quad (3)$$

Let $l' = \{1, 2, 3, 4\}$ be a layer in final network, $z^{(l')}$ the incoming vector into layer l' , $y^{(l')}$ the output vector of the layer l' , $W'^{(l')}$ the weights of the layer l' , $b'^{(l')}$ the biases of the layer l' , f the activation function (ReLU), o the output function (Sigmoid). The equation for the feed-forward operation of the final network is as follows:

$$y^{(0)} = [y^{(l^1)}, y^{(l^2)}, y^{(l^3)}, y^{(l^4)}, y^{(l^5)}] \quad (4)$$

$$z'^{(l'+1)} = W'^{(l'+1)}y^{(l')} + b'^{(l'+1)} \quad (5)$$

$$y'^{(l'+1)} = f(z'^{(l'+1)}) = \max(0, z'^{(l'+1)}), \text{ where } l' \neq 4 \quad (6)$$

$$y'^{(4)} = o(z'^{(4)}) = 1/(1 + e^{-z'^{(4)}}) \quad (7)$$

The labels for the input vector is determined as shown in Equations (8) and (9). L denotes the label of input vector, and

L' denotes the predicted label result.

$$\begin{cases} y'^{(4)} \geq 0.5, & L' = 1 \\ y'^{(4)} < 0.5, & L' = 0 \end{cases} \quad (8)$$

During the learning process, our multimodal neural network model is tuned to minimize the value of the loss function, i.e. the cross-entropy function. The loss function is described as follows:

$$J(L, L') = -\sum_{i=1}^2 L^{(i)} \log L'^{(i)} + (1 - L^{(i)}) \log(1 - L'^{(i)}) \quad (9)$$

B. Regularization

Overfitting is one of the major problems of neural network models. If a neural network model is overfitted with a particular train set, then the neural network model cannot generally be used in classification. To avoid the overfitting problem, Dropout regularization [49] is applied to our multimodal neural network model. Dropout is a technique to skip some units randomly while the neural network is trained. With a fixed probability, some incoming and outgoing connections of a neuron in the network are removed. This makes the model become not too dependent on a specific set of units and their associated weights and the biases. In our model, the DNNs are modeled with the dropout rate of 0.2. This rate is generally used in the typical DNN based models.

In addition, when tuning the neural network model, we used the validation set as well as the training set. While the training set is used to fit the model, the classification accuracy using the validation set is also measured together. The validation set does not update the weights and the biases of the model, but by monitoring the trends of the accuracies of both of the training set and the validation set, it can be checked whether the model fitting is done correctly without the overfitting problem. If the accuracy growths of both sets are not similar, then the tuning process is terminated, and modeling parameters are changed to avoid the overfitting problem. In the detection experiments in Section V-D, we used the validation set for the model tuning.

C. Learning Strategy for the Multimodal Neural Network

In practice, although malware and benign applications are given to be analyzed, it is not guaranteed that all of the features can be extracted from the given applications. In our cases, there were some applications that the decompiler tool or the disassembler tool cannot analyze properly. Only 78% of dex files, 79% of manifest files, and 69% of .so files were translated properly. The detailed statistical information is explained in Table II. It is possible to build the multimodal neural network by using only applications from which all types of features can be extracted or by padding zeros to feature vectors when the features cannot be obtained. In order to handle the problem of partially extracted features, when we modeled the multimodal neural network, each initial neural network for a specific feature type is first learned, and then the final neural network is learned with previously built initial networks. This learning strategy can enhance the overall accuracy of the

TABLE II
THE NUMBER OF FILES THAT ARE CORRECTLY ANALYZED

File type	# of Correctly analyzed apps	# of Total apps
Dex file	31,277 (78%)	40,000
Manifest file	31,839 (78%)	40,000
.so file	3,630 (69%)	5,243

APKtool and IDA pro were used to extract the raw data

model as well as can reduce the training time in the situation that only a few initial network is required to be updated.

D. Processing the Zero-Padded Feature Vectors in Detection

Unlike the learning process, in the online detection, even though some feature vectors consist of only zeros, our framework can still process them normally if there is at least one non-zero feature vector. In our experiments, zero-padded feature vectors are not excluded in the detection, but our framework produced high detection accuracy. This is because the zero-padded feature vectors hardly affect the final decision since the initial networks that correspond to the zero-padded feature vectors generate minimum output values that only calculated with the biases of the neurons.

V. PERFORMANCE EVALUATION

A. Dataset

For the evaluation of our model, 20,000 malware samples from VirusShare [38] and 1,260 from the Malgenome project [37] were used. In addition, 20,000 benign samples downloaded from Google Play App Store [39] between March 2015 and April 2016 were used. To find out whether the downloaded applications are benign or not, the online malware scanning service of VirusTotal [40] was used. VirusTotal provides APIs for the scanning service. We utilized these APIs to request the tests and to obtain the scanning results. When all of the virus-scanners in VirusTotal consider an application as benign, the application is included in the benign application set.

We had many experiments for the performance evaluation, and the size and the source of samples were slightly different, so we summarized the information of datasets in each experiment at Tables XI and XII in APPENDIX C.

B. Experimental Environment

We used an Ubuntu 14.04 machine with Intel Core i7-5820k CPU, GeForce GTX TITAN X GPU and 104GB RAM to deploy our proposed framework. In the experiments, the GPU was utilized to accelerate machine learning algorithms including our multimodal neural network algorithm. We implemented the modules of the framework in Python; most of the modules were executed on PyPy Interpreter [41] that provides runtime optimizations. The modules that use libraries such as PyCuda are executed on naïve Python because those libraries do not support PyPy. The multimodal neural network modeling tool is implemented using the Keras library [42]. Scikit-learn [52] and Tensorflow [53] are also utilized to implement the clustering and other machine learning algorithms.

TABLE III

THE PORTION OF THE CLUSTERS THAT HAVE ONLY SAME LABEL

	Original family label	Merged family label	Malware or benign label
Shared lib. opcode freq.	74.47%	89.36%	97.87%
Method opcode freq.	73.61%	76.39%	91.67%
Method API freq.	95.53%	96.72%	98.36%
Manifest	62.60%	66.41%	79.39%
String	88.10%	88.10%	94.05%

Original family label: Malgenome project label, Merged family label: labels of which the sub-families are merged into one, Malware or Benign label: labels for distinguish malware or benign

C. Effectiveness of the Usage of Feature in the Framework

1) *Topological Data Analysis for the Selected Features*: We also conducted experiments to figure out how the features of our framework are useful. We performed the topological data analysis (TDA) [61] that is devised for the high-dimensional and noisy data. The malware samples used here had been analyzed in many previous studies, so it is easy to compare our proposed model with the other studies. In TDA, the original data are projected, and the projected data are clustered and visualized. Each cluster is expressed as a node, and a node's color and size are assigned by the cluster's property. The edge between the nodes represents that the connected cluster nodes have same intersected elements. Fig. 3 and Table III shows the results of the TDA. In the figure, each cluster node's size means the log-scaled number of the cluster members, and the color of the nodes means the portion of the labels of the cluster members. The color of the cluster node with malicious samples is red; otherwise, the color is blue. The clustering algorithm, DBSCAN [47] was used in TDA. As shown in Fig. 3, most of the cluster nodes are expressed as pure red and blue. This means that the feature data have a property that can distinguish malicious and benign samples. There are some orange or green nodes, but the number of these nodes is very small. To check the exact TDA clearly, we measured the portion of the clusters that have same labels. As shown in Table III, when we specified the original labels using the Malgenome project reports, the average portion of clusters with same labels was about 74.47%. Among the malicious samples, there are some mutually-related families like Droid-KungFu (1/2/3/4/ Sapp/Update). These samples share common malicious behavior, so their features are also similar. For this reason, we merged the labels of those samples into one and the average portions were measured as 83.40%. Lastly, when using the malware or benign labels, the average portion was 92.26%. According to the results, our features are effective to capture applications' characteristics for the malware detection.

2) *Effectiveness of the Feature Vector of the Framework*: Our framework uses complicated feature extraction methods to analyze malware. The following are two main reasons of our feature extractions. First, it was necessary to provide the method that generates the fixed sized feature vector that can be used in the detection model. The size of the raw data such as naïve binary files of each application varies greatly, so the resizing algorithms are necessary to provide the fixed sized feature vectors which fit in our neural network model. The previously proposed resizing algorithms utilize the spatial or temporal characteristics of images or speech waveforms,

and these resizing algorithms cannot be directly used in our model, so we developed our own resizing algorithms. Second, common characteristics of malware should be included in the feature vectors, so that we developed the similarity-based feature vector generation method.

To show the effectiveness of our feature vector generation method including feature extraction, we conducted experiments to compare our framework with other methods: the native binary-based detection method, the bag-of-words based detection method, and an open-sourced opcode sequence-based detection method [30]. The naïve binary-based method uses the binary files such as manifest files, dex files, .so files directly as feature vectors, and the bag-of-words based method uses the frequencies per feature (e.g. APIs, Dalvik opcodes, and ARM opcodes) as feature vectors. The opcode sequence-based method uses disassembled opcode sequences as feature vectors.

Table IV shows the comparison results of the experiments. The first row shows the experimental results when all possible feature types are used to generate feature vectors, and the second row shows the results when only the Android method opcode information is used. In both cases, our framework has higher accuracy than the other methods. Especially, the naïve binary-based detection and the bag-of-words based detection produced very low accuracies. The accuracy of the opcode sequence-based detection is about 3% lower than that of our framework with only Android method opcode information.

To find out the reason for these results, we analyzed the malware samples that were classified correctly in our framework but were misclassified commonly in the other detection methods. Among the Android methods used in the malware samples, we calculated the ratio of the Android methods which appear in both malware samples benign samples. The results showed that about 79.9% of Android methods in the malware samples also appear in benign samples. Since the other detection methods use dex files themselves or all the opcode sequences to generate feature vectors, these 79.9% Android methods also affect the feature vector generations. As a consequence, the detection rate is degraded due to common Android methods in both malware and benign applications.

In contrast, in our framework, these Android methods appeared in benign samples have similarity values less than a certain threshold with malware representatives, so these methods are not used to generate feature vectors. We calculated the ratio of the elements set as one in our similarity-based feature vectors of the misclassified malware samples by the other methods. Fig. 7 in APPENDIX D shows the standard normal distribution derived from the ratio results. This figure includes the ratio results of all the benign and malware samples to compare with the result of the misclassified malware samples by the other methods. As shown in the figure, we can find that the shape of the standard normal distribution of the misclassified malware samples is almost same with that of all the malware samples, whereas there is a definite difference between the shape of the standard normal distributions of all the benign samples and the misclassified malware samples. In conclusion, the feature vectors that are generated with naïve

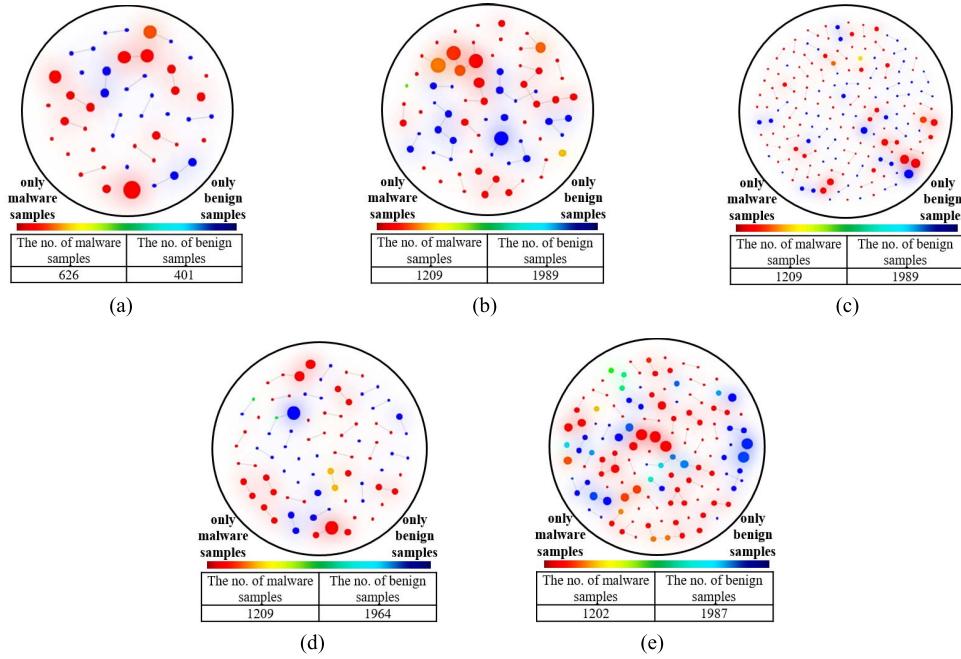


Fig. 3. Topological data analysis (TDA) result of each feature data. Density-based spatial clustering algorithm was utilized in the TDA. (a) - (e): the visualized result for each feature type. Malicious samples from Malgenome project were used. The size of dataset used in this experiment is described in Table XI.

TABLE IV
THE PERFORMANCE COMPARISON WITH OTHER FEATURE VECTOR GENERATION METHODS

Method	Basic Feature Type	Classification Model	Accuracy (%)
Ours	Manifest/String/Method Opcode Freq./Method API Freq./Share Lib. Func. Opcode Freq.	MNN	98%
Bin.-based detection	Manifest/Dex/So as like an image	MNN	43%
BoW-based detection	Manifest/String/App Dalvik Opcode Freq. /App API Freq. /App ARM Opcode Freq.	MNN	62%
Ours	Method Opcode Freq. (only sim based representation)	DNN	93%
Bin.-based detection	Dex as like an image	DNN	57%
BoW-based detection	Method Opcode Freq. (BoW representation)	DNN	60%
Opseq. based detection	Method Opcode Seq. as like an image	CNN	90%

Ours: our proposed detection, Bin.-based detection: the method that uses a naïve binary as a feature vector, BoW-based detection: the method that uses the frequencies of the words, Opseq-based detection: the method uses opcode sequences of an application as a feature vector. MNN: multimodal neural network, DNN: deep neural network, CNN: convolutional neural network. The size of dataset used in this experiment is described in Table XI

binary files or with simple refined raw data are not useful to distinguish malware and benign applications because those feature vectors can be dominated by common characteristics of benign applications. Our framework generates feature vectors not only to reflect the malware characteristics but also to exclude the benign characteristics by assigning zeros to the values of the features that have low similarity with the feature representatives of the malware. Due to this reason, it is expected that our method has shown high accuracy compared with others.

D. Effectiveness of the Multimodal Neural Model

1) Comparison With Different Neural Networks: To evaluate the effectiveness of the proposed multimodal neural network and its learning strategy, we measured the accuracies of three different neural network models. Our framework uses the five feature types for detection. For the evaluation, we measured the performance using all feature combinations as shown in Fig. 4.

The models used in the experiments are explained below:

- MNN-z: The multimodal neural network model trained with the zero-padded feature vectors

MNN-s

- (used by our framework): The multimodal neural network model separately trained only with the non-zero feature vectors
- DNN: The fully-connected deep neural network model trained with the zero-padded feature vectors

As shown in Fig. 4, when all feature types were used, each neural network model produced the maximum accuracy. In addition, the models detected malware more accurately when they used more feature types. As we expected, the performance of the multimodal neural model was best when the initial networks were trained separately. Especially, the performance differences with other models were increased whenever a new feature type is added. The trend lines of the histograms in Fig. 4 shows that the accuracy of the multimodal neural network model is improved significantly, compared with the other two models. Even though the number of features is same, accuracy results can be different depending on the combinations of features used. In our experiments, accuracy results were better when the manifest feature was used.

We checked whether the accuracy is increased in all the cases when a certain feature is added. This will show that multiple features should be used together to improve accuracy, as in the case of our framework. From this point of view,

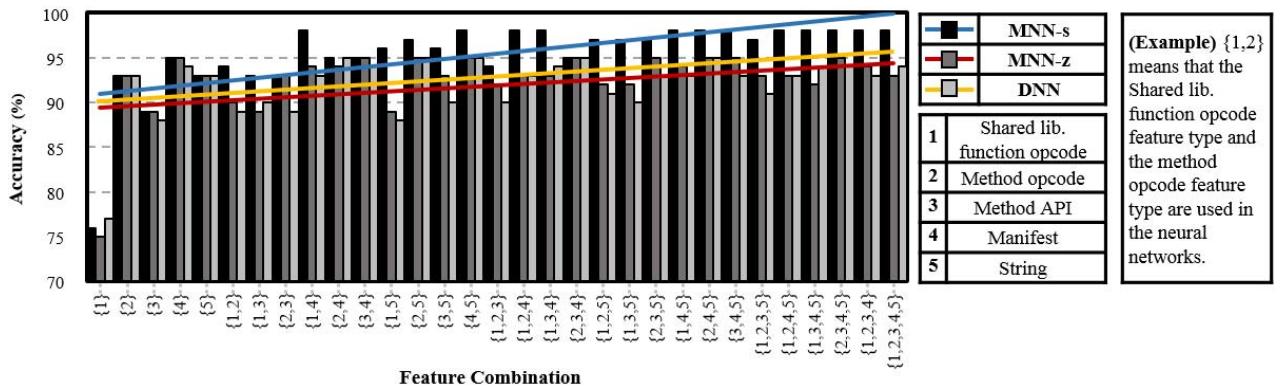


Fig. 4. Accuracy results of three different neural networks; Multimodal neural network tuned with zero padded vectors, multimodal neural network that uses the separately trained initial network, and deep neural network. All the neural networks have the same number of layers that have the same number of neurons. (Validation set/ test set: 20% each, training set: 60%, the size of dataset used in this experiment is described in Table XI.)

TABLE V
AVERAGE ACCURACY OF EACH MODEL BY
THE NUMBER OF FEATURE TYPES

# of Feature Types	Average Accuracy (%)		
	MNN-s	MNN-z	DNN
1	89.2	89	89
2	95.5	92.7	91.8
3	97	93.6	92.7
4	97.8	93.4	93
5	98	93	94

the experimental results showed that the MNN-s was most suitable to learn the various feature types altogether. The average accuracy of each model was also measured with respect to the number of feature types, and the results are in Table V. The average accuracy was about 89% when only one feature type was used, and the average accuracy was increased to about 98%, 93%, and 94% respectively when all feature types were used. In addition, there was no case that the accuracy is degraded even when a new feature type is added to our MNN-s model. In contrast, among 75 cases that a new feature type is added in the classification, we found 21 degradation cases in the MNN-z model and 17 degradation cases in the DNN model. Fig. 6 in APPENDIX D shows that our MNN-s consistently improves the accuracy as a new feature type is added, whereas the DNN model does not. In the figure, the experimental results of shallow learning models are also described to show the effect of the MNN-s more clearly.

Two reasons why the experimental results show that MNN-s is best can be inferred as follows: First, if features are not extracted from samples of either class (malware class or benign class), the corresponding feature vectors for those samples will be zero-padded accordingly. For these feature vectors, zero-padded feature vectors may affect the model during the learning process. Second, in the DNN model, neurons of the different feature types are connected with neurons in the next layer together. When the model is trained with malicious samples like Trojan that inject malicious code into benign applications, the neurons of the benign features and the malicious features will affect the neurons in the next layers together. This means that the characteristics from the given malicious samples are attenuated in the detection model during the learning process.

TABLE VI
THE LEARNING TIME FOR UPDATING THE MODEL

	MNN-s		DNN
	1,458 seconds		1,807 seconds
Case 1	F1 update	378 seconds	
Case 2	F2 update	560 seconds	
	F3 update	484 seconds	1,807 seconds
	F4 update	327 seconds	
	F5 update	579 seconds	

F1: shared lib opcode frequency feature vector, F2: method opcode frequency feature vector, F3: method API frequency feature vector, F4: manifest feature vector, F5: string feature vector

2) *Learning Time for Updating the Detection Model:* Many malicious applications or benign applications are created and distributed every day. Since the malware detection model should reflect the characteristics of those new applications for accurate and prompt detection, it is necessary to update the model continuously. In addition, if security experts want to add a new type of features, the detection model can be updated. Our proposed multimodal neural network model is designed to respond to new malicious applications flexibly and efficiently to handle these situations. The model can be updated by the changing the network that needs to be trained newly. To evaluate the efficiency of our multimodal neural network model, we compared the performance of our model with the deep neural network model in two cases while updating the detection model.

We considered the following two cases:

- Case 1: when all types of features can be extracted from given applications to update the model
- Case 2: when an additional feature type can be extracted from given applications to update or when an additional feature type is added to be used in the detection model

Table VI shows the learning time for model updating in two cases. For the deep neural network model, all networks of the model should be re-trained in both two cases. So, the learning times for the cases are equal naturally. For the multimodal neural network, the learning time for the case 1 is the sum of the training time for five initial networks and a final network. For the case 2, the only training time for one initial network and a final network is required in model updating.

As a result, it was confirmed that the time for model updating could be significantly reduced for the case 2. For

TABLE VII
THE PERFORMANCE COMPARISON WITH OTHER PROPOSED SYSTEMS (PAPER SURVEY)

System	Data set		Classification algorithm	Capacity for feature diversity	Accuracy (%) / F-measure
	Malware	Benign			
Ours	13,075 (V.S.) 1,209 (M.P.)	19,747 (G.P.) 1,300 (G.P.)	MNN-s (proposed)	High	98% / 0.99 99% / 0.99
Z. Yuan [28]	1,760 (C.C./M.P.)	20,000 (G.P.)	DBN	Low	96.8% / NA
W. Yu [29]	92 (M.P.)	96 (G.P.)	DNN/RNN (separately used)	Medium	90% / NA
N. Mchaughlin [30]	9,902 (M.P./Mc.)	9,268 (G.P./Mc.)	CNN	Low	87% / 0.86
H. Fereidooni [31]	18,677 (M.P./M0./ V.T./D.)	11,187 (D./M0.)	XGboost(best)/Adaboost/RF/SVM / K-NN/LR/NB /DT/DNN	Low	97% / 0.97 (XGboost)
Ch.-Y. Huang [2]	480 (M.P.)	124,769 (G.P./3 rd party)	Adaboost/NB/DT(best)/SVM	Low	NA / 0.78
M. Zhang [5]	2,200 (M.P./Mc.)	13,500 (Mc./G.P.)	NB	Low	93% / NA
D. Arp [7]	5,560 (M.P./3 rd party)	123,453 (G.P.)	SVM	Low	93.9% / NA
D-J. Wu [8]	238 (C.C.)	1,500 (G.P.)	K-NN+K-Means	Low	NA / 0.91
S. Y. Yerima [58]	1,000 (M.P.)	1,000 (G.P. or 3 rd party)	Bayesian based classifier	Low	92% / NA
Q. Jerome [59]	1,246 (M.P.)	1,260 (G.P.)	SVM	Low	NA / 0.98
S. Y. Yerima [60]	2,925 (Mc.)	3,938 (Mc.)	RF(best)/LR/NB/DT	Low	97.5% / NA (RF)

Abbreviation & Terms: Ours – our framework, MNN – multimodal neural network, DBN – deep belief network, DNN – deep neural network, RNN – recurrent neural network, CNN – convolutional neural network, RF – random forest, SVM – support vector machine, K-NN – K-nearest neighbor, DT – decision tree, LR – logistic regression, NB – Naïve Bayes, V.S. – VirusShare, M.P. – Malgenome project, C.C. – contagio community, G.P. – Googleplay store, Mc. – McAfee, M0 – M0Droid, V.T. – VirusTotal, D. – Drebin, 3rd party – third party market, M.P. – malicious repository.

example, the time when the manifest feature type is added was reduced by about 80% compared to that of the deep neural network model, i.e. case 1. In our experiments, a small set of samples were used in the model update; however, the number of applications that are collected newly every day is huge in practice. Therefore, more time is expected to be saved by training the multimodal neural network model because only the specific initial networks and the final network are updated.

E. Comparison With Other Existing Detection Methods

To show the performance of our framework compared with state-of-art detection systems, we investigated the similar approaches that have been previously proposed. From the deep learning-based methods to the general classification-based methods, various kinds of the Android malware detection methods were surveyed. Table VII shows the results of the investigations. Many existing methods utilize the malware samples from the Malgenome project. Therefore we included the performance results in the table when the samples from the Malgenome project were used in the detection test. Naturally, the accuracy and the F-measure values of previously explained experiments were also included together. As shown in Table VII, the detection accuracy or the F-measure values of our framework were higher than the other methods including the deep learning based methods [28]–[31]. Like DNN described in Section V-D, all the classification models treat the feature vectors consist of the different types of feature without any isolation processes. The method in [29] uses two classification models per different feature types, but there is no process to combine the results of each model. So, the comprehensive final decision cannot be derived from the classification models, and the method cannot properly handle the situations that the two models produce conflicting decisions.

As we explained in subsection 2 in Section V-C, we also measured the performance of the deep learning based detection method proposed in [30]. As shown in Table VIII, the detection

TABLE VIII
THE PERFORMANCE COMPARISON WITH [30]

	Precision	Recall	F-measure	Accuracy (%)
Ours	0.98	0.99	0.99	98%
[30]	0.90	0.84	0.87	90%

The size of dataset used in this experiment is described in Table XI.

accuracy of our framework was 8% higher than that of their proposed method. In detail, the precision, recall, F-measure, and accuracy of our framework were 0.08, 0.15, 0.12, and 8% higher each. As we described in the previous section, our framework generates feature vectors that are not dominated by the characteristics commonly appeared in both malware and benign applications while the method proposed in [30] uses naïve opcode sequences without such considerations. In addition, our framework uses various kinds of feature type to reflect the various aspects of Android applications and to deal with the situations like a certain raw data cannot be extracted properly. This is possible because our MNN model is designed to take advantage of these various feature types in terms of the accuracy and learning time efficiency.

F. Further Analysis to Evaluate the Framework

1) *Obfuscated Resilience of the Framework:* To show the obfuscation resilience of our proposed framework, we experimented with two sets of malware: a naïve malware set and an obfuscated malware set. Feature vectors are extracted from each malware set, and we calculated the similarities between them. We compared our similarity results with these three previous works such as ViewDroid [57], MassVet [58] and Droid-Sec [59]. In obfuscated malware generation, we applied the Android-specific obfuscation techniques that are introduced in [60] individually to samples in the naïve malware set to measure the impacts of each obfuscation technique. Six obfuscation techniques in [60] and one technique (PI) we defined were tested in the experiments.

Table IX shows the experimental results with the obfuscated malware samples. The average similarities of our framework

TABLE IX
THE AVERAGE SIMILARITY OF THE NAÏVE MALWARE AND THE OBFUSCATED MALWARE

Method	Call Indirection (CI)	Code Reordering (CR)	Data Encryption (ED)	Junk Insertion (JN)	Permission Insertion (PI)	Identifier Renaming (RI)	Package Renaming (RP)
Ours	1.0	0.99	0.93	0.92	0.99	0.99	1.0
ViewDroid	1.0	0.96	0.86	1.0	1.0	0.41	1.0
MassVet	1.0	0.0003	0.45	0.004	1.0	0.99	1.0
Droid-Sec	0.95	0.95	0.95	0.90	0.92	0.95	0.92

Ours: our proposed feature vector based similarity calculation (Jaccard coefficient similarity measure used), ViewDroid [61]: view graph based similarity calculation, MassVet [62]: CFG geometric center based similarity calculation (view graph based similarity calculation is excluded for the duplication, matching ratio is used as similarity), Droid-Sec [63]: Their proposed feature vector based similarity calculation (Permission/API(our dangerous API list)/Dropbox Behavior information based feature vector and Jaccard coefficient similarity measure were used), The size of dataset used in this experiment is described in Table XI. PI is a technique that inserts additional permissions to the manifest.

exceed 0.92, and the results show that our framework is as good as or better than previously proposed work. Our framework shows similar results with ViewDroid, and average similarities of the three obfuscation techniques, i.e. CR, ED, and RI, are slightly better than ViewDroid. These three obfuscation techniques do not change view graphs directly, but some view graphs may be no generated because view names and invoked functions related views are obfuscated. As a result, obfuscation causes some similarity degradations of ViewDroid. MassVet was implemented based on ViewDroid, and MassVet first analyzes malware with view graphs proposed in ViewDroid, and then analyzes reused code using their diff procedure. We already measured the performance of ViewDroid, so we only calculated the similarity, the code matching ratio using their diff procedure. MassVet uses features from control flow graphs (CFGs), and obfuscation techniques such as CR and JN affect these features. So, MassVet has very low similarity scores when these techniques are applied. Droid-Sec generates feature vectors based on frequencies of declared permissions, API invocations and dynamic behaviors. In most cases, the similarity results are more than 0.90, but the result of JN is lowest, because the JN technique may affect frequency information. Our approach is slightly better than Droid-Sec because we focused on features of centroid methods instead of frequency features of the whole application.

Our framework uses various kinds of feature, and the similarity-based feature vector generation process can alleviate impacts of the obfuscation techniques even though some features are affected by obfuscation techniques. Even though the experimental results show that our framework is effective against some obfuscation techniques, static analysis based approaches still have limitations if obfuscation techniques are developed for specific detection frameworks or methods.

2) *The Applicability on the Unsupervised Learning:* Our framework is devised for the malware detection using the supervised classification, however, we also had an experiment to evaluate the applicability of our proposed model to the unsupervised training-based classification. We implemented and compared two unsupervised learning-based classification models; The MNN shaped autoencoder and the DNN shaped autoencoder. The detailed information about these autoencoders is included in APPENDIX E. We followed the general unsupervised learning based classification approach that is introduced in [62]. Each autoencoder is trained to reconstruct positive data, i.e. malware only, and classification

is performed by comparing its reconstruction error to a threshold. If the error is smaller than the threshold, the input instance is assigned to the positive class. As shown in Table X, the detection accuracy of the MNN shaped autoencoder was about 6% higher than the DNN shaped autoencoder.

VI. RELATED WORK

A. Static Analysis

Huang *et al.* [2] proposed a malware detection method that uses the permission information and the number of files with common machine learning algorithms. Zarni and Zaw [3] developed a system that collects the required permissions from the applications. Their system applies clustering algorithm and classification algorithm step by step. DroidLegacy [4] decomposes malware into loosely coupled modules and matches the API call frequencies of each module. DroidSIFT [5] classifies malware and benign applications by analyzing API dependency graphs of each application. RiskRanker [6] analyses dangerous behaviors of the applications. DREBIN [7] uses features like hardware or software components, permissions, intents, and API calls to detect malware. DroidMat [8] uses the API call traces, intent message passing, and information of components (activity, service, and receiver) to detect malware. DroidMoss [9] is a system that uses a fuzzy hashing technique to measure similarities of applications to detect the repacked malware. Sum *et al.* [17] presented a method that uses keywords correlation distance in feature vector generation for the SVM classification. Narayanan *et al.* [18] proposed a method that uses CFGs as features, and the CFGs are applied to the online SVM algorithm for the malware detection. Demontis *et al.* [19] proposed a method to mitigate evasion attacks such as malware data manipulation. The method utilizes a secure SVM algorithm which enforces its features to have evenly-distributed weight. Compared with our framework, there are some differences that these methods use a small set of feature types and do not consider the flexibility of the detection model for feature type addition or partial model updates.

In addition to the above studies on signature-based detection, there are some studies to protect benign applications from malware. DroidAlarm [11] analyses to identify privilege escalation attacks. This system analyses execution paths in inter-procedural call graphs from a sensitive permission to a public interface. Chin *et al.* [12] proposed ComDroid that analyses the manifest file of an application to check

TABLE X
THE ACCURACY OF MNN SHAPED AE AND DNN SHAPED AE

	FPR	TPR	FNR	TNR	ACC
MNN	15%	85%	14%	96%	85%
DNN	17%	60%	40%	89%	79%

The size of dataset used in this experiment is described in Table XI.

whether security-critical declarations for application communications exist. CHEX [13], DroidChecker [14], AAPL [15], and AmAndroid [16] are methods to verify Android applications to defend against the component hijacking attacks. These studies focus on how to detect the potentially vulnerable points. The purpose of the researches is different from ours.

B. Dynamic Analysis

DroidScope [20] is a system that provides the semantic views for malware analysis. The system extracts system call traces as well as changes in environments such as processes and threads, and Dalvik instruction traces also are extracted and used in taint analysis in the system. TaintDroid [21] performs the data-flow analysis of applications and detects the information leakage of sensitive data. AASandbox [22] is a system that uses both static analysis and dynamic analysis. In its dynamic analysis, system call frequencies of benign applications and malware are collected and compared. Shabtai *et al.* [23] and Schmidt *et al.* [24] proposed the anomaly detection systems. Each system analyses mobile devices to identify abnormalities of the device usage. The system uses the metrics such as CPU usage, the number of sent packets, and so on. Even though our framework only uses static features currently, the dynamic features like those presented in the previous studies can be utilized additionally.

C. Deep Learning Based Malware Detection

Previous approaches that use deep learning algorithm are explained in turn. Pascanu *et al.* [25] uses the recurrent neural network to detect Windows-based malware. They use API events as the feature in detection. DeepSign [26] is a Windows-based malware detection method that uses dynamic API calls and their parameters as features. The deep belief network is used to classify malware and benign files. Saxe *et al.* [27] proposed a deep neural network-based malware detection method. In their method, PE import functions, strings, entropy and metadata of Windows binaries are used as features. The method uses different kinds of feature altogether, so it seems difficult to detect malware such as Trojan which has many features of normal programs. Droid-detector [28], a machine learning based method, is proposed to detect Android malware. This method extracts three types of features and uses them in the deep belief network. Yu *et al.* [29] proposed an Android malware detection system that models neural network with permissions and system call traces from applications. In their system, only permissions are used as static features, even though there is much information that can be used and tested for their usefulness in detection. McLaughlin *et al.* [30] proposed a system that uses a convolutional neural network (CNN) for the Android malware detection. In the system, raw opcode sequences of

applications are used as a feature without any refinement. Fereidooni *et al.* [31] proposed ANASTASIA, a system to detect Android malware using features such as intents, permissions, system commands, and API calls. The system uses many classifiers including the deep neural network. Even though many kinds of feature can be extracted from Android applications, most of the previous methods use a small number of kinds of feature in detection. In addition, the previous methods do not consider the situations when adding new feature types.

VII. CONCLUSION

In this paper, we propose a novel Android malware detection framework that utilizes many static features to reflect the properties of applications in various aspects. Total seven kinds of feature extracted by analyzing files such as a manifest file, a dex file, and a .so file from an APK file, and these features enrich the extracted information to express applications' characteristics. In addition, we suggested the effective feature vector generation method which is appropriate to detect malware that is similar to benign applications. Through our proposed feature representation, it is possible to prevent feature vector of malware from containing the common properties that appear in benign applications. Finally, we used the multimodal deep learning method, which is designed to deal with various kinds of feature type. Different types of feature are exclusively used to train the initial networks, and the results of the initial networks are subsequently used to train the final network. This architecture of the model is suitable for our framework to improve the malware detection accuracy. To the best of our knowledge, this research is the first application of the multimodal deep learning to the Android malware detection.

In the evaluation, we carried out many experiments. We compared the detection accuracy of many different detection models. And, we performed an experiment to demonstrate our detection model can be efficiently updated. In addition, we had experiments to confirm the usefulness of the feature and our proposed feature vector generation method. And we also carried out experiments about the applicability on the unsupervised learning based classification and the obfuscation resilience. As a result, our framework was effective enough to be used in the Android malware detection. The source code and other auxiliaries were released in [34].

APPENDIX A FEATURE VECTOR GENERATION ALGORITHMS

Algorithm 1 Existence Based Feature Vector Generation

Input: Feature set in DB, F_{db} & Feature set of an App, F_{app}

Output: A existence based feature vector

```

1:   feature_vector ← [0 | 0 | ... | 0]
2:   index ← 0
3:   for ∀f1 ∈ Fdb do // for all features in database
4:     if f1 ∈ Fapp then
5:       feature_vector[index] ← 1
6:   return feature_vector

```

Algorithm 2 Similarity Based Feature Vector Generation

```

Input: Feature set in DB,  $F_{db}$  & Feature set of an App,  $F_{app}$ 
Output: A similarity based feature vector
1:    $Centroids \leftarrow k\_means(k, F_{db})$  // preprocessing
2:    $feature\_vector \leftarrow [0 | 0 | \dots | 0]$ 
3:    $index \leftarrow 0$ 
4:   for  $\forall c \in Centroids$  do // for all centroids
5:      $min\_sim \leftarrow 0$ 
6:     for  $\forall f \in F_{app}$  do // for all features
7:        $dist \leftarrow get\_euclidean\_dist(c, f)$ 
8:        $sim \leftarrow 1/(dist+1)$ 
9:       if  $sim < min\_sim$  then
10:         $min\_sim \leftarrow sim$ 
11:       if  $min\_sum > threshold$  then
12:          $feature\_vector[index] \leftarrow 1$ 
13:       else
14:          $feature\_vector[index] \leftarrow 0$ 
15:        $index \leftarrow index+1$ 
16:   return  $feature\_vector$ 

```

APPENDIX B
MALICIOUS FEATURE DATABASE

	String	Permission	
	[hash, size, str]	[<uses-permission>, <permission>]	

Tables for the ex. based feature vector generation	Component		
	<activity>	<service>	<receiver>
	<intent-filter>	<intent-filter>	<provider>
	<intent-filter>
			<grant-uri-permission>
			...
Tables for the sim. based feature vector generation	Environment		
	<uses-features>	<uses-library>	<uses-sdk>

Tables for the ex. based feature vector generation	Method opcode freq.		Method API freq.
	move	goto	invoke

			sendmessage/socket
			...
			...
Tables for the sim. based feature vector generation	Shared lib. func. opcode freq.		
	bl	add	sub

Fig. 5. The structure of tables in the malicious feature DB. Tables for method opcode frequency, method API frequency, and shared library function opcode frequency contains the centroid in each row. The centroids are calculated by K-Means algorithm. The other tables contain the real values of the attributes.

APPENDIX C
DATASET INFORMATION
TABLE XI

THE DATASET INFORMATION USED IN ALL THE EXPERIMENTS

Exp. (Section)	Test	Validation	Train	Dataset size (malware/benign)	Dataset Source
V-C-1				1,260/2,000	M.P./G.P.
V-C-2		Not classification experiment		13,036/18,803 (in all the cases)	V.S./G.P.
V-D-1	20%	20%	60%	shown in Table XII	V.S./G.P.
V-D-2	0%	0%	100%	1,209/1,791	V.S./G.P.
V-E	20%	20%	60%	13,036/18,803 (comparison to [30])	V.S./G.P.
V-F-1		Not classification experiment		1,000/0	M.P./G.P.
V-F-2	20%	0%	80% (only malware)	1,209/1,791	V.S./G.P.

V.S – VirusShare, M.P. – Malgenome project, G.P. – Google Play Store

TABLE XII
THE SIZE OF DATASET USED IN THE EXPERIMENTS IN SECTION V-D

Feature Combination	Benign set	Malware set	Total
{1,2,4},{1,3,4},{1,4,5}, {1,2,3,4},{1,2,3,4,5} {1,2,4,5},{1,3,4,5}, {1,2,3,4,5}	19,747	13,075	32,822
{1,4}	18,831	12,593	31,424
{4}	18,756	12,521	31,277
{1,5},{1,2},{1,3}, {1,2,3},{1,2,5},{1,3,5}	18,815	13,036	31,851
{2,4},{3,4},{4,5}, {2,3,4},{2,4,5},{3,4,5} {1,2,3,5},{2,3,4,5}, {1}	19,741	13,075	32,816
{2},{3},{5}, {2,3},{2,5},{3,5}, {2,3,5}	1,463	2,167	3,630
	18,803	13,036	31,839

Feature number: 1 - Share lib. opcode feature, 2 – Method opcode feature, 3 – Method API feature, 4 – Manifest feature, 5 – String feature

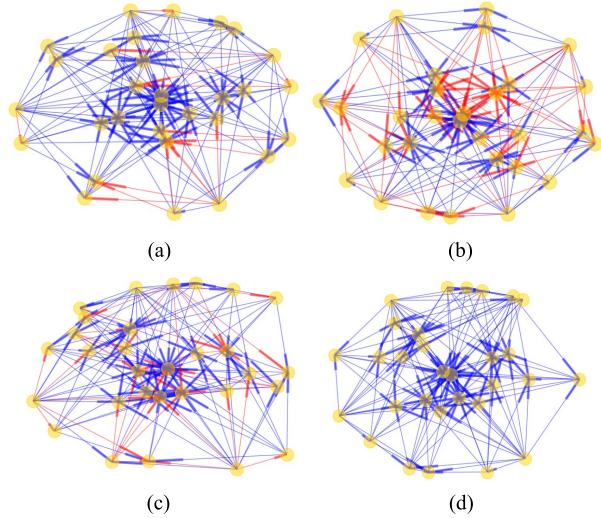
APPENDIX D
EXPERIMENTAL RESULTS

Fig. 6. The directed graph for detection accuracy changes according to feature type addition in random forest/support vector machine/deep neural network/multimodal neural network. Each node means the accuracy result of a certain feature combination, and only nodes in inclusion relation are connected with edges. The direction of edge is expressed as the bold line. Red edge means the accuracy decrease and blue edge means the accuracy increase (including equal accuracy). (a) Random forest. (b) Support vector machine. (c) Deep neural net. (d) Multimodal neural net.

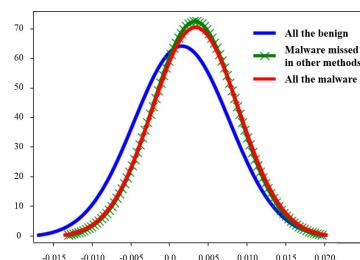


Fig. 7. The standard normal distribution of the total benign apps, the total malware, and the malware missed by binary-based detection method and bag-of-words based detection method. The standard normal distribution is computed using the ratio of the elements set as one in our similarity-based feature vector. The samples from VirusShare and Google Play Store were used.

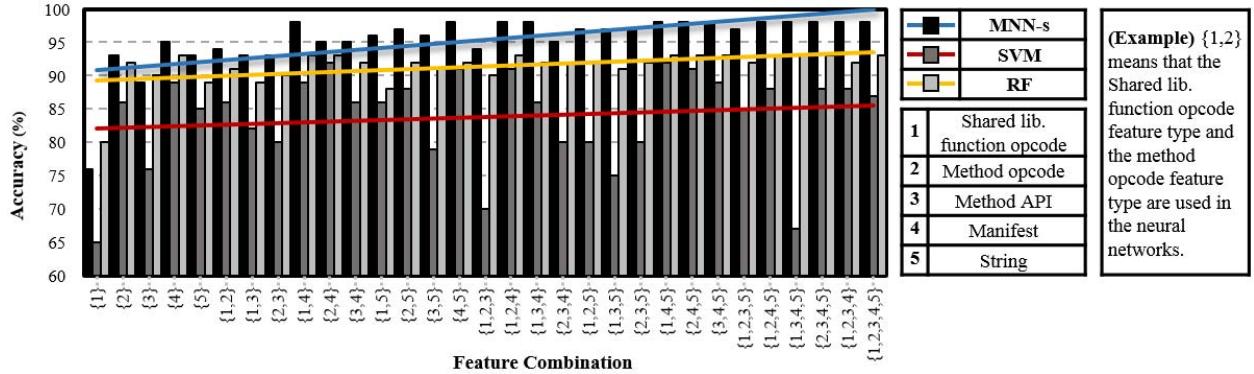


Fig. 8. Accuracy results of the multimodal neural network model, the support vector machine model, and the random forest model. (validation set/ test set: 20% each, training set: 60%, the size of dataset used in this experiment is described in Table XI)

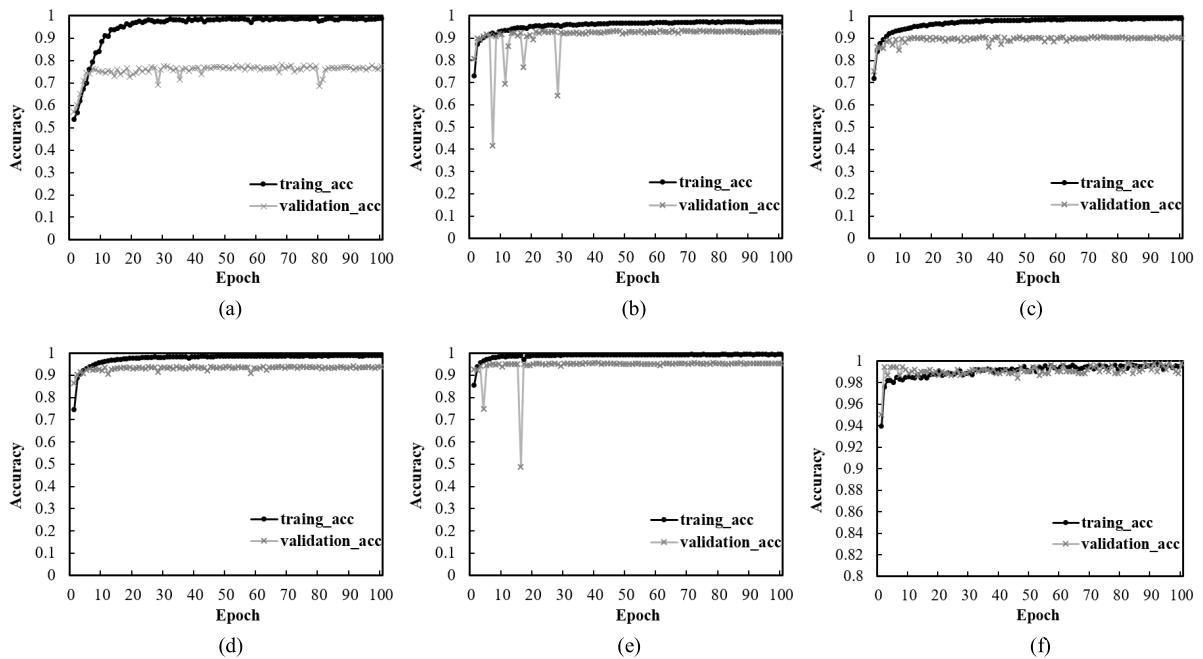


Fig. 9. Accuracy growth of the training set and the validation set by epochs. (a) - (e): initial networks of feature types, (f): final network (validation set: 20%, training set: 80%, the size of dataset used in this experiment is described in Table IX).

APPENDIX E AUTOENCODERS

TABLE XIII

THE STRUCTURE OF THE AUTOENCODERS

MNN shaped AE (No. of neurons)					DNN shaped AE (No. of neurons)
Init net1	Init net2	Init net3	Init net4	Init net5	
Layer1/1	200	200	200	200	1,000
Layer2/2	100	100	100	100	500
Layer3/3	20	20	20	20	100
Final network					
Layer1/4	500				500
Layer2/5	1,000				1000
Layer3/6	3,400				17,000

All the layers use the ReLU function except for the last layer. The last layer uses the sigmoid function.

REFERENCES

- [1] C. Lueg, “8,400 new Android malware samples every day,” G Data, Bochum, Germany, Tech. Rep., Apr. 2014. [Online]. Available: <https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day>
- [2] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu, “Performance evaluation on permission-based detection for Android malware,” in *Advances in Intelligent Systems and Applications* (Smart Innovation, Systems and Technologies), vol. 2. Berlin, Germany: Springer, 2013, pp. 111–120.
- [3] Z. Aung and W. Zaw, “Permission-based Android malware detection,” *Int. J. Sci. Technol. Res.*, vol. 2, no. 3, pp. 228–234, 2013.
- [4] L. Deshotels, V. Notani, and A. Lakhotia, “DroidLegacy: Automated familial classification of Android malware,” in *Proc. ACM SIGPLAN Program Protection Reverse Eng. Workshop*, 2014, Art. no. 3.
- [5] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, “Semantics-aware Android malware classification using weighted contextual API dependency graphs,” in *Proc. ACM Conf. Comput. Commun. Secur. (CCS)*, 2014, pp. 1105–1116.
- [6] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, “RiskRanker: Scalable and accurate zero-day Android malware detection,” in *Proc. ACM 10th Int. Conf. Mobile Syst., Appl., Service (Mobicys)*, 2012, pp. 281–294.

- [7] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, “DREBIN: Effective and explainable detection of Android malware in your pocket,” in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, vol. 14, 2014, pp. 23–26.
- [8] 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.
- [9] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, “Detecting repackaged smartphone applications in third-party Android marketplaces,” in *Proc. ACM Conf. Data Appl. Secur. Privacy*, 2012, pp. 317–326.
- [10] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan, “PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps,” in *Proc. ACM Int. Conf. Mobile Syst., Appl., Services (MobiSys)*, 2014, pp. 204–217.
- [11] Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, “DroidAlarm: An all-sided static analysis tool for Android privilege-escalation malware,” in *Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Secur.*, 2013, pp. 353–358.
- [12] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, “Analyzing inter-application communication in Android,” in *Proc. 9th Int. Conf. Mobile Syst., Appl., Services*, 2011, pp. 239–252.
- [13] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “CHEX: Statically vetting Android apps for component hijacking vulnerabilities,” in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 229–240.
- [14] P. P. F. Chan, L. C. K. Hui, and S. M. Yiu, “DroidChecker: Analyzing Android applications for capability leak,” in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2012, pp. 125–136.
- [15] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, and L. Lu, “Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting,” in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2015, pp. 4:1–4:15.
- [16] F. Wei, S. Roy, X. Ou, and R. Robby, “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps,” in *Proc. ACM Conf. Comput. Commun. Secur.*, 2014, pp. 1329–1341.
- [17] J. Sun, K. Yan, X. Liu, C. Yang, and Y. Fu, “Malware detection on Android smartphones using keywords vector and SVM,” in *Proc. IEEE/ACIS 16th Int. Conf. Comput. Inf. Sci.*, May 2017, pp. 833–838.
- [18] A. Narayanan, L. Yang, L. Chen, and L. Jinliang, “Adaptive and scalable Android malware detection through Online learning,” in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2016, pp. 2484–2491.
- [19] A. Demontis et al., “Yes, machine learning can be more secure! A case study on Android malware detection,” *IEEE Trans. Dependable Secure Comput.*, to be published.
- [20] L. K. Yan and H. Yin, “DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis,” in *Proc. 21st USENIX Secur. Symp.*, 2012, pp. 569–584.
- [21] W. Enck et al., “TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” *ACM Trans. Comput. Syst.*, vol. 32, no. 2, p. 5, 2014.
- [22] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, “An Android application sandbox system for suspicious software detection,” in *Proc. 5th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2010, pp. 55–62.
- [23] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, “Andromaly: A behavioral malware detection framework for Android devices,” *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, 2012.
- [24] A.-D. Schmidt, F. Peters, F. Lamour, C. Scheel, S. A. Çamtepe, and S. Albayrak, “Monitoring smartphones for anomaly detection,” *Mobile Netw. Appl.*, vol. 14, no. 1, pp. 92–106, 2009.
- [25] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, “Malware classification with recurrent networks,” in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2015, pp. 1916–1920.
- [26] O. E. David and N. S. Netanyahu, “DeepSign: Deep learning for automatic malware signature generation and classification,” in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2015, pp. 1–8.
- [27] J. Saxe and K. Berlin, “Deep neural network based malware detection using two dimensional binary program features,” in *Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2015, pp. 11–20.
- [28] Z. Yuan, Y. Lu, and Y. Xue, “Droiddetector: Android malware characterization and detection using deep learning,” *Tsinghua Sci. Technol.*, vol. 21, no. 1, pp. 114–123, Feb. 2016.
- [29] W. Yu, L. Ge, G. Xu, and X. Fu, “Towards neural network based malware detection on Android mobile devices,” in *Cybersecurity Systems for Human Cognition Augmentation*. Cham, Switzerland: Springer, 2014, pp. 99–117.
- [30] N. McLaughlin et al., “Deep Android malware detection,” in *Proc. ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, 2017, pp. 301–308.
- [31] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, “ANASTASIA: Android malware detection using static analysis of applications,” in *Proc. 8th IFIP Int. Conf. New Technol., Mobility Secur.*, Nov. 2016, pp. 1–5.
- [32] *APKtool*. Accessed: Sep. 2017. [Online]. Available: <https://ibotpeaches.github.io/APKtool>
- [33] *IDA Pro*. Accessed: Sep. 2017. [Online]. Available: <https://www.hex-rays.com/products/ida>
- [34] *Released Code*. Accessed: Sep. 2017. [Online]. Available: <https://github.com/cloudio17/A-Multimodal-Deep-Learning-Method-for-Android-Malware-Detection>
- [35] Y. Aafer, W. Du, and H. Yin, “DroidAPIMiner: Mining API-level features for robust malware detection in Android,” in *Proc. Int. Conf. Secur. Privacy Commun. Syst.*, 2013, pp. 86–103.
- [36] Y. Bengio, “Learning deep architectures for AI,” *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, 2009.
- [37] *Mal-Genome Project*. Accessed: Sep. 2017. [Online]. Available: <http://www.Malgenomeproject.org>
- [38] *VirusShare*. Accessed: Sep. 2017. [Online]. Available: <https://virusshare.com>
- [39] *Google Play Store*. Accessed: Sep. 2017. [Online]. Available: <https://play.google.com/store>
- [40] *VirusTotal*. Accessed: Sep. 2017. [Online]. Available: <https://www.virustotal.com/ko>
- [41] *PyPy Interpreter*. Accessed: Sep. 2017. [Online]. Available: <http://pypy.org>
- [42] *Keras*. Accessed: Sep. 2017. [Online]. Available: <https://keras.io>
- [43] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, “N-Gram-based detection of new malicious code,” in *Proc. 28th Annu. Int. Comput. Softw. Appl. Conf.*, vol. 2, Sep. 2004, pp. 41–42.
- [44] I. B. Mohamad and D. Usman, “Standardization and its effects on K-means clustering algorithm,” *Res. J. Appl. Sci., Eng. Technol.*, vol. 6, no. 17, pp. 3299–3303, 2013.
- [45] C. Robert, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA, USA: MIT Press, 2012.
- [46] D. P. Kingma and J. Ba. (2014). “Adam: A method for stochastic optimization.” [Online]. Available: <https://arxiv.org/abs/1412.6980>
- [47] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise,” in *Proc. KDD*, vol. 96, 1996, pp. 226–231.
- [48] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng, “Multimodal deep learning,” in *Proc. 28th Int. Conf. Mach. Learn.*, 2011, pp. 689–696.
- [49] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.
- [50] *Android Developer Reference Page*. Accessed: Sep. 2017. [Online]. Available: <https://developer.android.com/reference/packages.html>
- [51] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proc. 27th Int. Conf. Mach. Learn.*, 2010, pp. 807–814.
- [52] *Scikit-Learn*. Accessed: Sep. 2017. [Online]. Available: <https://scikit-learn.org>
- [53] *TensorFlow*. Accessed: Sep. 2017. [Online]. Available: <https://tensorflow.org>
- [54] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik, “A new Android malware detection approach using Bayesian classification,” in *Proc. IEEE 27th Int. Conf. Adv. Inf. Netw. Appl.*, Mar. 2013, pp. 121–128.
- [55] Q. Jerome, K. Allix, R. State, and T. Engel, “Using opcode-sequences to detect malicious Android applications,” in *Proc. IEEE Int. Conf. Commun.*, Jun. 2014, pp. 914–919.
- [56] S. Y. Yerima, S. Sezer, and I. Muttik, “High accuracy Android malware detection using ensemble learning,” *IET Inf. Secur.*, vol. 9, no. 6, pp. 313–320, 2015.
- [57] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, “ViewDroid: Towards obfuscation-resilient mobile application repackaging detection,” in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2014, pp. 25–36.
- [58] K. Chen et al., “Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale,” in *Proc. USENIX Conf. Secur. Symp.*, 2014, pp. 659–674.
- [59] 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.

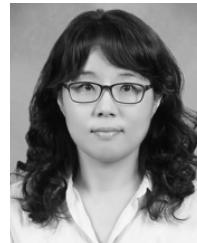
- [60] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android anti-malware against transformation attacks," in *Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Secur.*, 2013, pp. 329–334.
- [61] A. Zomorodian, "Topological data analysis," *Adv. Appl. Comput. Topology*, vol. 70, pp. 1–39, Jul. 2012.
- [62] J. Nathalie, "Supervised versus unsupervised binary-learning by feed-forward neural networks," *Mach. Learn.*, vol. 42, nos. 1–2, pp. 97–122, 2001.



TaeGuen Kim received the B.S. degree in electronics and computer engineering and the M.S. degree in computer and software from Hanyang University, South Korea, in 2011 and 2013, respectively. His research interests include malware analysis and machine learning.



BooJoong Kang received the B.S., M.S., and Ph.D. degrees in electronics and computer engineering from Hanyang University, South Korea, in 2007, 2009, and 2013, respectively. He is currently a Research Fellow at the Centre for Secure Information Technologies, Queen's University of Belfast, U.K. His research interests include malware analysis, threat analysis, intrusion detection, and cyber-physical resilience measures.



Mina Rho received the B.S. degree in computer science from Ewha Womans University, Seoul, South Korea, the M.S. degree in computer engineering from Boston University, Boston, MA, USA, and the Ph.D. degree in computer science from Indiana University, Bloomington, IN, USA. From 2012 to 2013, she was an Assistant Professor at the Roswell Park Cancer Institute, Buffalo, NY, USA. She is a Faculty Member with the Department of Computer Science and Engineering, Hanyang University, Seoul. Her research interests include the development of algorithms for analyzing genomic sequence data, and application of machine learning to analyze big data.



Sakir Sezer received the Ph.D. degree in electronic engineering from Queen's University of Belfast. He is currently a Professor and the Head of network and cyber-security research with the Center for Secure Information Technologies, Queen's University of Belfast. He is also a Co-Founder and the CTO of Titan IC Systems and a member of various research and executive committees, including the IEEE International System-on-Chip Conference Executive Committee. His research is leading major advances in the field of high-performance content and security processing.



Eul Gyu Im received the B.S. and M.S. degrees from Seoul National University in 1992 and 1994, respectively, and the Ph.D. degree from the University of Southern California in 2002. He is a Faculty Member of the Division of Computer Science and Engineering, Hanyang University, Seoul, South Korea. Before joining Hanyang University, he was with the National Security Research Institute, Daejeon, South Korea. His research interests include malware analysis, malicious traffic analysis, and smart grid security.