HADM: Hybrid Analysis for Detection of Malware

Lifan Xu
University of Delaware
xulifan@udel.edu

Dongping Zhang
AMD Research
dongping.zhang@amd.com

Nuwan Jayasena
AMD Research
nuwan.jayasena@amd.com

John Cavazos
University of Delaware
cavazos@udel.edu

***Abstract*—Android is the most popular mobile operating system with a market share of over 80% [1]. Due to its popularity and also its open source nature, Android is now the platform most targeted by malware, creating an urgent need for effective defense mechanisms to protect Android-enabled devices.**

In this paper, we propose a novel Android malware classification method called HADM, Hybrid Analysis for Detection of Malware. We first extract static and dynamic information, and convert this information into vector-based representations. It has been shown that combining advanced features derived by deep learning with the original features provides significant gains [2]. Therefore, we feed both the original dynamic and static feature vector sets to a Deep Neural Network (DNN) which outputs a new set of features. These features are then concatenated with the original features to construct DNN vector sets. Different kernels are then applied onto the DNN vector sets. We also convert the dynamic information into graph-based representations and apply graph kernels onto the graph sets. Learning results from various vector and graph feature sets are combined using hierarchical Multiple Kernel Learning (MKL) [3] to build a final hybrid classifier.

I. Introduction

With over 260 million shipments, Android has dominated the smart phone market with a 78.0% share in the first quarter of 2015 [4]. Unfortunately, the growing popularity of Android smart phones and tablets has made this popular OS a prime target for security attacks. In 2014, nearly one million unique malicious applications were produced, a 391% increase from 2013. Some estimates say that Android has been targeted by 97% of the developed mobile malware [5], creating an urgent need for effective defense mechanisms to protect Android-enabled devices.

Researchers have proposed various characterization methods to counter the increasing amount and sophistication of Android malware. These methods can be categorized into: static analysis, dynamic analysis, and hybrid techniques. Static analysis is based on extracting features by inspecting an application's manifest and disassembled code [6], [7], [8], [9], [10]. By contrast, dynamic analysis methods monitor the application's behavior during its execution [11], [12], [13], [14], [15], [16], [17]. Hybrid methods typically analyze an application before installation and also record its execution behavior [18], [19], [20], [21], [22], [23], [24]. These sets of static and dynamic information are then used together to detect malicious behavior. Static analysis is usually lightweight and can be performed on a user's device while dynamic analysis is usually performed in an offline emulator due to simulation overhead. Static and dynamic analysis both have their disadvantages. Static analysis techniques can be defeated by malware packing and other malware obfuscation techniques. On the other hand, dynamic analysis techniques can be defeated if the malware notices it is running in an emulator or sandboxed environment [25]. The hybrid analysis method is gaining more popularity for its combined advantages from both static and dynamic analysis and its capability to yield better accuracy in detecting malware.

In this paper, we propose **HADM**, **H**ybrid **A**nalysis for **D**etection of **M**alware. . We first extract a set of static and dynamic features. For static features, they are converted into vector-based representations. For dynamic features, in particular system call invocations, they are converted into vector-based and graph-based representations. Then, for all vector-based representations, we apply deep learning techniques to train a neural network for each of the vector set. Deep learning has been widely studied and shown to perform well on machine learning domains including speech recognition, natural language processing, and image classification in the past two decades. In our method, we train one Deep Neural Network (DNN) constructed by stacking Restricted Boltzmann Machines (RBM) for each of our feature vector sets including system call feature vectors and static feature vectors. The DNN learned features are concatenated to the original features to form the new DNN feature vector sets. Experiments show that higher level features learned from DNN in conjunction with the original features can improve the classification accuracy of each individual feature vector set. Different kernels are then applied on the new DNN feature vector sets to compute similarities of the Android applications. Similarly, different graph kernels are applied on the graph feature sets. The similarity output from each vector kernel or graph kernel can be subsequently constructed as a kernel matrix and fed into a machine learning model, e.g., Support Vector Machine (SVM), for classification. In HADM, a two-level MKL is applied to combine the discriminative power of different kernel matrices. In the first level, kernel matrices from different kernels are combined as the learning result of the corresponding feature vector set. Similarly, kernel matrices from different graph kernels are combined. In the second level, MKL is applied again to combine all learning results from the first level. The final kernel matrix is then fed into an SVM to construct our hybrid classification model. Figure 1 shows the framework of our HADM method.

For static analysis, we extract nine different features including requested permissions, permission request APIs, used permissions, advertising networks, intent filters, suspicious calls, network APIs, providers, and low level instruction sequences. For dynamic analysis, we run each application in an Android

emulator named Genymotion[1], and collect its system call invocations using a Linux utility called *strace*. Among these features, the instruction sequences and system call sequences are represented using n-gram vectors and all the other features are converted into histograms that are essentially 1-grams. In addition, the system call invocations are also converted into n-gram graphs. For the n-gram vectors and n-gram graphs, we evaluate four different n values including 1, 2, 3, and 4. As a result, we generate four vector sets for instruction sequences, four vector and four graph sets for system call sequences, and one vector set for each of the other features. The 16 feature vector sets are subsequently fed into DNN training and combined with the 4 graph sets using hierarchical MKL at the end.

To evaluate the performance of HADM, we collected thousands of Android applications across all categories of Google Play. We also collected a large number of Android malware from VirusShare [2]. In our dataset, 4002 samples are categorized as benign applications and 1886 samples are categorized as malware. Experiments on this dataset show, for dynamic features, the best classification accuracy that can be achieved is 83.3% by feature-vector-based representations and 87.3% by graph-based representations. On average, graph-based representations are able to achieve 5.2% absolute classification accuracy improvement over the feature-vector-based representations. For original static feature vector sets, the best classification accuracy that can be achieved is 93.5%. Finally, by applying hierarchical MKL, classification accuracy of the final hybrid classifier is further improved to 94.7%.

Our major contributions are summarized below:

- We propose HADM, a hybrid Android malware classification method utilizing multiple feature sets and different representations.
- We apply deep learning to learn new features for the vector sets and concatenate new features with the original features to boost classification accuracy.
- We apply hierarchical MKL to combine different kernel learning results from different features and thus further improve classification accuracy.

This paper is organized as follows. In Section 2, we present different feature sets and representations. In Section 3, we introduce the deep learning model used in our method. In Section 4, machine learning models used for classification are described. Section 5 presents our experimentation and the analysis of the results. In Section 6, we present an overview of related work. Finally in Section 7, we discuss our conclusions and future work.

II. Feature Sets

In total, 10 static and dynamic feature sets are extracted from our malicious and benign Android applications including requested permissions, permission request APIs, used permissions, advertising networks, intent filters, suspicious calls, network APIs, providers, instruction sequences, and system call sequences. Among them, instruction sequences and system call sequences are converted into 1, 2, 3, and 4-gram vectors. The other non-sequence features are represented using 1-gram vectors. For system call sequences, we also convert them into 1, 2, 3, and 4-gram graphs. In total, we generate 16 feature vector sets and 4 graph sets.

A. Hybrid Analysis Features

Requested permissions: Permission system is the first barrier and one of the most important security mechanisms introduced by Android. Therefore, the requested permission is one of the most used static features in Android application analysis [26]. Prior to installation of an application, it provides users with a list of requested permissions (e.g., *SEND_SMS, RECEIVE_SMS, INSTALL_PACKAGE*). Users normally grant the permissions without knowledge of the capabilities of these permissions, therefore an application can install itself and perform malicious behaviors such as sending premium SMS messages. In our experiments, we collect 1304 requested permissions listed in manifest files of our Android samples.

Permission request APIs: The Android permission can be requested by a series of critical API calls. For example, a *installPackage* API call can request permission *INSTALL_PACKAGE* and a *sendDataMessage* call requests permission *SEND_SMS*. In total, 246 such API calls are collected from our benign and malicious samples.

Used permissions: Some Android applications request multiple permissions, but only use a subset of the requested permissions. By extracting the used permissions, we can obtain a more precise observation of an application's intention. In total 66 used permissions are collected from our dataset.

Advertising networks: Advertising networks are increasing in numbers in the Android platform to offer developers a variety of monetization models and to help them maximize their revenues. This feature may not be necessarily related to malicious behaviors, but we collect 76 different advertising networks from our samples. The most popular networks are Google Ads, AdMob, and MobClik.

Intent filters: Intent is information about inter-process and intra-process communication. It is a passive data structure holding an abstract description of an action to be performed. Therefore, we can infer it as the intentions of the application. For example, an application can take a picture or can dial a phone number. In total, we collect 1016 different intent filters from our dataset.

Suspicious calls: A subset of API calls is capable of accessing sensitive data, communicating over the network, sending and receiving messages, and executing external commands. These suspicious API calls are frequently used by malware developers. For example, *readSMS* can read SMS messages, *sendSMS* can send SMS messages, *getCellLocation* is able to get your location, *Runtime→exec* is able to execute external commands, and *System→load* is able to load external libraries. In total, we collect 394 such calls.

Network APIs: We extract the used network APIs because malware tends to access the network and send out sensitive data. For instance, *android.net.wifi.STATE_CHANGE* broadcasts an intent action indicating that the state of Wi-Fi connectivity has changed;

[1]http://www.genymotion.com
[2]http://virusshare.com

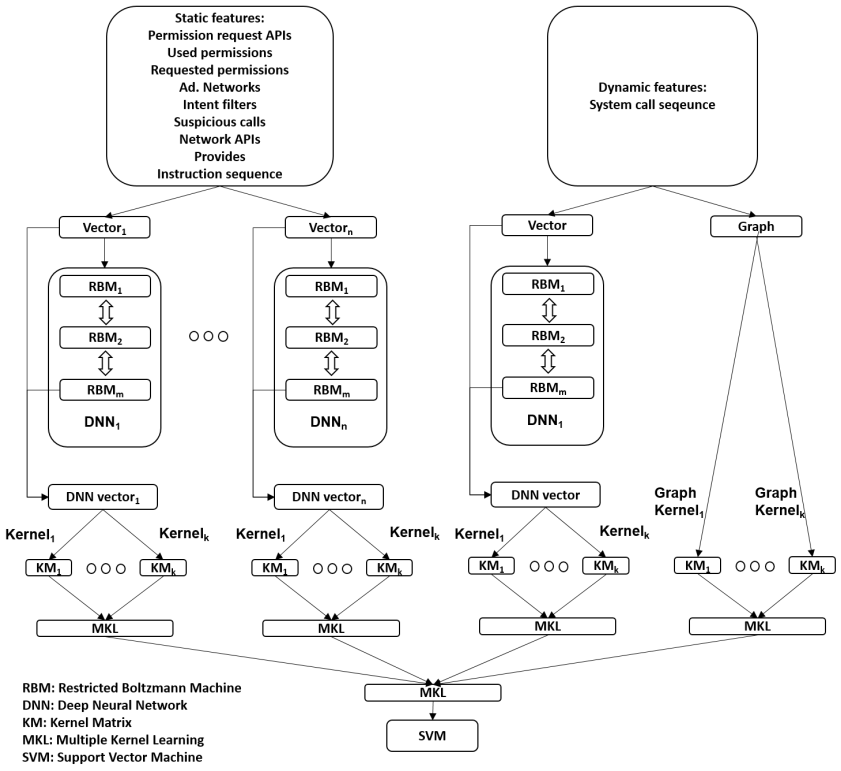


Fig. 1: This figure shows framework of HADM. Static features are converted to feature vector representations and dynamic features are converted to feature vector and graph representations. Each feature vector set is fed into a DNN for learning. The DNN features are concatenated with the original feature vectors to construct DNN feature vector sets. Multiple kernels and graph kernels are applied to each DNN or graph feature set. The learning results are then combined using a two-level MKL.

android.net.wifi.supplicant.CONNECTION_CHANGE notices both connections to and disconnections from a wifi network. In total, we collect 29 such APIs from our samples.

Providers: The provider declares a component which supplies structured access to data managed by the application. For example, *android.provider.Telephony.SMS_RECEIVED* broadcasts that a new text-based SMS message has been received by the device and this intent will be delivered to all registered receivers as a notification. In total, we are able to collect 966 providers from our samples.

Instruction sequences: We utilize an Android tool called Androguard[3] to extract low level instructions (also known as Dalvik bytecode) from an application. For each instruction, we keep only its name, while parameters and output are abandoned. We are able to collect 159 unique instructions from our samples.

System call sequences: For dynamic analysis, system calls are the most used features [26]. To capture runtime execution behaviors of an application, we record the system call invocations during execution of the application using the Linux strace tool [4]. The applications are emulated in an Android emulator named Genymotion. To record all the system call invocations, we run strace on the $zygote$ process. It starts during the Android initialization and is used to launch applications. We then execute the testing application. During the emulation, an application is first executed without user interaction for 20 seconds. We then stimulate hundreds of events generated by the Monkey toolkit provided by the Android SDK. Monkey is able to generate different types of events including touch events, motion events, trackball events, navigation events, system key events, and activity launching events. Other than the stimulation from Monkey, we feed the emulation additional phone call, SMS message, and movement events. All system call invocations in the emulation are recorded by strace. From a strace log of the $zygote$ process, we extract invocations only belong to the testing application. Similarly, the parameter and output are ignored. In total, we are able to collect 213 unique system calls.

B. Feature Vector Representations

After extracting the features, we embed them into vector space using an n-gram representation. An n-gram is a contiguous sequence of n items from a given sequence of features. There are two parameters associated with n-gram: n as the number of items in the sequence, and L as the number of unique items in the feature set. Given n and L, there can be L^n unique n-grams. Therefore, the dimension of the resulting n-gram feature vector goes exponentially as we increase the n value.

In HADM, we first build 1-gram vectors for all features. Then for instruction and system call sequences, we extract the

[3]https://code.google.com/p/androguard/
[4]http://linux.die.net/man/1/strace

top 20 instructions and system calls, and build 2-gram, 3-gram, and 4-gram vectors for both. In total, we generate four feature vector sets for instruction sequences, four feature vector sets for system call sequences, and one feature vector set for each of the other features. The 12 static and 4 dynamic feature vector sets are inputs for subsequent deep learning.

C. Graph Representation

For dynamic system call invocations, we also convert them into a graph-based representation called the n-gram graph. To construct the graph, a process tree of the Android application is first extracted from the strace log. Each process is represented as a vertex and connected with its child processes. Then, for each vertex, we collect system call invocations belonging to the corresponding process, and convert them to an n-gram vector. The resulting n-gram vector is attached to the vertex as its label. In total, we generate 1-gram, 2-gram, 3-gram, and 4-gram graphs for the system call sequences.

III. Deep Learning Model

Deep learning has shown promise in speech recognition, image classification, and other machine learning domains. It has also been shown that combining advanced features derived by deep learning with the original features provides significant gains. For example, Sarikaya et al. obtained 0.1% to 1.9% absolute classification accuracy improvements on a problem of natural language understanding using combined features [2]. After generating 16 feature vector sets, we train one Deep Neural Network (DNN) for each of the vector sets. Then the DNN learned features are concatenated with the original feature vectors and used for classification.

In our experiment, we select Deep Auto-encoder as our deep learning model. It is a DNN whose output target is the input data itself which serves our purpose of learning new features and combining new features with the original features for classification. The building block of a deep auto-encoder is a probabilistic model called Restricted Boltzmann Machine (RBM). DNN is often initialized or pre-trained using stacked RBMs. In some literature, DNN is also referred to as Deep Belief Network (DBN) [27].

A. Restricted Boltzmann Machine

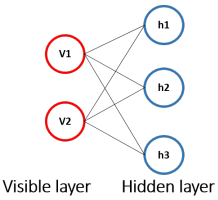
An RBM is an energy-based generative model that consists of two layers: a layer of binary visible units v and a layer of binary hidden units h. The units in different layers are fully connected with no connection between units in the same layer. Figure 2(a) shows an RBM with 2 units in the visible layer and 3 units in the hidden layer.

Details of how to train an RBM can be found in [28]. In our experiments, the standard Contrastive Divergence (CD) learning procedure is applied. In the training process, input vectors are first divided into batches and then fed into a training process for a number of iterations until convergence. The training process consists of three steps. The first step is called positive or forward propagation. In this step, probabilities of hidden units are sampled from the input and the positive gradient is computed. The second step is negative or backward propagation where the visible units are reconstructed from the hidden units and then the hidden activities are re-sampled from the reconstructed visible units. The negative gradient is also computed in this step. In the third step, the weight matrix is updated based on the difference of the positive gradient and the negative gradient. Algorithm 1 shows the training process of RBM.

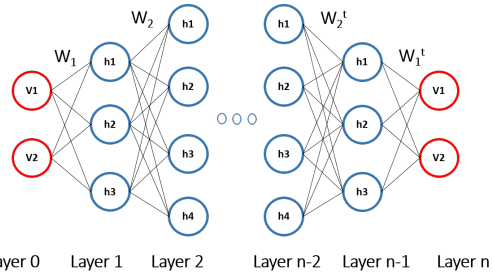
Algorithm 1 RBM Training Process

Input: batch set, Weight matrix W, learning rate l
Output: Weight matrix W

for number of iterations **do**
  for number of batches **do**
    From batch V, compute hidden activation $H = V \times W$
    Compute positive gradient $G_p = V \times H$
    From H, sample a reconstruction $V^{'} = H \times W$
    From $V^{'}$, re-sample hidden activation $H^{'} = V^{'} \times W$
    Compute negative gradient $G_n = V^{'} \times H^{'}$
    Update weight matrix W, $W+ = l \times (G_p - G_n)$
  end for
end for



(a) RBM



(b) Deep Auto-encoder

Fig. 2: An example of RBM and Deep Auto-encoder. (a): a RBM with 2 units in the visible layer and 3 units in the hidden layer. (b): Deep auto-encoder constructed by flipping the stacked RBMs.

In our experiments, we use the same parameters for training RBMs as used in [29] . The RBMs are initialized with very small random weights and trained for 80 iterations using mini-batches of size 128. The learning rate is set to be 0.001. We also use a momentum of 0.9 to speedup the learning process. All the training in our experiments are performed on an AMD Radeon™ HD 7970 GPU.

B. Deep Auto-encoder

Deep auto-encoder can be constructed by stacking independently trained RBMs. Each RBM is stacked on top of previous RBM such that the hidden layer of previous RBM become the visible layer of the current RBM. Each new layer of deep auto-encoder aims to extract higher-level dependencies between the original input vectors, thereby improving the ability of the network to capture the underlying regularities in the data [30]. The first layer of the network is expected to extract low-level features from the input vectors while each new layer is expected to gradually refine previously learned concepts, and therefore produce more abstract concepts [31].

After layer-by-layer pre-training the RBMs, we stack them and then "unroll" the generative model to form a deep auto-encoder. In our experiments, we first train four RBMs and stack them to form a five-layer network. The weight matrices of RBMs are used as the initial weight matrices for the five-layer network. By "unrolling" the stacked RBMs, we flip the five-layer network and create a deep nine-layer network whose lower layers use the matrices to encode the input and whose upper layers use the matrices in reverse order to decode the input. Figure 2(b) shows a deep auto-encoder constructed by stacking multiple RBMs and then "unrolling" the stacked RBMs. The auto-encoder can be fine-tuned using back-propagation of error derivatives [32]. Algorithm 2 shows the training process of the deep auto-encoder. To fine-tune the auto-encoder, we use a learning rate of 10^{-6} for all layers and train for 5 iterations. The fine-tune processes in our experiments are also performed on an AMD Radeon™ HD 7970 GPU. Output from the central layer of the deep auto-encoder is concatenated to the original input to improve the classification accuracy. We refer to the resulting feature vectors as DNN vectors in the remaining sections of this paper.

Algorithm 2 Deep Auto-encoder Training Process

Input: batch set, Weight matrices, learning rate l
Output: Weight matrices

for number of iterations **do**
  for number of batches **do**
    Assign batch V to be A_0, the activity of layer 0
    for layer i from 1 to n **do**
      Compute the activity of layer i,$A_i = A_{i-1} \times W_{i-1}$
    end for
    Compute error for last layer, $E_n = A_n - A_0$
    for layer i from n-1 to 0 **do**
      Back propagate error, $E_i = E_{i+1} \times W_i$
    end for
    for layer i from 0 to n-1 **do**
      Update weight matrix, $W_i += l \times A_i \times E_{i+1}$
    end for
  end for
end for

IV. Classification

To automatically classify the Android applications into benign or malicious applications, we calculate similarities between feature vectors and similarities between graphs depending on the representation we are using. The similarity measures are constructed as a kernel matrix and fed into a Support Vector Machine (SVM) for classification. We choose the SVM algorithm due to its accuracy as a supervised approach for binary classification. Additionally, SVMs can perform classification based on a precomputed kernel matrix constructed using graph kernels or Multiple Kernel Learning (MKL) in our context while most of the other machine learning models cannot. To further improve the accuracy, we also apply a hierarchical MKL method to combine different kernel matrices and learn the final classifier.

A. Kernel Matrix Construction for Vectors

For feature vector representations, we use different kernels to calculate similarities between each pair of vectors. The similarity measures are constructed as kernel matrices and fed into an SVM. For a given dataset $D = \{v_1, v_2, \ldots, v_n\}$ of vectors, a kernel matrix $M_{n\times n}$ is a symmetrical matrix where every element $M(i,j) = k(v_i, v_j)$ refers to the kernel function applied to a pair of vectors v_i and v_j. We evaluated three popular kernels including the Gaussian kernel (Eq. 1), the Intersect kernel (Eq. 2) and the Linear kernel (Eq. 3).

$$k_{gaussian}(x,y) = exp(-\sum_{i=1}^{n} \frac{(x_i - y_i)^2}{\sigma}) \tag{1}$$

$$k_{intersect}(x,y) = \sum_{i=1}^{n} min(x_i, y_i) \tag{2}$$

$$k_{linear}(x,y) = \sum_{i=1}^{n} x_i * y_i \tag{3}$$

B. Kernel Matrix Construction for Graphs

For the graph sets converted from dynamic system call sequences, we use Shortest Path Graph Kernel (SPGK) to compute graph similarities and construct kernel matrices. In the SPGK algorithm, an input graph is converted to all pair shortest path graph using Floyd-Washall algorithm first. Given a graph $G = \langle V, E\rangle$ comprising a set V of vertices together with a set E of edges, a shortest path graph is a graph $S = \langle V', E'\rangle$, where $V' = V$ and $E' = \{e'_1, \ldots, e'_m\}$ such that $e'_i = (u_i, v_i)$ if the corresponding vertices u_i and v_i are connected by a path in G. The edges in the shortest path graph are labeled with the shortest distance between the two nodes in the original graph.

The SPGK algorithm for two shortest path graphs $S_1 = \langle V_1, E_1\rangle$ and $S_2 = \langle V_2, E_2\rangle$ is computed as:

$$K_{SPGK}(S_1, S_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2) \tag{4}$$

where k_{walk} is a kernel for comparing two edge walks. The edge walk kernel k_{walk} is the product of kernels on the vertices and edges along the walk. It can be calculated based on the starting vertex, the ending vertex, and the edge connecting both. Let e_1 be the edge connecting nodes u_1 and v_1 of graph S_1, and e_2 be the edge connecting nodes u_2 and v_2 of graph S_2. The edge walk kernel is defined as follows:

$$k_{walk}(e_1, e_2) = k_{node}(u_1, u_2) \cdot k_{edge}(e_1, e_2) \cdot k_{node}(v_1, v_2) \tag{5}$$

where k_{node} and k_{edge} are kernel functions for comparing vertices and edges respectively. The same notations are also applied in the following sections.

In our experiments, we pick the Brownian Bridge kernel (Eq. 6) as used in Borgwardt et al. [33] with a c value of 2 for k_{edge}. For k_{node}, we evaluated the same kernels used on constructing kernel matrices for vector sets including the Gaussian kernel (Eq. 1), the Intersect kernel (Eq. 2) and the Linear kernel (Eq. 3). To speedup the graph kernel computation, we implement the parallelization of SPGK on multi-core CPUs and GPUs according to [34].

$$k_{brownian}(e_1, e_2) = max(0, c - |e_1 - e_2|) \tag{6}$$

C. Support Vector Machine

SVMs consist of two phases: training and testing. Given positive and negative samples in the training phase, an SVM finds a hyperplane which is specified by the normal vector w and perpendicular distance b to the origin that separates the two classes with the largest margin γ [35]. Figure 3 shows a schematic depiction of an SVM. During the testing phase, the samples are classified by the SVM prediction model and assigned either a positive or negative label. The decision function f of the linear SVM is given by

$$f(x) = \langle w, x \rangle + b \tag{7}$$

where x is a feature vector representing the sample. It is classified as positive if $f(x) > 0$ and negative otherwise. In the training phase, $\langle w, b \rangle$ are computed as the SVM prediction model from the training data. In the testing phase, the samples are classified using Eq. 7 with w and b from the prediction model. To use a kernel matrix as input, the decision function can be transformed to Eq. 8. In this equation, y_i is the class label of training data, w^* and α_i are parameters of the prediction model computed from the training data. $K(R_i, R)$ is the kernel value between a testing representation R and a training representation R_i [36]. Once we fill the kernel values with the kernel matrix, we can classify the testing applications.

$$f(R) = (w^* + \sum_{i=0}^{N} \alpha_i y_i K(R_i, R)) \tag{8}$$

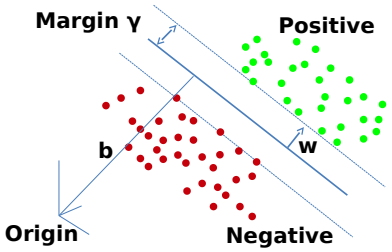


Fig. 3: This figure shows an illustration of the SVM method. w is the normal vector and b is the perpendicular distance to the origin.

D. Multiple Kernel Learning

One simple way to combine learning results from different features is to concatenate different feature vectors to create a large vector and use it for classification. However, this simple method assigns the same weight to different features which may lead to suboptimal learning results compared to training on individual features because some features may play more important roles in the learning than other features. Therefore, we need to assign different weights to different features based on their significance during learning. Such optimal weights can be calculated by the MKL algorithm.

MKL is an SVM based method for use with multiple kernels. An SVM takes one kernel matrix as input to build a classifier. However, when it comes to learning, it makes more sense to extract different features from all available sources, learn these features separately and then combine the learning results. MKL does this by taking kernel matrices constructed from different features and different kernels, and is able to find an optimal kernel combination to build the classifier. In addition to the SVM α_i and bias term w^*, MKL learns one more parameter which is the kernel weights β_j in training. Eq. 9 shows the resulting kernel method from MKL.

$$f(R) = (w^* + \sum_{i=0}^{N} \alpha_i \sum_{j=0}^{M} \beta_j y_i K_j(R_i, R)) \tag{9}$$

E. Hierarchical MKL

In our experiments, we choose to use Generalized MKL with the Spectral Projected Gradient decent optimization algorithm (SPG-GMKL) [37] to perform MKL. Since we construct multiple kernel matrices for each vector set and each graph set, we first use SPG-GMKL to combine different kernel matrices of the same vector or graph feature set. Experiments show we gain limited classification accuracy improvement in the first level of MKL. Because we explore multiple kernels with different parameters and reported only the best result. Combining different kernels learned from the same features may not necessary improve the performance. However, in the second level, SPG-GMKL is applied again to combine all MKL kernel matrices generated in the first level to learn the final hybrid classifier and much better improvement is achieved.

V. Experimental Results

In our method, we extract 10 static and dynamic features and convert them to 16 feature vector sets and 4 graph feature sets. We first evaluate the performance of each individual feature vector set. Then we train one DNN for each vector set and build the DNN vector set by concatenating DNN learned features with the original features. We show that using the DNN features are able to help improving the classification accuracy. Furthermore, for each of the DNN vector sets, learning occurs by multiple kernels and the learning results are combined using MKL. Similarly, we apply SPGK on the graph sets and construct multiple kernel matrices for each graph set. The kernel matrices of the graph sets are also combined using MKL. Finally, all resulting MKL kernel matrices are combined by applying MKL again to build the final hybrid classifier. These kernel matrices are evaluated with an SVM algorithm

for a ten-fold cross validation. We also evaluate 15 different values for the regularization parameter C in SVM, varying from 2^{-2} to 2^{12} with a step value of 2. The experiments are repeated five times with different cross validation partitions and the average classification accuracy rates are reported.

A. Dataset

We collected 5888 applications from Google Play and VirusShare [5]. To reveal malicious and benign applications, we submit our samples to the VirusTotal [6] web service and inspect the output of 51 commercial Anti-Virus (AV) scanners. We label all applications as malicious that are detected by at least two of the scanners. The other applications are labeled as benign. We end up with 1886 malicious applications and 4002 benign applications. The malicious samples were mostly discovered in 2014, and they are categorized into 39 families by a commercial AV scanner named AVG [7].

For the dynamic system call sequences of our samples, we convert them into n-gram graphs in addition to the n-gram vector representation. Table. I records the statistics including number of vertices, edges, shortest paths, and heights for the graphs generated from our malicious and benign samples. Since 1-gram, 2-gram, 3-gram, and 4-gram graphs have exactly the same structure, we only show numbers for one set. On the statistics table, the graphs generated from malware are slightly larger than the graphs that came from benign samples on average. We hypothesize that malware tends to spawn additional processes to perform malicious behaviors.

B. Evaluation Metrics

In our experiments, we train our SVM on a classification problem with two classes, malicious or benign. A confusion matrix is used in our method to evaluate the effectiveness of different kernels. From the confusion matrix, we can calculate False Positive Rate (FPR) and Accuracy.

We let $True\ Positive\ (TP)$ be the number of Android malware that are correctly detected, $True\ Negative\ (TN)$ be the number of benign applications that are correctly classified, $False\ Negative\ (FN)$ be the number of malware that are predicted as benign application, and $False\ Positive\ (FP)$ be the number of benign applications that are classified as malware. Then our evaluation metrics are defined as follows:

$$FPR = \frac{FP}{FP + TN} \tag{10}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{11}$$

C. Results from Original Vector and Graph set

First, we evaluate the performance of each original feature vector and graph set. As described before, three kernels consisting of Gaussian kernels, Intersect kernels, and Linear kernels are applied to our feature vector sets. Similarly, SPGK-Gaussian, SPGK-Intersect, and SPGK-Linear kernels are applied on the graph feature sets. For Gaussian kernel, we also evaluate different σ values from 2^{-6} to 2^{9} with a step value of 2^3. These kernel matrices are fed into an SVM for five runs of ten-fold cross validation.

In Table II, the first column lists the different feature sets. The second column shows the best accuracy achieved by each of the original feature vector sets or graph sets. We observe that the overall best accuracy is achieved by 4-gram vector set converted from instruction sequences. It reaches 93.5% accuracy. Among the static features other than instruction sequences, requested permissions, the most used static feature in previous static analysis methods [26] performs best and reaches 86.6% accuracy. For the vector and graph sets converted from system call sequences, 4-gram graph performs best with an 87.3% accuracy. We can also notice that the graph set performs better than the corresponding vector set by about 5% on average. This shows that the topology of the graph-based techniques adds predictive power to the model.

D. Results from DNN

Second, we evaluate the performance of each DNN vector set. In our deep auto-encoder, number of units in the first layer equals to the dimension of input feature vector. We only need to select the layer sizes for the hidden layers of four stacked RBMs. In our experiment, we evaluate all combinations of 4 layer sizes selected from 128, 256, 512, 1024, 2048, and 4096. The DNN learned features in conjunction with original features are then evaluated using the same method as described in Section V-C. In Table II, the third column shows the best accuracy achieved by combining original and DNN vectors. The fourth column lists the corresponding network sizes. In the third column, the data marked in bold shows DNN improves the performance of the original feature vector. We observe that 15 out of 16 feature vector sets can be improved by appending DNN learned features. By using DNN features the maximum accuracy improvement can be achieved is 0.5% by intent filters and 3-gram system call vectors. The best performance is also achieved by 4-gram instruction feature vectors at 93.8%.

E. Results from first level MKL

In the third experiment, we apply MKL on each DNN vector set and each graph set. Since we evaluate multiple Gaussian kernels with different σ values, we first select a Gaussian kernel matrix with the best performance, then combine it with Intersect and Linear kernel matrices using SPG-GMKL. The same process is applied to SPGK-Gaussian, SPGK-Intersect, and SPGK-Linear for graph sets.

The fifth column of Table II shows the results of applying MKL on each vector or graph feature set. Similarly, the data marked in bold means a performance improvement was achieved. In total, 9 out of 20 DNN vector sets and graph sets can be improved by MKL, and the maximum absolute improvement is 0.3%. The sixth column of Table II shows the weights of Gaussian, Intersect, and Linear kernel matrices learned by MKL. It should be noted that in vector and graph sets converted from system call sequences, the weight of Gaussian kernel is 0.00 because it can actually degrade the

[5] http://virusshare.com

[6] https://www.virustotal.com/

[7] http://free.avg.com/us-en/homepage

TABLE I: Detailed Statistics of Vertices, Edges, Heights, and Shortest Paths for graph representations of Malicious (M) and Benign (B) applications.

	Vertices			Edges			Shortest Paths			Heights		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Graph (M)	7	114	29	6	113	28	6	229	42	2	17	4
Graph (B)	7	109	24	6	108	23	6	411	33	2	18	3

accuracy if we include Gaussian kernel in MKL for these sets. We observe that improvement from the first level of MKL is limited because we evaluated multiple kernels with different parameters and reported the best results. After such a large kernel search, we may not be able to improve the performance further even with MKL.

F. Result from second level MKL

After applying the first level MKL to combine kernel matrices for each vector or graph set, we apply SPG-GMKL again to combine the 20 kernel matrices generated by the first level MKL. The second level MKL weights learned from SPG-GMKL for static and dynamic features are listed in Table III. And, classification results of the final hybrid classifier are shown in Table IV. The best classification accuracy we are able to achieve using HADM on our dataset is 94.7% with a FPR of 1.8%. Compared to the best accuracy that can be achieved by the original features, which is 93.5%, we obtain a 1.2% absolute improvement.

TABLE III: This table shows MKL weights of the dynamic features for the final classifier.

Permission APIs	Used permissions	Req. permissions	Ad. networks
3.104	2.152	4.888	1.905
Intent filters	Suspicious calls	Network APIs	Providers
1.465	2.678	1.266	1.481
Inst. 1-gram	Inst. 2-gram	Inst. 3-gram	Inst. 4-gram
1.137	1.410	2.699	4.392
syscall 1-gram vect.	syscall 2-gram vect.	syscall 3-gram vect.	syscall 4-gram vect.
0.578	0.673	1.041	1.357
syscall 1-gram graph	syscall 2-gram graph	syscall 3-gram graph	syscall 4-gram graph
1.221	1.498	2.054	2.434

TABLE IV: This table shows classification results from the final classifier.

TP	FN	FP	TN	FPR	Accuracy
1647	239	71	3931	1.8%	94.7%

G. Results from concatenating Original Feature Vectors

To compare our HADM method with traditional feature-vector-based methods, we perform experiments by concatenating original feature vectors and feeding them to an SVM for classification. A total of five experiments were performed. In the first experiment, we concatenated all original vector sets. In the second experiment, we concatenated all static feature vector sets. In the third experiment, we concatenated all static feature vector sets other than the instruction vector sets and in the fourth experiment we concatenated all instruction vectors. Finally, in the last experiment, we concatenated all system call vector sets.

Results of these experiments are shown in Table V. A check mark means the corresponding vector set is included in concatenation. The table shows by simply concatenating all vector sets, the best accuracy that can be reached is 93.4%, while concatenating only static features achieves 93.6% accuracy. These results are close to using just 4-gram instruction vector set which reaches 93.5% accuracy. These experiments show that adding more features may not necessarily increase the classification accuracy. However, in our HADM method, we refine each feature vector set with DNN and combine different features using weights learned by MKL. Therefore, we are able to improve classification accuracy over individual feature vector sets or graph sets or simply combining them.

TABLE V: This table shows classification results from simply concatenating the original feature vector sets.

Permission APIs	✓	✓	✓		
Used permissions	✓	✓	✓		
Req. permissions	✓	✓	✓		
Ad. networks	✓	✓	✓		
Intent filters	✓	✓	✓		
Suspicious calls	✓	✓	✓		
Network APIs	✓	✓	✓		
Providers	✓	✓	✓		
Inst. 1-gram	✓	✓		✓	
Inst. 2-gram	✓	✓		✓	
Inst. 3-gram	✓	✓		✓	
Inst. 4-gram	✓	✓		✓	
syscall 1-gram	✓				✓
syscall 2-gram	✓				✓
syscall 3-gram	✓				✓
syscall 4-gram	✓				✓
Accuracy	93.4%	93.6%	92.5%	93.4%	83.8%

H. Comparison with State-of-the-art

There is one well-known Android malware classification method using hybrid analysis that provides public access: Andrubis[8]. We submitted all of our 5888 samples to Andrubis for analysis. For each application, Andrubis is able to return a maliciousness rating between 0 and 10. In their rating system, 0 means likely benign and 10 means likely malicious. After we get the maliciousness ratings for our samples, we set a threshold t. We evaluate $t = \{1 - 9\}$ where if $t = 1$ means that any application returning 1 or higher is classified as

[8]https://anubis.iseclab.org/

TABLE II: This table shows classification results of different representations. Acc. means accuracy, Perm. means permissions, Req. means requested, Inst. means instructions, g. means gram, Sys. means system call sequence, and vect. means vector.

	Original Acc.	DNN Acc.	DNN Network Sizes	MKL Acc.	MKL Weights
Static Features					
Perm. APIs	83.9%	**84.3%**	4096-512-4096-512	**84.4%**	[0.32, 5.63, 5.63]
Used Perm.	80.8%	**80.9%**	4096-512-4096-512	**81.0%**	[0.33, 5.62, 5.62]
Req. Perm.	86.6%	**87.1%**	1024-512-256-128	**87.3%**	[0.66, 5.82, 5.59]
Ad. networks	72.8%	**72.9%**	128-128-128-128	72.9%	[0.07, 4.54, 4.54]
Intent filters	84.0%	**84.5%**	1024-1024-1024-1024	84.5%	[3.12, 4.51, 7.68]
Suspicious calls	81.3%	**81.5%**	4096-4096-4096-4096	81.5%	[0.53, 5.88, 5.88]
Network APIs	75.6%	**75.7%**	512-256-512-256	75.7%	[0.01, 8.39, 11.55]
Providers	69.1%	69.1%	4096-512-4096-512	69.1%	[0.00, 7.82, 9.74]
Inst. 1-g.	87.0%	**87.4%**	4096-4096-4096-4096	87.4%	[9.97, 4.66, 1.75]
Inst. 2-g.	90.2%	**90.3%**	512-1024-512-1024	**90.6%**	[9.67, 8.12, 3.79]
Inst. 3-g.	92.3%	**92.5%**	256-512-256-512	92.5%	[8.07, 11.37, 4.86]
Inst. 4-g.	93.5%	**93.8%**	2048-2048-2048-2048	93.8%	[6.90, 11.95, 4.44]
Dynamic System Calls					
Sys. 1-g. vect.	80.5%	**80.8%**	2048-2048-2048-2048	80.8%	[0.00, 5.35, 2.56]
Sys. 2-g. vect.	80.9%	**81.0%**	4096-4096-4096-4096	**81.3%**	[0.00, 6.11, 2.76]
Sys. 3-g. vect.	82.8%	**83.3%**	512-256-512-256	**83.6%**	[0.00, 7.51, 3.20]
Sys. 4-g. vect.	83.3%	**83.7%**	256-512-256-512	**83.9%**	[0.00, 8.20, 3.39]
Sys. 1-g. graph	85.3%			**85.5%**	[0.00, 7.84, 3.95]
Sys. 2-g. graph	85.9%			**86.2%**	[0.00, 8.99, 4.22]
Sys. 3-g. graph	87.1%			87.1%	[0.00, 9.83, 4.41]
Sys. 4-g. graph	87.3%			87.3%	[0.00, 10.43, 4.59]

malicious. Table VI shows the classification results obtained using different thresholds. First, we notice that 572 samples failed to be executed by Andrubis. However, these samples are emulated fine in our method. We believe this is because Andrubis can only analyze applications using API level 8 (Android 2.3) or lower [20]. In our method, API level 17 (Android 4.2) is used. Hence, we are able to emulate more recent applications. The third row of Table VI shows the True Positive. The fourth row shows the True Negative. The fifth row shows classification accuracy without failure which is calculated as (TP+TN)/(5888-572). The last row shows overall accuracy which is calculated as (TP+TN)/5888. The best accuracy Andrubis achieved is 85.2% when we ignore failed samples and 76.9% when we count the failures as wrong detections. Consequently, the HADM proposed in this paper is able to reach a significantly better accuracy on our dataset than the Andrubis method.

VI. Related Work

A few hybrid methods have been proposed before for Android malware classification including AASandbox [18], DroidRanger [38], SmartDroid [19], Andrubis [21], [22], [23], and Mobile-Sandbox [20]. In contrast to these methods, HADM applies deep learning techniques to improve the performance of each feature vector set, and it combines the results from feature vector sets and graph sets using hierarchical MKL.

Droid-Sec [39] is the first work to apply deep learning to Android malware classification. It extracts over 200 features from both static and dynamic analyses and then feeds these into a DNN for classification. Experiments on 250 malicious and 250 benign applications show Droid-Sec is able to reach 96.5% accuracy. DroidDetector [40] uses the same method as proposed in Droid-Sec. It extracts 192 features from both static and dynamic analyses and characterizes malware using a DNN-based model. DeepSign is another work that applies deep learning [41] on Windows malware signature generation and classification. It uses the Cuckoo sandbox [9] to record the execution behavior of each malware. Then, it treats the behavior report as a raw text file and uses uni-grams to convert each report into a 20,000 bit vector. The bit vectors are then fed into DNN to generate signatures. At the end, the signatures are fed into an SVM for classification. Experiments on 1800 malware samples without benign applications show that DeepSign is able to reach 96.4% accuracy. Saxe. et al. [42] also applied deep learning to Windows binary analysis. It extracts four different sets of static features and converts them into 1024-length vectors. The vectors are then fed into a DNN with two hidden layers for classification. In our method, we extract significantly more features from many more samples. More importantly, we train a DNN for each individual feature vector set and combine the DNN learned features with the original features and then perform classification using hierarchical MKL. This method has been shown to improve deep learning results.

MKL has been studied previously in Windows malware analysis by Anderson et. al. [43]. They apply Gaussian kernel and Spectral kernel on the same features and combine them using MKL. We improve upon their method by using hierarchical MKL and multiple features.

VII. Conclusion and Future Work

In this paper, we propose a hybrid Android malware classification method named HADM. We first evaluate the performance of 16 feature vector sets and 4 graph sets generated

[9] https://cuckoosandbox.org

TABLE VI: This table shows classification results from Andrubis. Acc. means accuracy and F. means failure.

Threshold	1	2	3	4	5	6	7	8	9
Failure	572	572	572	572	572	572	572	572	572
TP	**1253**	1212	1176	1146	1130	1114	1083	1040	979
TN	2993	3166	3261	3327	3367	3406	3447	3483	**3518**
Acc. w/o F.	80.0%	82.4%	83.5%	84.1%	84.6%	85.0%	**85.2%**	85.1%	84.6%
Overall Acc.	72.1%	74.4%	75.4%	76.0%	76.4%	76.8%	**76.9%**	76.8%	76.4%

from 10 static and dynamic features collected from Android applications. To improve the classification accuracy, we train one DNN for each feature vector set and concatenate the DNN learned features with the original features. Multiple kernels are then applied on the DNN vector sets and multiple graph kernels are applied on the graph sets. The kernel learning results are combined using MKL to further improve accuracy. At the end, MKL is applied again to the combined resulting MKL kernel matrices to build the final hybrid classifier.

Evaluation of different features on our dataset show that the best classification accuracy that can be achieved using static analysis is 93.5% by 4-gram instruction feature vectors, and the best accuracy that can be achieved using dynamic analysis is 87.3% by 4-gram system call graphs. Furthermore, the application of hierarchical MKL is able to yield a best classification accuracy among all our models by achieving 94.7%.

Future work includes but not limited to: extracting more features, experimenting other deep learning techniques, and trying to improve the performance of system call sequences.

REFERENCES

[1] N. Mawston. Android captured record 85 percent share of global smartphone shipments in q2 2014. Technical report, Strategy Analytics.

[2] R. Sarikaya, G. E. Hinton, and A. Deoras. Application of deep belief networks for natural language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(4):778–784, April 2014.

[3] Mehmet Gonen and Ethem Alpaydin. Multiple kernel learning algorithms. *J. Mach. Learn. Res.*, 12:2211–2268, July 2011.

[4] IDC. Smartphone os market share, q1 2015. Technical report.

[5] PulseSecure. 2015 mobile threat report. Technical report, 2015.

[6] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proceedings of the 7th Asia Joint Conference on Information Security (Asia JCIS)*, pages 62–69, Aug 2012.

[7] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[8] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[9] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS)*, 2014.

[10] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Computer Security - ESORICS 2014*, Lecture Notes in Computer Science. 2014.

[11] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[12] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)*, pages 15–26, 2011.

[13] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[14] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[15] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on Systems Security (EuroSec)*, 2013.

[16] K. Tam, S. J Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[17] M. Dimjaševic, S. Atzeni, I. Ugrina, and Z. Rakamaric. Android malware detection based on system calls. Technical report, University of Utah, 2015.

[18] T. Blasing, L. Batyuk, A. D. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALCON)*, pages 55–62, Oct 2010.

[19] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 93–104, New York, NY, USA, 2012.

[20] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp, and J. Hoffmann. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, pages 1–13, 2014.

[21] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[22] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Techical Report, TRISECLAB-0414-001*, 2014.

[23] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis. In *Proceedings of the 39th Annual International Computers, Software and Applications Conference (COMPSAC)*, 2015.

[24] S. Zhao, X. Li, G. Xu, L. Zhang, and Z. Feng. Attack tree based android malware detection with hybrid analysis. In *Proceedings of the IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2014.

[25] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, Jan 2014.

[26] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab. A review on feature selection in mobile malware detection. *Digital Investigation*, 13(0):22 – 37, 2015.

[27] L. Deng and D. Yu. Deep learning: Methods and applications. *Found. Trends Signal Process.*, 7:197–387, June 2014.

[28] G. E. Hinton. A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012.

[29] A. Krizhevsky and G. E. Hinton. Using very deep autoencoders for content-based image retrieval. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*, 2011.

[30] M. Ranzato, Y. Boureau, and Y. L. Cun. Sparse feature learning for deep belief networks. In *Proceedings of the Neural Information Processing Systems (NIPS)*, pages 1185–1192. 2007.

[31] N. Le Roux and Y. Bengio. Representational power of restricted boltzmann machines and deep belief networks. *Neural Computation*, 20(6):1631–1649, June 2008.

[32] L. Deng, M. L. Seltzer, D. Yu, A. Acero, A. R. Mohamed, and G. E. Hinton. Binary coding of speech spectrograms using a deep auto-encoder. In *INTERSPEECH*, pages 1692–1695, 2010.

[33] K. M. Borgwardt and H. P. Kriegel. Shortest-path kernels on graphs. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 74–81, 2005.

[34] L. Xu, W. Wang, M. Alvarez, J. Cavazos, and D. Zhang. Parallelization of shortest path graph kernels on multi-core cpus and gpus. In *Proceedings of the Programmability Issues for Heterogeneous Multicores (MultiProg)*, Vienna, Austria, 2014.

[35] N. Cristianini and J. Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[36] B. Scholkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001.

[37] A. Jain, S.V.N. Vishwanathan, and M. Varma. Spg-gmkl: Generalized multiple kernel learning with a million kernels. In *Proceedings of the 18th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*.

[38] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb 2012.

[39] Z. Yuan, Y. Lu, X. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. In *Proceedings of the ACM conference on SIGCOMM*, 2014.

[40] Z. Yuan, Y. Lu, and Y. Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(01):114–123, 2016.

[41] O.E. David and N.S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2015.

[42] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. *CoRR*, abs/1508.03096, 2015.

[43] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 7(4):247–258, 2011.