Deep4MalDroid: A Deep Learning Framework for Android Malware Detection Based on Linux Kernel System Call Graphs

Shifu Hou, Aaron Saas
Department of Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV, 26506, USA
{shhou, assas@mix.wvu.edu}

Lifei Chen
School of Mathematics and Computer Science
Fujian Normal University
Fuzhou, FJ, 350117, China
clfei@fjnu.edu.cn

Yanfang Ye *
Department of Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV, 26506, USA
yanfang.ye@mail.wvu.edu

***Abstract*—With explosive growth of Android malware and due to its damage to smart phone users (e.g., stealing user credentials, resource abuse), Android malware detection is one of the cyber security topics that are of great interests. Currently, the most significant line of defense against Android malware is anti-malware software products, such as Norton, Lookout, and Comodo Mobile Security, which mainly use the signature-based method to recognize threats. However, malware attackers increasingly employ techniques such as repackaging and obfuscation to bypass signatures and defeat attempts to analyze their inner mechanisms. The increasing sophistication of Android malware calls for new defensive techniques that are harder to evade, and are capable of protecting users against novel threats. In this paper, we propose a novel dynamic analysis method named *Component Traversal* that can automatically execute the code routines of each given Android application (app) as completely as possible. Based on the extracted Linux kernel system calls, we further construct the weighted directed graphs and then apply a deep learning framework resting on the graph based features for newly unknown Android malware detection. A comprehensive experimental study on a real sample collection from Comodo Cloud Security Center is performed to compare various malware detection approaches. Promising experimental results demonstrate that our proposed method outperforms other alternative Android malware detection techniques. Our developed system *Deep4MalDroid* has also been integrated into a commercial Android anti-malware software.**

I. Introduction

Smart phones have been widely used to perform the tasks such as banking, automated home control, and bill paying in people's daily life. In recent years, there has been an exponential growth in the number of smart phone users around the world: according to a recent report [27], there were over 1.91 billion smart phone users across the globe in 2015. Designed as an open, free, and programmable operating system, Android as one of the most popular smart phone platforms dominates the current market share [3]. However, the openness of Android not only attracts the developers for producing legitimate apps, but also attackers to deliver malware (short for ***ma***licious soft***ware***) onto unsuspecting users. Google's Android market is the official online platform for delivering apps to an Android based smart phone. Because of the lack in trustworthiness review methods, developers can upload their Android apps including cracked apps, repackaged apps, or trojans to the market easily. The presence of other third-party Android markets (e.g., Opera Mobile Store, Wandoujia) makes this problem worse. Today, a lot of android malware (e.g., Geinimi, DriodKungfu and Hongtoutou) is released on the markets, which poses serious threats to smart phone users, such as stealing user information, making premium calls, and sending SMS advertisement spams without the user's permission [14]. According to Symantec's latest Internet Security Threat Report [29], one in every five Android apps (nearly one million total) were actually malware. To protect legitimate users from the attacks of Android malware, currently, the most significant line of defense is anti-malware software products, such as Norton, Lookout, and Comodo Mobile Security, which mainly use the signature-based method to recognize threats. However, malware attackers increasingly employ techniques such as repackaging and obfuscation to bypass signatures and defeat attempts to analyze their inner mechanisms. The increasing sophistication of Android malware calls for new defensive techniques that are harder to evade, and are capable of protecting users against novel threats.

In this paper, we propose a novel dynamic analysis method named *Component Traversal* that can automatically execute the code routines of each given Android app as completely as possible. Based on the extracted Linux kernel system calls, we further construct the weighted directed graphs and then apply a deep learning framework resting on the graph based features for newly unknown Android malware detection. A comprehensive experimental study on a real sample collection from Comodo Cloud Security Center is performed to compare various malware detection approaches. Promising experimental results demonstrate that our proposed method outperforms other alternative Android malware detection techniques. Our developed system *Deep4MalDroid* has also been integrated into a commercial Android anti-malware software. The major contributions of our work can be summarized as follows:

* Corresponding author

- *A novel dynamic behavior extraction method:* Compared with the framework Application Programming Interface (API) calls, Linux kernel system calls are more resilient towards malware evasion techniques (e.g., the attacker can substitute the framework APIs to evade the detection). Moreover, the framework APIs vary from version to version of Android Operating System (OS), while Linux kernel system calls are version independent of Android OS. Therefore, for dynamic analysis, we extract the Linux kernel system calls from the executing apps instead of framework APIs. To monitor the system calls of an Android app through dynamic analysis, it always requires interactions with the app [36]. Given a large number of Android apps, it's infeasible to execute all of them manually. Although Android Testing Tool (ADT) Monkey has been commonly used to automatically execute the Android apps, it is not a sufficient approach to executing all the components of an app for malware analysis. To solve this problem, we propose the *Component Traversal* method as a way of automating the execution of an entire app code. By locating all the executable app components from the manifest file, a more complete system call list can be generated since all the runnable code may have an opportunity to be executed.
- *Graph representation of the extracted Linux kernel system calls:* For each Android app, based on the extracted system calls, in order to capture the relationships among them, a weighted directed graph will be constructed. Each graph node will represent a Linux kernel system call and its size will indicate its frequency, while a directed edge will indicate the sequential flow of Linux kernel system calls made and include a weight that implies the frequency the successor node called after the predecessor node.
- *An exploration of deep learning framework:* Due to its superior ability in feature learning through multilayer deep architecture, deep learning is feasible to learn higher level concepts based on the local feature representations [6], [18]. Based on the constructed Linux kernel system call graphs, we explore a deep learning architecture with the Stacked AutoEncoders (SAEs) model to learn generic patterns of Android malware and thus to detect newly unknown Android malware.
- *Comprehensive experimental study on a real sample set from an anti-malware industry company:* We develop an intelligent Android malware detection system *Deep4MalDroid* which integrates our proposed method and provide a comprehensive experimental study on the real sample collection from Comodo Cloud Security Center, which consists of 1,500 benign apps and 1,500 Android malware including the families of Geinimi, GinMaster, DriodKungfu, Hongtoutou, FakePlayer etc.

The rest of the paper is organized as follows. Section II discusses the related work. Section III presents the overview of the system architecture. Section IV introduces our proposed methods. In Section V, based on the real sample collection from Comodo Cloud Security Center, we systematically evaluate the performance of our developed Android malware detection system in comparison with other popular detection methods. Finally, Section VI concludes.

II. Related Work

Currently, the most significant line of defense against Android malware is anti-malware software products which mainly use the signature-based method to recognize threats. However, the signature-based solution is unable to prevent zero-day attacks. In order to remain effective, in the past few years, there have been many researches on intelligent Android malware analysis and detection [34], [30], [31], [32], [17].

In general, these researches can be categorized into two kinds of approaches: static analysis and dynamic analysis. *Static analysis* is the process of analyzing an Android app without actually executing it. This is a popular technique for Android malware analysis. Through static analysis, the representative features such as permissions [24], API calls [30], [24], [17], and functional call graphs [32] can be directly extracted from the source codes or binaries without execution. Though it is much more efficient and with high coverage, malware attackers can adopt techniques like repackaging, data encryption, or obfuscation (e.g., code reordering, junk code insertion, changing control flow logics) to evade static detection [21], [22]. In contrast, *dynamic analysis* focuses on the runtime behaviors and the system metrics of the Android apps. Compared with static analysis, dynamic analysis can provide an advantage in accurately studying runtime behaviors and thus more resilient towards typical evasion techniques [13], [33], [8], [26], [31]. DroidDolphin was a dynamic Android malware analysis framework which used DroidBox [10] and APE_BOX [9] to record different features from the collected Android apps [31]; TaintDroid used multi-level taint tracking to extract features from Android apps [13]: sensitive information was marked as a taint source and then was tracked as it was transferred across the system; Crowdroid extracted API system calls as a feature set for k-means clustering to detect Android malware [8], however, six users had to manually run the Android apps to help for the feature extraction. For dynamic analysis, as an app runs, it typically requires some user input(s) [36]. To keep the analysis process automated, we need some method, other than users, to provide the necessary input(s). Most of the existing works (e.g., [7], [12]) used ADT Monkey which can provide randomized user input to automatically execute the Android apps. However, this is not sufficient to reach all components of the Android app. Different from the existing works, in this paper, we propose a novel dynamic analysis method named *Component Traversal* that can automatically execute the code routines of each given app as completely as possible.

Detecting Android malware using data mining techniques is a fairly common practice [24], [30], [31], [11], [12], [34]. Shallow learning methods such as Support Vector Machine (SVM), Artificial Neural Network (ANN), Naïve Bayes (NB), and Decision Tree (DT) have successfully applied in Android malware detection [24], [30], [31], [11], [12]. Deep learning, a new frontier in machine learning and data mining, is starting to be leveraged in industrial and academic research for different applications (e.g., Computer Vision) [6], [23]. Yuan et al. have worked with the deep learning methods in Android malware detection [34], [35]. Their works combined static and dynamic analysis of files with Deep Belief Network (DBN) and experienced a 15.5% higher accuracy compared to traditional shallow learning models. Based on our collected sample set, we intend to explore a deep learning framework resting on our constructed Linux kernel system call graphs.

III. System Architecture

In this paper, based on the collected Android apps, we extract their Linux kernel system calls and construct the corresponding weighted directed graphs for feature representations, and then apply a deep learning framework for model construction and thus for newly unknown Android malware detection. Figure 1 shows the system architecture of our developed Android malware detection system *Deep4MalDroid*, which consists of the following five major components.

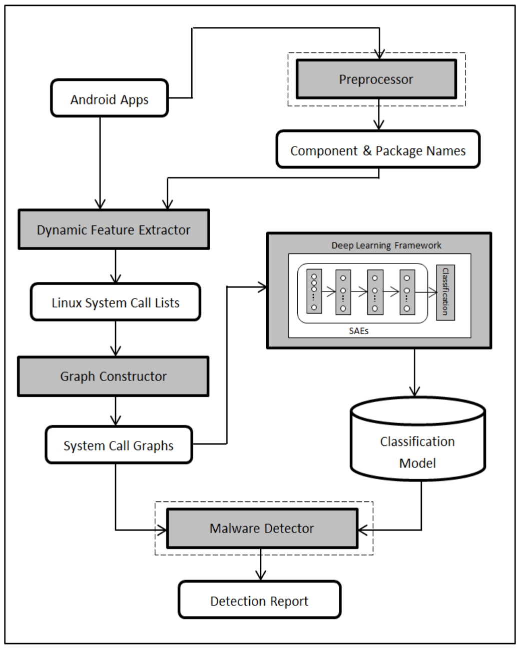


Fig. 1: System architecture of *Deep4MalDroid*

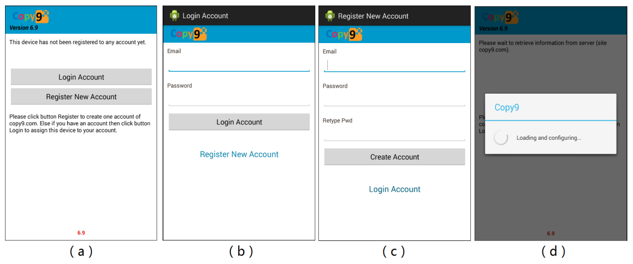
- *Preprocessor:* It accesses the manifest file of the Android Application Package (APK) to retrieve the list of all components, with which all the runnable code routines may have an opportunity to be executed. (See Section IV-A for details.)
- *Dynamic Feature Extractor:* It automatically extracts the Linux kernel system calls of each component from the given Android app. The app is executed in Genymotion which is a fast and robust Android emulator and the system calls are recorded by using Strace that is a diagnostic, debugging and instructional userspace utility for Linux. (See Section IV-A for details.)
- *Graph Constructor:* For each Android app, based the extracted system calls, a weighted directed graph will be constructed. Each vertex in the graph is a unique Linux kernel system weighted by its frequency, while the directed edge indicates which system call is performed afterwards and is weighted to show how frequently that sequence appears. (See Section IV-B for details.)
- *Deep Learning Classifier:* Resting on the constructed system call graphs, a deep learning framework is used for model construction and thus for newly unknown Android malware detection. (See Section IV-C for details.)
- *Malware Detector:* For the new collected unknown Android app, it will be parsed through the preprocessor, dynamic feature extractor and graph constructor for its feature extraction, and then the classification model will be used to predict whether it is benign or malicious.

IV. Proposed Method

To be resilient towards typical malware evasion techniques (e.g., repackaging, obfuscation), in this paper, we choose dynamic analysis for Android malware detection and propose a novel method named *Component Traversal* that can automatically execute the code routines of each given Android app as completely as possible. Based on the extracted Linux kernel system calls, the weighted directed graphs will be constructed. Then, resting on the constructed system call graphs, a deep learning framework is used for model construction and thus for newly unknown Android malware detection.

A. Component Traversal for Linux Kernel System Call Extraction and its Implementation

To monitor the system calls of an Android app through dynamic analysis, interaction with the app is required [36]. Interactions are either conducted by users [8] or executed automatically [11]. Performing dynamic analysis with user's actively interacting with the app might provide the best execution, however, it would be a labor consuming and resource intensive task. Given a large number of Android apps, it's infeasible to execute all of them manually. Most of the existing works (e.g., Andlantis [7], MALINE [12]) use the ADT Monkey, which generates pseudo-random streams of user events (e.g., clicks, touches, or gestures) and a number of system-level events [1], to automatically execute the apps for dynamic analysis. ADT Monkey is generally helpful for performing stress test, but it is not a sufficient approach to executing all the components of an app for malware analysis. A simple login screen could prevent the execution with ADT Monkey. For example, "SecureKid.apk" (*MD5: fb350aa40610c64fe18ef6005dcc98bc*) is a malicious repackaged app of "Copy9.apk" that provides remote spy on mobile. To execute this app, first it is required to execute its main activity which is a login page that loads as the app starts (as shown in Figure 2 (a)). There are two buttons for this starting page: Login Account (Figure 2 (b)) with "com.ispyoo.android.activity.LoginActivity" and Register New Account (Figure 2 (c)) with "com.ispyoo.android.activity.LoginActivity". When using ADT Monkey to auto-execute this app, no matter which event generated from ADT Monkey, it cannot cross the login page and thus fails to extract the sensitive behaviors from the app. Actually, the sensitive behaviors hide in the services of "com.ispyoo.common.monitor.AndroidWatchdogService" and "com.ispyoo.common.calltracker.receiver.ProcessCall", which are triggered in an activity called "com.ispyoo.android.activity.Splash" (as shown in Figure 2 (d)). In order to solve this problem, we propose the *Component Traversal* method as a way of automating the execution of an entire app's code. By locating all the executable components from the manifest file, a more complete system call list can be generated since all the runnable code may have an opportunity to be executed.



This device has not been registered to any account yet.

Please click button Register to create one account of copy9.com. Else if you have an account then click button Login to assign this device to your account.

Please wait to retrieve information from server (site copy9.com).

Fig. 2: Screen shots of "SecureKid" App

1) Preprocessing: Android app is compiled and packaged in a single archive file with an .apk suffix which includes all of the app code (.dex files), resources, assets, and manifest file. Android defines a component-based framework for developing mobile apps [32] and an Android app is composed of four different types of components [2]: *Activities* provide Graphical User Interface (GUI) functionality to enable user interactivity; *Services* are background communication processes that pass messages between the components of the app and communicate with other apps; *Broadcast Receivers* are background processes that respond to system-wide broadcast messages as necessary; *Content Providers* act as database management systems that manage the app data. The Android app must declare its components in a manifest file ("AndroidManifest.xml") which is one of the most important files in the Android project structure. Before the Android system can start an app component, the system must know that the component exists by reading the app's manifest file. The manifest file actually works as a road map to ensure that each app can function properly in the Android system.

By using the APKTool [4], which is a tool for reverse engineering the binary Android apps, we first unzip the APK file and then access the manifest file to retrieve the package name and the list of runnable components, specifically the Services and Activities. The *Component List* including the extracted package name and all runnable component names will be stored locally for use in manipulating the Android app during execution in the emulator.

2) Emulation and Feature Extraction: Android provides a sandboxed app execution environment [19]. A customized embedded Linux system interacts with the hardware in the smart phone. The middleware and application framework API runs on top of Linux. Compared with the framework APIs, Linux kernel system calls are more resilient towards malware evasion techniques (e.g., the attacker can substitute the framework APIs to evade the detection). The Linux operating system has about 300 system calls which can be classified into different categories depending on function of operating system, such as process management, memory management and device management [19]. If any Android app needs to request services (e.g., network transmission), developers can use "HttpUrlConnection" which is network API provided by Java Development Kit (JDK), or "HttpComponents" that is Java request networking framework provided by Apache, or they can even develop a networking framework by themselves. The framework APIs can be substituted, but the implementations of such request service have to rely on the system calls provided by the Linux kernel such as "sendto()"and "recvfrom()". Moreover, framework APIs vary from version to version of Android OS, while Linux kernel system calls are version independent of Android OS. Therefore, in this paper, we will extract the Linux kernel system calls from the executing Android apps, other than the framework APIs.

a) Emulation: Since Genymotion is a relatively fast and robust Android emulator with over two million users worldwide [16], we use it to execute the Android app for dynamic behavior extraction. Using the Android Debug Bridge (ADB), a versatile command line tool that allows the communication with an emulator instance or connected Android-powered device [20], the app to be analyzed will be loaded into the Genymotion emulator.

b) Linux kernel system call extraction: After successfully loading, by traversing each component in the *Component List* generated in the preprocessing step, each Activity and Service of the Android app will be executed to completion. To record the executed Linux kernel system calls, Strace is used for collecting logs. Strace is a diagnostic, debugging and instructional userspace utility for Linux, which is used to monitor interactions between processes and the Linux kernel, including system calls, signal deliveries, and changes of process state [25]. When all of the components have completed execution, using ADB and the Visual Studio file pipe, the full Linux kernel system call list is offloaded from the emulator and brought back to the host machine. Genymotion is then reset to prevent malware infection from skewing further extractions.

The proposed *Component Transversal* method allows for an automated, complete approach to dynamic analysis of Android apps rather than relying on user interactions or a random event generator. In order to enhance the process, a multi-threaded approach is implemented: one thread is used for Android app execution, while a separate thread is created to run Strace for logging Linux kernel system calls.

B. Graph Construction

In the previous step of our method, an Android app is analyzed and a list of its Linux kernel system calls is automatically extracted. Using independent Linux kernel system calls alone, however, is not sufficient information to model the app's behaviors [15]. Simply keeping a count of the system calls ignores any sequential relationships among them. In order to capture the relationships among the extracted system calls, a weighted directed graph $G = \{V, E\}$ is generated, where V is a set of nodes and $v \in V$ is unique Linux kernel system call, and $E \subseteq V \times V$ represents a set of directed weighted edges, where an edge $\overrightarrow{v_i v_j}$ indicates a sequential pair of system calls. Using a weighted directed graph maintains not just the number of each Linux kernel call made, but also the sequence of the system calls and the frequency of that sequence.

To construct the graph, each extracted unique Linux kernel system call is first mapped to an integer node. The frequency of the system call is directly proportional to the size of the node. For each sequential pair of system calls extracted, a directed edge between the nodes is created. This edge will be weighted and incremented each time that sequence occurs in the sample app. To further illustrate the process, Figure 3 is a segment of the collected Linux kernel system call list from "Live_wallpaper.APK" (*MD5: 2d38973d442ae070f76399ec4ef730e7*) which is a live wallpaper app embedded with malicious code that can steal user's credentials. In this paper, we only consider the names of Linux kernel system call while ignoring the parameters. This log contains five Linux system calls: "clock_gettime", "read", "getuid32", "ioctl", "fcntl64" with corresponding frequency of $\langle 4, 1, 4, 4, 1 \rangle$. A node is created for each system call in Figure 4 with a different size based on its frequency. In the recorded log, the system call "read" (line 2) is followed by "clock_gettime" (line 3), which means a directed edge is created from the "read" node to the "clock_gettime" node. The weight of this edge is the frequency of this pair of Linux kernel system calls appears in the generated log.

```
clock_gettime(CLOCK_MONOTONIC,  {1066,  546005435}) = 0
read(11,  "W",  16)                                 = 1
clock_gettime(CLOCK_MONOTONIC,  {1067,  994972273}) = 0
getuid32()                                          = 10814
clock_gettime(CLOCK_REALTIME,  {1431910070,  997161378}) = 0
getuid32()                                          = 10814
ioctl(9,  BINDER_WRITE_READ,  0xbfacdcc8)           = 0
ioctl(9,  BINDER_WRITE_READ,  0xbfacdcc8)           = 0
getuid32()                                          = 10814
clock_gettime(CLOCK_MONOTONIC,  {1068,  10819539}) = 0
ioctl(9,  BINDER_WRITE_READ,  0xbfacdb98)           = 0
ioctl(9,  BINDER_WRITE_READ,  0xbfacdb98)           = 0
getuid32()                                          = 10814
fcntl64(89,  F_GETFD)                               = 0
```

Fig. 3: Example of Linux kernel system call list

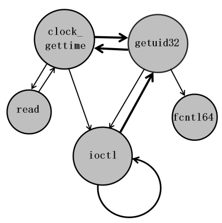


Fig. 4: Example of constructed system call graph

After graph construction, we use each node with its weight, each edge with its weight, as well as the in-degree and out-degree of each of the nodes as the feature of the Android app (note that all the feature values are logarithmically normalized). The generated graph based features will be used as the inputs for the classification model construction in the following section.

C. Deep Learning Framework for Android Malware Detection

Although classification methods based on shallow learning architectures, such as Support Vector Machine (SVM), Artificial Neural Network (ANN), Naïve Bayes (NB), and Decision Tree (DT), can be used to solve the Android malware detection problem [24], [30], [31], [11], [12], deep learning has been demonstrated to be one of the most promising architectures for its superior layerwise feature learning models and can thus achieve comparable or better performance [18]. In this paper, we explore a deep learning architecture with Stacked AutoEncoders (SAEs) model to detect Android malware. The SAEs model is a stack of AutoEncoders, which are used as building blocks to create a deep network.

1) AutoEncoder: An AutoEncoder, also called AutoAssociator, is an artificial neural network used for learning efficient codings [28]. Architecturally, an AutoEncoder is composed of an input layer, an output layer, and one or more hidden layers connecting them. The goal of an AutoEncoder is to encode a representation of the input layer into the hidden layer, which is then decoded into the output layer, yielding the same (or as close as possible) value as the input layer [5]. In this way, the hidden layer acts as another representation of the feature space, and in the case when the hidden layer is narrower (has fewer nodes) than the input/output layers, the activations of the final hidden layer can be regarded as a compressed representation of the input [5], [28], [6]. Figure 5 illustrates a one-layer AutoEncoder model with one input layer, one hidden layer, and one output layer. Typically, the number of hidden units (d') is much less then number of visible (input/output) ones (d). As a result, when passing data through such a network, it first compresses (encodes) input vector to fit in a smaller representation, and then tries to reconstruct (decode) it back. The task of training is to minimize an error or reconstruction (using Equation 1), i.e. find the most efficient compact representation (encoding) for input data (Equation 2).

$$E(\mathbf{x}, \mathbf{z}) = \frac{1}{2} \sum_{i=1}^{n} ||\mathbf{x}_i - \mathbf{z}_i||^2, \tag{1}$$

where $\mathbf{x}$ is an input vector, $\mathbf{z}$ is a reconstructed d-dimensional vector in the input space, n is the number of training samples.

$$\theta = \{\mathbf{w}, \mathbf{b}\} = \underset{\theta}{\arg\min}\, E(\mathbf{x}, \mathbf{z}), \tag{2}$$

where $\mathbf{w}$ is a $d' \times d$ weight matrix, $\mathbf{b}$ is an offset vector of dimensionality d', and θ is the mapping parameter set $\theta = \{\mathbf{w}, \mathbf{b}\}$.

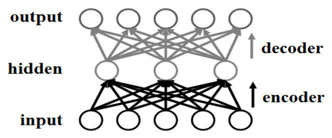


Fig. 5: A one-layer AutoEncoder model

2) Deep Learning Framework with SAEs: To form a deep network, an SAEs model is created by daisy chaining AutoEncoders together, known as stacking: the output of one AutoEncoder in the current layer is used as the input of the AutoEncoder in the next [6]. More rigorously, with an SAEs deep network with h hidden layers, the first layer takes the input from the training dataset and is trained simply as an AutoEncoder. Then, after the k^{th} hidden layer is obtained, its output is used as the input of the $(k+1)^{th}$ hidden layer, which is trained similarly. Finally, the h^{th} layer's output is used as the output of the entire SAEs model. In this manner, AutoEncoders can form a hierarchical stack. Figure 6 illustrates a SAEs model with h hidden layers. To use the SAEs for Android malware detection, a classifier needs to be added on the top layer. In this paper, we combine the SAEs and the classifier together as the entire deep architecture model for Android malware detection, which is illustrated in Figure 7.

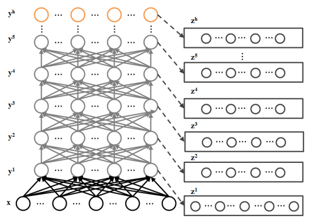


Fig. 6: Framework of Stacked AutoEncoders

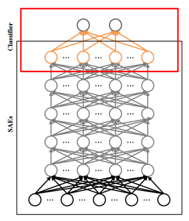


Fig. 7: Deep learning framework for Android malware detection

V. Experimental Results and Analysis

In this section, we conduct three sets of experimental studies using a real sample collection obtained from Comodo Cloud Security Center to fully evaluate the performance of our developed Android malware detection system: (1) In the first set of experiments, we compare the detection performances using ADT Monkey and our proposed *Component Traversal* method for Linux kernel system call extraction; (2) In the second set of experiments, we evaluate the detection performance of our graph construction method; (3) In the last set of experiments, we evaluate the detection performance of the deep learning framework by comparisons with typical shallow learning methods.

A. Experimental Setup

The sample set obtained from Comodo Cloud Security Center includes 3,000 android apps, half of which are benign, while the other half are malicious including the popular malware families of Geinimi, GinMaster, DriodKungfu, Hongtoutou, FakePlayer etc. We evaluate the Android malware detection performance of different methods using the measures shown in Table I. The emulations are conducted on the Genymotion. All the experiments are performed under the environment of 64 Bit Windows 8.1 operating system with Inter(R) Core(TM) i7-4790 CPU @ 3.60GHZ plus 16 GB of RAM.

TABLE I: Performance indices of Android malware detection

Indices	Specification
True Positive (TP)	Num. of apps correctly classified as malicious
True Negative (TN)	Num. of apps correctly classified as benign
False Positive (FP)	Num. of apps mistakenly classified as malicious
False Negative (FN)	Num. of apps mistakenly classified as benign
TP Rate (TPR)	$TP/(TP+FN)$
FP Rate (FPR)	$FP/(FP+TN)$
Accuracy (ACY)	$(TP+TN)/(TP+TN+FP+FN)$

B. Comparisons of Different Dynamic Feature Extraction Methods

In this set of experiments, we compare the performance of the two methods of automatic Android app execution: ADT Monkey and our proposed *Component Traversal* method (short for *CompTrav* in the following), using four typical classification methods (i.e., SVM, ANN, NB and DT). Based on the sample set described in Section V-A, we use both ADT Monkey and *CompTrav* method to extract the Linux kernel system calls from the Android apps. The extracted Linux kernel system calls are directly used as the inputs for each classifier. We conduct 10-fold cross validations for the evaluation. The experiment results shown in Table II demonstrate that our proposed *CompTrav* method outperforms using ADT Monkey for the Linux kernel system call extraction in Android malware detection. This should come as no surprise since *CompTrav* method allows more of the app components to be executed resulting in a more complete listing of Linux kernel system calls. The *CompTrav* method will therefore be used for the remaining experiments.

TABLE II: Comparisons of different dynamic feature extraction methods

Method	Accuracy	TP	FN	TN	FP
ADT Monkey+SVM (M_SVM)	67.46%	1667	833	1706	794
ADT Monkey+ANN (M_ANN)	66.36%	1635	865	1683	817
ADT Monkey+NB (M_NB)	62.74%	1553	947	1584	916
ADT Monkey+DT (M_SVM)	66.86%	1660	840	1683	817
CompTrav+SVM (C_SVM)	73.66%	1829	671	1854	646
CompTrav+ANN (C_ANN)	71.98%	1779	721	1820	680
CompTrav+NB (C_NB)	67.3%	1653	847	1712	788
CompTrav+DT (C_DT)	71.92%	1787	713	1809	691

For feature extraction efficiency, *CompTrav* also shows better performance than ADT Monkey. In our experiments, we find that in order to activate the code routines as completely as possible, ADT Monkey usually needs to generate at least 500 random events, which takes more than 2 minutes for executing an app. If using *CompTrav*, 5 seconds will be enough to execute an activity or a service. For our collected sample set, most of the malicious apps have less than 10 components, and the average execution time of an app using *CompTrav* is about 41 seconds whose speed is three times of using ADT Monkey.

C. *Evaluation of Graph Based Features*

In this set of experiments, we further evaluate the detection performance of the graph based features. Based on the sample set described in Section V-A, we first use *CompTrav* method to extract the Linux kernel system calls from the Android apps and then construct the graphs using the method described in Section IV-B. The directly extracted Linux kernel system calls are also used for comparisons. The 10-fold cross validations are conducted for the evaluation. The experimental results shown in Table III show that graph based features outperform the directly extracted Linux kernel system call features. This indicates that a simple count of Linux kernel system calls does not reveal the intent of an app, but the order of the system call execution and the other graph based features associated (e.g., in-degree, out-degree) can more accurately ascertain app functionality. Therefore, the graph based features will be used for the further experiments.

TABLE III: Evaluation of graph based features

Structure	Accuracy	TP	FN	TN	FP
CompTrav+SVM (C_SVM)	73.66%	1829	671	1854	646
CompTrav+ANN (C_ANN)	71.98%	1779	721	1820	680
CompTrav+NB (C_NB)	67.3%	1653	847	1712	788
CompTrav+DT (C_DT)	71.92%	1787	713	1809	691
*CompTrav*_Graph+SVM (CG_SVM)	88.24%	2181	319	2231	269
*CompTrav*_Graph+ANN (CG_ANN)	87.88%	2190	310	2204	296
*CompTrav*_Graph+NB (CG_NB)	77.94%	1942	558	1955	545
*CompTrav*_Graph+DT (CG_DT)	87.42%	2185	315	2186	314

Figure 8 shows an example that the generated graph based features can well discriminate the malware from the benign app. Using the example in Section IV-B, Figure 8 (a) shows the system call graph generated from the malicious app "Live_wallpaper.apk", while Figure 8 (e) displays the system call graph constructed from a benign app "Digimon_Unlimited.apk" (*MD5: 4b0b653094f06eb969715a0d604f77b4*) which is a Android game developed by Bulzipke. Figure 8 (b)(c)(d) show the graph based features of the malware "Live_wallpaper.apk" and Figure 8 (f)(g)(h) illustrate the graph based features of the benign app "Digimon_Unlimited.apk".

D. *Evaluation of Deep Learning Framework*

In this set of experiments, based on the sample set described in Section V-A, we first use *ComTrav* method to extract the Linux kernel system calls from the Android apps and then construct the graphs using the method described in Section IV-B. The generated graph based features are used as the inputs for

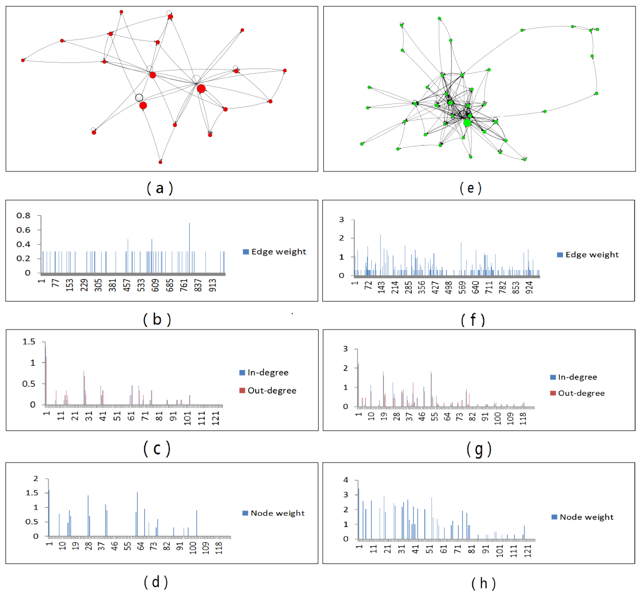


Fig. 8: Graph based features: malware vs. benign app

the different deep learning models. While building the deep learning model, there are two key parameters: the number of hidden layers and the number of neurons in each hidden layer. Table IV shows the detection performance changes with different deep learning model constructions. We also conduct the comparisons between the deep learning model and other four typical shallow learning classification models as shown in Table V. Figure 9 shows the ROC curves of different classification methods. From Table V and Figure 9, we can see that, compared with the typical shallow learning methods, the detection performance is greatly improved by using deep learning framework.

TABLE IV: Comparisons between different deep learning model constructions

Num of Layers	Num of Neurons	Accuracy	TP	FN	TN	FP
2	[300,300]	91.9%	2273	227	2322	178
2	[200,200]	92.56%	2304	196	2324	176
2	[100,100]	92.12%	2305	195	2301	199
3	[300,300,300]	92.84%	2301	199	2341	159
3	[200,200,200]	93.68%	2334	166	2350	150
3	[100,100,100]	92.38%	2306	194	2313	187
4	[200,200,200,200]	91.36%	2297	203	2271	229
5	[200,200,200,200,200]	91.4%	2277	223	2293	207

VI. Conclusion and Future work

In this paper, we develop an automatic Android malware detection system *Deep4MalDroid* using deep learning framework based on the Linux kernel system call graphs. A novel dynamic

TABLE V: Comparisons between deep learning and shallow learning models

Method	Accuracy	TP	FN	TN	FP
Deep Learning	93.68%	2334	166	2350	150
*CompTrav*_Graph+SVM (CG_SVM)	88.24%	2181	319	2231	269
*CompTrav*_Graph+ANN (CG_ANN)	87.88%	2190	310	2204	296
*CompTrav*_Graph+NB (CG_NB)	77.94%	1942	558	1955	545
*CompTrav*_Graph+DT (CG_DT)	87.42%	2185	315	2186	314

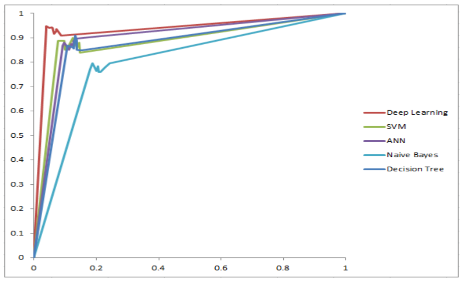


Fig. 9: ROC curves of different classification models

analysis method named *Component Traversal* is proposed which can automatically execute the code routines of each given Android app as completely as possible. Based on the extracted system calls, we construct the weighted directed graphs and then apply a deep learning framework for newly unknown Android malware detection. To the best of our knowledge, this is a unique approach to automated dynamic analysis. A comprehensive experimental study on a real sample collection from Comodo Cloud Security Center is performed to compare various malware detection approaches. Promising experimental results demonstrate that our proposed method outperforms other alternative Android malware detection techniques. The developed system *Deep4MalDroid* has also been integrated into a commercial Android anti-malware software.

For future work, we want to further improve *Component Traversal* method by introducing generation of random events on each component. We will also further explore how sparsity constraints are imposed on AutoEncoder to yield better detection performance. Meanwhile, it would be interesting to investigate other deep learning models for Android malware detection.

Acknowledgment

The authors would also like to thank the anti-malware experts of Comodo Security Lab for the data collection as well as helpful discussions and supports. This work is partially supported by the U.S. National Science Foundation under grant CNS-1618629.

References

[1] ADT Monkey, http://developer.android.com/tools/help/monkey.html.
[2] Android application fundamentals, http://developer.android.com/guide/components/fundamentals.html.
[3] Android, iOS combine for 91 percent of market, http://www.cnet.com.
[4] Apktool, http://ibotpeaches.github.io/Apktool/.
[5] Y. Bengio. Learning Deep Architectures for AI. In Foundations and Trends in Machine Learning, Vol 2(1), 1-127, (2009).
[6] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy Layer-Wise Training of Deep Networks. In NIPS, (2007).
[7] M. Bierma, E. Gustafson, J. Erickson, D. Fritz, Y. Choe. Andlantis: Large-scale Android Dynamic Analysis. In SPW, (2014).
[8] I. Burguera, U. Zurutuza, S. Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. In ICDM, (2011).
[9] S.J. Chang. Ape: A smart automatic testing environment for android malware. (2013).
[10] DroidBox, https://github.com/pjlantz/droidbox.
[11] M. Dimjaevi, S. Atzeni, I. Ugrina, and Z. Rakamari. Evaluation of android malware detection based on system calls. In IWSPA, (2016).
[12] M. Dimjasevic, S. Atzeni, I. Ugrina, Z. Rakamaric. Android Malware Detection Based on System Calls. In UUCS, (2015).
[13] 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 OSDI, (2010).
[14] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In SPSM (2011).
[15] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In AISec, 2013.
[16] Genymotion, https://www.genymotion.com/.
[17] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: scalable and accurate zero-day android malware detection. In MobiSys, (2012).
[18] G. E. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. In Neural Computation, Vol 18, 1527-1554, (2006).
[19] T. Isohara, K. Takemori, A. Kubota. Kernel-based Behavior Analysis for Android Malware Detection. In CIS, (2011).
[20] Kaspersky Security Bulletin 2015. https://securelist.com/.
[21] A. Krizhevsky, I. Sutskever and G. Hinton. Imagenet classification with deep convolutional neural networks. Proc. Adv. Neural Inf. Process. Syst. Vol. 25, 1106–1114, (2012).
[22] H. Larochelle, Y. Bengio, J. Louradour and P. Lamblin. Exploring strategies for training deep neural networks. J. Mach. Learn. Res. Vol. 10, 1–40, (2009).
[23] Y. Lv, Y. Duan, W. Kang, Z. Li, F. Wang. Traffic Flow Prediction With Big Data: A Deep Learning Approach. In Intelligent Transportation Systems, 865 - 873, (2014).
[24] N. Peiravian, X. Zhu. Machine Learning for Android Malware Detection Using Permission and API Calls. In ICDM, (2013).
[25] Strace, https://en.wikipedia.org/wiki/Strace.
[26] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss. Andromaly: a behavioral malware detection framework for android device. J.Intell Inf Syst, 161-190 (2012).
[27] Two Billion Consumers Worldwide to Get Smart(phones) by 2016, http://www.emarketer.com.
[28] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P. Manzagol. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. In Journal of Machine Learning Research, Vol 11, 3371-3408, (2010).
[29] P. Wood. Internet Security Threat Report 2015. Symantec, California (2015).
[30] D. Wu, C. Mao, T. Wei, H. Lee, K. Wu. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In Asia JCIS, (2012).
[31] W. Wu, S. Hung. DroidDolphin: a dynamic Android malware detection framework using big data and machine learning. In RACS, (2014.
[32] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, P. Porras. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. In Computer Science, Vol 8712, 163–182 (2014).
[33] R. Xu, H. Sadi, R. Anderson. Aurasium: practical policy enforcement for Android applications. In USENIX, (2012).
[34] Z. Yuan, Y. Lu, Z. Wang, Y. Xue. Droid-Sec: Deep Learning in Android Malware Detection. In SIGCOMM, (2014).
[35] Z. Yuan, Y. Lu , Y. Xue. Droiddetector: android malware characterization and detection using deep learning. Tsinghua Science and Technology, Vol 21 ,114–123 (2016).
[36] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, W. Zou. SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In SPSM, (2012).