Sequencing System Calls for Effective Malware Detection in Android

A. S. M. Ahsan-Ul-Haque[1], Md. Shohrab Hossain[1] and Mohammed Atiquzzaman[2]
[1]Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Bangladesh
[2] School of Computer Science, University of Oklahoma, Norman, OK, USA
Email: ahsanhaquetarique@gmail.com, mshohrabhossain@cse.buet.ac.bd, atiq@ou.edu

***Abstract*—Malware is one of the biggest threats for the privacy and security of the smart-phone users. Android is currently the most popular operating system for smart-phones; consequently, many malwares are directed toward Android devices. However, the existing malware detection techniques tend to compromise between accuracy and computational complexity. In this paper, we have proposed a novel technique to monitor the behavior of both malwares and benign applications using system calls and have developed a mathematical model that can detect mobile malwares. We have extracted features by sequencing the system calls of these application. We have proposed a novel way of feature reduction using Gaussian dissimilarity and compared our feature selection technique with existing methods. Using the extracted set of features, we have implemented a machine learning classifier, namely Gaussian Bayes classifier, on two different malware data-sets (obtained from Malware Genome Project and Android Malware Dataset by Arguslab) and on non-malware samples (obtained from Google Play Store). We have found that our model is quite lightweight yet powerful to detect malwares with significant accuracy of 98%.**

***Index Terms*—Android, Malware, Strace, ADB, System Call**

I. Introduction

Malware, a generic term for malicious software, refers to any software designed to cause damage to a single computer, server, or a computer network [1]. The activities of a malware may range from corrupting system files, manipulating data of other applications, to the extent of tracing keystrokes entered by the user, tracking data sent over Bluetooth, Wireless LAN, USB or monitoring network data used by other applications. The survey, conducted by Kaspersky Lab and Interpol, shows that 77% of accesses to the internet are performed by the users through the help of a smart-phones and tablets [2]. Mobile applications are more preferable for social networking services, managing emails, making transactions, etc. When infected with malwares, these applications may breach privacy and leak sensitive data to other parties, track financial records, such as bank account number, related PIN number, etc.

The global use of smart-phone is on the rise. Smart-phone users rely greatly on the efficiency of the tools and techniques used for the detection of malwares. The main *objective* of our work is to develop a technique for effective and accurate malware detection.

There exist several popular operating systems for smart-phones, such as iOS, Windows and Android. However, we have chosen Android platform for malware detection, mainly because of the following reasons: (1) most smart-phone devices run on Android. As per reports of International Data Corporation (IDC) [3], Android had total share of 81.4% in smart-phone market in last quarter of 2016 and it grew to 85.0% by the beginning of 2017. (2) Most smart-phone malwares are created for Android operating system. In 2014, Kaspersky Lab reported that the target of almost 98.05% of all existing mobile malwares is to attack Android users [2]. (3) Android is based on one of the Linux kernel's Long-Term Support (LTS) branches [4]. Hence, developing a malware detection model for Android would allow us to use the same for other Linux-based operating systems as well.

There exists many static and dynamic methods for malware detection. In static detection techniques [5]–[7], the detection is done without running the applications. Although static detection is very fast in case of known malwares, it is obsolete against repackaged malwares. Dynamic detection techniques [8]–[11], are more immune to repackaging and steganography, but they require more time for training and detection. Thus, existing methods tend to compromise between accuracy and computational complexity. On the contrary, our proposed method *differs* from the existing works as we have implemented a dynamic method which is efficient and fast, without compromising the accuracy.

The *contributions* of our work are as follows:

- We have extracted a set of distinctive features using Markov model by analyzing the sequence of system call logs that are made by malwares and benign applications.
- We have proposed a novel and simple way of feature reduction technique using Gaussian dissimilarity.
- Using the reduced set of features in a standard machine learning classifier, we have validated the effectiveness of the features.
- We have validated the model using two different malware data sets from Malware Genome Project [12] and Android Malware Dataset (AMD) by ArgusLab [13].

We have trained and tested our detection method on a set of 981 malwares acquired from Android Malware Genome Project [12] and 234 malwares acquired from Android Malware Dataset provided by ArgusLab [13], and 319 non-malware applications acquired from Google Play Store. Results show that our malware detection approach have found significant improvement over the previous works with an accu-

racy rate of 98%. Our approaches will be helpful especially for those who face trouble with limited processing capabilities and battery constrained Android devices, since the set of features we have used in our model is easy to extract (using Android Device Bridge utility).

The rest of the paper is organized as follows. In Section II, we have outlined the existing malware detection techniques. Our proposed classification model (feature extraction process using Markov model) is explained in Section III. Section IV describes our approach for malware detection using Gaussian Bayes classifier. Section VI discusses the experimental setups and algorithms employed for the feature extraction, training and testing phases. In Section VII, the experimental results are presented to validate our model. Finally, Section VIII provides concluding remarks.

II. Related Works

Malware utilizes a combination of techniques to avoid detection and analysis, such as: dynamic execution [14], code obfuscation [15], repackaging [16] and steganography [17]. According to the nature of the analysis, the existing malware detection techniques in Android can be classified into static and dynamic methods. In static methods, only some fixed attributes of a potential malware application are tested against a malware signature database. Examples of static features include permissions and API calls which can be extracted from the AndroidManifest.xml file. SigPID is a framework presented by Lichao Sun et al. [5] for static permission based detection. In the dynamic detection method [10], [11], the run-time behaviors of an application (like network traffic, system calls, CPU usage etc.) are monitored.

While surveying the literature, we found that there are mainly four detection techniques used, which are as follows:

1) Permission based detection
2) Signature based detection
3) Network traffic analysis
4) Based on system calls

In this paper, system call based detection technique has been employed. Machine learning using system call based classification methods usually result in a high degree of accuracy since they are able to capture the run-time behavior of the mobile apps. Wahanggara and Prayud [9] presented system call based model that uses Support Vector Machine method where they had achieved 90% accuracy using polynomial kernel. Deep4MalDroid is a deep learning framework for Android malware detection based on system call graphs presented by Hou et al. [8] where they have achieved an accuracy of 93.68%. Markov model-based system call graph analysis with complex machine learning models get higher accuracy with the expense of greater time for training. Some approaches, such as by F. Ahmed et. al. [18], have tried to reduce the training time by reducing the feature set using Correlation Coefficient and Information Gain which resulted in an accuracy of 96.3% using API calls in Windows. In this paper, We have reduced the time by proposing a simpler way of feature reduction. We have also used a very simple machine learning model (i.e. Naive Gaussian Bayes Model) which does not require any hyper-parameter. Consequently, we have improved the training time without compromising the accuracy.

III. Classification Model: Sequencing System Calls Using Markov Chain

This section establishes the relationship between our malware detection model with Markov model and shows the steps of the feature extraction process.

A. Preprocessing

A system call is a request to kernel made by an active process. Linux kernel implements around 300 system calls [19]. We collected system call traces of all applications of the data set consisting of both malware and non-malware, the process of which is described in details in Section VI. A sample trace log may look like this: [ioctl, writev, getuid32, ...], where each entry is a system call.

Table I lists all the symbols and corresponding descriptions which are defined in this Section and in Section IV, and used throughout the rest of the paper.

TABLE I
Summary of Notations

Symbol	Description
N	Number of unique system calls
L	Length of system call sequence
$[A_{i,j}]_{i=1,j=1}^{i=r,j=c}$	matrix A of dimension $r \times c$
$\delta_l^{(m)}(s_i, s_j)$	Transition freq. from s_i to s_j, upto l, for m^{th} sample
$\Delta^{(m)}$	Transition freq. matrix for m^{th} sample
$f_l^{(m)}(s_i)$	Freq. of system call s_i upto length l, for m^{th} sample
$F_m(s_i)$	Freq. of system call s_i upto length K, for m^{th} sample
$p^{(m)}(s_i, s_j)$	Transition probability from s_i to s_j for m^{th} sample
P_m	1-step transition probability matrix for m^{th} sample
$P_m^{(\lambda)}$	λ-step transition probability matrix for m^{th} sample
$\phi(A)$	Unrolling operation to turn A into column vector
X_i	i^{th} step feature vector
$X_i^{(m)}$	Value of i^{th} step feature vector for m^{th} sample
X	$X = [X_1, ..., X_\gamma]$, feature vector list of order γ
y	Class label, $y = 0$ for non-malware, $y = 1$ for malware
x_i	i^th feature of feature list X
$x_i^{(m)}$	Value of i^th feature of feature list X
M_y	Number of samples of class y during training
$\mu_i(y)$	Mean of x_i for class y
$\sigma_i^2(y)$	Variance of x_i for class y
$x^{(n)}$	n^{th} test sample or data point
$p(x^{(n)}\|y)$	Likelihood of n^{th} test sample generated from class y

B. Transition Frequency Matrix

Let $\{s_1, s_2, ..., s_N\}$ be the set of all the system calls (N being number of unique system calls) and $[s_{m_1}, s_{m_2}, ..., s_{m_L}]$ be the sequence of system calls for the m^{th} application, where L is the total number of system calls made by that application.

The transition frequency from s_i to s_j, up to length l for this application, $\delta_l^{(m)}(s_i, s_j)$, can be determined using Eqn. (1).

$$\delta_l^{(m)}(s_i, s_j) = \begin{cases} 0, & \text{if } l = 0, \text{ or } 1 \\ \delta_{l-1}^{(m)}(s_i, s_j) + 1, & \text{if } s_j = s_{m_l}, s_i = s_{m_{l-1}} \\ \delta_{l-1}^{(m)}(s_i, s_j), & \text{otherwise} \end{cases} \tag{1}$$

The full transition frequency matrix, $\Delta^{(m)}$, is given by Eqn. (2.) The entry $\Delta_{i,j}^{(m)}$ refers to the number of times the application uses j^{th} system call (s_j) immediately after i^{th} system call (s_i).

$$\Delta^{(m)} = \begin{bmatrix} \Delta_{1,1}^{(m)} & \dots & \Delta_{1,N}^{(m)} \\ \Delta_{2,1}^{(m)} & \dots & \Delta_{2,N}^{(m)} \\ \vdots & \ddots & \vdots \\ \Delta_{N,1}^{(m)} & \dots & \Delta_{N,N}^{(m)} \end{bmatrix} = [\delta_K^{(m)}(s_i, s_j)]_{i=1,j=1}^{i=N,j=N} \tag{2}$$

C. Transition Probability Matrix

For the m^{th} sample, we also need to record the frequency of i^{th} system call, $F^m(s_i)$, which is given by Eqn. (4). $f_l^{(m)}(s_i)$ is the frequency up to length l, as given by Eqn. (3).

$$f_l^{(m)}(s_i) = \begin{cases} 0, & \text{if } l = 0 \\ f_{l-1}^{(m)}(s_i) + 1, & \text{if } s_i = s_{m_l} \\ f_{l-1}^{(m)}(s_i), & \text{otherwise} \end{cases} \tag{3}$$

$$F_m(s_i) = f_L^{(m)}(s_i) \tag{4}$$

Finally, we get the transition probability of j^{th} system call (s_j) immediately after the i^{th} system call (s_i), by using Eqn. (5).

$$p^{(m)}(s_i, s_j) = \begin{cases} \frac{\Delta_{i,j}^{(m)}}{F^m(s_i)}, & \text{if } F^m(s_i) \neq 0 \\ 0, & \text{if } F^m(s_i) = 0 \end{cases} \tag{5}$$

P_m denotes the matrix of one-step transition probabilities for the m^{th} application, and is given by Eqn. (6).

$$P_m = P_m^{(1)} = [p^{(m)}(s_i, s_j)]_{i=1,j=1}^{i=N,j=N} \tag{6}$$

We know from Markov chain, two-step probability matrix can be obtained by

$$P_m^{(2)} = P_m^{(1)} \cdot P_m^{(1)} = P_m^2 \tag{7}$$

Similarly, we can obtain any λ-step probability matrix by raising P_m to the power of λ,

$$P_m^{(\lambda)} = P_m^\lambda \tag{8}$$

Markov transition probability matrix can be shown using Complete Directed Graph. Fig. 1 shows the graphical representation of a Markov transition probability matrix with only three system calls. The nodes S_1, S_2 and S_3 in the graph represent the system calls. The labels on the directed edges represent the transition probabilities; for example: $P_{m1,2}$ denotes the 1-step probability of system call S_2 right after S_1 for the m^{th} application.

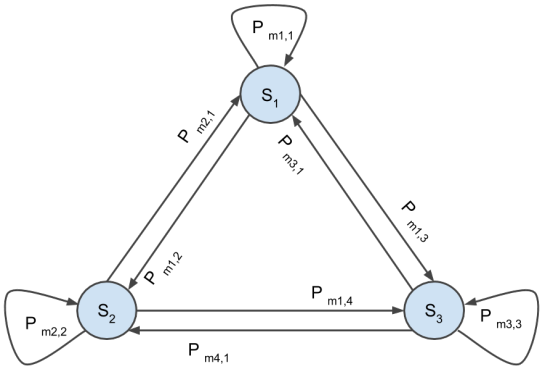

Fig. 1. The P_m matrix for the m^{th} sample can be viewed as a complete Markov chain. Here is an example showing 3 system calls.

Next, we define the unrolling operation of a matrix which converts any matrix to a column vector. Let, A be any matrix of dimension $d \times n$.

$$A = [A_{i,j}]_{i=1,j=1}^{i=d,j=n} = \begin{bmatrix} A_{1,1} & \dots & A_{1,n} \\ A_{2,1} & \dots & A_{2,n} \\ \vdots & \ddots & \vdots \\ A_{d,1} & \dots & A_{d,n} \end{bmatrix} \tag{9}$$

The unrolling operation, ϕ of a matrix is defined as,

$$\phi(A) = [A_{1,1} \dots A_{1,n} A_{2,1} \dots A_{2,n} \dots A_{d,1} \dots A_{d,n}]^T \tag{10}$$

D. Feature Extraction

After the unrolling operation of i^{th} step transition probability matrix, we get the i^{th} step feature vector. $X_i^{(m)}$ is the data point corresponding to the m^{th} application. $X_i^{(m)}$ lies in a hyper-plane of dimension $[0, 1]^{N^2}$.

$$X_i^{(m)} = \phi(P_m^{(i)})^T \tag{11}$$

The features in X_i denote the probabilities of one system call occurring right after i^{th}-step of another system call, as previously described. Since all the entries in the feature matrices denote probability, we do not need to further normalize the features.

In general, we have used the list of feature vectors X of order γ, as given by Eqn. (12), which consists of a total of γN^2 features.

$$X = [X_1, X_2, ..., X_\gamma] \tag{12}$$

IV. Gaussian Bayes Classification Method

In the classification step, we assume that each feature follows a normal (Gaussian) distribution (The Gaussian approximation of a sample feature distribution is shown in Fig. 7). We discuss the classification task using the features

of order $\gamma = 1$, i.e., $X = [X_1]$. So, the number of features is limited to N^2.

A. Training

Let x_i be the i^{th} feature of the training samples and $x_i^{(m)}$ is the value of the feature for m^{th} sample. Now, let y denote the class (or label) of the samples. We have two classes, defined as in Eqn. (13).

$$y = \begin{cases} 1, & \text{for malware} \\ 0, & \text{for non-malware} \end{cases} \tag{13}$$

Let M_0 and M_1 be the number of non-malwares and the number of malwares used in the training respectively. $\mu_i(y)$ is the mean and $\sigma_i^2(y)$ is the variance of the i^{th} feature, given class y.

We get $\mu_i(y)$ from Eqn. (14).

$$\mu_i(y) = \frac{1}{M_y} \sum_{m=1}^{M_y} x_i^{(m)} \tag{14}$$

Then, we calculate the variance using Eqn. (15).

$$\sigma_i^2(y) = \frac{1}{M_y} \sum_{m=1}^{M_y} (x_i^{(m)} - \mu_i(y))^2 \tag{15}$$

The normal distribution over the i^{th} feature x_i, given the class y is

$$p(x_i : \mu_i(y), \sigma_i^2(y)|y) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp(-\frac{(x_i - \mu_i)^2}{2\sigma_i^2}) \tag{16}$$

Eqn. (16) represents the likelihood of x_i being generated from class y.

B. Classification

We have used maximum likelihood estimation in the classification method. For a new n^{th} sample with data-point $x^{(n)}$ (with i^{th} feature value being $x_i^{(n)}$) we calculate $p(x_i^{(n)}|y = 0)$ and $p(x_i^{(n)}|y = 1)$ using Eqn. (16).

Thus, assuming independence among the features, the likelihood of the new sample being malware is given by Eqn. (17).

$$p(x^{(n)}|y = 1) = \prod_{i=1}^{N^2} p(x_i^{(n)} : \mu_i(y), \sigma_i^2(y)|y = 1) \tag{17}$$

Similarly, the likelihood of the new sample being non-malware is given by Eqn. (18).

$$p(x^{(n)}|y = 0) = \prod_{i=1}^{N^2} p(x_i^{(n)} : \mu_i(y), \sigma_i^2(y)|y = 0) \tag{18}$$

Finally we classify the new sample as,

$$\begin{cases} \text{app is a malware,} & \text{if } p(x^{(n)}|y = 1) > p(x^{(n)}|y = 0) \\ \text{app is not a malware,} & \text{otherwise} \end{cases}$$

We can simplify Eqn. (17) and Eqn. (18) by using maximum log likelihood as shown in Eqn. (19).

$$\prod_{i=1}^{N^2} p(x_i^{(n)}|y = 1) > \prod_{i=1}^{N^2} p(x_i^{(n)}|y = 0) \tag{19}$$

$$\Rightarrow \sum_{i=1}^{N^2} log(p(x_i^{(n)}|y = 1)) > \sum_{i=1}^{N^2} log(p(x_i^{(n)}|y = 0)) \tag{20}$$

Hence, the classification of the new n^{th} sample is given by,

$$\begin{cases} \text{app is a malware,} & \text{if } \sum_{i=1}^{N^2} log(p(x_i^{(n)}|y = 1)) > \\ & \sum_{i=1}^{N^2} log(p(x_i^{(n)}|y = 0)) \\ \text{app is not a malware,} & \text{otherwise} \end{cases}$$

V. Feature Selection

In high dimensional data set feature selection is essential to reduce the training time. As we can see from the previous subsection, the extracted features have the space complexity of $O(\gamma N^2)$. There are many well known feature subset selection heuristics, such as Information Gain, Gain Ratio etc. In this paper, we propose another way of feature reduction, using Gaussian Dissimilarity (GD).

Let $\mu_i(y)$ and $\sigma_i^2(y)$ respectively be the mean and the variance of class y The dissimilarity heuristic for the i^{th} feature is defined in Eqn. (21).

$$\begin{aligned} GD(i) = -(&\frac{1}{\sqrt{2\pi\sigma_i^2(0)}} \exp(-\frac{(\mu_i(1) - \mu_i(0))^2}{2\sigma_i^2(0)}) \\ &+ \frac{1}{\sqrt{2\pi\sigma_i^2(1)}} \exp(-\frac{(\mu_i(1) - \mu_i(0))^2}{2\sigma_i^2(1)})) \end{aligned} \tag{21}$$

We compute Gaussian Dissimilarity for each element of X. Then, the features are sorted based on their GD values in a descending order. We select $X_r \subseteq X$, the first r features from the list. In this paper, we have used $r = 500$.

This heuristic is effective because if two features have relatively close mean and variance values for both malware and non-malware samples, then that feature will not be able to differentiate malwares from non-malwares effectively using Gaussian Bayes classifier. From Eqn. (21), we can see that for this type of features, the GD values will be very small. Thus, the features with higher GD values will be able to more effectively differentiate between malware and non-malware.

Since both of the terms on the right hand of Eqn. (21) are non-negative, we introduce a new heuristic Logarithmic Gaussian Dissimilarity (LGD), as given in Eqn (22) and (23). LGD will be computationally more efficient since the exponential terms are absent from this heuristic.

$$\begin{aligned} LGD(i) \sim &\frac{1}{2} log(2\pi\sigma_i^2(0)) + \frac{(\mu_i(1) - \mu_i(0))^2}{2\sigma_i^2(0)}) \\ &+ \frac{1}{2} log(2\pi\sigma_i^2(1)) + \frac{(\mu_i(1) - \mu_i(0))^2}{2\sigma_i^2(1)}) \end{aligned} \tag{22}$$

Removing the constant terms from Eqn. (21), we get the new Logarithmic Gaussian Dissimilarity (LGD) heuristic, as given in Eqn. (23).

$$\begin{aligned} LGD(i) &= log(\sigma_i(0)) + \frac{(\mu_i(1)-\mu_i(0))^2}{2\sigma_i^2(0)}) \\ &+ log(\sigma_i(1)) + \frac{(\mu_i(1)-\mu_i(0))^2}{2\sigma_i^2(1)}) \\ &= log(\sigma_i(0)\sigma_i(1)) \\ &+ (\mu_i(1)-\mu_i(0))^2(\frac{\sigma_i(0)^2+\sigma_i(1)^2}{2(\sigma_i(0)\sigma_i(1))^2})) \end{aligned} \tag{23}$$

VI. Experiments

In this Section, we discuss the process of implementing the model, developed in Section IV, using the physical Android device.

A. Experimental Setup

We let all the applications (both malware and non-malware) run sequentially on the same *rooted* Android device running on Lollipop 5.1 (API 22). Our device choice helped us test the behavior of more than 80% of the total Android devices at present [4]. The run-time was also same for all applications. This approach ensured the same environment for all of the applications. The complete workflow is outlined in Fig. 2. We used a total of 1215 malwares combined from two data-sets and 319 non-malwares acquired from Google Playstore, as shown in Table II.

TABLE II
Number of Mobile Apps

	Malware Genome Project	Android Malware Dataset	Google Play Store	Total
Malware	981	234	-	1215
Non-malware	-	-	319	319

We used standard Linux utility *strace* to collect system call information of the applications. The strace log of a sample application is shown in Fig. 3.

From the strace logs, we filtered out only the system calls made by the applications. After this step, the system call log of a sample application is shown in the Fig. 4.

Algorithm 1 shows how to parse the strace logs and get the sequence of system calls in terms of unique ids using a database. After this step, we found a total of 61 unique system calls (N) used by the applications, as shown in Fig. 5.

After creating the Δ, F and P relational matrices for each application, we extracted the feature vector X and finally acquired the desired reduced set of features X_r. Every entry in each of the data set denotes a probability, as explained in Subsection III-C. The feature values of a sample application is shown in Fig 6.

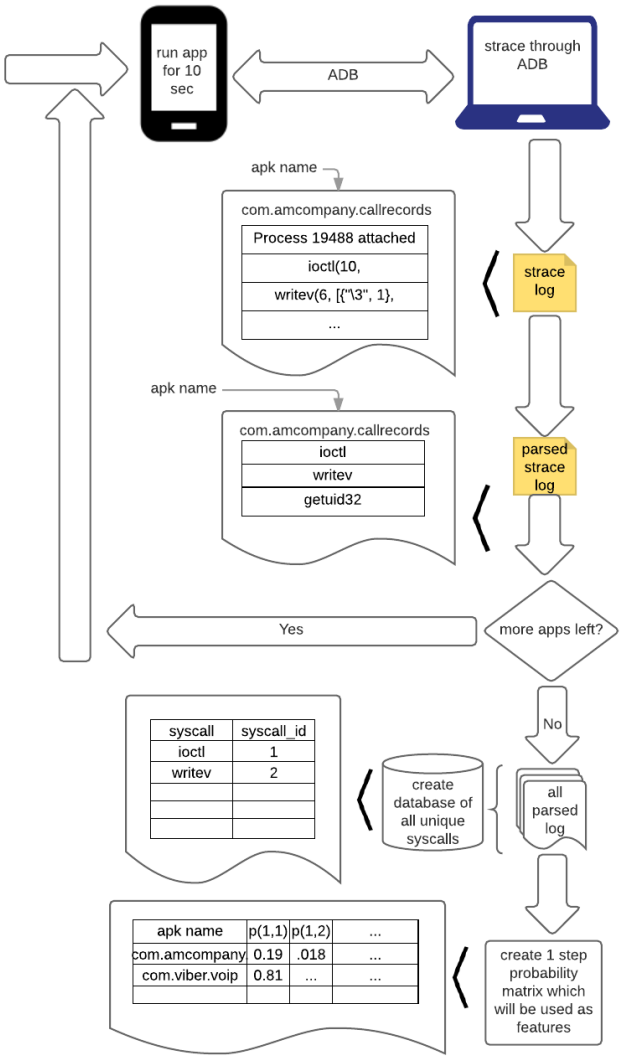

Fig. 2. Work-flow diagram of preprocessing and feature extraction

```
1  Process 19488 attached
2
3  ioctl(10, _IOC(_IOC_READ|_IOC_WRITE, 0x62, 0x01, 0x18),
   0xbebfdf30) = 0
4
5  writev(6, [{"\3", 1}, {"ActivityThread\0", 15}, {"ACT-
   RESUME_ACTIVITY handled : 1 "..., 65}], 3) = 81
6
7  getuid32()                              = 10167
8
9  epoll_pwait(22, {{EPOLLIN, {u32=84, u64=84}}}, 16, 0, NULL, 8) = 1
10
11 recvfrom(84, "nysv\0\0\0\0S\235\273a\232\5\0\0\312U\3\0====",
   2400, MSG_DONTWAIT, NULL, NULL) = 24
12
13 recvfrom(84, 0xbebfd670, 2400, 64, 0, 0) = -1 EAGAIN (Resource
   temporarily unavailable)
```

Fig. 3. System call log of an application

B. Implementing Gaussian Bayes Classifier

We used Scikit-Learn [20], which is a Python library for machine learning, for training and testing and for measuring the classification performance on the reduced feature sets. We chose the Naive Gaussian Bayes classifier with full training

```
1  ioctl
2  writev
3  getuid32
4  epoll_pwait
5  recvfrom
6  recvfrom
7  clock_gettime
```

Fig. 4. Parsed System call trace of an application

Algorithm 1 Parse System Call

```
procedure PARSE-SYSCALL(stracelogs)
    id ← 1
    for each log in stracelogs do
        do
            read system call s from log
            if s does not match any previous system calls then
                assign id to system call s and update database
                current_id ← id
                id ← id + 1
            else
                current_id ← id of s from database
            end if
            add id to file straceid[log]
        while there are system calls remaining in s
    end for
    return straceid
end procedure
```

set and K-fold cross validation for the testing purpose, with $K = 7$ to 10, the results of which are shown in Table III. Table II shows the sources of the data-set used during the experiments.

VII. Experimental Results

The results of feature selection and K-fold cross validation is discussed in this section.

A. Experimental values and Gaussian approximations

The probability density distribution of one of the sample features and the Gaussian approximation of the sample feature are shown in Fig. 7, for both malware and non-malware classes. The low value of variance of malwares, in comparison with non-malwares, denotes that there are not much difference between the training samples for the selected feature, i.e., the system call transitions for the malwares are very similar for this feature.

B. Performance Metrics

In K-fold cross validation the True Positives (TP), False Positives (FP), True negatives (TN) and False Negatives (FN) values are actually averaged over the $K-1$ test subsets. We calculated the following metrics to determine the performance of our model, as shown in Table III.

1) **Accuracy (α):** It refers to the proportion of correct labeling of all the test data, which can be expressed as follows:

$$\alpha = \frac{TP + TN}{TP + TN + FP + FN}.$$

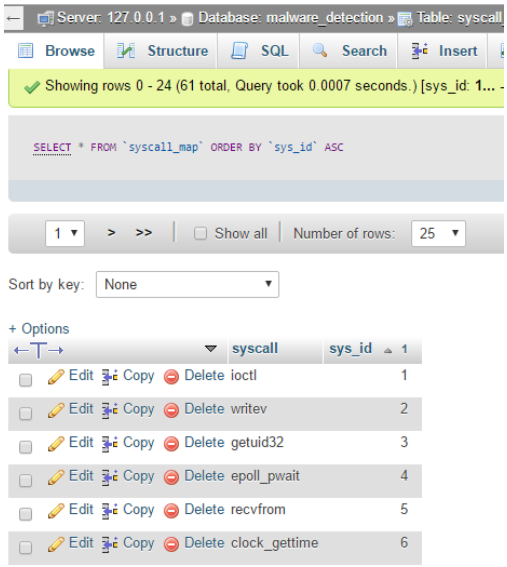

Fig. 5. The database containing all the unique system calls

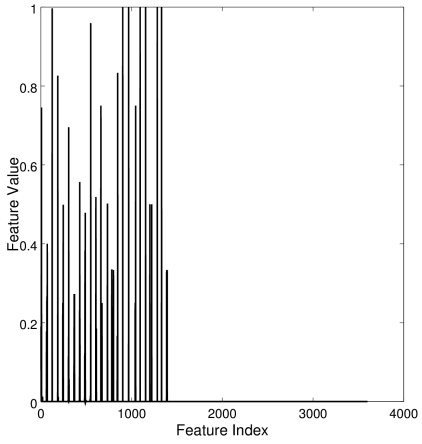

Fig. 6. Feature values vs indices of a sample application

2) **Recall (r):** It measures the proportion of malwares that are correctly identified and are computed as follows:

$$r = \frac{TP}{TP + FN}.$$

3) **Specificity (ψ):** Specificity (ψ) measures the proportion of non-malware that are correctly identified.

$$\psi = \frac{TN}{TN + FP}$$

4) **Precision (ρ):** It measures the proportion of the detected malware that are actually malware.

$$\rho = \frac{TP}{TP + FP}$$

5) **F1-score (η):** F1-score is the harmonic mean of precision and recall.

$$\eta = 2 \cdot \frac{\rho \times r}{\rho + r}$$

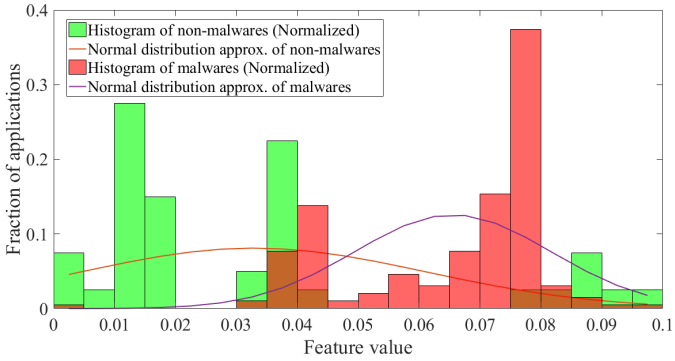

Fig. 7. Histogram and distribution approximation for a single feature

TABLE III
PERFORMANCE METRICS (%)

K	α	r	ψ	ρ	η
10	98	100	89.2	97.6	98.8
9	98.4	100	91.4	98.1	99
8	99	100	94.6	98.8	99.4
7	99.7	100	98.2	99.6	99.8

Using the proposed feature selection heuristic and 10-fold cross validation, our model has achieved an accuracy of 98% whereas DREBIN [21] claims an accuracy of 95.9% and DroidAPIMiner [22] reached an accuracy of 99%. So, our model outperforms the existing frameworks on the same dataset obtain from Malware Genome Project.

VIII. CONCLUSION

In this paper, we proposed a definitive and novel approach of malware detection in Android devices. We extracted a distinctive set of features which takes into account the probability of one system call occurring after another. To improve the training time, we proposed a novel way of feature selection using Gaussian Dissimilarity and Logarithmic Gaussian Dissimilarity. We used the set of features in a standard yet simple machine learning model to evaluate the effectiveness of the features. We evaluated the performance of our model using two different malware data sets obtained from Malware Genome Project [12] and Android Malware Dataset provided by ArgusLab [13]. The experimental results show that our model has a significant accuracy of 98% to detect mobile malwares. Furthermore, the graphical representations of the features give us an insight regarding the high accuracy of the model that will help other researchers with the required quantitative information and guidelines.

REFERENCES

[1] Defining Malware. Web page. [Online]. Available: https://technet.microsoft.com/en-us/library/dd632948.aspx

[2] Kaspersky Lab and INTERPOL, "Mobile Cyber Threats," October, 2014. [Online]. Available: https://media.kaspersky.com/pdf/Kaspersky-Lab-KSN-Report-mobile-cyberthreats-web.pdf

[3] International Data Corporation, Web page. [Online]. Available: http://www.idc.com/promo/smartphone-market-share/os

[4] Google Developer Site. Web page. [Online]. Available: https://developer.android.com/guide/platform/index.html/#linux-kernel

[5] L. Sun, Z. Li, Q. Yan, W. Srisa-an, and Y. Pan, "SigPID: Significant Permission Identification for Android Malware Detection," in *11th International Conference on Malicious and Unwanted Software (MALCON)*, Puerto Rico, USA, October, 2016.

[6] Z. Aung and W. Zaw, "Permission-Based Android Malware Detection," *International Journal of Scientific & Technology Research*, vol. 2, March, 2013.

[7] S. Hou, T. Lu, Y. Du, and J. Guo, "Static Detection of Android Malware Based on Improved Random Forest Algorithm," in *IEEE International Conference on Intelligence and Security Informatics (ISI)*, Beijing, China, July, 2017, pp. 200–200.

[8] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4MalDroid: A Deep Learning Framework for Android Malware Detection Based on Linux Kernel System Call Graphs," in *IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, Omaha, NE, USA, Oct 13-16, 2016.

[9] V. Wahanggara and Y. Prayudi, "Malware Detection Through Call System on Android Smartphone Using Vector Machine Method," in *Fourth International Conference on Cyber Security, Cyber Warfare, and Digital Forensic (CyberSec)*, Jakarta, Indonesia, October, 2015, pp. 62–67.

[10] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: Having a Deeper Look into Android Applications," in *ACM Symposium on Applied Computing*, New York, NY, USA, 2013.

[11] G. Cabau, M. Buhu, and C. P. Oprisa, "Malware Classification Based on Dynamic Behavior," in *18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, Timisoara, Romania, Sept, 2016, pp. 315–318.

[12] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *IEEE Symposium on Security and Privacy*, San Francisco, California, USA, May, 2012.

[13] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep Ground Truth Analysis of Current Android Malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Bonn, Germany: Springer, July, 2017, pp. 252–276.

[14] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, March, 2007.

[15] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Int. Conference on Broadband, Wireless Computing, Communication and Applications*, Fukuoka, Japan, November, 2010, pp. 297–300.

[16] M. Zheng, P. P. C. Lee, and J. C. S. Lui, "ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems," in *9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Heraklion, Crete, Greece: Springer, July 26-27, 2012.

[17] S. Nagaraja, A. Houmansadr, P. Piyawongwisal, V. Singh, P. Agarwal, and N. Borisov, "Stegobot: A Covert Social Network Botnet," *Information Hiding, Lecture Notes in Computer Science*, vol. 6958, pp. 299–313, 2011.

[18] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq, "Using Spatio-temporal Information in API Calls with Machine Learning Algorithms for Malware Detection," in *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*, New York, NY, USA, 2009, pp. 55–62.

[19] R. Love, *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly, 2013.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[21] D. Arp, M. Spreitzenbarth, M. Hbner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *Symposium on Network and Distributed System Security (NDSS)*, 02 2014.

[22] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android," in *Security and Privacy in Communication Networks*, T. Zia, A. Zomaya, V. Varadharajan, and M. Mao, Eds. Cham: Springer International Publishing, 2013, pp. 86–103.