Dynamic Android Malware Classification Using Graph-Based Representations

Lifan Xu
University of Delaware
xulifan@udel.edu

Dongping Zhang
AMD Research
dongping.zhang@amd.com

Marco A. Alvarez
University of Delaware
malvarez@udel.edu

Jose Andre Morales
Carnegie Mellon University
jamorales@cert.org

Xudong Ma
Clifford International School
xudong.ma@hotmail.com

John Cavazos
University of Delaware
cavazos@udel.edu

***Abstract*—Malware classification for the Android ecosystem can be performed using a range of techniques. One major technique that has been gaining ground recently is dynamic analysis based on system call invocations recorded during the executions of Android applications. Dynamic analysis has traditionally been based on converting system calls into flat feature vectors and feeding the vectors into machine learning algorithms for classification.**

In this paper, we implement three traditional feature-vector-based representations for Android system calls. For each feature vector representation, we also propose a novel graph-based representation. We then use graph kernels to compute pair-wise similarities and feed these similarity measures into a Support Vector Machine (SVM) for classification. To speed up the graph kernel computation, we compress the graphs using the Compressed Row Storage format, and then we apply OpenMP to parallelize the computation. Experiments show that the graph-based representations are able to improve the classification accuracy over the corresponding feature-vector-based representations from the same input. Finally we show that different representations can be combined together to further improve classification accuracy.

I. Introduction

Android has increasingly grown in popularity and has become the most popular smart phone platform with 295.2 million shipments and an 84.6% market share in the second quarter of 2014 [18]. Unfortunately, the growing popularity of Android smart phones and tablets has made this popular OS a prime target for cybersecurity attacks. By the first quarter of 2014, Android's share of new global mobile malware had risen to 99% [12], creating an urgent need for effective defense mechanisms to protect Android enabled devices.

In recent years, several researchers have proposed defensive strategies to counter the increasing amount and sophistication of Android malware. These methods can be categorized into: static analysis, dynamic analysis, and a hybrid of both techniques. Static analysis is based on extracting features by inspecting application's manifest and disassembled code [3]. By contrast, dynamic analysis methods monitor the application behavior during its execution [10]. Hybrid methods typically analyze the application before installation and also record the execution behavior [25], [17], [29], [16]. Both sets of information are then used together for detecting a malicious application.

One key behavioral feature used in dynamic analysis of malware is the system call invocations [26], [22], [5], [31], [13], [7], [14], [28], [10]. In previous work on Android malware analysis, the most common representation of the set of system call invocations is to convert them into histograms. However, other representations including signatures, n-grams, and Markov Chains have been studied previously in Windows-based malware analysis research [11], [2], [23], [6], [21]. Despite the differences between all these techniques, all these methods have varying degrees of accuracies and false positive rates, and each of the approaches has its drawbacks. The histogram representation can capture the distribution of system calls, but ignores the structural information. The signature method inherently prevents the detection of unknown malware of which no signatures exist. N-gram analysis is not only unable to capture malware structural information, but also introduces pressure on computational resources due to its large feature space. A Markov Chain-based representation takes advantage of the transition probabilities between system calls, but it cannot record their order and structure. In this work, we show that using of structure is important. Using structure we are able to achieve up to 87.3% classification accuracy while without using structure we only achieve 83.3%.

Most previous work that used histograms, signatures, n-grams, and Markov Chains used a feature-vector-based representation. In this paper, we first describe these representations from previous work that we reimplemented for our study, namely system call histograms, n-grams, and Markov Chains for Android malware analysis. Then, we describe novel graph-based representations, one for each of the three feature-vector-based representation, we call the three new graph-based representations we developed as follows: the Histogram System Call Graph (HSCG), the N-gram System Call Graph (NSCG), and the Markov Chain System Call Graph (MCSCG). In the HSCG, processes executed with direct ancestral lineage to the main process are recorded. The main process is the first process created for the application, and thus the first node in the graph. Each process is treated as a vertex and labeled with a histogram of its system call invocations. The graph is formed by connecting nodes representing parent/children processes. The NSCG is similar to HSCG except the nodes in the NSCG are labeled with n-gram vectors. Similarly, nodes in the MCSCG are labeled with the Markov Chain vectors.

For this research, we first use *strace*, the Linux system call utility to dynamically collect system call invocations from the execution of an Android application. We run each application in an Android emulator called Genymotion[1]. At the beginning of emulation, we run each application for a certain amount of time without interference. We then simulate a series of interactive events, and all the system call invocations that happened during the emulation are recorded. We present a set of methods to convert the system call trace from execution of a specific application into different feature-vector-based and graph-based representations. After conversion, we use a graph kernel to compute the pairwise graph similarities of the Android applications for each graph representation. The similarity measures are subsequently constructed and provided as a kernel matrix to a Support Vector Machine (SVM) for classification. Similarly, vector representations are also fed into an SVM to construct the classification model.

For feature-vector-based representations, Gaussian kernel, Linear kernel, and Intersect kernel algorithms are evaluated. For graph-based representations, we use the Shortest Path Graph Kernel (SPGK) algorithm on the HSCG, NSCG, and MCSCG graphs. SPGK is suitable for similarity computations of graphs with continuous labels (e.g., vectors). Since SPGK is expensive because of its $O(n^4)$ running time, we apply two techniques to speedup computation. First, we observe the vertex label for the HSCG, NSCG, and MCSCG graphs are long and sparse, so we compress these graphs and represent them using the Compressed Row Storage (CRS) format. Second, we utilize a multi-core CPU and parallelize the computation using OpenMP.

We show the kernel matrices of different representations can be combined together using multiple kernel learning method to further improve the classification accuracy.

Our major contributions are summarized below:

- We perform a quantitative evaluation of different feature-vector-based representations for Android malware analysis.
- We develop novel graph-based representations of system calls to improve classification accuracies of traditional feature-vector-based representations.
- We compress the feature vector labels of the graph representations and parallelize the graph kernel computation.
- We perform a thorough evaluation on an expressive dataset, demonstrating that graph representations offer better classification accuracy than the corresponding feature-vector-based ones.
- We show graph-based and feature-vector-based representations can be combined to further improve the performance.

II. Android Application Emulation

To capture runtime execution behavior of an application, we record system call invocations during the execution of the application using the Linux strace tool [1]. Our dynamic analysis of Android applications is performed in an emulated Android OS environment named Genymotion. It is well known that malware tends to carry out critical tasks upon initial execution. However, some malware may not execute malicious code until after user interaction or until it is triggered by particular events. Therefore, during our emulation, we first start the application and keep it running for some time without any interference. Then we perform a series of user interaction events using the Monkey toolkit provided by the Android SDK. We also perform other events including making phone calls, sending SMS, and movements while the app is running.

A. Emulation Procedure

The process of dynamically analyzing Android APK files is performed in a fully automated manner. Before analysis begins, the system is initialized as follows: First, from a Ubuntu 14.04 desktop, the Genymotion emulator is launched. Second, a folder of APK files is created, which serves as the sample repository to analyze. Third, a folder is created as the output repository to store the analysis result files. Given these initialization steps, a python script is executed to manage the analysis procedure and issues all necessary commands to the runtime environment. Communication between the python script and the runtime environment is performed using the Android Debug Bridge (ADB) tool. Given the initialization described above, the following steps are performed by the python script to load and launch an APK in the runtime environment and collect the execution data:

1) Install application from input repository
2) Retrieve $zygote$ process id (PID)
3) Run strace on $zygote$
4) Start an application and run for 20 seconds
5) Retrieve the application PID
6) Simulate user interactions using Monkey and run for 10 seconds
7) Simulate phone call events and run for 10 seconds
8) Simulate SMS message events and run for 10 seconds
9) Simulate phone movement events and run for 10 seconds
10) Simulate a second run of user interactions using Monkey and run for 10 seconds
11) Stop strace, move log files to output repository

We iterate on Steps 1 through 11 until all samples in the repository have been analyzed. Step 1 copies and installs the APK file into the Android emulator. Step 2 applies the ps command to retrieve the PID of the $zygote$ process. Step 3 starts the $strace$ command to record all system call invocations of $zygote$ and its descendant processes. The $zygote$ process is a standard process running in the Dalvik Virtual Machine that is a component of the Android runtime environment. Whenever an application is launched in Android, the associated process of the application is created and assigned a PID by $zygote$. By recording execution behavior of $zygote$, we are able to record the runtime execution behaviors of all the applications that are going to start later from the moment of their creation by $zygote$. Step 3 assures that we collect all relevant data of the application from its launching time because we trace its parent process $zygote$. Step 4 launches the application under analysis. The application is executed for 20 seconds without any interference. Step 5 issues a ps command

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

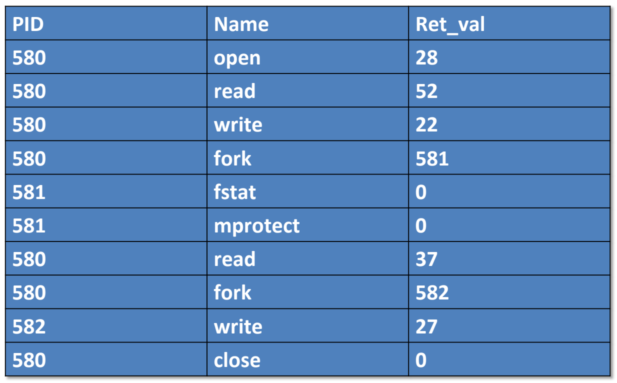
which provides a list of all currently running processes and their PID. This list is saved to a file used for identifying the PID of the application under analysis. Since the strace records all processes with direct ancestral lineage to $zygote$, there can be multiple processes that we can ignore during analysis. By recording the list of PIDs we are able to retrieve information only related to the application under analysis and its descendant processes and threads. The package name for the application under analysis is used to identify the running process in the ps output. We conduct our analysis by using the PID associated with the application's package name in the ps output. Steps 6-10 simulate different events. For Step 6 and 10, we use Monkey to generate 200 random events including touch events, motion events, trackball events, navigation events, system key events, and activity launching events. We also set the delay between events to 100 milliseconds. For Step 7, we simulate four phone call events including incoming phone calls, answering phone calls, making phone calls, and rejecting phone calls. For Step 8, we simulate receiving and sending SMS messages containing sensitive information like password and bank account information. For Step 9, we simulate the movement of a phone from one location to another. After each interaction step, we keep the application running for 10 seconds without any interference. Step 11 stops the strace and places the resulting files in the output repository. Two files are retrieved, one containing the strace data and the other contains the output of ps.

III. Strace Log Conversion

After we collect the strace data and the ps output for each application, the package name, retrieved from each application's manifest file, along with the strace and ps files, are used as input to a script that converts the strace files to multiple representations. This strace conversion method has two parts. The first extracts system call invocations only belonging to the testing application since the strace log file contains system call invocations of $zygote$ and all its child processes. A system call trace returned by the first part serves as input to the second part, which converts the trace to feature-vector-based or graph-based representations.

A. System Call Invocation Extraction

An important step in our strace conversion script is to look up the package name in the ps output to identify the PID of the application. With the strace data and PID of the application under analysis, our script can extract only the processes and system call invocations belonging to the testing application. In the strace log file, each line records one system call invoked by a particular PID. The lines can be parsed into columns that record PID, invocation time, system call name, parameters and return values respectively. Since the strace log file contains not only system call invocations of the application under analysis, but all processes with direct ancestral lineage to $zygote$, we need to extract information only related to the application of interest. To achieve this, our strace conversion script first creates a process list containing only the PID of the application. Then, it traverses the strace log file. If an invocation is made by a process in the process list, then the PID, name, and return value of this invocation are added as an entry into the system call trace. If the name of this invocation is $fork$ or $clone$, it means a child process is spawned. On success, both system calls return the PID of the child process. On failure, -1 is returned. If the return value of $fork$ or $clone$ is not -1, our conversion script adds the return value that is the PID of the spawned child process into the process list, then it continues traversing the strace log. Finally, a system call trace serving as an input for the subsequent conversion part is returned. For the purpose of demonstration, we create one synthetic system call trace shown in Figure 1.



PID	Name	Ret_val
580	open	28
580	read	52
580	write	22
580	fork	581
581	fstat	0
581	mprotect	0
580	read	37
580	fork	582
582	write	27
580	close	0

Fig. 1: Example System Call Trace.

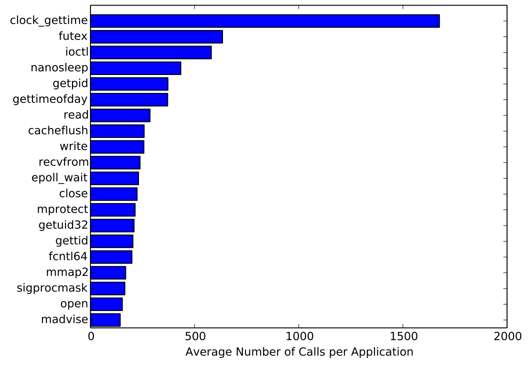
B. System Call List

From the full dataset of system call traces, we collect a system call list containing 213 unique system calls referred to as the full system call list. Since the length of the system call list plays a key role in most representations in terms of computation time, we want to keep the list as short as possible while maintaining the same classification accuracy. To find out the appropriate list, we compute the average number of invocations per application for each system call and sort them. The sorting is done separately for benign traces and malicious traces. We extract the top K system calls from both traces and merge them to get the reduced system call list. We then perform experiments by setting K to be 5, 10, 20, and 213. Results show that using the top 20 system calls is not as accurate as using the full system call list, but it is able to reach a very close classification accuracy with significantly reduced computation time. Detailed results are in Section VI-F and the top 20 system calls for benign and malicious applications are shown in Figure 2.

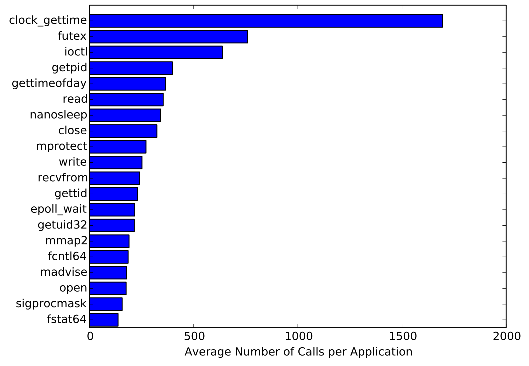
C. Feature-vector-based Representations

We first convert the system call traces to three previously studied feature-vector-based representations used in Android and Windows malware analysis.

1) System Call Histogram: The first previously studied representation of system call usage in Android malware analysis we look at is histograms. To convert a system call trace into a histogram, our strace conversion script takes the system call trace and a system call list as input. Then it parses each line of



(a) Benign application



(b) Malicious application

Fig. 2: Top 20 system calls per application on average.

the trace in order, finds the index of each system call name, and increments the corresponding element of the histogram by one. For demonstration purposes, we show the resulting histogram in Figure 3(a) converted from the system call trace listed in Figure 1. In our experiments, we feed our script both the full system call list and the top 20 system call list, and the resulting histograms are named $histogram\text{-}full$ and $histogram\text{-}top20$, respectively.

2) N-gram: Another previously studied representation in malware analysis is an n-gram [28]. An n-gram is a contiguous sequence of n system calls from the system call trace. There are two parameters associated with n-gram: n as the number of system call invocations in the sequence and L as the number of unique system calls which is also the size of the system call list. Given n and L, there can be L^n different n-grams. Therefore, the dimension of the resulting n-gram feature vector grows exponentially as we increase the value of n. In our experiments, we merge the top 20 system calls from benign and malicious applications, which gives us a value of $L = 23$. This is a significant reduction from 213, which is the original number of unique system calls. For n, we test 2-grams, 3-grams, and 4-grams. Figure 3(b) shows the 2-gram histogram converted from the given system call trace in Figure 1.

3) Markov Chain: Another representation of system calls that has been studied, in particular, in Windows malware analysis research, is Markov Chain [2]. It can be viewed as a directed graph where the vertices are the system calls and the edges are the transition probabilities calculated by the data contained in the trace. For a Markov Chain graph, $G = \langle V, E\rangle$, it consists of two sets, the vertex set V and the edge set E. V corresponds to the system calls, while E, corresponds to the transition probability from one system call to another. Given n system calls in the system call list, an adjacency matrix $A_{n\times n}$ can be used to represent the Markov Chain graph. For each element $A_{i,j}$ in the matrix, it presents the transition probability from system call i to system call j. The adjacency matrix can be treated as a 1D feature vector and fed into a machine learning model for classification. Fig. 3(c) shows the resulting Markov Chain converted from the given trace of system calls in Figure 1. In our experiments, we generate the Markov Chains using the top 20 system calls and the full system call list, which we have named $MarkovChain\text{-}top20$ and $MarkovChain\text{-}full$, respectively.

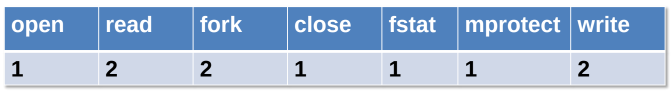
D. Graph-based Representations

To improve the classification accuracy when using the feature-vector-based representations, we propose a graph-based representation that augments each traditional feature vector representation. In these graphs, each vertex represents a process of the Android application and each vertex is labeled with a feature vector.

1) Histogram System Call Graph: First, we construct the Histogram System Call Graph (HSCG) as an alternative to a simple system call histogram representation. To generate an HSCG, our conversion script reads the trace line by line and parses each one to the triple of PID, system call name, and return value. If the process corresponding to the PID of this record is not yet a vertex of the graph, it is added as a vertex to the graph. If this is a $fork$ or $clone$ system call, the return value, which is the PID of the spawned child process, is also added as a vertex into the graph, and an edge from parent to child is also added. In an HSCG graph, each vertex is associated with a system call histogram generated from the system calls performed by the corresponding process. Fig. 4(a) shows the resulting Histogram System Call Graph converted from the given system call trace in Figure 1.

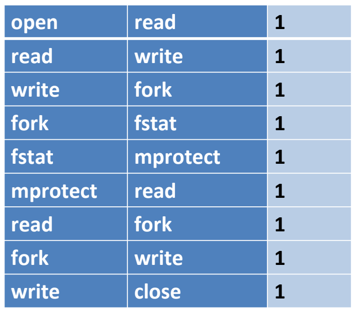
2) N-gram System Call Graph: We also create an N-gram System Call Graph (NSCG) as a graph-based alternative to the previously studied n-gram histogram. An NSCG shares the graph structure with an HSCG. The only difference is that the vertices in NSCG are labeled with an n-gram histogram for the corresponding process instead of the system call histogram. In our experiments, we created 2-gram, 3-gram, and 4-gram graphs for the corresponding n-gram histogram using the top 20 system call list which sets the L to be 23. Dimensions of vertex labels in these graphs are therefore 23^2, 23^3, and 23^4 respectively. Fig. 4(b) shows the resulting 2-gram System Call Graph converted from the system call trace.

3) Markov Chain System Call Graph: Similarly, we create the Markov Chain System Call Graph (MCSCG) as an alternative to the traditional Markov Chain representation. MCSCG shares the same graph structure with HSCG and NSCG. However, the vertices in MCSCG are labeled with a Markov Chain, instead of a histogram or an n-gram. In



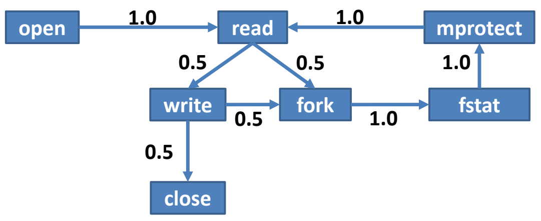
open	read	fork	close	fstat	mprotect	write
1	2	2	1	1	1	2

(a) System Call Histogram



open	read	1
read	write	1
write	fork	1
fork	fstat	1
fstat	mprotect	1
mprotect	read	1
read	fork	1
fork	write	1
write	close	1

(b) 2-gram Histogram



(c) Markov Chain

Fig. 3: Different Feature-vector-based Representations Converted from Fig. 1

our experiments, we create MCSCG for both $MarkovChain$-$top20$ and $MarkovChain$-$full$. Dimensions of vertex labels in the resulting graphs are 23^2 and 213^2 respectively. Fig. 4(c) shows the resulting MCSCG converted from the given system call trace.

IV. Graph Similarity Computation

After converting the system call traces into graphs, we use graph kernels to calculate the similarities between each pair of graphs. One existing graph kernel is applied for graphs with feature vector labels, e.g. HSCG, NSCG, and MCSCG. It is named Shortest Path Graph Kernel (SPGK) [4]. Since the labels in HSCG, NSCG, and MCSCG can be sparse, we adapt the Compressed Row Storage (CRS) format to compress these graphs and parallelize the computation by utilizing a multi-core CPU with OpenMP.

A. Shortest Path Graph Kernel Algorithm

For 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:

$$K_{SPGK}(S_1, S_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2) \tag{1}$$

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:

$$k_{walk}(e_1, e_2) = k_{node}(u_1, u_2) \cdot k_{edge}(e_1, e_2) \cdot k_{node}(v_1, v_2) \tag{2}$$

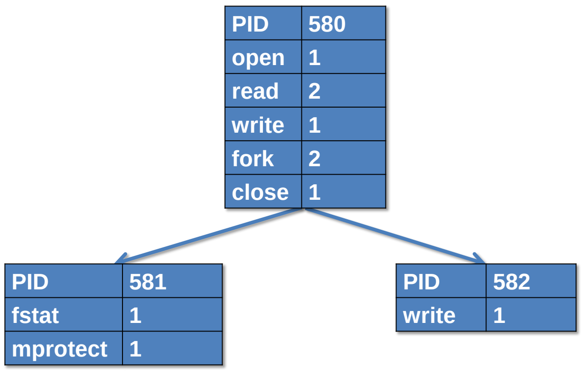
where k_{node} and k_{edge} are kernel functions for comparing vertices and edges respectively. The same notations are also applied in the following sections.

In our experiments, we pick the Brownian Bridge kernel (Eq. 3) as used in Borgwardt et al. [4] with a c value of 2 for k_{edge}. For k_{node}, given that the dimension of the feature vectors is n, we evaluated three popular kernels including the Gaussian kernel, the Intersect kernel and the Linear kernel.

$$k_{brownian}(e_1, e_2) = max(0, c - |e_1 - e_2|) \tag{3}$$

B. Computation Speedup

Two techniques are applied in our experiments to speedup the graph similarity computation. First, we observe the labels in HSCG, NSCG, and MCSCG are usually sparse, therefore we compress these graphs using the Compressed Row Storage

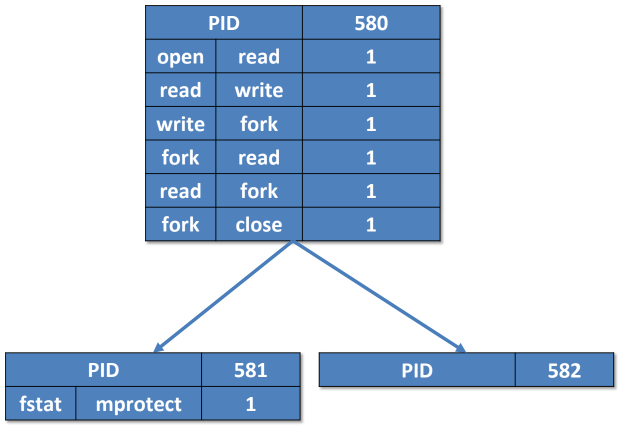


PID	580
open	1
read	2
write	1
fork	2
close	1

PID	581
fstat	1
mprotect	1

PID	582
write	1

(a) Histogram System Call Graph

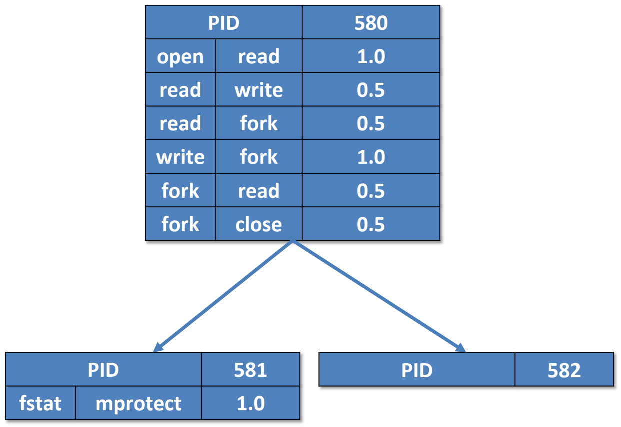


PID		580
open	read	1
read	write	1
write	fork	1
fork	read	1
read	fork	1
fork	close	1

PID		581
fstat	mprotect	1

PID	582

(b) 2-gram System Call Graph



PID		580
open	read	1.0
read	write	0.5
read	fork	0.5
write	fork	1.0
fork	read	0.5
fork	close	0.5

PID		581
fstat	mprotect	1.0

PID	582

(c) Markov Chain System Call Graph

Fig. 4: Process-based Graph Representations Converted from Fig. 1

(CRS) format. By adapting the CRS format, we are able to significantly reduce the average dimension of vertex labels by eliminating all zero elements. Consequently, the computation time of SPGK is greatly decreased. To further speedup the computation, we adapt the method proposed by Xu, et al., [30] and parallelize SPGK by utilizing a multi-core CPU with OpenMP. Since we have multiple input graphs and the computation between different pairs of graphs is independent, we can parallelize SPGK naturally by assigning each CPU core a different pair of graphs at a time.

V. Classification

To automatically classify 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 the 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).

A. 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 [9]. Figure 5 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

$$f(x) = \langle w, x \rangle + b \tag{4}$$

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. 4 with w and b from the prediction model. To use a kernel matrix as input, the decision function can be transformed to Eq. 5. 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 [24]. Once we fill the kernel values with the kernel matrix, we can classify the testing applications.

$$f(R) = (w^* + \sum_{i=0}^{N} \alpha_i y_i K(R_i, R)) \tag{5}$$

For HSCG, NSCG, and MCSCG, we use *SPGK-Gaussian*, *SPGK-Intersect*, and *SPGK-Linear* to compute the kernel matrix. To make it consistent with the graph-based representations, we also use *Gaussian*, *Intersect*, and *Linear* kernels to construct the kernel matrices for all the feature-vector-based representations. These kernel matrices are then fed into SVM for classification using Eq. 5. In our experiments, we observe that the *Intersect* kernel achieves the best classification accuracy, therefore, we only report the results from the *Intersect* kernel.

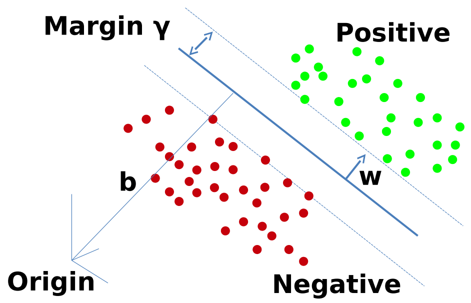


Fig. 5: This figure shows an illustration of the SVM method. w is the normal vector and b is the perpendicular distance to the origin.

B. 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. 6 shows the resulting kernel method from MKL.

$$f(R) = (w^* + \sum_{i=0}^{N} \alpha_i \sum_{j=0}^{M} \beta_j y_i K_j(R_i, R)) \tag{6}$$

In our experiments, we use the state-of-the-art Generalized MKL with Spectral Projected Gradient decent optimization (SPG-GMKL) [15] to perform MKL.

VI. Experimental Results

In our experiments, we train our SVM on a classification problem with two classes, malicious or benign. For each representation, we construct a kernel matrix. These kernel matrices are then fed into an SVM algorithm using ten-fold cross validation. We also evaluate 15 different values for the regularization parameter C in the 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.

The experiments are performed on a workstation with 128 GB memory, an AMD Opteron 6386 CPU with 32 Piledriver cores clocked at 3.2 GHz, an AMD Radeon HD 7970 GPU with 32 compute units and 3 GB global memory, and a 2 TB hard drive.

A. Dataset

We collected 5888 applications from Google Play and VirusShare[2]. To classify the binaries as malicious and benign applications, we submit our samples to the VirusTotal[3] 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 the commercial AV scanner, AVG[4].

We recorded the runtime execution behaviors of the Android applications and then converted them to different representations using our strace conversion scripts. Table I shows statistics, including number of vertices, edges, and shortest paths for the graphs generated from our malicious and benign samples. Since HSCG, NSCG, and MCSCG graphs have exactly the same graph structures, we only show numbers for HSCG in the table. 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 this is the case because 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 $FalsePositiveRate(FPR)$ 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:

$$FPR = \frac{FP}{FP + TN} \tag{7}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{8}$$

[2]http://virusshare.com
[3]https://www.virustotal.com/
[4]http://free.avg.com/us-en/homepage

TABLE I: Detailed Statistics of Vertices, Edges, Heights, and Shortest Paths for different graph representations of Malicious (M) and Benign (B) applications. HSCG, NSCG, and MCSCG graphs have the same statistics

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

C. Different Kernels

For the SPGK algorithm, we need to pick a valid kernel function k_{node} for comparing vertices and another valid kernel function k_{edge} for comparing edges. In our experiments, we pick a Brownian Bridge kernel for k_{edge} as used in Borgwardt et al. [4] with a c value of 2 for k_{edge}. For k_{node}, we evaluate Gaussian kernels with different σ values from 2^{-6} to 2^9 with a step value of 2^3. We also evaluate k_{node} using the Intersect kernel and the Linear kernel. However, we report only the best classification results although they may be achieved by different kernels in different graph sets.

Similarly, we apply different Gaussian kernels, an Intersect kernel, and a Linear Kernel on feature-vector-based representations. Only the best classification accuracies are reported.

D. Results from Interaction Steps

To understand the importance of interaction events, we first emulate each application for 20 seconds without any interference and then apply various interaction events. All system call invocations are recorded in one strace log file. We generate HSCG graphs using only the first 20 seconds of the strace log files and refer to these as $HSCG$-$nointeraction$. We also generate HSCG graphs using the whole strace log files and refer to these as $HSCG$-$interaction$. By applying graph kernels on these two graph sets, the SVM results show $HSCG$-$nointeraction$ can reach 80.2% classification accuracy while $HSCG$-$interaction$ reaches 85.3%. This 5.1% improvement reveals that by applying different interaction events, we are able to expose more malicious behaviors.

E. Results from Incomplete Strace

In our experiments, we run strace on $zygote$ so we can record all system call invocations of the testing application from the moment it is launched. This is an important step because malware tends to carry out malicious tasks upon initial execution. If we only record the execution behavior after the application has been launched, for example, like the method proposed by Wei et al. [28], important malicious behavior may not be recorded. To understand the importance of complete strace log, we ignore all system call invocations that happen during the first second of the strace log files and generate HSCG graphs named $HSCG$-$incomplete$. We then compare its performance with $HSCG$-$interaction$. Experiments show $HSCG$-$incomplete$ reaches 84.5% classification accuracy which is 0.8% less than $HSCG$-$interaction$. Therefore, recording system call invocations during initial execution can help reveal more malicious behaviors. The experiments in the rest of the paper are all based on full strace log files.

F. Results from Top K System Call List

As mentioned in Section III-B, we extract the top K system calls to reduce the computation time. We generate HSCG graphs from strace log files using the top 5, top 10, and top 20 most frequent system calls. Then, we compare the classification results using these graphs to using the HSCG graphs generated using the full system call list.

Table II shows the accuracy achieved by using different system call lists and the corresponding graph kernel computation time on our experimental machine. It shows that using the top 20 system calls cannot reach the same accuracy as using the full system call list. Nevertheless, using the top 20 system calls is able to reach an accuracy level of about 1% better than using top 15 system calls, 2% better than using top 10 system calls, and 3.3% better than using top 5 system calls. In terms of computational cost, using only the most frequent 20 system calls is 1.8x faster than using the full list. Therefore, we use only the top 20 system call list for most of our experiments, unless otherwise noted. For some representations that are not computationally expensive, we experiment with both the full system call list and the top 20 system call list.

TABLE II: Best Classification Accuracy and Graph Kernel Computation Time for HSCG Graphs Generated using Different System Call Lists

graph	Accuracy	Time (sec)
HSCG-full	85.3%	68
HSCG-top20	83.3%	38
HSCG-top15	82.3%	30
HSCG-top10	81.3%	23
HSCG-top5	80.0%	17

G. Results from Feature Vector Representation

Here, we evaluate previously studied feature-vector-based representations including histogram, n-gram, and the Markov Chain. We build system call histograms with top 20 system calls and the full system call list. We name them $histogram$-$top20$ and $histogram$-$full$, respectively. For n-grams, we use the top 20 system call list and evaluate different N values, where N is 2, 3, and 4. The resulting vector sets are named 2-$gram$-$histogram$, 3-$gram$-$histogram$, and 4-$gram$-$histogram$. To build our Markov Chains, we use the top 20 system calls and the full list. The resulting feature vectors are named $MarkovChain$-$top20$ and $MarkovChain$-$full$, respectively. These feature vectors are fed into different kernels for constructing the kernel matrices. Then, the matrices are fed into an SVM for five runs of ten-fold cross validation. We report the best classification accuracy that each representation can achieve and the corresponding False Positive Rate (FPR) in Table III. The results show that an n-gram representation performs better for larger values of n. This is reasonable because larger values of n means more system call combinations are taken into consideration. Please note that $histogram\text{-}top20$ is essentially $1\text{-}gram\text{-}histogram$ in our experiment. We also observe that using a full system call list can achieve better classification accuracy for histogram and Markov Chain compared to using top 20 system calls for these representations. However, the FPR rates are also increased using the full system call list.

TABLE III: Best Classification Accuracy Achieved by Different Feature-vector-based Representations

Vector set	Accuracy	FPR
histogram-top20	74.5%	8.2%
histogram-full	80.5%	9.1%
MarkovChain-top20	81.3%	8.4%
MarkovChain-full	82.6%	9.2%
2-gram-histogram	80.9%	8.1%
3-gram-histogram	82.8%	8.0%
4-gram-histogram	83.3%	8.0%

H. Results from Graph Representations

For each feature vector representation, we generate its corresponding graph representation. HSCG is the graph representation for the system call histogram. In particular, we generate HSCG using the top 20 system call list and the full list. We call these graph representations $HSCG\text{-}top20$ and $HSCG\text{-}full$, respectively. The N-gram System Call Graph (NSCG) is the graph representation for n-grams. We generate NSCG graphs for $2\text{-}gram\text{-}histograms$, $3\text{-}gram\text{-}histograms$, and $4\text{-}gram\text{-}histograms$ using the top 20 system call list. We call these graph representations $NSCG\text{-}2$, $NSCG\text{-}3$, and $NSCG\text{-}4$, respectively. For Markov Chains, we build a Markov Chain System Call Graph (MCSCG) and experiment with $MCSCG\text{-}top20$ and $MCSCG\text{-}full$ using the top 20 and the full system call list, respectively. The kernel matrices for these graph representations are fed to an SVM algorithm for five runs of ten-fold cross validation. We also evaluated 15 different values for the regularization parameter C in our SVM algorithm. Figure 6 shows the classification accuracy for these different graph representations for different values of C. From the figure, we observe that $HSCG\text{-}top20$ performs the worst and $HSCG\text{-}full$ is slightly better. $NSCG\text{-}2$ is not as effective as $MCSCG\text{-}top20$ because MCSCG has encoded transitional probability information. $MCSCG\text{-}full$ outperforms $MCSCG\text{-}top20$ due to the utilization of a full system call list. Although, $MCSCG\text{-}full$ is inferior to $NSCG\text{-}3$. Overall, $NSCG\text{-}4$ reaches the best accuracy at 87.3%.

We directly compare the classification accuracy between feature-vector-based representations and their corresponding graph-based representations in Table IV. On average, a graph-based representation is able to reach **5.2%** classification accuracy improvement over the corresponding feature-vector-based representation. Thus, we can conclude graph-based representation performs better than flat feature vectors using the same strace information. This shows that the structure of the calls represented in the graph-based techniques adds predictive power to the model.

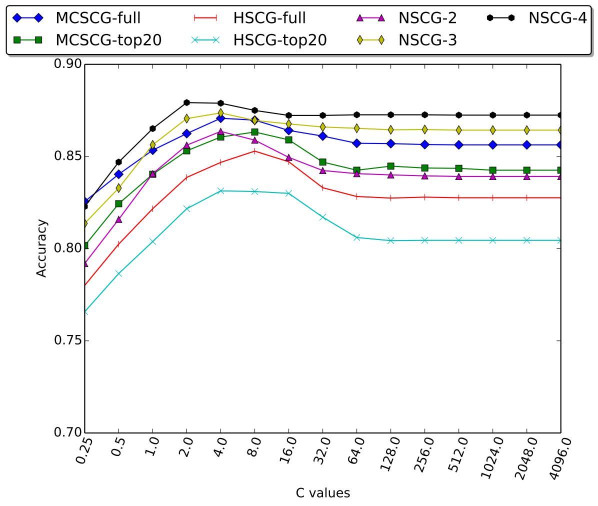


Fig. 6: Classification Accuracy with Different C values for HSCG, NSCG, and MCSCG

TABLE IV: Classification Accuracy Comparison between Feature-vector-based Representation and its corresponding Graph-based Representation

	Vector	Graph	Improvement
histogram-top20	74.5%	83.3%	8.8%
histogram-full	80.5%	85.3%	4.8%
MarkovChain-top20	81.3%	85.9%	4.6%
MarkovChain-full	82.6%	87.2%	4.6%
2-gram-histogram	80.9%	85.9%	5.0%
3-gram-histogram	82.8%	87.1%	4.3%
4-gram-histogram	83.3%	87.3%	4.0%
Average			5.2%

I. Results from Multiple Kernel Learning

To demonstrate different representations can be used together and further improve classification accuracy, we apply the MKL method to combine different similarity kernel matrices and feed the resulting matrix into an SVM. In particular, we evaluate the linear combinations of kernel matrices obtained from $MarkovChain\text{-}full$, $MCSCG\text{-}full$, $histogram\text{-}full$, $2\text{-}gram\text{-}histogram$, $3\text{-}gram\text{-}histogram$, $4\text{-}gram\text{-}histogram$, $HSCG\text{-}full$, $NSCG\text{-}2$, $NSCG\text{-}3$, and $NSCG\text{-}4$ using SPG-GMKL. The weights obtained from SPG-GMKL for each kernel matrix are listed in Table V. By linearly combining these kernel matrices, we are able to reach a classification accuracy of **87.7%** with a FPR of **3.8%**. Therefore, feature-vector-based representation and graph-based representation can be combined together to achieve better accuracy.

J. Graph Kernel Running Time

For n input graphs, each graph kernel returns a kernel matrix of size $n \times n$. The kernel matrix is symmetric, therefore we only compute its diagonal and the top half corresponding to $(n^2 + n)/2$ entries. In our dataset, we have 5888 samples in total. Therefore, we generate 5888 system call traces, one trace for each application execution and then convert these

TABLE V: Multiple Kernel Learning Weights and Classification Result. MC stands for MarkovChain, hist. stands for histogram.

	MC	MCSCG	hist.	2-hist.	3-hist.	4-hist.	HSCG	NSCG-2	NSCG-3	NSCG-4	Accuracy	FPR
Weight	0.34	0.60	0.12	0.14	0.22	0.30	0.25	0.32	0.47	0.59	87.7%	3.8%

into 5888 graphs for each graph representation and feed the graphs into the corresponding graph kernels. Consequently, each graph kernel needs to compute the similarity between $(5888^2 + 5888)/2$ pairs of graphs given n is 5888.

To speedup the computation, we compress the graphs and parallelize the graph kernels using OpenMP. In our experiments, we use one CPU thread for the initialization which consists of reading input graphs into CRS format and converting each graph to its shortest path graph, and then we create 32 threads for graph kernel computation.

Table VI shows the kernel matrix computation time for HSCG, NSCG, MCSCG, and 4-gram-histogram. $SPGK$-$Intersect$ is applied on HSCG, NSCG, and MCSCG while $Intersect$ is applied on the 4-gram-histogram. The second row in the table shows the dimensions of vertex labels for different graph representations and feature vector length for 4-gram-histogram. The third row shows the average vertex label dimensions for CRS format. The last row shows the computation time spent on kernel matrix construction for $SPGK$-$Intersect$ and $Intersect$. By adapting the CRS format, we can dramatically reduce the dimensions of vertex labels as shown in the table. It also shows that the computation time of $SPGK$-$Intersect$ algorithm increases as the vertex label dimension increases. Overall, the overhead constructing a kernel matrix for the whole training set is reasonable. On top of that, once the training is done, the time spent on computing graph similarities versus the training graphs for one testing graph should be much faster.

VII. LIMITATIONS

In this paper, we focus on analyzing different representations of system call invocations. However, there are a few limitations of the dynamic analysis technique.

First, there are some advanced malware that employ anti-analysis techniques to evade dynamic analysis in emulated Android environments [19]. Once an emulator is detected, the malicious code does not get triggered.

Second, we observe that some of Android applications cannot be straced. After analyzing the failed samples, we discovered that some of these applications could not be installed due to one of several reasons, such as one or more missing shared libraries, no certificates present, a malformed manifest, a failed dexopt, or an invalid APK.

Third, the best classification accuracy we were able achieve on our dataset using dynamic analysis is 87.7%. However, the state-of-the-art static method is able to achieve a better performance on a larger dataset. For example, DREBIN [3] reaches a detection rate of 94% on their dataset consisting of 123,453 benign applications and 5,560 malware samples. The emulation overhead of dynamic analysis prevents us from running experiments on tens of thousands of applications in a short time. Nevertheless, static analysis techniques can be defeated by malware packing and other malware obfuscation techniques [20]. Ideally, dynamic analysis and static analysis should be complementary to each other. By using them together, we can ensure improved malware analysis and detection.

In conclusion, dynamic analysis has its drawbacks. However, the purpose of this paper is to evaluate different representations and improve the classification accuracy for dynamic analysis. We feed different interaction events into dynamic analysis and evaluate the performance of traditional feature-vector-based representations converted from system call invocations. We then propose different novel graph-based representations and combine them with traditional feature-vector-based representations to further improve the classification accuracy of dynamic analysis.

VIII. RELATED WORK

Android malware can be analyzed using two different and complementary methods: static and dynamic analysis. Static methods mainly focus on extracting features from the manifest and dex files of the application package. DREBIN [3] and DroidSIFT [32] are good examples of current static analysis methods. Dynamic methods concentrate on scrutinizing behavior of malware during its execution in an emulation environment. CrowDriod [5] and CopperDroid [22] are examples of dynamic analysis. A few hybrid methods combining the advantages of static and dynamic analysis have also been proposed. Andrubis [17], [29], [16] and Mobile-Sandbox [25] are good examples of hybrid analysis method.

To the best of our knowledge, there is no existing work that systematically analyzing different representations of system call invocations for Android malware. There is also no prior work comparing the feature-vector-based representation with the graph-based representation for system calls.

Wei et al., recorded system call invocations for 96 benign applications and 92 malware samples by manually installing and executing each application on an Android phone [28]. They converted the system call invocations into 1-gram, 2-gram, 3-gram, and 4-gram feature vectors. However, their method was not automated and thus cannot be applied to a larger number of Android applications. They only recorded the system call invocations after the application had been started. Information about how the application was launched and executed was ignored by the authors, and as a result their strace log files are incomplete. In our method, we automatically analyze each application. Since we strace the $zygote$ process, instead of the testing application, we are able to record the complete set of system call invocations. Dimjasevic et al., also recorded system call invocations for Android malware, and then converted them into two representations [10]. One was histograms and the other one was a variant of our Markov

TABLE VI: Timing statistics For HSCG, NSCG, MCSCG, and 4-gram-histogram

	HSCG-top20	HSCG-full	MCSCG-top20	MCSCG-full	NSCG-2	NSCG-3	NSCG-4	4-gram-histogram
Label dimension	23	213	23^2	213^2	23^2	23^3	23^4	23^4
CRS dimension	8	17	26	40	27	48	69	2114
Compute (sec)	38	68	177	225	172	276	369	12

Chain representation. Canzanese et al. [8], recorded system call traces for Windows binaries and convert them into n-gram vectors. Classification algorithms including logistic regression, naive Bayes, random forests, nearest neighbors, and nearest centroid were tested. The methods proposed in Zhang et al., [28], Dimjasevic et al., [10], and Canzanese et al., [8] are reimplemented as the baseline in this paper.

Canali et al., performed a thorough evaluation of accuracy of system-call-based Windows malware detection [6]. They built different signatures of system calls based on n-grams, n-bags, and n-tuples. Then, a signature matching is performed to detect malware. Our method is different because our method is not signature-based. Anderson et al., compared Markov Chains with n-gram representations based on instruction traces collected from Windows executables [2]. Wagner et al., proposed a graph model based on Linux system call traces which is very similar to our HSCG [27]. However, their random walk graph kernel is expensive and does not scale. In our work, we adapt previous representations and propose novel representations for comparison. We also compress the graphs and parallelize the graph kernel to achieve reasonable computation time.

IX. Conclusion and Future Work

In this paper, we evaluate the classification performance of traditional feature-vector-based representations and novel graph-based representations for system call invocations. We first implement the traditional histogram, n-gram, and Markov chain representations for system call usage in Android malware analysis. To improve the classification accuracy of the traditional feature-vector-based representations, we propose three graph-based representations where each process is treated as a vertex and labeled with a feature vector. Graph kernels are then applied on the graph-based representations to compute graph similarities that are subsequently classified with SVM algorithm. To speed up the graph kernel computation, we compress the graphs and parallelize the computation by utilizing a multi-core CPU.

To evaluate these representations, we collected a dataset consisting of 4002 benign and 1886 malicious Android applications. We first show by feeding interaction events into dynamic analysis, the classification accuracy can be greatly improved. Subsequent experiments on this dataset showed graph-based representations are capable of improving the classification accuracies of the corresponding feature-vector-based representations by 5.2% on average. We also show that different representations can be combined together to further improve the performance by 0.4%.

Future work includes but not limited to: comparing n-bag and n-tuple representations with their corresponding graph ones, collecting more applications for experiments, and extracting the processes specifically involved in the malicious behavior from the graph representation to reduce the graph size.

X. Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM-0003599

References

[1] Strace linux man page. http://linux.die.net/man/1/strace. Accessed on Jan-21-2016.

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

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

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

[5] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. 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)* (2011), pp. 15–26.

[6] Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., and Kirda, E. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)* (New York, NY, USA, 2012), pp. 122–132.

[7] Canfora, G., Mercaldo, F., and Visaggio, C. A. A classifier of malicious android applications. In *Proceedings of the 8th International Conference on Availability, Reliability and Security (ARES)* (Sept 2013), pp. 607–614.

[8] Canzanese, R., Mancoridis, S., and Kam, M. Run-time classification of malicious processes using system call analysis. In *Proceedings of the 10th International conference on Malicious and Unwanted Software (MALCON)* (Oct. 2015).

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

[10] Dimjašević, M., Atzeni, S., Ugrina, I., and Rakamaric, Z. Android malware detection based on system calls. Tech. rep., University of Utah, 2015.

[11] EGELE, M., SCHOLTE, T., KIRDA, E., AND KRUEGEL, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys 44*, 2 (Mar. 2008), 6:1–6:42.

[12] F-SECURE. Mobile thread report q1 2014.

[13] HAM, Y. J., AND LEE, H. Detection of malicious android mobile applications based on aggregated system call events. *International Journal of Computer and Communication Engineering 3* (2014).

[14] HAM, Y. J., MOON, D., LEE, H., LIM, J. D., AND KIM, J. N. Android mobile application system call event pattern analysis for determination of malicious attack. *International Journal of Security and Its Applications (SERSC) 8*, 1 (2014), 213–246.

[15] JAIN, A., VISHWANATHAN, S., AND VARMA, M. 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)*.

[16] LINDORFER, M., NEUGSCHWANDTNER, M., AND PLATZER, C. 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).

[17] LINDORFER, M., NEUGSCHWANDTNER, M., WEICHSELBAUM, L., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. 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).

[18] MAWSTON, N. Android captured record 85 percent share of global smartphone shipments in q2 2014. Smartphone report, Strategy Analystics, 2014.

[19] PETSAS, T., VOYATZIS, G., ATHANASOPOULOS, E., POLYCHRONAKIS, M., AND IOANNIDIS, S. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security* (New York, NY, USA, 2014), EuroSec '14, pp. 5:1–5:6.

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

[21] REDDY, D. K. S., DASH, S. K., AND PUJARI, A. K. New malicious code detection using variable length n-grams. In *Information Systems Security*, vol. 4332 of *Lecture Notes in Computer Science*. 2006, pp. 276–288.

[22] REINA, A., FATTORI, A., AND CAVALLARO, L. 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).

[23] RIECK, K., TRINIUS, P., WILLEMS, C., AND HOLZ, T. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security 19*, 4 (Dec 2011), 639–668.

[24] SCHOLKOPF, B., AND SMOLA, A. J. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001.

[25] SPREITZENBARTH, M., SCHRECK, T., ECHTLER, F., ARP, D., AND HOFFMANN, J. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security* (2014), 1–13.

[26] TCHAKOUNTÉ, F., AND DAYANG, P. System calls analysis of malwares on android. *International Journal of Science and Technology 2*, 9 (2013).

[27] WAGNER, C., WAGENER, G., STATE, R., AND ENGEL, T. Malware analysis with graph kernels and support vector machines. In *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE)* (Oct 2009), pp. 63–68.

[28] WEI, Y., ZHANG, H., GE, L., AND HARDY, R. On behavior-based detection of malware on android platform. In *Proceedings of the IEEE Global Communications Conference (GLOBECOM)* (Dec 2013), pp. 814–819.

[29] WEICHSELBAUM, L., NEUGSCHWANDTNER, M., LINDORFER, M., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Techical Report, TRISECLAB-0414-001* (2014).

[30] XU, L., WANG, W., ALVAREZ, M., CAVAZOS, J., AND ZHANG, D. 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).

[31] YAN, L. K., AND YIN, H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium* (2012).

[32] ZHANG, M., DUAN, Y., YIN, H., AND ZHAO, Z. 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).