Malware Detection in Android System

Sapna Malik
Department of Computer Science & Engineering
MSIT

Abstract*-With the advent of the digital age smartphones have become a necessity for our life. Smartphones are not only used to communicate with each other but also used for banking, payment etc. They also contain a lot of sensitive data such as our contacts, messages, tracking information, etc. Thus they have become a highly vulnerable area for attacks. These threats can disrupt the working of the phone, modify or retransmit the data without the knowledge of the user. With the android system, which is the world's most popular Smartphone operating system and covers of about 80% of the global market share.*
In this paper presented a machine learning-based method for the detection of malicious Android applications and successfully classified the android malware fakeInstaller with an accuracy of 69% using SVM and a linear- time graph kernel Neighborhood Hash Graph Kernel(NHGK

Keyword: Malware Detection; Graph Kernels; Machine Learning;Structural Analysis of android application

I. INTRODUCTION

Smartphones have taken a crucial role and have replaced a variety of banking and other sensitive services, they have also become an active target for various attacks. According to a research firm IDC, antivirus program have been installed in only 5% of the smartphones and tablets. Despite the rapid growth of the Android platform, little focus has been given on the security part of the system. Android malware, such as DroidDream, has been discovered in over 50 applications on the official Android market in March 2011[1].The android security is not strong enough to protect our data and even safe applications can unintentionally expose our information [2].A study has shown that around 211 applications out of 204,040 applications on the official Android market and alternative marketplaces are found to be malicious[3].

The demand for the tools required to deal with such threats is increasing and different approaches like taint analyses and signatures cannot automatically distinguish benign apps from malware and they can be easily circumvented using simple program obfuscations techniques[4].

The technique presents in this research work based on structural analysis of malwares. Each type of malware has similar structure and we try to identify the structure of the malware in the function call graph of an android application. This method, however, is not efficient, solely since there does not exist any known solution to test whether two graphs are isomorphic in polynomial time. To solve this problem, in this research work NHGK (Neighborhood Hash Graph Kernel) has been used to reduce the time complexity in linear time[5] and done hashing on control flow graph to measure code similarity.

The rest of this paper is structured as follows: In Section II we will give an overview of our approach. In section III we will give a detailed description of creating control flow graph of an android application. In Section IV we will explain labeling and hashing of control flow graph and In Section V we will explain structural analysis using SVM. In Section VI methodology and Section VII list limitation of our approach. We will conclude the research work in Section VIII.

II. OVERVIEW

In this paper, the following steps has been taken for malware detection in an apk:

Reverse engineering of the android applications to get java bytecode from android dex code

In this research work ,DEX2JAR API is used to convert DEX code to Bytecode and used JADX API to convert .apk files to JAVA Files.

Creation of function call graph with labels from the byte code.

After getting the byte code we created a function call graph. A function call graph is a directed graph [5] where a node exists for each of the application's functions and edges from the callers to callers.

To visualize the call graph we have used. (Dot) language to represent the graph in textual format and the used zgrviewer to visualize it. For creating the graph, we also need to have installed GraphViz.

Flowdroid is used to create a call graph directly from the .apk. After getting the flow graph we label each node (function) of the application to 15 bit vector depending upon the types of instruction used in that function.

Implementation of Support Vector Machine with NHGK kernel.

After receiving the feature vector, it is necessary to convert it into a linearly separable data. To do so, we pass it though the Neighborhood Hash Graph Kernel (NHGK) kernel. After processing in the kernel we receive a 400*900 large vector each bit corresponding to a frequency of 900 hashes possible. Now, we compare this vector with the centroid of the FakeInstaller and apply it to the SVM to find the degree of similarity to it.

III. TRAINING SVM AND CONCLUDING THE RESULT

After the implementation of the SVM it is necessary to train it according to a malware so that the weights are changed accordingly. For doing this we have various methods, in this research work we are using QP programming to find the most optimized set of weights to be used. This may be a onetime process, but it may be invoked again for a new set of data for better weights. After the training, we expect results, so for each application under scrutiny, we apply the entire process again except the training to get a value, corresponding to which we classify the application.

CREATING CONTROL FLOW GRAPH

A function call graph is a directed graph [5] where a node exists for each of the application’s functions and edges from the callers to callas. A call graph is a directed graph whose vertices, representing the functions a program is composed of, are interconnected through directed edges which symbolize function calls .Call graphs are generated from a binary executable through static analysis of the decompiled bytecode. The bytecode of each application is generated from dex2jar. To visualize the call graph we have used. (Dot) language to represent the graph in textual format and the used zgrviewer to visualize it. The flowdroid is used to generate function call graph directly from the apk.

IV. LABELING OF FUNCTION CALL GRAPH

Furthermore, a labeled function call graph can be constructed by attaching a label to each node. Formally, this graph can be represented as a 4-tuple G = (V, E, L, l), where V is a finite set of nodes and each node v is associated with one of the application's functions. E belongs to VxV, denotes the set of directed edges, where an edge from a node v1 to a node v2 indicates a call from the function represented by v1 to the function represented by v2. Finally, L is the multiset of labels in the graph and l: V -> L is a labeling function, which assigns a label to each node by considering properties of the function it represents.

Labeling of nodes is done according to the type of the instructions contained in their respective functions. Reviewing the Dalvik specification, we define 15 distinct categories of instructions based on their functionality as shown in Table 1. Each node can thus be labeled using a 15-bit field, where each bit is associated with one of the categories:

Consequently, the set of labels L is given by a subset of all possible 15-bit sequences.

V. CLASSIFICATION

This research work uses the Support Vector Machine to classify malicious and non-malicious applications as SVM have the property of not getting over fitting into the dataset as in case of other learning algorithm. Another advantage is that with the introduction of kernel in SVM non-linear data can be curved into a linear dataset. This makes the process more flexible and robust. A brief explanation of how the SVM works is as follows

Support Vector Machine

SVM is responsible for finding a hyperplane that will differentiate between positive and negative examples. The goal is to separate the two classes by a function which is induced from available examples. The second goal is to produce a classifier that will work well on unseen examples, i.e. it generalizes well. This linear classifier is termed the optimal separating hyper plane. Intuitively, we would expect this boundary to generalize well as opposed to the other possible boundaries.

But one of the problems in this is that the dataset may not be linear enough to find a hyperplane for the data. So one needs to use a kernel function which will transform the input to a high dimensional implementation and convert the inputs into linearly separable data. This project makes use of Neighborhood Hash Graph Kernel.

Neighborhood Hash graph Kernel(NHGK)

The main idea of the NHGK is that the functions closer to one function also have an effect on the structure of the graph. So for each node a hashed value is created using the labels of its neighboring nodes and itself.
Hashing of Neighbors

After labelling nodes each function stores information about the code in it, but we also need a way to store the information about the functions that are directly connected to a function. To do this we calculate a hash over all of its neighbors in the function call graph [6]. The main advantage NHGK kernel is that it reduce the time complexity of comparing two graphs O (2 n) in linear time.

The computation of the hash for a given node v and its set of adjacent nodes Vv is defined by the operation

$$h(v) = r(\ell(v)) \oplus \left(\bigoplus_{z \in V_v} \ell(z) \right)$$

Where $\oplus$represents a bit wise XOR of labels and r denotes a single-bit rotation to the left. We calculate the hash for each node individually and replace the label with the hash value.

Similarity with malicious application

After hashing the kernel needs to return degree of similarity of the application with a malicious application[7].
We have analyzed the malicious data to find out a centroid of the hash vector of the malicious applications. Now to compare this with our graph we use the function

$K (G_h, G_{h'}) =|L_h \cap L_{h'}|$

Where G_h corresponds to the graph corresponding to the centroid of malicious applications, $G_{h'}$ corresponds to the graph corresponding to the application to classify and L_h and $L_{h'}$ corresponds to their hash vector, $\{a_1, a_2, \ldots, a_N\}$, where $a_i \mathcal{E} \mathbb{N}$ indicates the occurrences of the i-th hash in Gh, respectively.

To find the intersection of the two vectors we have made use of the formula:

$h_i = \min (a_i, a_i') \; \forall \; a_i \; \mathcal{E} \; L_h$ and $\forall \; a_i' \; \mathcal{E} \; L_{h'}$.

Barla et al. [3] show that this histogram intersection can be indeed adopted in kernel-based methods and propose a feature mapping, such that S is an inner product in the induced vector space. For this purpose, each histogram H is mapped to a P-dimensional vector$\varphi(H)$ as follows

$$\phi(H) = \left(\underbrace{\overbrace{1, \cdots, 1}^{a_1}, \overbrace{0, \cdots, 0}^{M-a_1}}_{\text{bin } 1}, \cdots, \underbrace{\overbrace{1, \cdots, 1}^{a_N}, \overbrace{0, \cdots, 0}^{M-a_N}}_{\text{bin } N} \right)$$

where M is the maximum value of all bins in the dataset, N is the number of bins in each histogram This is the feature vector returned by kernel to be used by the SVM.

VI. METHODOLOGY

This research work is an open source project with code available on github (https://github.com/nitish2794/MALDIAS). We have used version control (git) to track the changes in the project and to revert them if needed.

Structure of project and the files required for processing

1).Processing of android application

Input Files required for the processing of android application

Android Platforms SDK:- Each android application uses android's library to implement these features, so to separate these methods from used defined ones we need android sdk to find the default android methods.

Source and Sinks.txt:- This file is used by flowdroid to find all possible entry and exit points in an application which helps in accurately implementation of call graph.

SVM training module: - We need to train SVM before we can use it to classify, to train the SVM we need to preprocess the entire dataset available and then train the SVM according to the dataset.

2). Creating a flow graph of the android application

We have used to flowdroid to create a control flow graph of the android application.

3 a).Implementation of NHGK kernel

We create a label to each function of the application, depending upon the types of instruction used in the function. After that we do the hashing of the labels of a node with the labels of its neighbors.

3 b).Sending the feature vector SVM to classify.

After receiving the feature vector, it is necessary to convert it into a linearly separable data. To do so, we pass it though the NHGK kernel. After processing in the kernel we receive a 400*900 large vector each bit corresponding to a frequency of 900 hashes possible. Now, we compare this vector with the centroid of the FakeInstaller and apply it to the SVM to find the degree of similarity to tithe various files used in this are;-

Default.txt:-This contains a feature vector corresponding to the centroid of the malware fakeInstaller.

Weights.txt:- This contains the weights corresponding to the 400*900 size vector. This weights are already trained one and as the project continues these weights may be progressed as more and more data is found.

4). Display the result

The SVM returns a value to the system corresponding to the feature vector of the application and according to this value we classify it into malicious or non-malicious or safe application.

DATA SET

Our dataset consisted of 91 applications out of which 28 were benign rest were malicious and belonged to the group of FakeInstaller.

VII. LIMITATION

In this static analysis technique call graph processed are the approximation of the real functions and their neighbor. So there is a limit to what extent you can store relevant information in a neighborhood hash. A call graph can be obfuscated by adding unreachable calls. Moreover, function inlining can be used to hide the graph structure. Invalid bytecode sequences can be deliberately added by the attacker to prevent the successful decompilation of android application.

VIII. CONCLUSION AND FUTURE WORK

In this paper we have presented a machine learning-based method for the detection of malicious Android applications. Our method is inspired by the neighbourhood hash graph kernel to represent applications based on their function call graphs. This representation is shown to be both, efficient and effective, for training an SVM that ultimately enables us to automatically identify Android malware with a detection rate of 69% with 1% false positives, corresponding to one false alarm in 100 installed applications on a Smartphone.

Further, this same procedure can be applied to many different types of malwares to create a collection of SVM each of which will correspond to a malware type. Then, we can use one-vs-all strategy to analyze an application.

REFERENCES

[1] Lookout Mobile Security. Security alert: Droiddream malware found in officialandroid market.http://blog.mylookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/

[2] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri "A study of android application security", In Proceedings of the 20th USENIX Security Symposium, 2011.

[3] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets", In Proceedings of the 19th Network and Distributed System Security Symposium, 2012.

[4] C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Efective and efficient malware detection at the end host", In Proc. of USENIX Security Symposium, 2009.

[5] X. Hu, T.C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs", In Proc. of the ACM conference on Computer and communications security, 2009.

[6] A. Barla, F. Odone, and A. Verri, "Histogram intersection kernel for image classification" In Proc. of International Conference on Image Processing, ICIP, volume 2, pp. 513-516, 2003.

[7] V. Rastogi, Y. Chen, and X. Jiang. "DroidChameleon: evaluating Android anti-malware against transformation attacks," In ASIACCS, pages 329–334. ACM, 2013.