Automatic Malware Detection for Android

Abstract— **With Android being the most widespread mobile computing device platform today, malicious software (malware) inevitably exists for it. While there have been various attempts, with a certain degree of success, to provide a safe environment for Android users by detecting malware in numerous ways, none of them is able to detect malware that has yet unseen or rare behavior. In this project we propose a completely automatic framework that would detect a malware Android app with such behavior. The framework uses a machine learning classification algorithm and a testing technique. It operates on the Android application repository level (e.g., Google Play) where it detects a newly uploaded malware app automatically. Our results show that, in a collection of more than 1800 Android applications, the framework is able to correctly determine the application's behavior consistently and accurately.**

I. Introduction

Android is an operating system primarily targeting mobile devices such as smartphones and tablets. The operating system has held a large share of the market for a few years now, and its market share continues to grow. Just like many other mobile device operating systems, Android's key selling point is a huge number of software applications — or apps for short — that are developed by various parties. The Internet connectivity on Android-based devices gives these devices and accompanying apps almost unlimited ways for interaction with other computer systems on the Internet.

These facts also make Android a very appealing target for malicious software (malware). A user may install a malware app on an Android device without knowing that the app is malicious. Some examples of malicious actions are sending unauthorized SMS messages, and sending sensitive private information stored in the device to a remote server without the user's knowledge and permission. In order to mitigate threats posed by a malware app, we propose a completely automatic framework to detect malicious behavior.

The framework consists of two basic components: testing and machine learning. The testing component is there to thoroughly test each app in a controlled environment in order to learn as much as possible about its behavior. The more thorough testing of the app can be carried out, the better feature vector capturing the true behavior of the app can be constructed. The machine learning component serves the purpose of extracting these feature vectors from Android apps, and then using the feature vectors to learn parameters of a classifier that discriminates benign from malicious apps.

Earlier research on malware [1] has shown that the software application behavior can be observed through system calls the application makes. Our framework monitors system calls an Android app makes during the testing phase, and constructs a system call dependency graph that represents data flow between the system calls. The graph is then converted to a feature vector of the app. Therefore, each feature vector is a member of either the benign app class or the malicious app class. Once the learning phase is finished, the classifier can tell which class the feature vector of any app — including of a yet unseen app — belongs to.

We evaluated our approach on 1824 Android apps in total, out of which 656 are benign and 1168 are malicious. Results from the evaluation show that our framework is very accurate in predicting the behavior of an app with an 86.8% accuracy on average.

II. Related work

Earlier research on malware detection focused on signature extraction from malware binary code. However, it is easy to mitigate such efforts by obfuscating the code. Android SDK [2] comes with ProGuard, a tool for Android app code obfuscation. Google encourages Android app developers to use ProGuard to obfuscate their apps.

Recent research focuses on behavior detection [3], [4], [5] by observing patterns in the way a software program executes. Such approach is justified by the fact that the behavior should not change at all or change very little after code obfuscation. We build on that idea in order to be able to distinguish benign from malicious Android apps.

One way to detect the program behavior is by tracing data flow between system calls [1], [6]. Palahan et al. [7] do the same with the addition of extracting statistically significant behavior, but their results might be biased due to the way they generate input data from benign programs. Reina et al. [8] also use system calls to determine program behavior, but only to detect differences in malware with and without stimuli applied, not to tell apart benign from malicious behavior. Unlike existing work, our approach provides a simplistic, yet effective data flow model between system calls that discriminates goodware from malware.

III. Preliminaries and Problem Definition

A. Problem Definition

Given a training set of Android apps with a label designating each app in the set as either benign or malicious, the goal is to learn how to discriminate benign from malicious apps in a new set that has no labels. Each app can be classified as benign or malicious based on its behavior, i.e. actions the app performs. In other words, the goal is to learn characteristics of both benign and malicious behavior in order to be able to detect a malicious app.

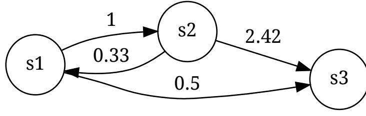

Fig. 1. A weighted directed graph with three vertices and four edges. Edge weights are written next to edges.

B. Graph Theory

In graph theory, graph $G = (V, E)$ is a representation of a set of objects V where these objects are all called vertices. Some pairs of objects from V are connected by links — also called edges — from a set E. An edge can have an associated weight with it and the weight is written next to an arch representing the edge. A graph with weighted edges is called a weighted graph.

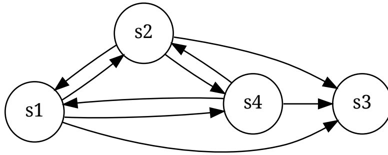

Fig. 2. A complete digraph with four vertices.

A directed graph is a graph where edges have orientation, i.e. the edges are directed. If there is a directed edge from a vertex s_1 to a vertex s_2, then we denote the edge by (s_1, s_2). An example of a weighted directed graph is given in Figure 1. The graph is given with $V = \{s_1, s_2, s_3\}$ and $E = \{(s_1, s_2), (s_1, s_3), (s_2, s_1), (s_2, s_3)\}$. A complete digraph is a graph in which every ordered pair of distinct vertices is connected by a directed edge. An example of a complete digraph is given in Figure 2.

C. System Calls

A system call is a mechanism for a software program to request a service from the operating system's kernel. Usually the program does not invoke system calls directly, but rather indirectly through a software library from the operating system. In the case of Android being the operating system, the kernel is a Linux kernel where system calls [9] form the fundamental interface between an Android app and the kernel.

The Android operating system has more than 200 system calls. We found a file in the Android source code that claims to list all the system calls that are supported by the Bionic C library. The library is a derivation of the BSD's standard C library code for Android. However, by running Android apps in an early phase of this work we observed system calls that are not on the list, i.e. the list is incomplete. Therefore, we extended the list with system calls that we had detected in the phase. The final list contains 209 system calls.

Formally, let $\mathcal{S}$ be a set of system call names containing all the system calls from the final list, $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$, where $n = 209$. We define a binary relation on system calls $s_{i_j}, s_{i_k} \in \mathcal{S}$, $i_j, i_k \in \{1 \dots n\}$ as $s_{i_j} < s_{i_k}$ if and only if $j < k$ for $j, k \in \mathbb{N}$. The relation denotes that system call s_{i_j} chronologically happens before system call s_{i_k}. Then a system call sequence q of length m is a sequence of instances of system calls $q = (s_{i_1}, s_{i_2}, \dots, s_{i_m})$, where $s_{i_1} < s_{i_2} < \dots < s_{i_m}$. For a pair of system calls s_{i_j} and s_{i_k} in sequence q, where $s_{i_j} < s_{i_k}$, we define the distance between the calls as $d(s_{i_j}, s_{i_k}) = ||s_{i_j} - s_{i_k}|| = k - j$.

We model data flows between a pair of system calls by using the distance between the calls. Let $w_{j,k}$ denote the weight of a directed edge (s_j, s_k) in a system call dependency graph. A system call dependency graph (SDG) is a complete digraph with $V = \mathcal{S}$. Then w_{i_j, i_k} for sequence q is defined as:

$$w_{i_j, i_k} = \sum_{s_{i_j} < s_{i_k}} \frac{1}{d(s_{i_j}, s_{i_k})} \tag{1}$$

where

$$s_{i_k} < s_{i_{j'}} \text{ s.t. } j' = \arg\min_a \{s_{i_a} = s_{i_j} | k < a\} \tag{2}$$

Informally, the closer the pair is in sequence q, the more it contributes to the edge weight in the graph. Instead of actually observing a data flow between system calls, we use the model of the flow. Our data flow model is based on a naïve observation that the closer a pair of system calls is in sequence q, the more likely it is that there is a data flow between the pair.

IV. Approach

In order to learn to discriminate benign from malicious Android apps, we capture the behavior of an app during its execution in the Android operating system. Our approach consists of three phases. We automatically execute an app, generate its feature vector, and use the vector in a supervised learning technique. The approach is illustrated in Figure 3.

Willems et al. [1] showed that software program behavior can be observed by tracking data flow between pairs of system calls the program makes. We use that information in modeling a data flow between system calls an Android app makes during its execution in the Android operating system. The data flow forms dependencies between system call pairs and results in a system call dependency graph. Each app execution has a corresponding system call dependency graph. The graph is an encoding of the app's behavior. We transform the graph into a feature vector that we use in a later training phase or for behavior detection of a yet unseen app.

A. App Execution

To execute an app one could use a physical device, manually install the app, start it, and manually interact with the app. Such approach would be biased, would leave side effects after every app execution, and would not scale to executing more than 1800 apps. Therefore, a different approach is needed. For that purpose, we use the Android software development kit [2], which includes a set of development

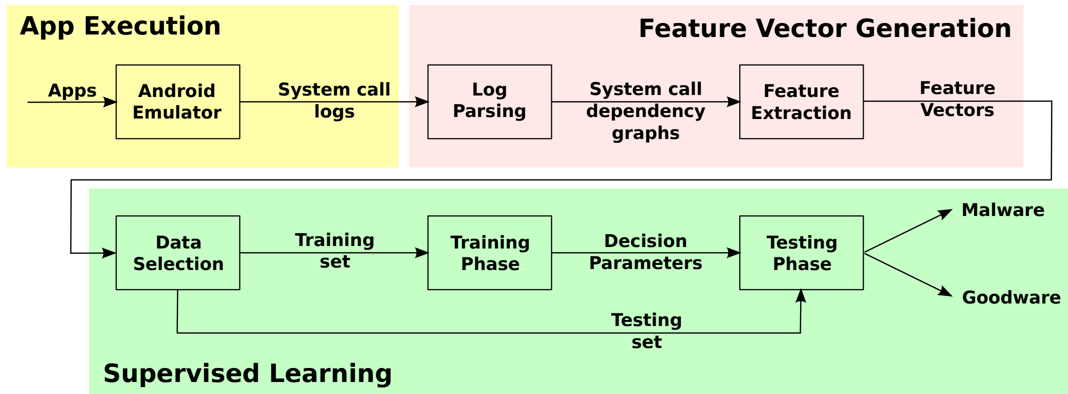

Fig. 3. There are three phases in our approach: App Execution, Feature Vector Generation, and Supervised Learning.

tools. In the kit are, among others, a handset emulator and a debugger.

The emulator provides an ability to use any of the existing Android operating system versions. It also has an ability to emulate any of the several mobile hardware platforms provided, giving an environment capable of emulating any of the capabilities of a hardware platform, thus making it realistic. This includes spoofing the sending of SMS text messages to the emulator as well as sending Global Positioning System (GPS) location updates. Another feature of the emulator is snapshot taking of the operating system and loading a previously taken snapshot. We heavily utilize the feature, as described later.

The debugger is used for a communication between a host machine and the emulator. It consists of client and server-side programs for a two-way communication. Checking the status of the emulator, installing an app, accessing a file system in a running Android system, uninstalling an app, and fetching files from the system is all done through the debugger.

There are two types of events we send to our execution environment during an app's execution. The first type of an event is internal. Internal events are sent to an app itself, such as screen clicks, touches, and gestures. The second type of an event is external. External events are sent to the emulator and include events that come from the interaction with the external environment. They can be anything from a phone call, text message, GPS location update, or a reading from an accelerometer, gyroscope, and proximity sensor.

We start by installing an Android app in a pristine sandbox environment. For that purpose we use a clean snapshot taken immediately after a creation of an android virtual device (AVD) that runs within the emulator. The snapshot is created only once throughout the whole process. It is important that each app is executed in a clean and controlled environment provided by an emulated snapshot for two reasons. The first one is so that there are no side effects from other app executions. The second reason is that we can monitor the execution.

Once an app is installed, we start the app with The Monkey. The Monkey is a part of the kit that runs in the emulator and generates pseudo-random streams of user events. The events are internal and range from clicks, touches, and gestures to system-level events. When the app is started, The Monkey sends 1000 of events with a short pause period before each event. In parallel to internal events, we send external events too. In particular, malware from our data set performs unauthorized activities related to SMS text messages and location updates [10], so we focus on these two external events. Every few seconds we send a text message or location update to the emulator. The combination of the internal and external events we use provides a realistic interaction of an Android user with her's phone or tablet and the whole external environment.

As the app's execution is driven by both internal and external events, we record a sequence of system calls the app sends to the system's kernel. Because there are various types of the events and because they are executed within different contexts of the app, resulting system calls form a rich palette. This richness enables us to distinguish each app from other apps. Even though the sequence contains more information, keeping only system call names and their chronological order is sufficient. When all of the 1000 internal events have been sent, we stop recording the sequence and uninstall the app.

B. Feature Vector Generation

After the app execution phase follows the feature vector generation phase. Our data flow model ignores all parameters a system call has and a possible return value the call outputs. From an observed sequence of system calls for every app — be it a benign or malicious app — during the execution phase we extract system call names only in the chronological order. The extracted names form a sequence q.

From sequence q we compute weight $w_{j,k}$ for every system call pair $(s_j, s_k) \in \mathcal{S}^2$ as explained earlier. Weight $w_{j,k}$ is the weight of an edge (s_j, s_k), i.e. of the directed edge from vertex s_j to vertex s_k in the system call dependency graph of an app. If there are j and k such that $w_{j,k} = 0$, we still considered edge (s_j, s_k) to exist, but with the weight of 0. That means that the graph is a complete digraph. Note that each app execution has a single corresponding system call dependency graph. Each app is executed only once, hence each app has only one system call dependency graph.

We create the feature vector $\mathbf{x}$ of an app by taking edge weights from the graph. For every directed edge (s_j, s_k) there is a corresponding feature in $\mathbf{x}$. Because the graph is a complete digraph, it means that the graph has $|\mathcal{S}|^2$ edges, hence the dimensionality of x is $|\mathcal{S}|^2 = 43681$. Therefore, $\mathbf{x}$ can be written as $\mathbf{x} = [x_1 x_2 \dots x_{|\mathcal{S}|^2}]^{\mathrm{T}}$. Each element x_l in $\mathbf{x}$ represents one feature. In particular, the value of x_l is $w_{j,k}$, where $1 \leq j, k \leq |\mathcal{S}|$, $k \equiv l \pmod{|\mathcal{S}|}$, $j \equiv l - k + 1 \pmod{|\mathcal{S}|}$. Therefore, values of j and k can be uniquely determined for a particular value of l.

C. Supervised Learning

The final phase of our approach is an application of a supervised learning technique on the feature vectors generated in the previous phase. We can apply the technique because we have a set of well-known benign apps and a set of well-known malicious apps. These sets are discussed in more detail in Section V. In other words, we avail ourselves of the information on what set each app belongs to in order to learn the characteristics of both benign and malicious behavior.

A supervised learning technique that we use is support vector machines (SVMs) [11]. There are numerous implementations of the SVMs technique. Our framework uses a readily available free software library LIBSVM [12]. The library is an essential component of the supervised learning phase code implementation that we carried out in GNU Octave, a MATLAB-like language.

In the learning phase of the technique we combine a subset of goodware and a subset of malware apps, both subsets of a varying size, into a learning set. We use the rest of our data set as a testing set. More details about subset selection are given in Section V.

V. Experimental results

Here we present an evaluation and experimental results of our approach. We obtained Android apps from two sources. Our evaluation consists of cross-validation, selection of training and testing set at random, and evaluate the performance of both experiments.

In order to be able to evaluate our approach, we collected both benign (goodware) and malicious (malware) Android apps. We obtained 656 benign apps from F-Droid, a free software-only online repository of Android apps [13]. Apps in the F-Droid repository belong to different categories, such as office, navigation, multimedia, internet, games, etc. We assume these apps to be goodware because the F-Droid community has a strict inclusion policy [13] of new apps submitted by an external developer. A developer that wants to upload a new app to the repository has to provide the accompanying source code. The community compiles every app from the source code themselves with the Android SDK.

Furthermore, all applications in the repository are free and open source software. A prerequisite for software to be free software is to make its source code available. The availability of the source code decreases incentives for someone to include malicious code. That is the case because thousands of Android developers have access to the code and are able to detect malicious aspects of it just by reading it. The F-Droid community also makes every effort to ensure that everything in the repository is safe to install and wherever possible, the source code is checked for potential security or privacy issues.

We obtained 1168 malicious apps from the Android Malware Genome Project [10]. The project focuses on the Android platform in order to systematize or characterize existing Android malware. In particular, the project authors have put more than one year of effort from August 2010 to October 2011 to collect about 1200 malware apps that cover the majority of existing Android malware families. They characterize them from various aspects, including their installation methods, activation mechanisms as well as the nature of carried malicious payloads. For example, these malicious apps leverage root-level exploits to fully compromise the Android security posing the highest level of threats to users' security and privacy, turn the compromised phones into a botnet controlled through network or short messages, have the built-in support of sending out background short messages (to premium-rate numbers) or making phone calls without user awareness.

To generate an SDG of an app, each malware and goodware binary was executed in a sandbox. The sandbox is a clean snapshot in the Android emulator. A clean snapshot of the Android operating system was loaded for the execution of each application in order to have a pristine environment and not to condition the app behavior in any way. Invoked system calls by an app were traced by the *strace* tool, which is available in the Android emulator. The binaries execution consisted of the injection of one thousand of random events that simulate a user interaction with an application as well as of sending text messages and location updates to the emulator.

As explained in detail in Section IV, by executing the applications we obtain one SDG per application that is converted into a feature vector. The vector is then fed into a machine learning algorithm. Since we know the nature of every app, in our approach we apply support vector machines, a supervised learning model with associated learning algorithms that analyze data and recognize patterns. We applied the SVMs technique with different kernels to classify the data, which resulted in different outcomes.

To classify our data set, we use a linear and multiple polynomial kernels. In particular, for the polynomial kernels we used polynomials from the first to the fourth degree.

The first experiment that we performed consists of randomly splitting the input data set into two groups. The first group is the training set and it is composed of 90% of apps. The second group is the testing set composed of the remaining 10% of apps. The SVMs algorithm learns with the training set and its associated labels and tests with the testing set. In this case we performed 50 iterations of the SVMs algorithm to compute the mean and standard deviation of the computed accuracy.

In Table I we report the results. The best value of accuracy is obtained with a polynomial kernel of the first degree.

Kernel Type	Mean (%)	Std. Dev. (%)
Linear	82.9	2.7
Polynomial 1st degree	86.8	2.1
Polynomial 2nd degree	83.9	3.3
Polynomial 3rd degree	80.6	2.9
Polynomial 4th degree	79.9	3.1

TABLE I
AVERAGE RESULTS FOR 50 ITERATIONS OF THE SVMS ALGORITHM WITH A DATA SET SPLIT INTO 90% BEING A TRAINING SET AND 10% A TESTING SET.

Kernel Type	Case 1	Case 2
Linear	80.7	79.1
Polynomial 1st degree	84.7	86.7
Polynomial 2nd degree	82.2	80.9
Polynomial 3rd degree	79.7	77.7
Polynomial 4th degree	79.6	76.9

TABLE III
RESULTS OF THE SVMS ALGORITHM APPLYING A CROSS-VALIDATION.

The difference between a polynomial kernel of the first degree and a linear kernel is in a default leading factor. The associated standard deviation is relatively small, thus showing that in general the obtained results are consistent.

		PREDICTED LABELS	
		GOODWARE	MALWARE
REAL	GOODWARE	48	18
LABELS	MALWARE	14	103

TABLE II
CONFUSION MATRIX — LABELING PERFORMED BY THE ALGORITHM FOR THE TESTING SET.

In order to better point out the performance of our approach, in Table II we show a confusion matrix, as commonly done in machine learning. Each row of the matrix represents the instances in an actual class, while each column represents the instances in a predicted class. In this way it is easy to see how the system is confusing the two classes. In our case we can see how many false positive (the goodware apps classified as malware apps, in the first row and second column) and how many false negatives we have (the malware apps classified as goodware apps, in the second row and first column). The top-left to bottom-right diagonal represents correctly classified data.

The second experiment that we performed is about selecting the training and testing set in a different way. For this experiment we divided the data set into two equally sized sets, each having 50% of malware and 50% of goodware. We have two cases. The first one consists of using the training set for learning and the testing set for testing. In the second case we swapped the roles of the sets, so that we used the testing set for training and the training set for testing. This is cross-validation, mainly used when the goal is prediction and when one wants to estimate how accurately a predictive model will perform in practice.

Indeed, in Table III we have the results of the SVMs algorithm for different kernels in both cases. Note that in this case we also got the best results for the polynomial kernel first degree and, as we can see, the values are within the standard deviation estimated in the previous experiment.

Looking at the numbers, we can say that our approach gives a very good accuracy of correctly identified malware apps. On the other hand, we can see from the confusion matrix (in the first experiment) that we have a relatively high number of false positives.

Our simple data flow model seems to give high accuracy in discrimination between benign and malicious Android apps. Existing work on Android malware detection makes no such comparison. Zhou et al. [10] learn from malware samples only obtaining a 79.6% accuracy of detected malware. Other work, e.g. Enck et al. [14], detect security and privacy issues, but their approach does not scale to a big number of apps.

VI. DISCUSSION AND CONCLUSION

In this paper, we proposed a framework for classifying goodware and malware apps for the Android system using a machine learning technique. Our framework exploits information contained in the system call dependency graph of an app execution. As discussed in Section V, we obtained good results with our approach, correctly classifying 86.8% of malware and goodware apps. However, our naïve hypothesis could be the main reason of a relatively high number of false positives that we obtained. Another weakness in our approach is random events that we send to an app. Such events likely miss some real behavior of an app, which results in a suboptimal sequence of the resulting system calls. That further propagates to a suboptimal generation of feature vectors that are used in the machine learning algorithm.

For these reasons, our future work is moving in the direction of taking a more strict hypothesis in order to refine the feature vector generation. For instance, performing focused test during the app execution would help to produce a more precise behavior of the program, which would result in a richer sequence of system calls, and eventually in a better system call dependency graph and a feature vector. Moreover, usage of formal methods to find rarely run code fragments could be another useful approach and better model the application behavior.

VII. ACKNOWLEDGMENTS

We are thankful to Zvonimir Rakamarić for his help in obtaining the malware from the Android Malware Genome Project [10].

REFERENCES

[1] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *2007 IEEE Symposium on Security and Privacy (SP)*, vol. 5, no. 2, pp. 32–39, 2007.
[2] "Android SDK." https://developer.android.com/sdk/index.html.
[3] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *Journal in Computer Virology*, vol. 7, no. 4, pp. 247–258, 2011.

[4] M. Christodorescu, S. Jha, and C. Kruegel, “Mining specifications of malicious behavior,” in *Proceedings of the 1st India Software Engineering Conference*, pp. 5–14, 2008.

[5] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, “Synthesizing near-optimal malware specifications from suspicious behaviors,” in *2010 IEEE Symposium on Security and Privacy (SP)*, pp. 45–60, 2010.

[6] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, “A view on current malware behaviors,” in *USENIX workshop on large-scale exploits and emergent threats (LEET)*, 2009.

[7] S. Palahan, D. Babić, S. Chaudhuri, and D. Kifer, “Extraction of statistically significant malware behaviors,” in *Proceedings of the Annual Computer Security Applications Conference*, 2013.

[8] A. Reina, A. Fattori, and L. Cavallaro, “A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors,” *2013 European Workshop on System Security*, 2013.

[9] “Linux system calls.” http://man7.org/linux/man-pages/man2/syscalls.2.html.

[10] Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” in *2012 IEEE Symposium on Security and Privacy (SP)*, pp. 95–109, 2012.

[11] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[12] C.-C. Chang and C.-J. Lin, “LIBSVM: a library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 27:1–27:27, 2011.

[13] “F-droid.” https://f-droid.org/.

[14] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones.,” in *OSDI*, vol. 10, pp. 255–270, 2010.