Android Malicious Application Detection Using Support Vector Machine and Active Learning

Bahman Rashidi*, Carol Fung*, Elisa Bertino†
*Department of Computer Science, Virginia Commonwealth University, Richmond, VA, USA
{rashidib, cfung}@vcu.edu
†Department of Computer Science, Purdue University, West Lafayette, IN, USA
bertino@purdue.edu

***Abstract*—The increasing popularity of Android phones and its open app market system have caused the proliferation of malicious Android apps. The increasing sophistication and diversity of the malicious Android apps render the conventional malware detection techniques ineffective, which results in a large number of malicious applications remaining undetected. This calls for more effective techniques for detection and classification of Android malware. Hence, in this paper, we present an Android malicious application detection framework based on the *Support Vector Machine* (SVM) and *Active Learning* technologies. In our approach, we extract applications' activities while in execution and map them into a feature set, we then attach timestamps to some features in the set. We show that our novel use of time-dependent behavior tracking can significantly improve the malware detection accuracy. In particular, we build an active learning model using *Expected error reduction* query strategy to integrate new informative instances of Android malware and retrain the model to be able to do adaptive online learning. We evaluate our model through a set of experiments on the DREBIN benchmark malware dataset. Our evaluation results show that the proposed approach can accurately detect malicious applications and improve updatability against new malware.**

I. Introduction

The number of the global smartphone users is growing rapidly and has reached 2.7 billion in 2016 [12]. Among them more than 85% are android smartphone users [17]. On the other hand, the number of mobile apps has been growing exponentially in the past few years. According to the number reported by Google Play Store, the number of apps in the store has reached 2.8 billion by March 2017, surpassing its competitor Apple App Store by 0.6 billion [19]. As the number of smartphone increases, privacy and security of the smartphone users have become a primary concern. Malicious apps can steal private (personal) information such as your contact list, photos, files an etc. In addition, they can also cause financial loss for the user by making secretive premium-rate phone calls and text messages [30]. This can also have an impact on the performance of the device as well.

In the most recent versions of Android OS, whenever an app needs to access a resource, it has to request the OS to get the access and the OS asks user through a pop up menu whether the user wants to grant or deny the permission request. There have been many studies on evaluating the performance of such strategy for managing resources. The studies have shown that users tend to rush through their permission requests decisions by granting all requested permissions to the apps [1], [10]. In addition to this fact, the important fact is that more than 70% of Android smartpohne apps request additional permissions beyond their actual need. An additional requested is a permission which is not necessary for the apps to function. An example of an additional requested permission can be a *tic tac toe* app requesting SMS and phone call permissions. Making decision on the maliciousness of apps without knowing sufficient knowledge about the apps is a challenging task. Therefore, an effective malicious app detection can provide additional information to help users and protect them from privacy breaches [11].

In general, malware detection techniques include *static analysis* and *dynamic analysis*. Analyzing Android apps' codes and the *Control Flow Graph* (CFG) is the key part of every static-based malware analysis. Using the CFG analysis, the model is able to find the malicious API calls and put a set of predefined restrictions on them as well as system calls [8], [20]. Since static-based malware analysis only focuses on the apps' code including API and syscalls, it is not able to detection malicious behaviors happening at the runtime [7]. In contrast, dynamic analysis captures all the runtime activities of apps and run a deep analysis on them to detect the malicious behaviors and activities during the runtime [8]. In this paper, we study Android apps' behaviors as they run on the device, and propose an active learning method using Support Vector Machines (SVM). SVM can be used to classify apps in order to distinguish between malicious and benign ones. The active learning method empowers the model to retrain itlself at a low cost.

In our approach, we not only consider apps' activities such as API cals and syscalls, but also the time that the activities occur. This can help to build a comprehensive set of features to be used as our training set. By introducing the time as a feature (timestamp of activities), we notice that this can significantly enhance the malicious app detection accuracy of the proposed model. The first step of our model is to log apps' activities using our own developed instrumentation tool called *DroidCat*. After capturing the logs a filtering and parsing mechanism is applied to syntheze and clean the captured activity logs. We train the SVM model and test it using a dataset of known malicious and benign apps. Our experimental results demonstrate that our proposed model achieves high accuracy

978-3-901882-98-2

in detecting malicious apps.

The major contributions of the work reported in this paper include: 1) An instrumentation tool that facilitates app behavior logging in order to generate high quality dataset for analysis. 2) A comprehensive time-aware Android app behavior analysis, which is based on the apps' intents and actions, as well as extra features that further improves detection accuracy. 3) A trained SVM model which can decide whether an app is malicious or not based on its behavior. 4) An Active Learning model which can retrain the SVM model to be able to detect malicious apps with behaviors different than the apps in training set.

To the best of our knowledge, this is the first time that a SVM model equipped with Active Learning is used on real data set for malicious smartphone apps detection.

The rest of the paper is organized as follows: Section II presents some background knowledge of SVM and Active Learning; Section III describes our proposed model, capturing apps' logs, training and testing the model and online hyper-parameter updating strategy; we present our evaluation results and the impact of the parameters in Section IV; related work overview is in Section V; Conclusion and future work are in Section VI.

II. Background

In this section we briefly go over some background knowledge about *Support Vector Machine (SVM)* and *Active Learning*, including how an SVM model works and can be evaluated. We also explain how active learning can help update the model by incoming new instances.

A. Support Vector Machines

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane [4]. In other words, given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which can be used to categorize new data examples. An SVM training algorithm builds a model that assigns new instances to one class or another, making it a non-probabilistic binary linear classifier. Regardless of the dimensions of the sets (finite or infinite), if the input sets are not linearly separable, SVM maps the original sets into a higher-dimensional space, presumably making the separation easier. This transformation to a high-dimensional space increases the computational load [4].

To reduce the computational load of the dot product operation which is needed in the dimension transformation and improve the accuracy of classifying data sets, SVM uses *kernel* functions. A kernel function helps accelerate the dimension transformation computation [4]. The mathematical definition of a kernel function is as follows:

$$K(x, y) = \langle (x), f(y) \rangle \tag{1}$$

where K is the kernel function, x, y are n dimensional inputs, f is a map from n-dimension to d-dimension space (d is much larger than n). In this equation, $\langle x, y \rangle$ denotes the dot product. In other words, a kernel function can also be understood as a measure of similarity between two data points. For example, a kernel function K takes two data points x_i and $y_j \in \mathbb{R}^d$, and produces a similarity score, which is a real number, i.e., $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$.

SVM has some advantages that make it unique. Some of the advantages are: (i) it has a lower computational complexity, (ii) it is effective in cases where the number of dimensions is greater than the number of samples, (iii) it uses a subset of training points in the decision function (called support vectors), so it is also memory efficient, and (iv) different kernel functions can be specified for the decision function. Next we elaborate the two key steps of SVM: training and evaluation [25].

1) Training: In order to classify datasets, the SVM model needs to be trained. As we described previously, SVM is a supervised learning model. The supervised learning is the process of inferring a function from labeled training dataset. The training dataset shall consist of a set of data points together with their labels (classes). The training dataset is then used by SVM to produce an inferring function, which can be used for classifying new instances [25].

In addition to the training set, a kernel function needs to be selected for the SVM model. The effectiveness of the selected kernel depends on the training datasets. The kernel selection and its regularization parameters is a challenging issue. Model overfitting may occur if the kernel model or its parameters are not selected appropriately. There are a few options for kernels such as Linear, Radial Basis Function (RBF), and Polynomial. Formal definition of these kernels are listed in Table I.

TABLE I
KERNEL DEFINITIONS

kernel	Mathematical Formulation
Linear	$K = (X, Y) = X^T Y$
Polynomial	$K = (X, Y) = (\gamma \cdot X^T Y + r)^d, \gamma > 0$
Radial Basis Function (RBF)	$K = (X, Y) = exp(-\gamma \cdot \parallel X - Y \parallel^2), \gamma > 0$

In the table r, d, and γ are the coefficient value, degree of polynomial, and the influence of a single training example. When training an SVM with the RBF kernel, two parameters must be considered: C and γ. The parameter C, which is common to all SVM kernels, controls the trade off between the misclassification of training examples and the simplicity of the decision surface. γ defines how much influence a single training example has. The larger γ is, the closer other examples must be to be affected.

2) Evaluation (Validation): In order to evaluate the performance and accuracy of a SVM model, cross-validation can be used. Cross-validation is a technique to assess how a statistical analysis will be generalized to an independent set. Cross-validation has different types such as *Leave-p-out*, *k-fold*, and etc [25], [4]. In the leave-p-out technique we use p data points as the validation set and the remaining data points as the training set. The model can be evaluated for different values of p. In k-fold validation, the dataset is randomly sliced into k equal sized data chunks. Of the k chunks, one chunk is retained as the validation set for testing the model, and the

remaining $k-1$ chunks are used as training data. The cross-validation process is then repeated k times (the number folds), with each of the k chunks used exactly once as the validation data. In the evaluation section of this paper, we evaluate our model using these techniques.

B. Active Learning

Active learning is sometimes called "query learning". It is a special case of semi-supervised machine learning in which a learning algorithm is able to interactively query the user (or some other information source) to obtain the desired outputs at new data points [6]. Active learning technique can be used as a tool to retrain a machine learning model with new instances (unlabeld data points). For example, to be able to detect new trends of "spam" in an email service, newly flagged emails as spam by users can be considered as new instances to retrain machine learning models. Here mailing service users are called "Oracle" and active learning techniques utilizes their opinions to label new instances and add them to the training sets. The labeling process refers to the process of measuring app's risk by experts and labeling them as malware or benign. In other words, the key idea behind active learning is that a machine learning model can achieve higher accuracy if it is allowed to choose the instances from which it learns. This is why the model chooses instances with a higher level of informativeness and achieve higher accuracy consecutively [27], [6].

There are several different problem scenarios in which the learner (model) may be able to ask queries on new data instances. Out of all the active learning techniques, the major two are the *stream-based selective sampling* and the *pool-based sampling*. The former is used when the model queries the unlabeld instance in an online (real-time) manner and the latter is used when new instances are stored in a pool (collection) of data and the learner queries the pool to label when needed. In our proposed model we use the stream-based technique. Next we elaborate the formal definition of active leaning and its query strategy [9].

1) Formal definition: Let T be set of all data in the original dataset. For example, in our model, T includes all apps that are known as malware or benign. During each retraining, say the i^{th}, the dataset T is split into three sub-datasets: T_{Ki} - known data points, T_{Ui} - unknown data points, and T_{Ci} - a subset of T_{Ui} selected to be labeled by the Oracle. If the active learning technique is based on a stream-based learning, then $\mid T_{Ui} \mid = 1$.

2) Query strategy: Query strategy, also called *"utility measures"*, is the process of choosing new incoming data instances to retrain the model. In other words, this process learner should determine which data point should be labeled [27]. There are several query strategies in active learning. For example, a query strategy based on *"uncertainty sampling"* selects query instances which have the least label certainty under the current trained model. This simple approach is not computationally expensive compared to others. We will describe the query strategy for our model in further details in Section III.

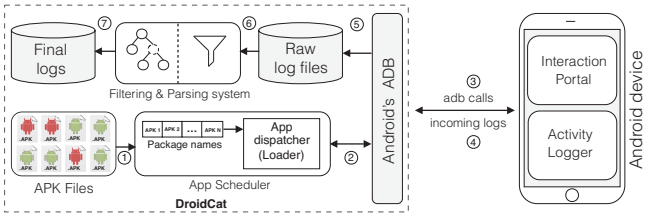

Fig. 1. DroidCat instrumentaiton tool architecture

III. Support Vector Machine Model

In this work, we use SVM and active learning for Android malicious app detection. We model the malicious app detection problem as a machine learning problem with two classes: *malicious* and *benign*. We map the app's behavior onto the SVM training dataset features. To train the SVM model, we capture the behaviors from both malicious and benign apps and use them to generate an initial trained SVM for malicious app detection. In this section we first present our SVM model and then explain how we can retrain the model using new instances to be able detect new types of malicious apps.

A. Data Collection

Our vision to specify app behavior is to view the running app as a black-box and focus on its interaction with the Android OS. In this case, a typical interface to monitor is the set of system and API calls that the app invokes during its running time. Every action that involves communication with the apps' resources (e.g., accessing the file system, sending SMS/MMS over the network, accessing the location services, calling Ads API libraries, and accessing the network) requires the app to launch OS services or API calls.

We used *DroidCat* instrumentation tool to capture apps' logs (behaviors). The DroidCat is developed by our team and it has been used in our previous projects [21], [22]. The reason we did not use the existing instrumentation tools in the market such as Robotium and uiautomator was because of their drawbacks and low accuracy. For example, Robotium cannot handle Flash or Web components nor simulate the clicking on soft keyboard, and it is not suitable for multi-process applications tests. Using DroidCat, we can capture the actual log activities of apps. DroidCat's architecture is illustrated in Figure 1. One of the main advantages of DroidCat is that it instruments apps through real human-interaction, so we capture the actual activities of apps which highly assimilate real-life apps' behaviors. As you can see in the architecture, DroidCat is composed of multiple components. Every one of the components is in charge of a task.

The first task is to extract the packages' names. We used `aapt` tool to accomplish it. The tool is designed to work with archive files. Since Android apps are in the format of APK (a type of archive files), we utilize the *appt* tool to read archive files. In addition, because this tool is embedded into the Android SDK, it does not impose a high performance overhead to the process.

After reading the packages' names and recording them, out next step is to run the apps. In order to be able run an

app, we should load it into the device's memory. We used ADB `logcat` tool to load the apps. The loading process also includes installing the apps and running them as well. *App dispatcher* component is in charge of the loading process. This component also determines the amount of time that the apps should run. By running the apps, we are able to capture the activities logs and record them at the time of instrumentation.

We call the collected logs from the previous step "raw" logs. The raw logs need to be filtered in order to eliminate the unnecessary information such as loading/installing/running logs.

1) Parsing: After filtering the log files, DroidCat eliminates unnecessary information and extracts important keywords. Each keyword refers to a sensitive resource access request, an API call, or Android action constants. We can also call them *features*. In our model, we focus not only on the generated Intents by apps but also on API library calls that cause permission escalation or generate unwanted Ads. In total we defined 150 keywords under various categories. When analyzing the parsed logs, we noticed that for some resources such as "WiFi", malicious and benign apps have different patterns in the timing of requests during app running. For example, the malicious apps tend to request the WiFi network during the first quarter of their running time period. Because of this, we include the timing of requests or library calls as an additional feature. Among of the 150 keywords, we added the timing feature to 56 of them. Therefore, we defined 206 *time-dependent* and *time-independent* observations in total.

B. Model Building

In this section we describe our RBF-based SVM model and its components. We start from the motivation of using SVM as a malicious app detection method and why RBF works the best for the model. We also elaborate the active learning component of the proposed model and its query strategy in details.

Figure 2 shows the overall view of our model. The key components of the model are *Model building*, *Model evaluation*, *Model optimization*. The model also has the capability of active learning using *Informativeness Measurement* and Oracle. The decision model is trained by a set of instances, called *Historical Data*, which is used as training set for the model. For new incoming instances, our model is able to label them using Oracle for active learning.

C. Model Training

For our model we choose SVM for classification algorithm. This is due to the large number of captured features from the data (206 features). It is difficult to find a separation boundary using other machine learning classifiers or clustering algorithms. A major advantage of SVMs is that the data can be transformed to a high-dimension space, where we can find a separation hyperplane using linear or RBF kernels.

By visualizing our collected data, we noticed that our data is not linearly separable. This is common when the training dataset has a large number of features. Figure 3 illustrates our

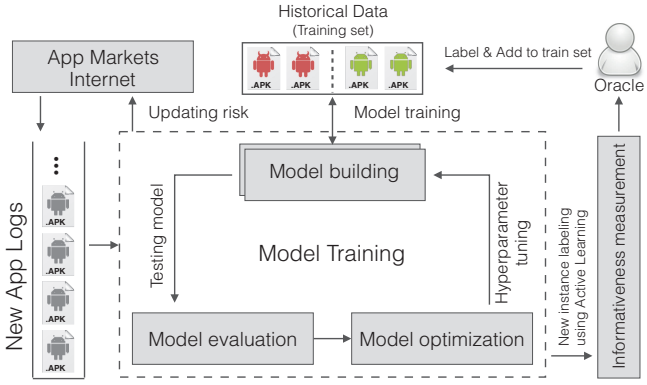

Fig. 2. SVM model and the active learning component architecture

training dataset using two features. The X axis and Y axis are the features. We trained our classification model using a sample set of our data with both linear and RBF kernels and plotted the separation boundaries. We can see that the RBF-based model can achieve cleaner separation compared to the linear kernel. Therefore, we use RBF as the kernel function in our model.

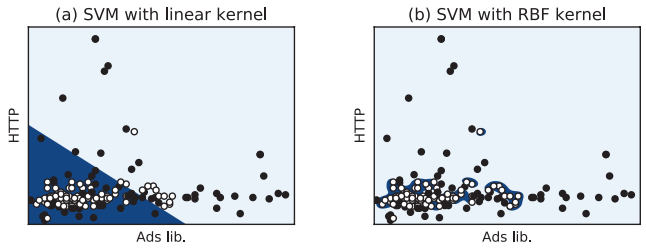

Fig. 3. Radial-Basis Function (RBF) and Linear kernel comparison on a sample dataset

D. Active Learning

We also integrate active learning into our model to be able to use new apps as training data points. To design an active learning model, two critical questions must be answered: 1) How often the model should be retrained to keep the detection rate high?, and 2) what query strategy should be used? We elaborate our answers to these questions in what follows.

1) Learning: Our strategy is to use a stream-based learning. This way the model queries new apps instantly and makes a decision on whether it should be labeled by the Oracle or not. There are two different types of stream-based learning: *Online learning* (one new app at a time) and *Batch learning* (a group of apps at a time). In online learning, the model decides whether to discard or process (label) a new data. In batch learning, the model collects new data until it reaches a certain size (batch size) and then decides whether to process them or not. Figure 4 (a) and (b) shows an overview of both online learning and batch learning respectively.

2) Query Strategy: Query strategy is the most influential part of an active learning-based model. Choosing a good strategy increases the accuracy of the model for future unknown apps. To achieve this goal, we chose the *Expected error reduction* strategy to query new apps. This strategy aims at labeling apps that minimizes the model's future generalization error. The idea is to estimate the expected future error of a

model trained using the original dataset $\mathcal{L} = \langle x, y \rangle$ and test it with the remaining data $\mathcal{U}$ (new data) and then query the new apps with minimal expected future error. An approach to minimize the expected 0/1-loss is as follows:

$$x_0^*/1 = \text{argmin} \sum_{u \in \mathcal{U}} E_y[H_{\theta^{+\langle x,y \rangle}}(Y \mid u)] \quad (2)$$

where $\theta^{+\langle x,y \rangle}$ denotes the updated model after it has been retrained with the training app $\langle u_x, u_y \rangle$ added to $\mathcal{L}$. In this equation, E_y and $H_{\theta^{+\langle x,y \rangle}}$ are the expectation over possible labeling of x and uncertainty of u after retraining with x respectively. Here, since we do not know the actual label of the query app, we approximate the label using expectation over all possible labels under the current model θ.

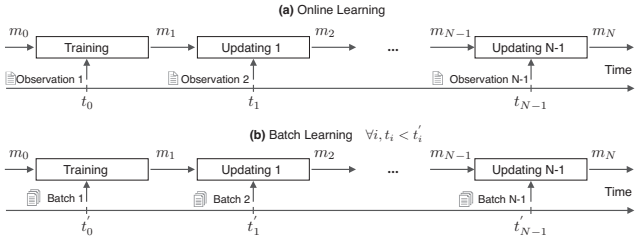

Fig. 4. Illustration of Batch learning and Online learning

IV. Evaluation

We present our evaluation results in this section. More specifically, we measure the accuracy of the model and then evaluate the stability of the model by adding noise to the training set. After that we evaluate the model under different settings of features using K-Best features. Finally we evaluate the active learning model under different settings of Oracle's expertise and batch sizes.

A. Experiment Setup

a) Software and Hardware: Our experiment environment is python 3.6 running on a same machine. We implemented all of our experiments using `scikit-learn` libraries powered by Google [18]. The libraries that are used in our experiments include *model_selection*, *train_test_split*, *confusion_matrix*, *SVC*, *cross_val_score*, *SelectKBest*, and etc. It is worth mentioning that to capture the actual logs of apps in our experiments, we used 4 LG Nexus 4 smartphones. We turned on all the sensitive resources such as Wifi, Bluetooth, and GPS.

b) Datasets: In order to conduct our evaluation experiments, we used a Android malware dataset called Drebin project [3]. The dataset includes more than 5K apps from 179 different malware families. We selected 700 apps from this dataset so that we have multiple apps from all the malware families. In addition to the 700 malicious apps, we collected 700 benign apps from various categories of Android apps. We randomly selected 500 malicious and 500 benign apps from both datasets (malicious and benign) and use them as *training sets* for the model, and the remaining 200 apps from each dataset are used as *test set* to test the performance of the model.

We defined the running time per app to be $2-5$ minutes in the App Dispatcher component. In total, the log capturing process (instrumentation) took around 79 hours for all the apps through human interactions.

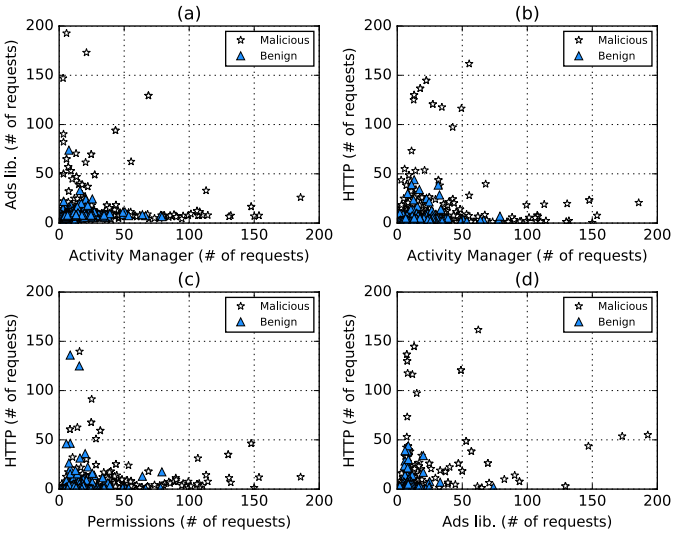

Fig. 5. Pairwise visualization of the dataset used for training and testing the model by four app behaviors (permission request, Ads lib., WiFi, and Activity Manager) as axises

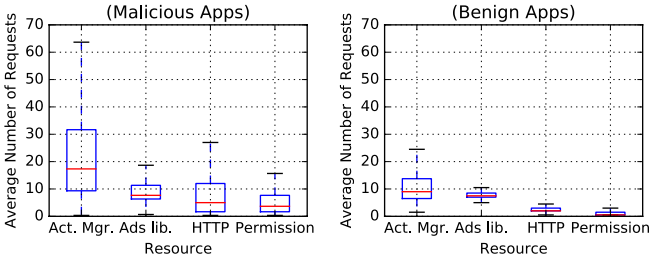

Fig. 6. Visualizing average resource request by malicious and benign apps for permission request, Ads lib., WiFi, and Activity Manager: (a) malicious apps; (b) benign apps.

B. Training Dataset Visualization

Before presenting the validation results, we visualize our data set using four features pairwisely. We selected permission manipulation, Ads lib. usage, HTTP request, and Activity manager interference features from the original dataset. Figure 5 shows the visualization results. In this figure, the x and y axis are a pair of the selected features representing the number of times that malicious and benign apps have requested those resource. We can see from the figures that the distribution of the malicious apps is more diverse than benign apps. Regardless of some differences, there is some overlap among malicious and benign apps. To visualize the distributions, we visualize the average usage of resources in Figure 6. We can see the comparison of the average resource usage by malicious and benign apps. The blue boxes in the figures represent the data range from the second quarter to the third quarter of the data sample, while the red bars are the medium values of the samples. The vertical whiskers indicate

the range of all data except outliers. The outliers are not plotted in the figures (which can be very large number).

C. Model Accuracy and Reliability

In the first experiment, we assess the accuracy of the model. We first tuned the model to find the best parameter configuration. Since we use RBF as our kernel function, we need to find the best values for C and γ. We used `scikit-learn` *model_selection* library to find the optimal kernel function and values for parameters. As a result, RBF was selected as the optimal kernel with $C = 1$ and $\gamma = 5.5e^{-4}$. After finding the optimal configuration for the model, we cross-validated our model on the training set to measure the average accuracy μ and standard deviation σ. Figure 7 shows the cross-validation results under different settings of C. We can see that with optimal setting for γ when $C = 1$, we have the highest average accuracy above 90% and a low deviation (Figure 7(a)) and when $C = 0.1$ (model is not tuned), the average accuracy drops and the standard deviation is higher (Figure 7(d)).

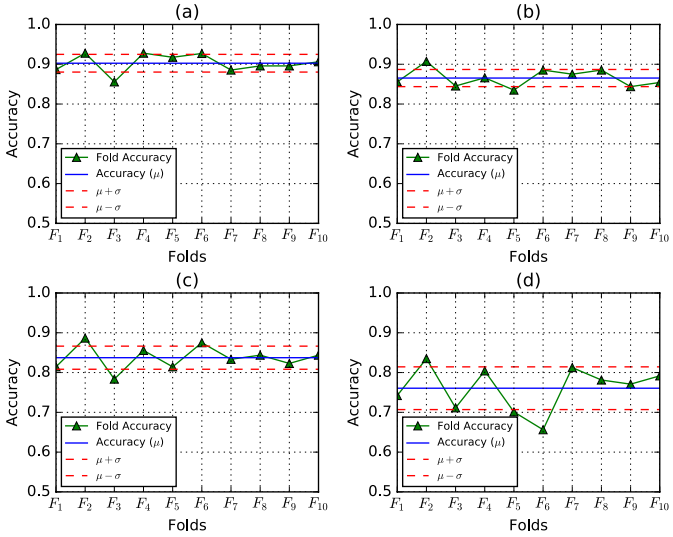

Fig. 7. Evaluation of model's accuracy using Cross-Validation and parameter (γ, C) tuning: (a) $(\gamma = 5.5e^{-4}, C = 1.0)$; (b) $(\gamma = 5.5e^{-4}, C = 0.5)$.(c) $(\gamma = 5.5e^{-4}, C = 0.3)$; (d) $(\gamma = 5.5e^{-4}, C = 0.1)$.

The second experiment is also on the model accuracy. We evaluated the reliability of the model using the leave-p-out validation technique. We split the training set into 10 subsets and train the model with one set or multiple sets. We increase the size of the training set start from 10% and increases by 10% each round. We validated the accuracy using the training as well as our validation sets. We ran the experiment for 10 times to plot the average accuracy and confidence interval in Figure 8(a). The results show that by increasing the training set size, the accuracy of model increases. We can also see that the accuracy of the model on the test set is lower than that of the the training set. The reason that we achieved such high accuracy is that, in contrast with existing approaches, we consider time in our features. We also see that with only 50% of the training set, the model is able to predict almost as accurately as that with 100% on the training set, which shows the reliability of the model.

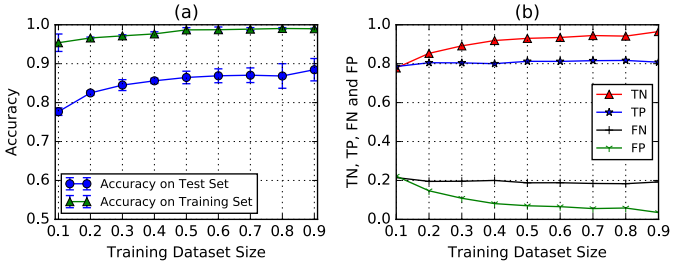

Fig. 8. Model accuracy evaluation: (a) accuracy of the model with different sizes of training dataset and evaluated by training and test datasets for 10 runs; (b) evaluated True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN) of the model.

We also carried out another experiment on the accuracy by measuring the True Positive (TP), True Negative (TN), False Positive (FP), and False Negative of our perdition. Figure 8(b) shows the evaluation results. Table II shows the performance of the model on the test dataset in terms of Recall, Precision, F_1-Measure for different training set sizes. We can see that all performance indicators increase with the training dataset size.

TABLE II
PERFORMANCE MEASUREMENT - RECALL, PRECISION, F1-MEASURE

Size	Recall	Precision	F1-Measure
10%	0.78	0.80	0.79
20%	0.80	0.86	0.83
30%	0.80	0.89	0.84
40%	0.80	0.91	0.85
50%	0.81	0.93	0.86
60%	0.81	0.93	0.87
70%	0.81	0.94	0.87
80%	0.82	0.94	0.87
90%	0.83	0.96	0.88

D. Model Stability Evaluation

In this subsection we assess the stability of the model. We define stability as the robustness of the model against noisy or incomplete malicious apps behaviors added to the training set. To evaluate the stability, we generated noisy data and mixed it with the training set. We generated the noise using the *Gaussian* distribution ($\mathcal{N} = (\mu, \sigma)$), a very common noise in machine learning validation process with different strengths 1 to 10. The generated noise is a vector of numbers in the size of ($|Features|$) that is added to the data points. Figure 9 shows the generated noise for this experiment. We also added the noise to the training set under different size settings from 5% of the dataset to 20% to see the effects of the noise if different portion of the dataset is affected. Figure 10(a) and (b) show the impact of the noise on the accuracy of the model using both training and test sets respectively. As you can see, when the strength and size of the noise increase, the accuracy slightly decreases. From this result, we can see that the model is still stable even after adding high strength noise to 20% of the training set.

In the second experiment on model's stability, we used *univariate feature selection* to measure the impact of features. Univariate feature selection works by selecting the best features based on univariate statistical tests. Among the univariate feature selection methods, we used *KBest* feature selection

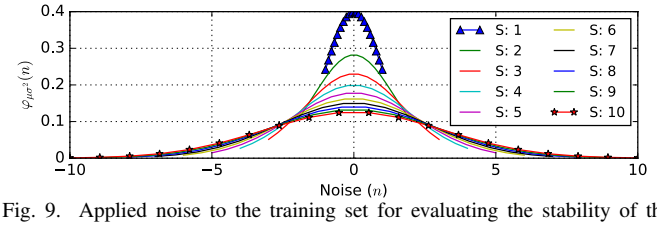

Fig. 9. Applied noise to the training set for evaluating the stability of the model under different strengths (S) (strength=1, $\cdots$, 10)

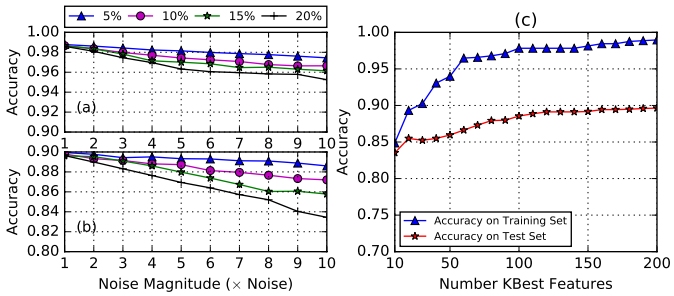

Fig. 10. Evaluation of noise and feature impacts: (a) accuracy of the model under different settings of noised data and noise strengths; (b) evaluating model's accuracy by training the model with different sets of the features using K-Best features ($K = 1, \cdots, 10$)

and evaluated the accuracy of the model for different sets of features on both training and test sets. Figure 10(c) shows the results of this experiment. From those results, we understand that a combination of features including high and low score features is necessary to keep the accuracy high. The reason is that, if the top K high score features do not represent the differences of classes, the result will be a biased and overfit trained model. As you can see the accuracy of the model increases by adding more features and at some point (starting from 100) it stays stable. Table III shows the top 10% best features together with the average requests by malicious and benign apps.

TABLE III
RESOURCE AVERAGE USAGE STATISTICS (TOP $K = 10$ BEST FEATURES)

Feature	WiFi	Ads	MMS	SMS	Blue.	Brow.	Root	Boot	Zygot	Call
Malicious	29.9	42	12.3	36.4	17.2	12.8	14.3	2.4	7.6	4.8
Benign	12	8.1	1.7	5.2	5.4	6.3	3.2	0.1	0.9	0.87

E. Active Learning Evaluation

In this section, we evaluate the active learning model of the model. We use three metrics to evaluate the performance of the active learning model. They are: 1) the speed of retraining the model to detect new instances; 2) the impact from the Oracle's expertise on the model's accuracy for the future apps; 3) the impact of the batch size when retraining the model using batch learning.

In the first experiment, we designed a scenario in which the model is initially trained by all the benign apps with different number of malicious apps. The rest of the malicious apps will be evaluated by the trained model and then verified by Oracle before they are added to the training set. Our goal is to assess how the active learning can help retrain the model to achieve higher accuracy in real time. In this experiment, since we are using labeled data, we can emulate an Oracle with expertise 1 who can accurately label all new apps. Figure 11 shows the evaluation results for this experiment. As you can see, the model is able to retrain the model by adding new labeled apps and improves the accuracy of the model. We can also see that when the initial model is trained with more malicious apps, the initial accuracy is higher. For example in Figure 11(d), when the number of initial malicious apps is 100, the accuracy increases very fast. We can also see that the detection accuracy fluctuates at the beginning in almost all cases. The reason is that the model is not sufficiently trained at the beginning and makes wrong predictions until more training data come in to improve the detection accuracy.

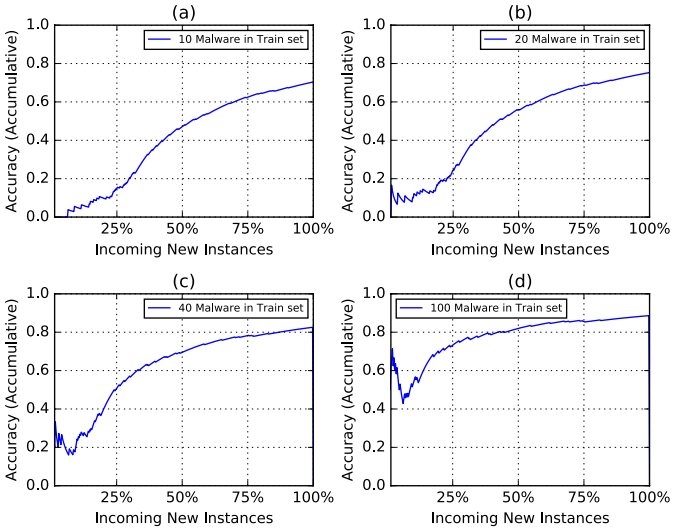

Fig. 11. Accumulative accuracy of the model using Active Learning for incoming new instances with different trained models and Oracle's expertise "1": (a) accuracy of model trained with 10 malware; (b) accuracy of model trained with 20 malware; (c) accuracy of model trained with 40 malware; (d) accuracy of model trained with 100 malware.

In the second experiment, we set the expertise of Oracle to different levels (0.6, 0.7, 0.8, and 0.9). In this scenario the initial model is trained with 50 malicious apps and all the benign apps. In this experiment we aim to see the impact of Oracle's expertise on the model's accuracy. Figure 12(a) shows the accumulative accuracy of the model for the new incoming malicious apps. We can see that with lower expertise, the accumulative accuracy increases slower than that with higher Oracle's expertise.

In the last experiment, we evaluate the accuracy of the model when our query strategy is based on batch learning. For this experiment, we fixed the Oracle's expertise to be 1 and 50 apps are used for the initial model. We also set the size of batch to be 10,30,60, and 90. Figure 12(b) shows the experiment results. We can see that the accumulative accuracy has a direct relation with the batch size. With smaller batch size, the model gets retrained more frequently than when using large batch sizes. Compared with the previous two experiments with online learning (batch size=1), the accumulative accuracy is proportionally lower even though it has higher computational

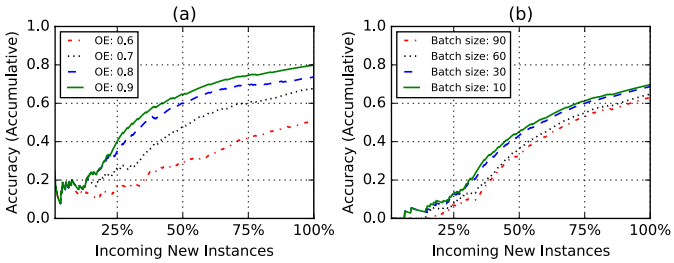

Fig. 12. Accumulative accuracy of the model using Active Learning for incoming new instances: under different settings of Oracle's expertise (OE) and batch sizes: (a) accuracy of model with different Oracle's expertise (0.6, 0.7, 0.8, and 0.9); (b) accuracy of the model with different batch sizes (10, 30, 60, and 90).

complexity. Therefore, choosing a proper batch size is critical for active learning.

V. Related Work

Several research efforts have focused on the principles and practices of managing resource usage and Android security [14], [5] and privacy protection [2], [24], [23]. Machine learning has been shown to be a powerful method to model security attacks or defend against attacks. These machine learning solutions have been widely discussed in literature. As a machine learning model SVMs have been extensively used to model security attacks. In the last two years, a few approaches [13], [26], [29], [15], [16] have been proposed using SVM to address the mobile security problems such as assuring users' privacy, and detecting malicious apps, and targeted malware.

Nissim et al. [16] proposed a machine learning model, called ALDROID, to detect Android malicious apps at runtime. The proposed model is the closest one to our model. The model is based on active learning and updates the machine learning model periodically. ALDROID uses active learning to reduce the labeling efforts of security experts, and enables a frequent process for enhancing the framework's detection model. ALDROID uses exploitation as query strategy. The model is trained by features that are based on the application's static code analysis. To construct the training dataset, they extracted requested permissions from `AndroidManifest.xml` as features. They claim that their model is behavioral-based. However static knowledge does not represent the actual apps' behaviors. They evaluated the model using simulation and the model is not well-evaluated, so it is not clear how stable and reliable the model is. In contrast, our model captures actual app behaviors and thus has higher accuracy and robustness. In addition, we conducted a set of comprehensive experiments to evaluate the stability and reliability of our model.

Narayanan et al. [15] proposed an adaptive Android malware detection solution, called DroidOL, based on online learning. The model aims at updating the learning model to be able to address the *malware population drift*. The training dataset was collected manually from official and unofficial app markets such as Google Play, FDroid, and SlideMe. To build the ground truth the collected apps were labeled using *VirusTotal*. They assumed that the model receives labeled data, which is not inter-procedural control flow sub-graph (static analysis). Also their model is based on a shaky ground, which means they assume that the model receives labeled data periodically and retrains the model. These all can be the reasons that the accuracy of the model is low. In contrast, our model uses active learning to handle the new unlabeled data.

The major difference between our proposed model and the existing ones is that they do not use a comprehensive app behavior analysis in their model training. The definition and discovery of proper observations, such as apps' intents, API calls, and time-stamp, make our model a unique solution in terms of Android malware detection. In addition, our proposed model updates the model's parameters using active online learning as its query strategy. To the best of our knowledge, ours is the first model to detect malware using a real-human app instrumentation data and active learning.

VI. Conclusion and Future Work

In this paper, we propose an Android malicious app detection framework using SVM and active learning. We first define and select features to represent behaviors of Android apps and collect them through our own developed human-oriented instrumentation tool DroidCat. A filtering and parsing method is then employed to synthesize and organize the captured behaviors. We train the model with a proper malicious app dataset using the RBF kernel and test it with test datasets. Through our model, we can detect malicious apps with high accuracy. On top of the SVM model we implemented an active learning method to improve the stability and robustness of the detection model against new instances of Android malware. Our experimental results demonstrate that our proposed model achieves satisfying accuracy in terms of true positive and false positive rate and adapts the detection model for new malware trends.

Optimal Query Strategy: In our model, the type of query strategy that we use has a high impact on the model's performance and accuracy. In this model, we used the expected reduce error strategy and achieved a reasonable performance. In order to improve the model, we can apply other query strategies to see if they can have better performance in terms of cost-effectiveness and accuracy. For example, some of the existing strategies measure the quality of the new data points and if they pass a quality threshold they can be included into the training set [28].

New Classes of Malware: One of the main important threats to any malware detection system is new trends (classes) of malware with very different behavioral patterns. In such a scenario, the model should be able to detect and retrain itself as soon as possible. One solution to this is using *progressive learning*. In progressive learning, the model is able to detect new classes and add them to the training set. Therefore, the training set will have more labels. This method can be implemented on top of the active learning method. Such a method for learning and training the model can be used if the model can be trained with a dataset with multiple (more than 2) labels.

REFERENCES

[1] What is the price of free. http://www.cam.ac.uk/research/news/what-is-the-price-of-free.

[2] Y. Agarwal and M. Hall. Protectmyprivacy: Detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 97–110, New York, NY, USA, 2013. ACM.

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*. The Internet Society, 2014.

[4] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[5] J. Crussell, R. Stevens, and H. Chen. MAdFraud: Investigating ad fraud in android applications. In *12th CMSAS*, MobiSys '14, pages 123–134, New York, NY, USA, 2014. ACM.

[6] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar. Learning in non-stationary environments: A survey. *IEEE Computational Intelligence Magazine*, 10(4):12–25, Nov 2015.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[8] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[9] J. a. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37, Mar. 2014.

[10] S. Gunasekera. *Android Apps Security*. Apress, Berkely, CA, USA, 1st edition, 2012.

[11] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[12] I. Lunden. 6.1b smartphone users globally by 2020, overtaking basic fixed phone subscriptions. http://techcrunch.com/2015/06/02/6-1b-smartphone-users-globally-by-2020-overtaking-basic-fixed-phone-subscriptions.

[13] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo. Machine learning aided android malware classification. *Computers & Electrical Engineering*, pages –, 2017.

[14] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *18th CMCN*, Mobicom '12, pages 317–328, New York, NY, USA, 2012. ACM.

[15] A. Narayanan, L. Yang, L. Chen, and L. Jinliang. Adaptive and scalable android malware detection through online learning. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 2484–2491, July 2016.

[16] N. Nissim, R. Moskovitch, O. BarAd, L. Rokach, and Y. Elovici. Aldroid: efficient update of android anti-virus software using designated active learning methods. *Knowledge and Information Systems*, 49(3):795–833, 2016.

[17] L. Nunez. Android just hit a record 88% market share of all smartphones. https://qz.com/826672/android-goog-just-hit-a-record-88-market-share-of-all-smartphones.

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

[19] T. S. Portal. Number of apps available in leading app stores as of march 2017. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores.

[20] B. Rashidi and C. Fung. A survey of android security threats and defenses. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 6(3):3–35, September 2015.

[21] B. Rashidi and C. Fung. Xdroid: An android permission control using hidden markov chain and online learning. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 46–54, Oct 2016.

[22] B. Rashidi, C. Fung, and E. Bertino. Android resource usage risk assessment using hidden markov model and online learning. *Computers & Security*, 65:90 – 107, 2017.

[23] B. Rashidi, C. Fung, and T. Vu. Recdroid: A resource access permission control portal and recommendation service for smartphone users. In *Proceedings of the ACM MobiCom Workshop on Security and Privacy in Mobile Environments*, SPME '14, pages 13–18, New York, NY, USA, 2014. ACM.

[24] B. Rashidi, C. Fung, and T. Vu. Dude, ask the experts!: Android resource access permission recommendation with recdroid. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 296–304, May 2015.

[25] M. Rychetsky. *Algorithms and Architectures for Machine Learning Based on Regularized Neural Networks and Support Vector Approaches*. Shaker Verlag GmbH, Germany, Dec. 2001.

[26] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference*, pages 141–147, Aug 2012.

[27] B. Settles. Active learning literature survey. *Computer Sciences Technical Report*, 1648, 2010.

[28] B. Settles and M. Craven. An analysis of active learning strategies for sequence labeling tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 1070–1079, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[29] S. Wu, P. Wang, X. Li, and Y. Zhang. Effective detection of android malware based on the usage of data flow {APIs} and machine learning. *Information and Software Technology*, 75:17 – 25, 2016.

[30] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 303–313, Piscataway, NJ, USA, 2015. IEEE Press.