Application of SVM Techniques a Machine learning method for analysis of data related to Android field

Abinash Parida

Project for industry purpose

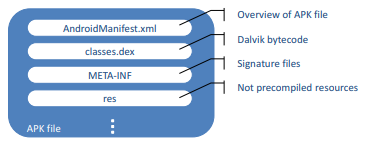
Introduction

Android is one of the most widely used operating systems (OSes) for smartphones, with a global market share of 87.7 % in terms of sales to end users. Its open specification facilitates the development of applications and their release on the Android application market. However, this makes it difficult to manage Android OSes and applications in a centralized manner. The difficulty of management enables Android malware to be distributed without being discovered. The amount of Android malware is increasing, and various types of threats exist for Android users . For example, Simplocker and LockerPin are two ransomware applications for Android. Simplocker encrypts user files while LockerPin changes a device’s personal identification number (PIN)2 lock. The new PIN is not known by the legitimate user or the attacker, so the user will be unable to obtain the PIN even if the requested ransom is paid. The impact of Android malware is not limited to smartphones. Although Android is currently used mainly on smartphone devices, it will be widely used by Internet of Things (IoT) devices as well. In fact, an Android-based OS for IoT, called “Android Things,” which was rebranded as “Brillo,” is already available.3 Consequently, Android malware will progressively affect more than just smartphones. In this chapter, we introduce techniques for detecting Android malware and describe how machine learning techniques, in particular support vector machines (SVMs) , can be used for analyzing Android applications.4 We also address dataset generation, which is essential for machine learning techniques because their performance largely depends on the size and quality of the dataset. We use not only permission requests and API calls, but also application categories and descriptions as the data source. Using the generated dataset, we demonstrate the effectiveness of using an SVM for analyzing Android applications by measuring the classification performance of an SVM and comparing it to that of a scheme that does not utilize machine learning. To improve the performance of the SVM further, we evaluate the effectiveness of the features used for analyzing Android applications and remove the non-contributing encoded features from the dataset. We then describe an experiment in which 94.15% classification accuracy was achieved. We close with a discussion of several issues and limitations on the practical use of machine learning techniques in this field.

Methodology

**Structure of Android Application Package**

Android applications are provided in the form of APKs. An APK file is a ZIP file consisting of multiple files, making it necessary to unzip it before use. As shown in Figure 1, each APK file contains the files “AndroidManifest.xml” and “classes.dex” as well as signatures files and resources that are not precompiled. AndroidManifest.xml and classes.dex are often used to analyze and evaluate the threats and vulnerabilities of APK files.



APK file structure

**AndroidManifest.xml: Central Configuration**

All APK files contain an AndroidManifest.xml file, containing assorted application information described in XML, although in a binary form; so it cannot be read as text. The text can be extracted using such tools as Apk-tool5 and Android Studio.6 . A permission system is used to restrict access to privileged system resources, and Android application developers have to explicitly declare the permissions in AndroidManifest.xml. The official Android permissions are categorized into four types, Normal, Dangerous, Signature, and Signature Or System, although the Signature Or System type was eliminated when Android 6.0 was introduced. The use of Dangerous permissions requires user approval because they allow access to restricted resources and may have a negative impact if used incorrectly. When taken as the input of a machine learning algorithm for malware detection, permissions are usually coded as binary variables; i.e., an element in the vector can take only one of two values: 1 for a requested permission and 0 otherwise. The number of possible Android permissions varies depending on the version of the OS. Android OS is continuously evolving, and the available permission requests and API calls may change from time to time. Therefore, one needs to refer to the uses-sdk tag to obtain the supported API versions, and run risk analysis using the supported permissions and API calls to conduct risk analysis efficiently.

**classes.dex: Dalvik Bytecode**

Android applications are developed in Java and compiled into Java bytecode. The bytecode is then translated into Dalvik bytecode and stored in Dalvik Executable (DEX) format, i.e., classes.dex. Dalvik bytecode is, like Java bytecode, reverse-engineering-friendly, enabling code analysis without the source code. There are tools that facilitate code analysis. As mentioned earlier, an APK file is a ZIP file, and the files in it are stored in a binary form. Therefore, the file content cannot be analyzed as is. The aforementioned Apktool converts AndroidManifest.xml into text. It can also generate smali code by reverse engineering the bytecode in classes.dex.

**Techniques for Identifying Android Malware**

Malware is distributed to Android terminals as APKs. To identify malware, APK analysis techniques are needed. Automated techniques are highly desired for malware analysis. In this section, analysis techniques in three categories (blacklisting, parameterizing, and classification) are introduced.

**Blacklisting**

Blacklist-based detection techniques are often used in a variety of fields, such as spam filtering and malware detection. There are specific blacklists for each application, e.g., blacklists of APK files, blacklists of URLs, and blacklists of application developer signatures. An APK file blacklist is a list of hash values of APK files identified as malware. A URL blacklist enumerates URLs that host malicious contents, such as malware, and APK files that communicate with these URLs are identified as malware. A blacklist of application developer is a list of the certificates of malware developers, and it is very likely that APK files with these certificates are actually malware.

**Parameterizing**

One automated approach to judging whether software is malware is to define a numerical parameter that represents the likelihood of the software being malware. If the value of this parameter exceeds a certain value, the software is considered malware.

**Classification**

In the classification approach, each sample is classified instead of defining and using key parameters. If malware detection is the only concern, schemes using this approach classify the samples into two groups. One might argue that the schemes described above can also be considered classification schemes, but unlike them, the schemes here usually do not provide human-friendly parameters and use machine learning techniques, which are well-known to outperform other types of techniques in classification tasks. There are several different machine learning techniques, but not all of them are suitable for classifying Android applications.. As described in Section 4, various features including API calls are used as SVM input in this chapter. Encoding these features as numerical attributes results in a very large dataset. This is particularly true for the API feature: more than 30,000 unique APIs are used by the APK files in our data. This high-dimensional and sparse data makes it difficult to use common machine learning techniques. However, according to Vapnik’s statistical learning theory, an SVM has guaranteed performance even for extremely high-dimensional data. A linear SVM is particularly preferable for such data because of its fast convergence rate and favorable generalization performance. Another reason to prefer a linear SVM over other methods is, that it facilitates fast feature selection for high-dimensional data. Therefore, only SVM is used for this purpose in this project.

Analysis

**Dataset Preparation**

Before running machine learning, a dataset must be prepared.

**APK File Analysis**

APK files are required to generate various datasets for the analysis. They are available in online APK markets, such as Google Play.8 There are several approaches to downloading these files, and there are tools and APIs especially designed for this purpose [8] that are worth considering. Some markets set several restrictions on the use of and access to APK files, which are described in the corresponding terms and conditions. By analyzing APK files, the data required to build a dataset can be extracted. There are basically two types of analyses for extracting features: static analysis and dynamic analysis. Static analysis inspects the files inside an APK file. Among these files, AndroidManifest.xml and classes.dex contain data suitable to be used as features. Permission request information can be extracted from AndroidManifest.xml, and API calls from classes.dex. Further information, such as the intent information, may also be extracted from AndroidManifest.xml, although this chapter utilizes permission request information and API call information only. In contrast to static analysis, dynamic analysis monitors the behavior and activities of running applications.

**Application Metadata**

APK analysis is not the only way to generate data for datasets. Other data sources include, for example, the metadata of APK files available in online APK markets. The APK files in APK markets are published with an application description. Application category information and the number of downloads are also often available. These pieces of information can be collected and used as features. Because application descriptions cannot be handled in their original form, they need to be converted so that they can be used as SVM input. One way to do this is to apply the bag-of-words model, which lists the frequencies of each word appearing in the description.

**Label Assignment**

To run supervised learning, label information is needed a priori. There are several techniques for obtaining this information. One is introduced here. It uses Virus Total, 14 an information aggregator that derives data from the output of various evaluation engines. If at least two of the results indicate that the file is malicious, the APK file is considered malware.

**Data Encoding**

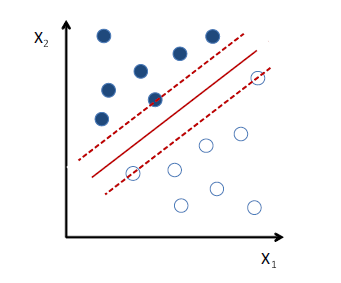
Encoding information from various sources into numerical features can be challenging: feature format and availability of features may vary from source to source. For example, while the weight and order may carry essential information for API calls, they are unavailable for features like permission requests and application categories. To encode all features consistently, we encode all the features as binary attributes. For permission requests and API calls, an attribute is set to 1 if the permission/API is declared in the manifest; otherwise it is set to 0. Application categories can be encoded as binary attributes following one-hot encoding: each application category is modelled as a binary attribute, and the attributes are mutually exclusive so that only one in the set can have a value of 1.

**Dataset**

Following the techniques mentioned earlier, a dataset can be generated. We collected 87,182 APK \_les from the Opera Mobile Store for the period January{September 2014. The \_les from which permission requests could not be extracted were excluded simply because permission requests are strictly required for our analysis schemes. The \_les that VirusTotal could not handle were also excluded from the dataset, because VirusTotal evaluation results are needed to label the dataset. Following the procedure described in Section 4.3, the \_les were labelled as malicious or benign, and adware was omitted. The result was a dataset of 61,730 APK \_les, consisting of 49,045 benign and 12,685 maliciously.

**Overview of Support Vector Machine**

An SVM is a machine learning model that maps features of data samples and draws a decision hyperplane that divide these features into groups. Figure 2 demonstrates how an SVM divides samples with 2-dimensional features into two groups. More precisely, an SVM draws hyperplanes (the dotted lines in the figure) that crosses the outermost sample for each of the two groups, and draws another hyperplane (the solid line) that maximizes the distance to each of the hyperplanes. The hyperplane serves as the borderline between the two groups. This model can be applied to Android malware identification: an SVM can divide APK \_les into two groups by using the features of the APKs and drawing a decision hyperplane between the features of benign software (benignware) and those of malware.



**Fig. 2** Division of samples with 2-dimensional features into two groups using an SVM

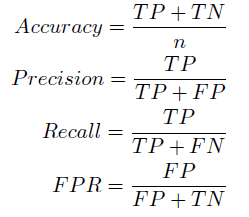
To classify samples with high accuracy, one needs to choose features that capture the characteristics of benignware and malware. Different types of features could be used, including requested permissions, application categories, application descriptions, and the number of downloads. This chapter explores the use of an SVM for detecting malware by preparing various features and evaluates which types of features effectively contribute to the detection performance.

**Tuning Hyperparameters**

Algorithm performance often depends strongly on the hyper parameters used to train the SVM model. Hyper parameter optimization a tuple of hyperparameters that yields a model that maximizes the generalization performance on independent data. Before performance evaluation, cross-validation is performed to estimate the generalization performance and identify the op-timal hyperparameters for later use.

**Evaluation Metrics**

There are common metrics for assessing classi\_er performance. They are calculated using four intermediate parameters: true positive (TP), false positive(FP), true negative (TN), and false negative (FN). TP is the number of predicted positive records classified correctly, FP is the number of predicted positive records classified incorrectly, TN is the number of predicted negative records classified correctly, and FN is the number of predicted negative records classified incorrectly.

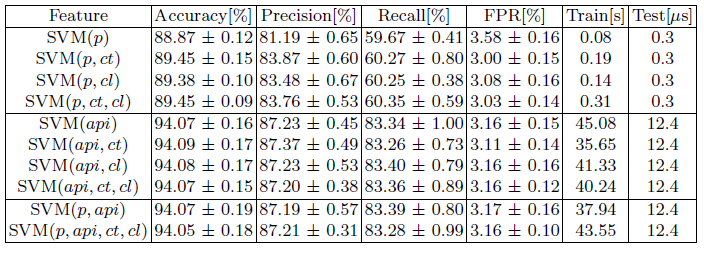


When measuring the performance in experiments, the hyper parameters that result in maximum accuracy are considered optimal. In practice, for applications that provide security alerts, it may be better to implement a scheme that minimizes the false negative rate (FNR), which is equivalent to (1-recall), while for applications that provide automated counter measures, it may be better to implement a scheme that minimizes FPR instead. In practical implementations, the individual use case determines which metric to use.

**Numerical Results**

To evaluate the generalization performance of the classifiers, the dataset can be randomly divided into training and test sets, and the experiment can be conducted several times. In our experiment, 70% of the data were used for training, and remainder were used for performance evaluation. The reported results were averaged over ten runs.

Performance of SVM-based schemes



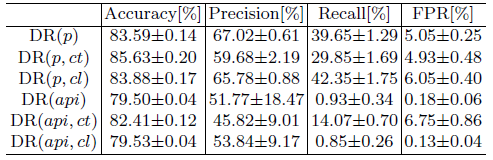
we compare the overall generalization performance using dif-ferent feature schemes. Because accuracy, precision, and recall show similar tendencies, accuracy is used in the following discussion. The upper block of the table shows the results for when the model was initially trained using the permission request feature, and then the results for when the application category feature and the cluster feature were used as well. On our dataset, the permission request feature yielded an accuracy of 88.87%. The accuracy increased to 89.45% when the application category feature was added and to 89.38% when the cluster feature was added. A comparison of these values indicates that the application category feature may carry slightly more discriminative information than the cluster feature. The accuracy remained at 89.45% when these two metadata features were used in addition to the permission request feature. In other words, the two metadata features carry similar information when used for malware detection. The middle block of Table 4 shows the results using the API feature. Here, the learning started with only the API feature; the other two metadata features were then added. Compared to the results for SVM(p), SVM(api) shows signi\_cant improvement for all four performance criteria. The 94.07% accuracy is a 5% performance gain compared to the accuracy with SVM(p). However, when the metadata features were added, there was only a slight improvement in accuracy. This suggests that metadata features carry little extra discriminative information useful for classification. The bottom block of Table 4 shows the results for SVM(p; api) and SVM(p; api; ct; cl). For both of these feature schemes, accuracy remained at the same level as with the API feature only. This con\_rms that API calls carry more \_ne-grained discriminative information than permission requests. Because of the associative relationship, permission means no additional discriminant information in terms of performance gain compared to classifiers based on the API feature alone. Compared to SVM(p; api), the accuracy of SVM(p; api; ct; cl) dropped slightly when the metadata features were also used for learning. To sum it up, using metadata features with the API feature provides little additional discriminative information useful for classification.

The last two columns of Table show the training and testing times for all settings. Due to the high performance of LIBLINEAR on mid-scale datasets, computation time is not considered a bottleneck for either training or testing. More advanced features can also be included in the learning as long as the generalization performance is improved.

**Comparison to Parameterizing Approach**

As mentioned above, machine-learning-based approaches outperform other schemes in classification. To understand the performance advantage, this section provides a comparison between a parameterizing approach and a machine learning approach (using DroidRisk and SVM, respectively). To enable a fair comparison, DroidRisk was extended to be able to incorporate features beyond permission requests. The following sections describe the extensions and then compare the performance to that of the SVM-based scheme.

Performance of DroidRisk (DR)-based schemes



Conclusion

In this project, techniques for identifying Android malware have been de- scribed, and the usability of machine learning techniques for this purpose have been demonstrated, with the emphasis on SVMs. The dataset used for running an SVM to identify malware was generated from permission requests,

API calls, application categories, and descriptions. Our evaluation measured the performances of SVM and DroidRisk on our dataset, leading to the conclusion that an SVM has a clear advantage over DroidRisk. To improve SVM performance further, SVM-RFE was used to evaluate attributes and identify their contributions. Removing the non-contributing feature achieved 94.15% of accuracy. The evaluation carried that API calls are the most dominant feature, and that permission requests and application categories also make some contributions to classification performance, while application clusters derived from application descriptions barely contribute to it. This chapter focused on SVM, but other machine learning techniques, such as deep learning, can also be considered for identifying Android malware. Different types of features and feature selection schemes could also be

explored. Such research efforts would greatly contribute to a more secure Android ecosystem.