ANDROID MALWARE DETECTION BASED ON FACTORIZATION MACHINE

by

Chenglin Li

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Department of Electrical and Computer Engineering
University of Alberta

Abstract

With the increasing popularity of Android smart phones in recent years, the amount of Android malware is growing rapidly. Due to its great threat and damage to mobile phone users, Android malware detection has become increasingly important in cyber security. Traditional methods for android malware detection, like signature-based ones, cannot protect users from the ever-increasing sophistication and rapid behavior changes in new types of Android malware. Therefore, lots of recent efforts have been made to use machine learning to characterize and discover the malicious behavior patterns of mobile apps for malware detection. In this thesis, we propose a novel and highly reliable machine learning algorithm for Android Malware detection based on the use of Factorization Machine and the extensive study of Android app features. We first extract 7 types of features that are highly relevant to malware detection from the manifest file and source code of each mobile app, including Application Programming Interface (API) calls and permissions. We have two observations. First, the numerical feature representation of an app usually forms a long and highly sparse vector. Second, the interactions among different features are critical to revealing some malicious behavior patterns. Based on these observations, we propose to use factorization machines, which fits the problem the best,

as a supervised classifier for malware detection. According to extensive performance evaluation, our proposed method achieved a test result of 99.01% detection rate with a false positive rate of 0.09% on the DREBIN dataset, and a 99.2% detection rate with only 0.93% false positive rate on the AMD dataset, significantly outperforming a number of state-of-the-art machine-learning-based Android malware detection methods as well as commercial antivirus engines.

Acknowledgments

I would like to thank all the people who contributed in some way to the work described in this thesis. First and foremost, I am deeply indebted to my academic advisor, Dr. Di Niu. His guidance and supervision helped me to learn how to be a good researcher and pursue an academic career. His willingness to discussion helped me through two important years of my life. Additionally, I would like to thank my committee members Dr. Cor-Paul Bezemer and Dr. Bruce Cockburn for their interest in my work.

I would also like to thank Dr. Rui Zhu, who is the cooperator in my research project on Android malware detection. Thanks for the discussions and his advice during the project, and I learned a lot from him through out the whole research work.

Thanks to Wedge Networks Inc. for giving me the internship through the Mitacs program. The internship during Wedge networks was a valuable experience for me, I really learned a lot from this program.

Finally, I would like to thank all my friends and family members who supported me during my time here. I would like to thank Mom, Dad and other family members for their constant love and support. I am lucky to have met lots of good friends here and have a happy life. Thanks for their friendship and unyielding support!

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Acronyms	Definition
FM	Factorization Machines
SVM	Support Vector Machine
AMD	Android Malware Dataset
t-SNE	t-distributed Stochastic Neighbor Embedding
API	Application Programming Interface
APK	Android Application Package
ICFG	Inter-procedural Control Flow Graph
CFG	Control Flow Graph
IC3	Inter-Component Communication Analysis for Android
JVM	Java Virtual Machine
IPC	Inter-process Communication
MLP	Multilayer Perceptron Classifier
NB-G	Naive Bayes with Gaussian kernel
NB-B	Naive Bayes with Bernoulli kernel
NB-M	Naive Bayes with Multinomial kernel
ROC	Receiver Operating Characteristic curve
AUC	Area Under Curve

Chapter 1

Introduction

1.1 Android Malware

In recent years, we have witnessed the explosive growth of smartphone usage after global sales surpassed the sales of basic mobile phones (or feature phones) in early 2013 [1]. Nowadays, smartphones are used everywhere in our daily life, e.g., for online shopping, mobile games, online banking, personal heath care, and even as remote controllers. According to a survey on global mobile OS market shares [2], Android is the dominant mobile operating system with a 87.7% market share as of the second quarter of 2017. Android is now powering not only smartphones but also tablets, TVs, wearable devices and even IoT with Android Things. The rapid growth of smartphone usage and the huge market share of the Android OS have not only brought about the opportunities for mobile application development, but also the challenges needed to defend devices from Android-targeting malware. According to Kaspersky's Mobile Malware Evolution 2016 Report [3], the number of malicious installation packages amounted to $8,526,221$ in 2016—almost three times more than that in 2015. Also, the distribution of malware through Google Play and other online app stores is growing rapidly.

1.2 Signature-Based Approaches

To win the battle and protect mobile phone users, a number of anti-virus companies (e.g., McAfee, Symantec) provide software products as a major defense against these kinds of threats. These products typically use a signature-based method [4] to recognize threats. For example, [4] proposed a signature-based malware detection method that is well suited for mobile devices. With signature-based methods, a unique signature is generated for each previously known malware, while detection involves scanning an app to match existing signatures in a malware database. However, this can be easily evaded by attackers. Example counter-methods involve changing signatures using code obfuscation or repackaging. To overcome this issue, the heuristic-based method, introduced in the late 1990s, uses explicit expert rules to recognize malware, although these rules are prone to human bias. In fact, both methods will be less effective if the development of the malware database or expert rules cannot keep pace with the speed at which new malware emerges and evolves.

1.3 Machine Learning-Based Approaches

An alternative emerging approach for malware detection is to develop intelligent malware detection techniques based on machine learning. Machine learning is potentially capable of discovering certain patterns in previously undetected malware samples. One major type of machine learning-based malware detection method use the so called static analysis [5], [6] to collect features, this kind of analysis method can make decisions about an app without executing it in a sandbox, thus incurring a low overhead. Static analysis has two phases: feature extraction and classification. In the first phase, various features such as API calls and binary strings are extracted from an original file. In the second phase, machine learning is used to automatically categorize the file sample into malware or benign-ware based on a vectorized representation of the file. Different machine-learning-based malware detection

methods could differ in both phases. For example, DroidMat [5] performs static analysis on the manifest file and the source code of an Android app to extract multiple features, including permissions, hardware resources, and API calls. It then uses k-means clustering and k-nearest neighbor (k-NN) classification to detect malware. DREBIN [6] extracts similar features from the manifest file and source code of an app and uses a support vector machine (SVM) for malware classification based on one-hot encoded feature vectors.

Many recent works are trying to find malicious behavior patterns through control flow graphs or call graphs. AppContext [7] classifies applications using machine learning based on the contexts that trigger security-sensitive behaviors. It builds a call graph from an application source code and extracts the context factors through information flow analysis. Then AppContext [7] is able to obtain the features for the machine learning algorithms from the extracted context. In the paper [7], 633 benign applications from the Google Play store and 202 malicious samples were analyzed. AppContext correctly identifies 192 of the malware applications with an 87.7% accuracy. Gascon et al. [8] also utilized call graphs to detect malware. After extraction of call graphs from Android applications, a linear-time graph kernel is applied in order to map call graphs to features. These features are given as input to SVMs to distinguish between benign and malicious applications. They conducted experiments on $135,792$ benign and $12,158$ malware applications, detecting 89% of the malware with 1% of false positives. This kind of method relies heavily on the accuracy of the call graph extraction. However, current works like FlowDroid [9] and IC3 [10] cannot fully solve the construction of Inter-component control flow graphs (ICFG), especially the inter-component links with intents and intent filters.

However, existing machine learning techniques for malware detection yields limited accuracy, mainly due to the use of a first-order model or linear classifiers, such as SVM [6]. These are insufficient to discover all malicious patterns. A natural idea to introduce nonlinearity into malware detection is to consider the interaction between features, or in other words, *feature crossing* or *basis expansion*. For ex-

ample, an app concurrently requesting both GPS and `SEND_SMS` permissions may be attempting to execute a location leakage, while the presence of either one of such requests alone does not point to any malicious behavior. However, machine learning models involving feature crossing are not scalable to long feature vectors.

For example, a total of $545,000$ features are used by DREBIN [6], the SVM-based detector, which means that more than 297 billion interactions need to be considered if feature crossing were to be used. One could expect this number to be even larger in a more recent dataset; the Android Malware Dataset (AMD) [11], which contains more file samples thus exposing more features. Moreover, although the total number of features is large, the number of features activated by each file sample is usually much smaller, leading to a sparse vectorized representation for each individual app. This will further lead to even sparser interaction terms (the cross terms), posing significant challenges to model training—there are not enough non-zero entries in the dataset to train the coefficient of each crossed term.

In this thesis, to effectively model feature interactions as well as efficiently handle long and sparse features, we propose a novel *factorization machine* (FM) model for Android malware detection. In contrast to feature crossing or basis expansion, which suffers from the model size issue and the sparsity issue mentioned above, factorization machines [12] aim to learn the coefficient of each interaction as the inner product of two latent variables corresponding to the two features, thus effectively reducing the number of parameters to be linear to n, where n is the length of the feature vector.

We fine-tuned the feature extraction process and performed extensive feature engineering on Android apk files. We evaluated our model on two typical malware datasets: the DREBIN dataset [6], involving $5,560$ malware samples, and the Android Malware Dataset (AMD) [11], involving $24,553$ malware samples. On the DREBIN dataset, we achieved a 99.01% detection rate with 0.09% false positives. For the AMD dataset, a detection rate of 99.2% was achieved with a 0.93% false positive rate. These results suggest that our proposed method is highly accurate and reliable, substantially outperforming all state-of-the-art machine learning methods

for Android malware detection (including the SVM-based DREBIN [6] with a reported detection rate of 94%) as well as most of the existing Anti-Virus engines uploaded to VirusTotal by commercial vendors including Cylance and Kaspersky.

We also evaluated the performance of our method on malware family identification, which is an important task in malware attribution. For this task, our model is trained to identify the apps that belong to a certain malware family, among samples from other families as well as clean files. We achieve an average detection rate of 98.73% with an average false positive rate of 0.17% for 7 malware families from the AMD dataset.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. We first introduce the background of the Android system and describe our malware detection system in detail in Chapter **2**. Details about the proposed machine learning models and the motivations to use a factorization machine are given in Chapter **3**. Experimental results regarding malware detection, malware family identification are presented in Chapter **4**. Chapter **5** discusses related work, while the limitations of our current system are discussed in Chapter **6**. Finally, we conclude the thesis in Chapter **7**.

Chapter 2

Background and System Overview

In this chapter we will first introduce some background information for the Android operating system and Android application files (`apk` files). Then we briefly introduce the architecture of our malware detection system.

2.1 Background

Android applications are written in Java and executed within a custom Java Virtual Machine (JVM), and each application package is contained in a jar file with the file extension of `apk`. Android applications consist of many components of differing types, which are the essential building blocks for the application. Each component has an entry point through which the system or a user can enter the application and applications interact via components. Therefore, it is critical to analyze the component APIs for security concerns. There are four fundamental building blocks of applications on the Android platform.

1. **Activities** serve as the entry point for a user's interaction with an app, and are also central to how a user navigates within an app or between apps.

2. **Services** are components that can perform long-running operations in the background without providing a user interface. A service can be started by other application components and will continue to run in the background even

if the user switches to another application. In addition, components can be bound to services to interact with them, and even perform inter-process communication (IPC). For example, services can handle network transactions, play music, perform file I/O, or interact with content providers, all of which can occur in the background.

3. **Broadcast receivers** Android apps can send and receive broadcast messages from the Android system and other Android apps. These broadcasts are sent when an event of interest occurs. For example, the Android system sends broadcasts when various system events occur, such as when the system boots up or the device starts charging. Apps can also send custom broadcasts, for example, to notify other apps of something that they might be interested in, such as the completion of a download.

   Apps can subscribe to receive specific broadcasts. When a broadcast is sent, the system automatically routes broadcasts to apps that have subscribed to receive that particular type of broadcast. Generally speaking, broadcasts can be used as a messaging system across apps and outside of the normal user flow

4. **Content providers** are components that are used to manage access to structured data sets, encapsulate data and provide mechanisms for defining data security. A content provider is a standard interface for connecting data in one process to code running in another process.

All components must be declared in the application manifest file before they can actually be used. Communications between different components are through intents and intent filters. Intents are messaging objects that can be used to request actions from other application components. An intent filter is an expression declared in the application manifest file that specifies the intent type that the component will receive.

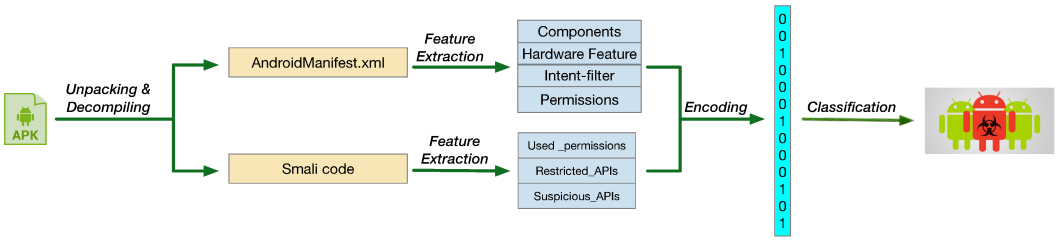

Fig. 2.1. System architecture of our malware detection model.

2.2 System Overview

Our malware detection system consists of four parts: Unpacking and Decompile, Feature Extraction, Encoding, and Prediction - all shown in Fig. 2.1. By the end of this section, we will have a detailed introduction for each part.

2.2.1 Unpacking and Decompiling

The original data we receive for each application is an Android `apk` file. Each `apk` file is actually a zipped file that consists of the application source code, resources, assets, and manifest file. The application source code is encoded as dex files (i.e., Dalvik Executable Files) that can be interpreted by the Dalvik VM. The manifest file consists of a number of declarations and specifications. Finally, other resources may contain images, HTML files, etc.. Unfortunately, the dex files, as executable code, are hard to understand and therefore need to be converted into readable formats such as smali code or even Java code. Smali code is an intermediate form decompiled from the dex files. The takeaway is that, after unzipping the `apk` file we still need to decompile dex code before we can continue to feature extraction.

There are some popular tools available for decompiling dex code such as APKTool [13] or `baksmali` [14]), which can unpack the `apk` file and decompile the dex files to smali code. In our system, we use `aapt`, which is a Android SDK tool, to extract manifest file information into a readable text file, and use APKtool to convert the classes.dex file into smali code. After this step, we obtain readable source code and the manifest file `AndroidManifest.txt` for each Android app, based on

that representative features will be extracted.

2.2.2 Feature Extraction

Feature engineering is the most important part for training a machine learning model. The upper bound of the performance of the model is depend directly on the used features. Through study of the Android system and tons of previous work, we finally decided to extract 7 kind of features from both the source code and manifest file. From the manifest file we extract the following four types of features:

1. **App components**: As we know, an app contains several components of four types: `activities`, `services`, `content providers` and `broadcast receivers`. Those components, declared in the manifest file, define different user interfaces and interfaces to the Android system. The names of these components are collected to help identifying variants of well-known malware, for example the DroidKungFu family share the name of particular services [6].

2. **Hardware features**: If an application wants to request access to the hardware components of the device, such as its camera, GPS or sensors, then those features must be declared in the manifest file. Requesting certain hardware components may have security implications. For example, requesting of GPS and network modules may be a sign of location leakage.

3. **Permissions**: Android uses a permission mechanisms to protect the privacy of users. An app must request permission to access sensitive data (e.g. SMS), system features (e.g. camera) and restricted APIs. Note that the permission system is one of the most important security mechanism in Android. Many operations need specific permissions to be executed and these permissions are granted by users upon installation. Malware usually tends to request a special set of permissions. Similar ideas also apply to hardware resources.

4. **Intent filter**: Intent filters declared within the declaration of components in the manifest file are important tools for inter-component and inter-application

communication. Intent filters define a special entry point for a component as well as the application. Intent filters can be used for eavesdropping specific intents. Malware is sensitive to a special set of system events. Thus, intent filters can be hints.

Furthermore, we also extract another three types of features from the decompiled application source code (e.g., smali code):

1. **Restricted APIs**: In the Android system, some special APIs related to sensitive data access are protected by permissions. If an app calls these APIs without requesting corresponding permissions, it may be a sign of root exploits.

2. **Suspicious APIs**: We should be aware of a special set of APIs that can lead to malicious behavior without requesting permissions. For example, cryptography functions in the Java library and some math functions need no permission to be used. However, these functions can be used by malware for code obfuscation. Thus, attention should be paid to the unusual usage of these functions. We will mark these types of functions as *suspicious APIs*, following in the footsteps of DREBIN [6].

3. **Used permissions**: We first extract all API calls from the app source code, and use this to build a set of permissions that are actually used in the app by looking up a predefined dictionary that links an API to its required permission(s).

2.2.3 Encoding

As seen in the previous section, all of the features are associated with string values, so we need to encode them before they can be fed into a classifier. Here we use an N-dimensional indicator to encode each application into a feature representation, where N is the feature dimension. To be specific, suppose all the extracted features form a feature set S with size $|S|$, then each app will be represented as a

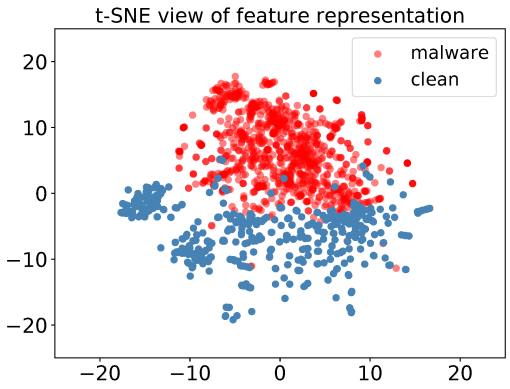

Fig. 2.2. t-SNE view of $2,000$ samples from the DREBIN dataset $(1,000)$ and clean file dataset $(1,000)$. All these samples have been encoded as a highly sparse vector and then t-SNE algorithm is applied for dimension reduction and visualization.

$|S|$-dimensional indicator, where each dimension is either 1 or 0 indicating whether the corresponding feature appears in the app. There are two things need to be noticed. First, the feature set size $|S|$ is often very large and grows as the dataset size becomes larger. Second, the number of features extracted for each app is relatively very small compared with the feature dimension, so we would often get a large, highly sparse vector representation for each app. We will further discuss this in the following chapter.

To show the effectiveness of our feature representation in distinguishing malware and clean files, we further apply t-SNE [15] algorithm on $2,000$ already encoded samples from the DREBIN dataset $(1,000)$ and clean dataset $(1,000)$ for visualization. The result is shown in Fig 2.2, it is not hard to tell that all the samples are nicely spaced apart and grouped together with their respective labels.

2.2.4 Classification

After encoding, we would get a vector representation of each application, based on which we can then apply machine learning algorithms for automatic malware detection. There are several learning algorithms that can be used for classification, for example, DREBIN [6] uses support vector machine (SVM), [16] uses one-class SVM with kernels. And as a general classifier, deep neural networks are also widely

used in malware detection [17], [18]. In this thesis, instead of randomly choosing a general classifier to get a good prediction model by tuning the parameters, we first make observations on the vector representations of the Android application and then choose the factorization machine model that fits our problem the best. Notice that FM model can also be used for malware family identification. Details are presented in the next chapter.

Chapter 3

Factorization Machine for Malware Detection

At the core of our malware detection scheme is classification. Generally, a classification problem in machine learning is to infer a function $h : \mathbb{R}^n \rightarrow \mathbb{R}$ for all possible $\mathbf{x} \in \mathbb{R}^n$ to predict how much it belongs to a class, e.g., the malware class in this thesis. To find such a function, we are given a set of samples, each of which has been marked as a "malicious" or "benign". This initial dataset is used to teach the machine.

After proper pre-processing has been performed on each Android application file, it is then converted into a feature vector $\mathbf{x}$ in accordance with chapter 2. In this chapter, we introduce the modeling of malware detection based on factorization machine [12]. This machine has demonstrated high efficiency in learning high-order interaction representation between sparse features with numerous applications in various fields.

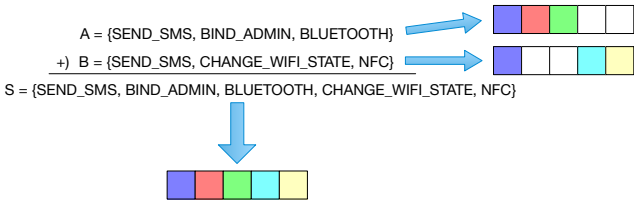

Fig. 3.1. One-hot encoding for string features. Here we use different color blocks to represent different feature values in permission set, and the block with color will be encoded as 1 and the white block will be encoded as 0.

3.1 Feature Representation and First-order Classifiers

We begin our modeling from feature representation in Android malware detection. Suppose we have two applications, A and B, and each requests three permissions, as illustrated in Fig. 3.1. As there are five unique permissions requested by A and B, we can then create a vector $\mathbf{x}_A, \mathbf{x}_B \in \{0,1\}^5$ such that each entry represents exactly one permission, e.g., the first entry as a blue block represents the permission `SEND_MSG` and the second entry represents the permission `BIND_ADMIN`. As a result, we can write $\mathbf{x}_A = (1,1,1,0,0)$ and $\mathbf{x}_B = (1,0,0,1,1)$. It is straightforward to extend this idea to all kinds of extracted features as discussed in chapter 2. The formal name for this scheme in literature is *one-hot* encoding.

There are some popular models for a scalable and stable solution of classification. One popular solution is *support vector machine* (SVM), which attempts to find a hyperplane that separates malware samples from benign ones with a maximal margin. A maximal margin solution usually performs better in many machine learning tasks, so SVM has been widely applied in many fields in addition to Android malware detection [6]. More specifically, an SVM model attempts to find a hyperplane such that

$$h(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i + w_0,$$

where $w_i, i = 0, 1, \ldots, n$ are trainable parameters such that it can maximize the margin. To predict the probability of whether a given $\mathbf{x}$ is a malware, we can further

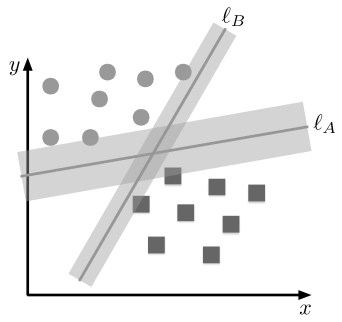

Fig. 3.2. SVM classifier leads to maximal margin solutions. Here ℓ_A leads to a larger margin than ℓ_B and is believed to be a better classifier.

use a sigmoid function to calculate such value:

$$\hat{y}(\mathbf{x}) := \sigma(h(\mathbf{x})) = \frac{1}{1 + \exp(-h(\mathbf{x}))}. \tag{3.1}$$

Here we denote $\hat{y}(\mathbf{x})$ (or $\hat{y}$ for short) as the estimated probability of being a malware. Given a set of samples $\mathcal{D} = \{(\mathbf{x}, y)\}$, the optimal coefficients of w_i can be obtained by solving the following optimization problem:

$$\begin{aligned} \min_{\mathbf{w}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{s.t.} \quad & y \cdot h(\mathbf{x}) \geq 1, \ \forall (\mathbf{x}, y) \in \mathcal{D}, \end{aligned} \tag{3.2}$$

where $\|\cdot\|$ denotes the ℓ_2 vector norm, and y is the sample label, if a sample is malware then $y = 1$, otherwise $y = 0$.

However, these models are not suitable for Android malware detection for two reasons. *First*, the feature vectors from one-hot encoding are highly sparse. For example, samples in the benchmark dataset DREBIN [6] will be encoded into vectors with $93,324$ entries, in which only 73 nonzero elements are found on average. *Second*, these models only exploit the first-order features, they do not take interactions among entries into account. To make matters worse, the severity of these problems is amplified in Android malware detection because the high sparsity of features implies that each feature vector can provide little information for classification should a model only exploit the information from nonzero values.

3.2 Second-order Feature Crossing and Factorization Machine

To overcome these issues, we attempted to incorporate feature interactions. Let us take some toy examples to see how the relationship between two features can facilitate the prediction of malware. If an application requests the GPS hardware feature as well as network modules permission, it is likely that this application may attempt to send geo-location information to a command & control server, therefore it is more prone to perform malicious behaviors. Another example is that some malware samples like `BaseBridge` can collect personal/device information and send it elsewhere via SMS messages. They will request two permissions, `READ_PHONE_STATE` and `SEND_SMS`.

A natural method for learning interactions of different features is through basis expansion or feature-crossing:

$$h(\mathbf{x}) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} W_{ij} x_i x_j. \tag{3.3}$$

By assigning a weight $W_{i,j}$ for each pair of x_i and x_j, we have the easiest way to capture pairwise interactions. However, it is not efficient here due to the large number of parameters: this model has $n(n-1)/2$ free parameters. In the DREBIN dataset, for example, the input vector has a length of $93,324$ but the number of nonzero entries is about 73 on average. In this case, full feature crossing like W would necessitate four billion weights. This would impose a very heavy burden on the training process since the model becomes too complicated and it requires a large scale of data for training. Needless to say, the end result is very time-consuming. In additon, it looks much worse for sparse data in the case of Android malware detection, as each sample only activates an extremely small portion of entries in W when using popular algorithms like stochastic gradient descent (SGD).

Popular techniques to overcome these issues are low-rank or dimension reduc-

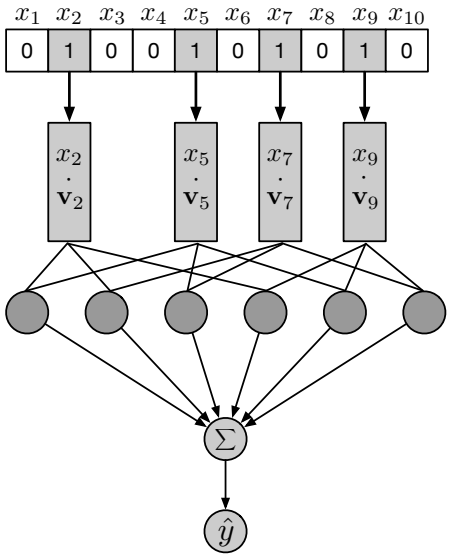

Fig. 3.3. The architecture of factorization machine model for malware detection. Here the dark gray node stands for the inner product operator, i.e., it calculates inner product of two incoming vectors.

tion methods, such as using a factorization machine (FM) [12]. More specifically, FM assumes that W is with the largest rank of k and therefore, we can decompose $W = VV^{\mathsf{T}}$. If we denote $\mathbf{v}_i$ as the i-th row of V, FM will train a hidden vector $\mathbf{v}_i$ for each x_i and the model the pairwise interaction weight w_{ij} as the inner product of the corresponding hidden vectors of entries x_i and x_j:

$$h(\mathbf{x}) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j, \tag{3.4}$$

where $\langle \cdot , \cdot \rangle$ denotes the dot product of two vectors of length k:

$$\langle \mathbf{v}_i, \mathbf{v}_j \rangle := \sum_{f=1}^{k} v_{i,f} v_{j,f}, \tag{3.5}$$

In practice, the hyperparameter k is much smaller than the feature dimension n ($k \ll n$). Thus, the number of parameters to be estimated is reduced from $O(n^2)$ to $O(nk)$.

We can further improve the performance of FM by using more sophisticated feature engineering schemes for cross terms. For example, by using "partial FM",

which only involves interactions between selected features, e.g., between `Used permissions` and `Permissions`, thus ignoring crossed terms that are not relevant to malicious behavior discovery.

Chapter 4

Evaluation

In this chapter, we evaluate the performance of our factorization-machine-based Android malware detection system. We apply our system to malware detection task and malware family identification task, based on two public benchmark datasets: DREBIN [6] and AMD [11]. In addition to detection performance evaluation, we further evaluate efficiency in terms of processing time and detection time for all tasks.

TABLE 4.1
PERFORMANCE METRICS FOR ANDROID MALWARE DETECTION.

Metrics	Description
TP	# of malicious apps correctly detected
TN	# of benign apps correctly classified
FP	# of false prediction as malicious
FN	# of false prediction as clean
$Precision$	$TP/(TP+FP)$
$Recall$	$TP/(TP+FN)$
$F1$	$2*Precision*Recall/(Precision+Recall)$
FPR	$FP/(FP+TN)$

4.1 Experiment Setup

4.1.1 Datasets

We will start with a brief description for each malware dataset:

- **DREBIN**: it is a dataset with $5,560$ malware files collected from August 2010 to October 2012. All malware samples are labeled by one of 179 malware families. This is one of the most popular benchmark dataset for Android malware detection.

- **AMD**: the Android Malware Dataset contains $24,553$ samples that are categorized in 135 varieties among 71 malware families. This dataset consists of samples that were collected from 2010 to 2016. This is one of the largest, and the newest dataset at April 2018. This dataset provides more recent Android malware evolving trends.

Along with these malware datasets, we also collected a number of real-world Android applications collected from the Internet. Resources of these files include Apkpure [19] with $5,400$ samples, 700 samples from 360.com and 13K commercial applications from the HKUST Wake Lock Misuse Detection Project [20]. In summary, we have collected $19,100$ real-world applications.

Although these Android applications are mostly collected from well-known Android markets and research projects, we should ensure whether they are clean. To do so, we uploaded all these collected files to the VirusTotal service, a public antivirus service with 78 popular engines, and inspected scanning reports from the VirusTotal service for each file. Each engine in VirusTotal would show one of three detection results: **True** for "malicious", **False** for "clean", and **NK** for "not known", respectively. If an application has more than one **True** result, we label it as **malware**; otherwise, we label it as **clean**. As a result, only $16,753$ out of 19K collected samples passed all scanners on the VirusTotal service, and we will only use these samples in further experiments.

Details of these two datasets are shown in Table 4.2. When doing experiments on the AMD dataset, we evaluated on all these clean files. When evaluating on the DREBIN dataset, we randomly sampled $5,600$ clean files to match the number of malware samples in this dataset. To simplify our terminologies, the DREBIN dataset (or the AMD dataset) consists of both clean samples and malware samples in the subsequent of this section.

Here we make some comparison between the DREBIN and the AMD datasets for further experiments. Table 4.3 shows a detailed breakdown of these two datasets in evaluation. As we can see in this table, the overall feature set size grows from $93,324$ to $294,019$ as the dataset size grows from $11,160$ to $41,306$. Note that app components, intent filters and permissions are the three sets that grow the most. The former two are defined manually by the developer so they tend to have different values. For permissions, even though there are a fixed number of system permissions developer can apply in manifest file, they can still declare self-defined permissions, e.g., `com.zing.znews.permission.C2D_MESSAGE`. So the permission set can also grow as the dataset grows. The remaining feature sets have a relevant fixed size because they are linked to the Android system APIs or smart phone hardware, which have a limited size.

4.1.2 Evaluation Tasks

We evaluate detection performance and run-time performance of our proposed system on two separate tasks, and compared them with several baseline algorithms as well as existing signature-based commercial anti-virus engines that available in the VirusTotal service. Specifically, we focus on the following three aspects:

- **Malware detection**: in this kind of experiments, we compare our trained factorization-machine-based system with some baseline machine-learning based detection algorithms. In addition, we also send all samples, including malware samples, to the VirusTotal service to compare with commercial anti-virus engines.

TABLE 4.2
DATASETS FOR DETECTION PERFORMANCE EVALUATION.

Dataset	# malware	# Clean files	Total	Feature size
DREBIN	5,560	5,600	11,160	93,324
AMD	24,553	16,753	41,306	294,019

- **Malware family identification**: in this kind of task, each sample will be sent into our system and our system will respond whether the input sample belongs to a specific malware family. Here we regard clean files as a special family named "clean".

- **Run-time**: to further evaluate efficiency of our proposed system, we analyze the run-time in terms of processing time and detection time. The processing time counts from the beginning to the phase of feature encoding, while the detection time counts on the classification phase.

For performance evaluation tasks, we evaluate the detection performance and family identification performance using the measures shown in Table 4.1, and we focus on the following four metrics: *precision*, *recall*, *F1* and *False Positive Rate* (FPR). Note that in the literature, recall and false positive rate correspond to malware detection rate and false alarm rate for the detection system.

Moreover, the dataset is split into training (80%) and testing (20%) sets in both experiments. All models are trained with 4-fold cross validation for hyper parameter tuning and then tested on the testing set for performance evaluation. We repeated this procedure 5 times and then averaged the results. The baseline algorithm and our proposed method are trained and tested in the same manner.

4.2 Detection Performance

In this subsection, we conduct two sets of experiments to show the performance of our proposed FM-based malware detection model as well as other baseline algorithms. Comparison with existing anti-virus engines are also included.

TABLE 4.3
SIZE OF THE EXTRACTED FEATURE SETS ON THE DREBIN AND AMD DATASETS.

Feature set	DREBIN	AMD
App components	79,523	250,612
Hardware features	86	143
Permissions	3,830	12,492
Intent filters	9,317	30,102
Restricted APIs	387	466
Suspicious APIs	42	43
Used permissions	139	161
Total feature	93,324	294,019

4.2.1 Comparison with Baseline Algorithms

We first evaluated our proposed FM-based method and compared it with other existing baseline algorithms, including SVM, which is used in DREBIN [6], classical machine learning algorithms such as Naive Bayes [5], and neural networks e.g., multi-layer perceptron [21] .

Table 4.4 shows the test result of different algorithms on the DREBIN set. As we can see, FM achieves the best performance for precision with a score of 99.91% and 0.09% false positive rate when the threshold is set to 0.5. The multilayer perceptron classifier (MLP) gives the same precision and false positive rate scores, but it gives even a better recall and the best $F1$ scores on this small DREBIN dataset. The SVM algorithm gives a recall score or detection rate of 92.35% with a 4.19% false positive rate close to the result given in DREBIN [6], which is 94% and 1% respectively. However, it is still not comparable with the result given by FM and MLP. Naive Bayes with three different kernels, Gaussian, Multinomial and Bernoulli, all have very high recall scores, but at the cost of high false positive rates and low precision, resulting in bad overall performance and low $F1$ scores.

ROC curves on the DREBIN test set are also shown in Fig. 4.1. Obviously, the FM and MLP algorithms give the best performance with an area under the curve (AUC) score of 1.0 under the accuracy of e^{-3}. the naive Bayes classifier with multinomial and Bernoulli kernel follow with AUC values of 0.998 and 0.997,

TABLE 4.4
TEST RESULT ON THE DREBIN SET WITH A THRESHOLD OF 0.5. ALL VALUES ARE MULTIPLIED BY 100.

Algorithm	FM	SVM	NB-G	NB-B	NB-M	MLP
Precision($\times 100$)	**99.91**	95.62	90.86	90.53	96.25	**99.91**
Recall($\times 100$)	99.01	92.35	99.37	**99.82**	99.28	99.64
F1($\times 100$)	99.46	93.96	94.93	94.95	97.74	**99.77**
FPR($\times 100$)	**0.09**	4.19	9.90	10.35	3.84	**0.09**

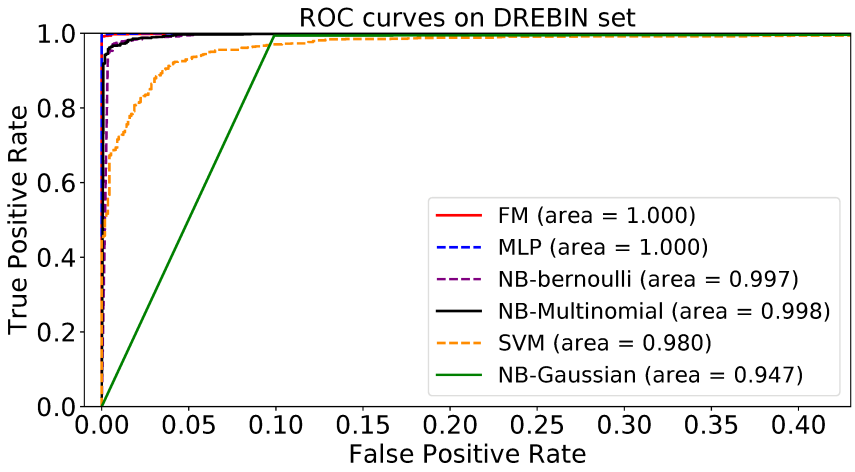

Fig. 4.1. ROC curves for all the baseline algorithms and our FM method on DREBIN test set.

respectively. Naive Bayes with the Gaussian kernel and SVM are the worst with a AUC scores of 0.947 and 0.98. For these two curves, the true positive rate grows slowly as the false positive grows. That is to say, a high true positive rate is at the cost of high false positive rate. Notice that in Table 4.4 SVM is better than Naive Bayes with Bernoulli kernel, with similar $F1$ score and much better false positive rate. But with the ROC curve we now can see by adjusting the threshold of Naive Bayes with Bernoulli kernel will always outperform SVM under the same false positive rate limitation.

The same experiment procedure was then repeated on the larger AMD data set. The result is shown in Table 4.5 from which we can see, on the large data set FM gives the best performance under all metrics with a recall score and FPR of 99.20% and 0.93%. MLP follows with a 99.16% and 1.52% recall score and FPR. Compared

with what shown using the DREBIN dataset, where MLP is slightly better than FM with a higher $F1$ score, one can see the FM's advantage in dealing with a highly sparse vector becoming more obvious as the feature space size or the sparsity of the vector representation grows, leading to a better performance than other algorithms including MLP. We can say with confidence that our FM method would outperform other algorithms with a larger margin on an even larger data set. The SVM method has similar results to what we got with the DREBIN set with a FPR of 4.24% and $F1$ score of 96.94%. The same with Naive Bayes with three different kernels, they still give high recall score at the cost of high false positive rate.

ROC curves on the AMD set are shown in Fig 4.2. Obviously, FM and MLP give the best performance with a 0.999 AUC value, and Naive Bayes with Multinomial and Bernoulli kernels follow with AUC values of 0.994 and 0.993 respectively, then SVM with 0.955.

The experiment results also support our claim that interaction terms are important for revealing malicious behavior patterns. SVM and Naive Bayes directly use the vector representation to learn the classifiers. On the other hand, FM achieved a much better performance by adding interaction terms. MLP as a universal approximator [24] can also output excellent results but more data will be needed to train the model, especially when the input vector is highly sparse. Even though we can not say, from current experiment result, that FM can definitely outperforms MLP, we can still see the trend that FM tends to beat MLP when dataset grows.

TABLE 4.5
TEST RESULT ON THE AMD SET WITH THRESHOLD 0.5. ALL NUMBERS ARE MULTIPLIED BY 100.

Algorithm	FM	SVM	NB-G	NB-B	NB-M	MLP
Precision($\times$100)	**99.35**	97.01	80.64	90.29	92.20	98.93
Recall($\times$100)	99.20	96.87	99.35	99.41	**99.54**	99.16
F1($\times$100)	**99.28**	96.94	89.02	94.63	95.73	99.05
FPR($\times$100)	**0.93**	4.24	33.88	15.19	11.97	1.52

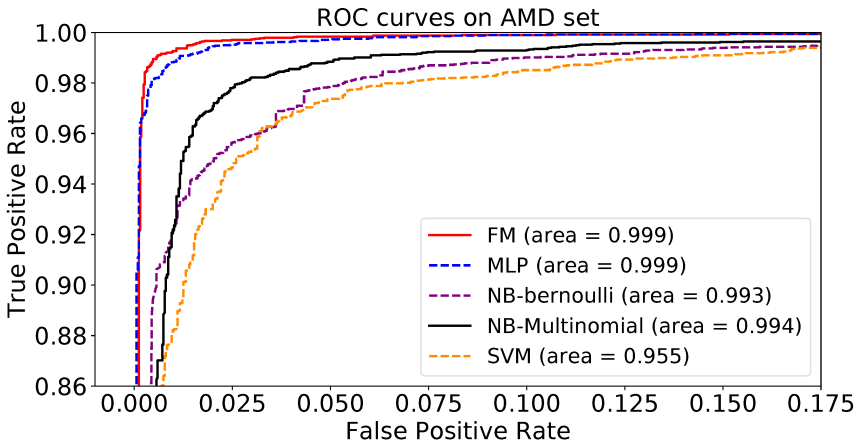

Fig. 4.2. Zoomed in upper-left part of the ROC curves for all the baseline algorithms and our FM method on AMD test set.

4.2.2 Comparison with AV engines

We also compared the performance of our malware detection algorithm with existing commercial Anti-Virus engines on VirusTotal [25]. The critical point to mention is that all of the *truly* clean files used in our experiments are actually labeled by these AV engines using the rule described in subsection 4.1.1. Therefore, AV engines are supposed to have a better false positive rate than their normal performance. Due to the page limit, we only list results of popular engines with best performance among all 78 AV engines from `VirusTotal`, such as `Kaspersky`, `Cylance` and `McAfee`.

Table 4.6 summarizes scanning results from commercial AV engines on the testing split of the DREBIN set. We can see that our method outperforms most of the AV engines with a precision of 99.91% and 0.09% FPR. And for those engines that have better recall or $F1$ score than what our method presented would often have either much worse precision or FPR, e.g. `Cylance`. Only several AV engines have a comparable overall results, e.g. `CAT-QuickHeal`, `Symantec`.

The results on the AMD dataset are shown in Table 4.7. This time, our FM model outperforms all of the AV engines with a $F1$ score of 99.28%. We also found that on this dataset most of the AV engines (include those that are not shown here) give worse recall scores compared with what they get on the DREBIN set, but still

have a good FPR. This is reasonable, because on the one hand, those AV engines all use signature-based malware detection method so they cannot discriminate against a piece of malware unless said malware has been seen before and recorded in the AV's database. To clarify, they can not detect newly emerged malware without a database update. On the other hand, as mentioned before, all malware in the DREBIN dataset was collected before 2014 and the AMD dataset was released only last year, containing a fresher stock of malware. So the detection rate for some AV engines could fall if they did not include the most recently malware in their data base.

TABLE 4.6
PERFORMANCE OF VIRUSTOTAL SCANNERS ON THE DREBIN TEST SET. ALL VALUES ARE MULTIPLIED BY 100.

Scanner	Precision%	Recall%	F1%	FPR%
FM	**99.91**	99.01	99.46	**0.09**
McAfee	**99.91**	98.74	99.32	**0.089**
CAT-QuickHeal	99.64	99.46	99.55	0.357
Symantec	**99.91**	99.28	**99.59**	**0.089**
Kaspersky	99.63	97.21	98.41	0.357
Cylance	50.09	**99.91**	66.73	98.66
Qihoo-360	97.78	94.96	96.35	2.141

TABLE 4.7
PERFORMANCE OF VIRUSTOTAL SCANNERS ON THE AMD TEST SET. ALL VALUES ARE MULTIPLED BY 100.

Scanner	Precision%	Recall%	F1%	FPR%
FM	99.35	99.20	**99.28**	0.93
McAfee	99.73	93.82	96.69	0.358
CAT-QuickHeal	99.70	98.84	99.27	0.418
Symantec	99.57	67.26	80.29	0.42
Kaspersky	**99.84**	53.35	69.54	**0.119**
Cylance	58.86	**99.64**	74.00	98.96
Qihoo-360	97.50	68.92	80.76	2.507

4.3 Malware Family Detection

Another important task for Android malware detection is malware family classification. To evaluate our model on this task, we built another two data sets. Details about those two data sets are shown in the first column of Table 4.8 and Table 4.9 respectively. All samples from the 6 largest malware families in the DREBIN set and another $1,500$ clean files form the first data set. The second dataset contains all samples from the 7 largest malware families in the AMD dataset and also $1,500$ clean applications. To shown our model's capacity to distinguish one malware family from other families as well as clean files, each time we label all the samples from one family as "True" and samples from all the other families as "False" then shuffle and randomly split the dataset into training (80%), to train a new model, and test set (20%) to evaluate the model. Notice that if the "clean" family is labeled as 'True' this is then actually a malware detection task.

Table 4.8 gives the experimental results of malware family detection on the DREBIN set. As is shown, our model can achieve a weighted average detection accuracy of 99.04% with an average false positive rate of 0.187%. In particular, all families show a recall score above 0.93, precision score above 0.98, $F1$ score above 0.96 and false-positive rate below 0.4%. For family `Plankton` our model produces a perfect result with an $F1$ score of 1.0 and FPR of 0.

The results on the AMD set are shown in Table 4.9. On this dataset, our method managed an average detection rate of 98.94% with a 0.095% false positive rate. All families show a recall score above 96%, precision above 96.7%, $F1$ score above 96.48% and false positive below 0.25%. We can tell that the overall performance on the AMD set is slightly better than what was produced with the DREBIN set. This may because the samples for each family on the AMD set are much larger than we have on DREBIN set.

In summary, the FM method achieves great performance for malware family classification and can be used to predict the family of a piece of malware if enough training samples are provided.

TABLE 4.8
FAMILY CLASSIFICATION RESULTS ON THE DREBIN DATASET.

Family	Samples	Precision%	Recall%	$F1\%$	FPR%
FakeInstaller	925	99.52	99.52	99.52	0.126
DroidKungFu	666	100.0	98.57	99.28	0.00
Plankton	625	100.0	100.0	100.0	0.00
Opfake	613	98.43	98.43	98.43	0.229
GinMaster	339	98.70	98.70	98.70	0.108
BaseBridge	330	100.0	93.90	96.86	0.00
Clean	1500	98.99	100.0	99.49	0.426
Average	—	99.33	99.04	99.17	0.187

TABLE 4.9
FAMILY CLASSIFICATION RESULTS ON THE AMD DATASET.

Family	Samples	Precision%	Recall%	$F1\%$	FPR%
Airpush	7606	99.73	99.54	99.64	0.162
Youmi	1256	99.60	97.66	98.62	0.027
Mecor	1762	100.0	100.0	100.0	0.00
FakeInstaller	2129	99.77	100.0	99.89	0.028
Fusob	1238	100.0	100.0	100.0	0.00
Kuguo	1122	100.0	96.73	98.34	0.00
Dowgin	3298	99.69	98.48	99.08	0.060
Clean	1500	96.97	96.00	96.48	0.244
Average	—	99.57	98.94	99.25	0.095

4.4 Processing Time Evaluation

In this section, we evaluate the processing time efficiency of our malware detection model. As is shown in previous sections, our system consists of four parts. Normally, the last two phases, encoding and classification, take much less time than feature extraction and decompiling. Also, for different applications these two phases would often take a fixed processing time due to the fixed feature space size. Therefore, we focused on evaluating the processing time for unpacking, decompiling and feature extraction, then give out an average processing time for all applications on the encoding and prediction phase.

The evaluation was done on a virtual machine hosted on ESXi. The VM is running Ubuntu 16.04 with a memory of 4G and 2 CPUs. We randomly sampled $3,794$ AMD samples, $6,120$ clean files and all the $5,560$ DREBIN samples to test this experiment. The results are shown in Fig 4.5; the three figures in the first row show the relation between dex source code size and processing time. The figures in the second row show the relation between `apk` file and processing time. We can tell from the figure that the processing time and dex code size almost have a linear relation and for samples in those three datasets the slops are approximately similar to 0.4. There is no fixed relation between `apk` file size and processing time and this is because apart from the dex code and manifest file, an `apk` also contains other resource files like HTML, figures, etc. In some applications these files may take a lot of space, for example games, while for others this is not the case. The histogram of processing time and dex code size of all $15,474$ samples are shown in Fig 4.3 and Fig 4.4 respectively. We can see, over 78% of samples have a dex code size of less than $3MB$ and over 70.6% samples have a processing time of less than 5 seconds. On the same samples we also measured the mean time for encoding and prediction. The former took our system an average 4.7ms and 0.021ms for the latter.

Compared with DREBIN [6], it seems that our system does not have much of an advantage in processing time. However, this is not the case. To begin with, the test is done on a system that is not fully integrated, the output of Smalisca is first

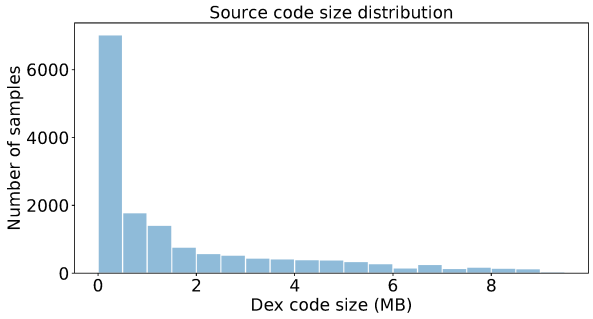

Fig. 4.3. Dex source code size distribution for the $15,474$ samples from AMD $(3,794)$, DREBIN $(5,560)$ and real-world apps $(6,120)$ from online app stores.

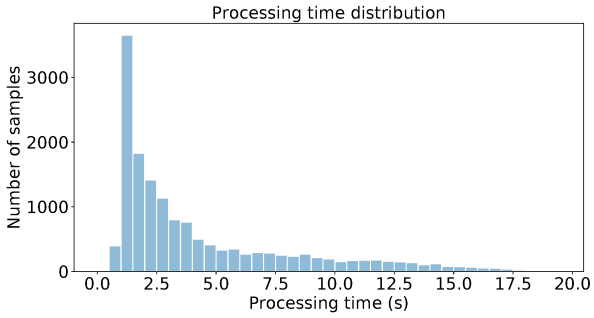

Fig. 4.4. Processing time distribution for the $15,474$ samples from AMD $(3,794)$, DREBIN $(5,560)$ and real-world apps $(6,120)$ from online app stores.

written into a json file and then reloaded into RAM for further processing. The I/O between RAM and flash storage would often take a long time. Second, the feature sets used in our system are simpler and smaller than sets used in DREBIN, so under the same conditions our system should take less processing time than DREBIN.

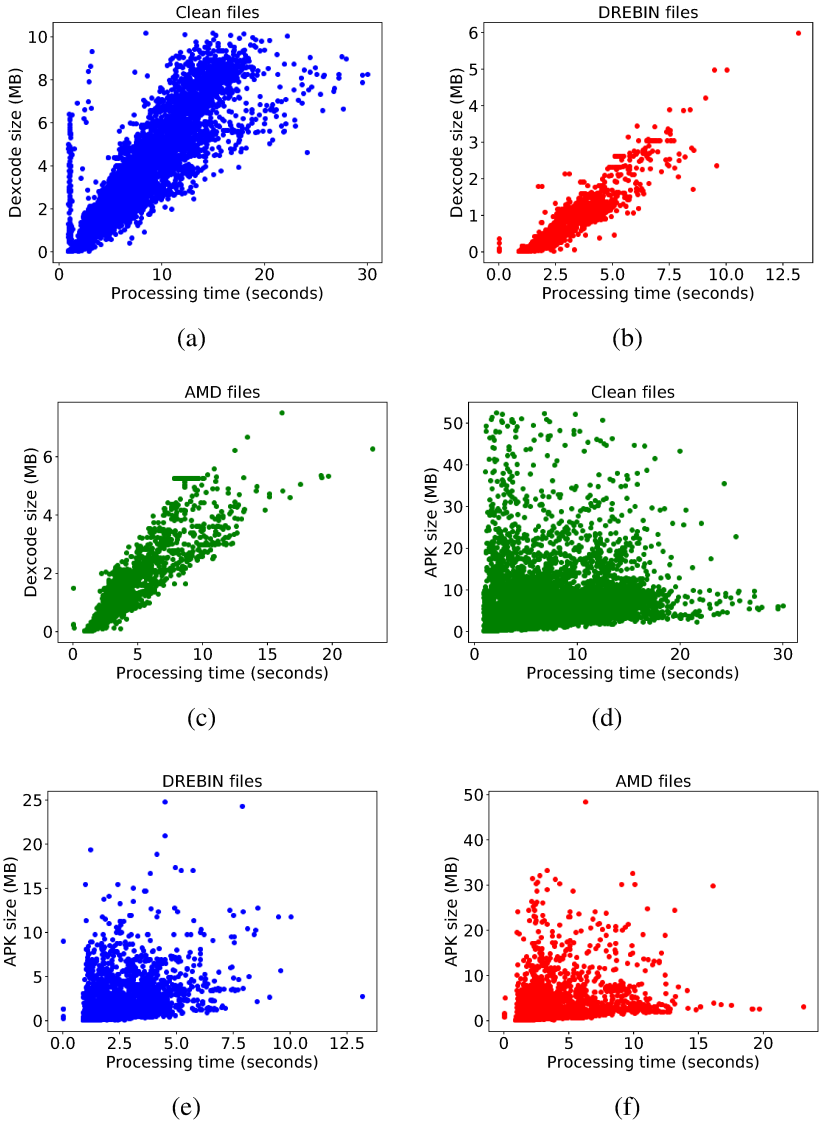

Fig. 4.5. Scatter plot of processing time vs. file size. Figures in the first row illustrate how long it takes in terms of dexcode file size, and those in the second row illustrate that of APK file size. Three columns refer to clean files, malware samples in the DREBIN dataset, and malware samples in the AMD dataset, respectively.

Chapter 5

Related Work

Static analysis of Android applications focuses on analyzing internal components of an application, it is able to explore all possible execution paths in malware samples and has long been used for detecting of malicious behaviors and application vulnerabilities. This analysis typically based on source code or binary analysis to search for malicious patterns.

Some works focus on the detection of specific malicious behavior such as privacy breach and over privilege. For example, [26], [27] goes through source code with a predefined source and sinks to find a potential privacy breach. [28] further examines all the URL addresses to see if the app is trying to steal users' private information. Stowaway [29] detects over privilege in Android applications by comparing the maximum set of permissions needed for an app with the actual request permissions. [30] uses data flow analysis for security certification. However, static taint-analysis and over privilege are prune to false positive.

Other works [5], [6], [22], [23], [31] try to directly classify an application as malicious or benign through permissions requests analysis for application installation (e.g. [32]–[34]), control flow (e.g. [35], [36]), or signature-based detection (e.g. [37], [38]). These works take different approaches in both the feature extraction and the classification phase. Peiravian and Zhu [23] used permission and API calls as features and SVM, decision trees and ensemble as classifiers. [22] tried random forest, SVM, LASSO and ridge regression on features based on system

calls. Hindroid [31] built a structured heterogeneous information network (HIN) with Android application and related system APIs as nodes and their rich relationships as links, and then used meta-path for malware detection. DREBIN [6], which extracted features from manifest files and source code, including permissions, hardware, system API calls and even all the URLs, and then uses SVM as the final classifier for malware detection. Differentiating ourselves from existing work and instead of only focusing on feature engineering and ignoring the importance of choosing a suitable algorithm. After acquiring the feature representations of apps, we first make two observations. Then the optimum machine learning algorithm that handles our problem the best is chosen for malware detection according to the observations.

Lots of recent works are trying to find malicious behavior patterns through control flow graphs or call graphs. AppContext [7] classifies applications using machine learning based on the contexts that trigger security-sensitive behaviors. It builds a call graph from an application source code and extracts the context factors through information flow analysis. It is then able to obtain the features for the machine learning algorithms from the extracted context. In this paper, 633 benign applications from the Google Play store and 202 malicious samples were analyzed. AppContext correctly identifies 192 of the malware applications with an 87.7% accuracy. Gascon et al. [8] also utilized call graphs to detect malware. After extraction of call graphs from Android applications, a linear-time graph kernel is applied in order to map call graphs to features. These features are given as input to SVMs to distinguish between benign and malicious applications. They conducted experiments on 135,792 benign and 12,158 malware applications, detecting 89% of the malware with 1% of false positives. This kind of method relies heavily on the accuracy of call graph extraction. However, current works like FlowDroid [9] and IC3 [10], [39] cannot fully solve the construction of Inter-component control flow graphs (ICFG), especially the inter-component links with intents and intent filters.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this project, we point out the importance of considering interaction terms across features for the discovery of malicious behavior patterns in an Android app. The features used to represent an application are app components, hardware features, permissions, intent filters from the manifest file and restricted APIs, suspicious APIs and used permissions from source code. Based on the extracted features, a highly sparse vector representation was constructed for each application using one-hot encoding. We then propose a factorization-machine-based malware detection system to handle the high sparsity of vector representation and model interaction terms at the same time. To the best of our knowledge, this is the first approach that uses FM models for malware detection. A comprehensive experimental study on real sample malware collections, DREBIN and AMD datasets, and clean applications collected from online app stores were performed to show the effectiveness of our system on malware detection and malware family identification tasks. Promising experimental results demonstrate that our method outperforms existing state-of-art Android malware detection techniques as well as most of the commercial antivirus engines on VirusTotal.

6.2 Limitations of Our Current Work

Our system takes advantage of machine learning to recognize malicious behavior patterns for malware detection. While machine learning techniques provide a powerful tool for automatically inferring models, they require a representative dataset for training. That is, the quality of the detection model depends on the availability and quality of both malware and benign applications. While it is straightforward to collect benign applications, gathering recent malware samples is not that easy and requires some technical effort. Fortunately, offline analysis methods, e.g. RiskRanker [38], can help to acquire malware and provide the samples for updating and maintaining a representative dataset in order to continuously update our model.

One limitation for our system is processing time. We plan to integrate our system into Wedge networks' in-line, real-time security solution which only allows us to have millisecond-scale processing time. For encoding and prediction our system takes about 4.8ms, however, decompiling and feature extraction takes on the order of seconds. Fortunately, we still have space to improve our system's time efficiency such as reducing I/O and finishing all work at once in main memory (RAM), or even using Application-specific integrated circuits (ASIC), such as FPGAs, for speed up. In addition, we note that decompiling apk files can fail when using some existing tools. In our experiments, we observed such failures for some files, and we found that malware samples are more likely to fail in decompiling. This is in our expectation as malware samples may use some additional techniques like code obfuscation that may lead to decompiling failures, which limits Android malware detection.

6.3 Future Work

Our future work will focus on the aforementioned limitations of our current system. To begin with, we will try to get more real world malware examples as well as clear files by either cooperating with tech companies or implement a sandbox ourself. Then, we also will try to improve the time efficiency of our system by implementing

specialized tools such as reverse engineering tools. New features are also important for discovering new malicious patterns in new emerging malware. So we need to pay attention to malware trends and keep our system up to date.

References

[1] Newshub. (April, 2013) Smartphones now outsell 'dumb' phones. [Online]. Available: http://www.newshub.co.nz/technology/smartphones-now-outsell-dumb-phones-2013042912

[2] statista. (2017) Global mobile os market share in sales to end users from 1st quarter 2009 to 2nd quarter 2017. [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[3] R. Unuchek, "Mobile malware evolution 2016," 2017. [Online]. Available: https://securelist.com/mobile-malware-evolution-2016/77681/

[4] D. Venugopal and G. Hu, "Efficient signature based malware detection on mobile devices," *Mobile Information Systems*, vol. 4, no. 1, pp. 33–49, 2008.

[5] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012, pp. 62–69.

[6] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[7] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *Soft-*

ware engineering (ICSE), 2015 IEEE/ACM 37th IEEE international conference on, vol. 1. IEEE, 2015, pp. 303–313.

[8] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. ACM, 2013, pp. 45–54.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[10] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 77–88.

[11] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*. Bonn, Germany: Springer, 2017, pp. 252–276.

[12] S. Rendle, "Factorization machines," in *Data Mining (ICDM), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 995–1000.

[13] Connor Tumbleson, Ryszard WiÅŻniewski, "Apktool," 2018, [Online; accessed 9-May-2018]. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[14] Ben Gruver et al., "Baksmali," 2018, [Online; accessed 9-May-2018]. [Online]. Available: https://github.com/JesusFreke/smali

[15] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[16] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *Intelligence and security informatics conference (eisic), 2012 european*. IEEE, 2012, pp. 141–147.

[17] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: deep learning in android malware detection," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 371–372.

[18] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[19] Apkpure.com, "apkpure," 2018, [Online; accessed 9-May-2018]. [Online]. Available: https://apkpure.com/

[20] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 396–409.

[21] D. W. Ruck, S. K. Rogers, M. Kabrisky, M. E. Oxley, and B. W. Suter, "The multilayer perceptron as an approximation to a bayes optimal discriminant function," *IEEE Transactions on Neural Networks*, vol. 1, no. 4, pp. 296–298, 1990.

[22] M. Dimjašević, S. Atzeni, I. Ugrina, and Z. Rakamaric, "Evaluation of android malware detection based on system calls," in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*. ACM, 2016, pp. 1–8.

[23] N. Peiravian and X. Zhu, "Machine learning for android malware detection using permission and api calls," in *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*. IEEE, 2013, pp. 300–305.

[24] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[25] "Virustotal," [Online; accessed 9-May-2018]. [Online]. Available: https://www.virustotal.com/#/home/upload

[26] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "Scandal: Static analyzer for detecting privacy leaks in android applications," *MoST*, vol. 12, 2012.

[27] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *International Conference on Trust and Trustworthy Computing*. Springer, 2012, pp. 291–307.

[28] H. Fu, Z. Zheng, S. Bose, M. Bishop, and P. Mohapatra, "Leaksemantic: Identifying abnormal sensitive network transmissions in mobile applications," in *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*. IEEE, 2017, pp. 1–9.

[29] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 627–638.

[30] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android," Tech. Rep., 2009.

[31] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Hindroid: An intelligent android malware detection system based on structured heterogeneous information network," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 1507–1515.

[32] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.

[33] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the eighth symposium on usable privacy and security*. ACM, 2012, p. 3.

[34] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1025–1035.

[35] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. Van Horn, "Sound and precise malware analysis for android via pushdown reachability and entry-point saturation," in *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*. ACM, 2013, pp. 21–32.

[36] S. Liang, W. Sun, and M. Might, "Fast flow analysis with godel hashes," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 225–234.

[37] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 576–587.

[38] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 281–294.

[39] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon, "Combining static analysis with probabilistic models to enable

market-scale android inter-component analysis," in *ACM SIGPLAN Notices*, vol. 51, no. 1. ACM, 2016, pp. 469–484.