Fest: A Feature Extraction and Selection Tool for Android Malware Detection

Kai Zhao*, Dafang Zhang*‡, Xin Su*, Wenjia Li†
*College of Computer Science and Electronics Engineering, Hunan University, Changsha, China
Email: {kzhao, dfzhang, suxin}@hnu.edu.cn
†Department of Computer Sciences, New York Institute of Technology, New York, NY, USA
Email: wli20@nyit.edu

***Abstract*—Android has become one of the most popular mobile operating systems because of numerous applications (apps) it provides. However, Android malware downloaded from third-party markets threatens users' privacy, and most of them remain undetected because of the lack of efficient and accurate detecting techniques. Prior efforts on Android malware detection attempted to build precise classification models by manually choosing features, and few of them has used any feature selection algorithms to help pick typical features. In this paper, we present *Feature Extraction and Selection Tool* (FEST), a feature-based machine learning approach for malware detection. We first implement a feature extraction tool, *AppExtractor*, which is designed to extract features, such as permissions or APIs, according to the predefined rules. Then we propose a feature selection algorithm, *FrequenSel*. Unlike existing selection algorithms which pick features by calculating their importance, *FrequenSel* selects features by finding the difference their frequencies between malware and benign apps, because features which are frequently used in malware and rarely used in benign apps are more important to distinguish malware from benign apps. In experiments, we evaluate our approach with 7972 apps, and the results show that FEST gets nearly 98% accuracy and recall, with only 2% false alarms. Moreover, FEST only takes 6.5s to analyze an app on a common PC, which is very time-efficient for malware detection in Android markets.**

***Index Terms*—malware, privacy, machine learning, feature selection**

I. INTRODUCTION

According to F-Secure's research [1] in 2014 Q1, 99% of malware was distributed on the Android platform , and this rate was 79% in 2012. On the other hand, as the main sources of the Android apps, third-party markets contain a lot of cracked or tampered apps, and there is no sign whether an app has been checked for security risks or not. The lack of security inspection on mobile apps intensifies the spreading of malware. Zhou et al. [2] had some Android malware checked by popular security softwares, and the precisions were only between 20.2% and 79.6%, indicating an urgent demand on malware accurate detection for Android app markets.

In recent years, there have been several efforts on malware detection by using various machine learning algorithms. Drebin [3] extracts permissions, APIs and IP address as features and uses the Support Vector Machine (SVM) algorithm to learn a classifier from the existing ground truth datasets, which can be used to detect unknown malware. DroidMat [4] alternatively uses permissions and intents mining with KNN (K-Nearest Neighbor) to perform malware detection. In addition, DoridAPIMiner [5] focuses on providing several lightweight classifiers based on the API level features. However, none of them was aware of the importance of feature selection, which had led to some major limitations in their works. First, the recall value of DroidMat is 10% lower than its precision value, which means that some malware cannot be correctly detected. Second, the high accuracy of DroidAPIMiner benefits from correctly classifying more benign apps than malware. Moreover, It takes a large amount of time for Drebin to build the classifier because of the high-dimensional feature vector. In summary, feature selection remains an unsolved problem on keeping the balance of efficiency and effectiveness in malware detection.

In this paper, we present FEST, which aims at detecting malware with both of high efficiency and accuracy. Previous works are inefficient due to lack of feature selection, which also results in the imbalance between accuracy and recall, and time overhead in building classifiers. To solve these problems, FEST focuses on two directions: (1) we extract all the features which indicate the functions and behaviors in apps by *AppExtractor*, and (2) we propose an algorithm *FrequenSel* to select typical features which help distinguish malware from benign apps.

In summary, the observations and contributions of this work are listed as follows:

- We design a scalable, policy-driven and automatic feature extraction tool *AppExtractor*, by which we can decompile apk format packages and extract features according to the policies written in regular expression.
- We propose a novel feature selection algorithm *FrequenSel*, which selects typical feature from all the available features. And it only takes several seconds to perform feature selection, which is much faster than the existing algorithms.
- In experiments, we evaluate FEST with a real-world dataset which contains 7972 apps. We automatically get 398 typical features by *FrequenSel* for building classifiers. The FEST can achieve 98% accuracy and 97.8% recall. Moreover, FEST analyzes an app in 6.5s on average, making it easy to handle 1000 apps in 1.8h.

‡ Dafang Zhang is corresponding author, email: dfzhang@hnu.edu.cn.

978-1-4673-7194-0/15/$31.00 ©2015 IEEE

The rest of this paper is organized as follows: Section II introduces the overall architecture of our approach. Section III describes the features extracted by *AppExtractor* and the details of *FrequenSel*. In Section IV, we evaluate FEST. We discuss some related works in Section V, and conclude in Section VI.

II. Approach Overview

In this section we first introduce the overall architecture of FEST, and then describe each module individually to explain how FEST works for malware detection.

A. Architecture Of Fest

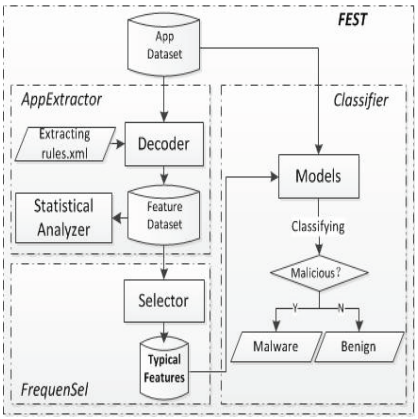

Fig. 1. The overview architecture of Fest

As shown in Figure 1, FEST contains three main components: *AppExtractor* is responsible for extracting all features, *FrequenSel* is used to select typical features, and *Classifier* focuses on building models with machine learning algorithms. The Android app dataset contains both malware and benign apps, and *AppExtractor* batches decoding apps to decrypted files, extracts origin features from these files into the feature dataset. Then the selector implements *FrequenSel*, traverses the entire feature dateset and selects the typical ones. Finally, we use machine learning algorithms to build models and evaluate them on the Android app dataset by classifying them into malware or benign apps.

B. AppExtractor

Android apps are packed into apk format, and the features we are interested in are encrypted, such as permissions, APIs, actions, IP and URLs. To extract these features, we implement the decoder based on an open source recompilation tool [6], which unpacks apps to readable xml or smali files. However, apps often contain lots of APIs, and we only need the features corresponding to their main functions. Therefore, we define several customized extraction rules(also policies) in xml files. For instance, we focus on the APIs provided by Android framework, so we can define a rule like this:

Listing 1. A part of extracting rule.xml

```
<rule>
   <id>5</id>
   <category>FrameworkAPI</category>
   <description>Extract APIs of Framework</description>
   <regex>Landroid(/\w+)+;->\w+|Landroid(/\w+)+\$\w+</regex>
   <targetfile>smali</targetfile>
   <multiMatch>true</multiMatch>
</rule>
```

In Listing 1, the value of regex is defined in regular expression. According to this rule, the decoder can extract APIs such as *android.telephony.TelephonyManager.getSimSerialNumber* from smali code. Features such as permissions or actions can be extracted by other similar rules. The node multiMatch indicates whether we want the regex to be matched more than once or not.

The statistical analyzer is an important module in *AppExtractor*. With decoder, we collect more than 32,000 features, before feature selection, we use the analyzer to give general reports about the feature dataset according to our demands. For example, we are often interested in what the most common feature is or which kinds of features are used more frequently in malware.

C. FrequenSel

The *FrequenSel* is an algorithm which is able to select typical features. Most of features extracted by *AppExtractor* are worthless for classification purpose, and only few of them can represent the functions and behaviours of apps, which are named as typical features. In FEST, selection algorithm is implemented in the selector, which takes the feature dateset as input, and runs *FrequenSel* to decide whether a feature satisfies with the specific conditions or not. These conditions require that the frequency and coverage of a feature should exceed several predefined thresholds (see Section III). Actually, we only use hundreds of features in our experiments, which is much less than over 545,000 features used in Drebin [3].

D. Classifier

This component is responsible for malware classification. Machine learning is widely used in many fields, such as text categorization or image identification. In this paper, machine learning algorithms are used for malware classification, because most of them process data with numerical vectors, we need to map our typical features into a joint *feature vector*. To address this, two definitions are introduced as follows:

$$F = \{f_1, f_2, ..., f_n\}, F^a = \{f_1^a, f_2^a, ..., f_m^a\} \tag{1}$$

first, all of selected features from Android apps are contained in F, for each given app a, it can be defined with the features it contains. As shown in Equation 1, n is the size of the feature set and m is the number of different features in a. Then we define feature vector of an Android app as V in Equation 2.

$$V = \{v_1, v_2, ..., v_i\}, v_i = \begin{cases} 1 & f_i \in F \text{ and } f_i^a \in F^a, 1 \le i \le n \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Thus a feature vector can be translated into $V = \{0, 1, 0, 0, 1, 0, 1, ...\}$, 1 indicates that the feature is contained in this app, whereas 0 indicates not. However, V is often a sparse vector, in order to reduce the storage overhead, we

transform V to a compressed format V^*. Assuming that the features are arranged in a fixed order, then we can index a feature by its position, and V^* is defined as follow:

$$V^* = \{1, 4, 6, ...\} \quad (3)$$

in Equation 3, the positions of non-zero elements in V are stored in V^*, which saves a great amount of memory space when modeling. Unlike Drebin's high-dimensional vector[1], V only contains several hundreds typical features benefiting from *FrequenSel*. According to our experiments, algorithm like SVM used in Drebin takes more than one hour to build a model on such a high-dimensional vectors, and after we perform feature selection by *FrequenSel*, it only takes less than one minute.

In FEST, several machine learning algorithms have been tested to find the most suitable ones for malware detection (see details in Section IV), and we find that the SVM can achieve the highest accuracy and the best stabilities regardless of the number of features. Moreover, the KNN is much faster than SVM with only 1%-2% decrease of accuracy. So SVM and KNN are both suitable algorithms: if the accuracy is more prioritized than the efficiency in some cases, SVM is better, and vice versa. It is noteworthy that we often build models once and use them for classification in a relatively long time, so the cost of model building is generally not a major concern.

III. Feature Extraction and Selection

In this section, we explain the two approaches of Android malware detection in our work, namely feature extraction implemented in *AppExtractor* and feature selection implemented in *FrequenSel*.

A. Feature Extraction

In feature extraction, we use the decoder to dissect Android apps, and extract all the features which correspond to their functions and behaviors. The extracted features can be divided into four different categories.

1) Permission Features: Android provides an install-time permission system to control the access to privacy and security relevant APIs. Normally, apps must explicitly declare what permissions they will use in the *AndroidManifest.xml* file based on the functions they provide. For example, according to the results of analyzer, we find that more than 93% of *SMS* permissions, including *p.SEND_SMS*, *p.RECEIVE_SMS*, *p.READ_SMS* and *p.WRITE_SMS* are requested by malware, indicating that malware uses these permissions more frequently than benign apps, because malware often consumes money by sending text messages and steals users' private information. In addition, we observe that *p.INSTALL_PACKAGES* and *p.PROCESS_OUTGOING_CALLS* are much more frequently declared in malware, which indicates that malware tends to install unknown apps and monitors phone calls at the background. Therefore, permission features can be used to classify malware from benign apps.

[1]The concept of sparse vector is mentioned, but the total number of different features remains too large when they build models.

2) API Features: In [7], the authors conclude that 1/3 of Android apps are over-privileged, and the results of [8] show that several privileged permissions are exploited to malware, thus they do not need to request these leaked permissions for actual usage. The two works indicate that permission is not enough for detecting malware. Therefore, we consider APIs which can actually reflect the behaviors of apps to help identify sensitive operations. For example, we find more than 87% of *android.provider.Browser.getAllBookmarks* and 97% of *android.content.pm.PackageManager* are requested by malware, showing that malware often steals private information about browser bookmarks or installed apps in user's devices. Obviously, we should focus on these kinds of features when we try to detect malware.

3) Action Features: Android apps can register actions they are concerned about in *AndroidManifest.xml* or code, and this mechanism allows malware monitors some system events. We find *a.SIG_STR* and *a.BATTERY_CHANGED_ACTION* are rarely used in benign apps. However, some malware may use these kinds of actions to hide themselves to be detected. For example, malware will suspend their malicious process when the battery runs out sharply. This kind of features may also be suitable for distinguishing malware and benign apps.

4) IP And URL Features: Malware regularly establishes network connections to retrieve commands or exfiltrate data collected from devices. In our apps dataset, we find that more than 93% of malware would request permissions to access the Internet. Therefore, all IP addresses, hostnames and URLs found in the disassembled code are included in the last kind of feature. Some of these addresses might be involved in botnets, for example, a well-known malware, named *Basebridge* [9] is known to secretly send privacy data (e.g., the IMEI number) to *http://b3.8866.org:8080*.

B. Feature Selection

With *AppExtractor*, we obtain more than 32,000 features. However, some features (e.g., p.DOWNLOAD_WITHOUT_NOTIFICATION) are only used by very few apps in our Android app dataset, which are not common enough, some other features (e.g., p.WRITE_EXTERNAL_STORAGE) are widely used by both malware and benign apps, which cannot be used to distinguish malware and benign apps. Moreover, these features consist a high-dimensional feature vector which may cause very complicated computation and cause low efficiency in building detection models. Therefore, feature selection algorithm is necessary before modeling.

In this subsection, we first describe two well-known feature selection algorithms, *Chi-Square* and *Information Gain*. Then, we present two phenomena based on observations on the results of the two algorithms. Finally, we propose our selection algorithm *FrequenSel* which is quite suitable for the feature dataset generated by *AppExtractor*.

1) Chi-Square and Information Gain: Chi-square derived from statistics is a method to test the independence of two variables. In feature selection, chi-square tests whether a feature f is relevant to a category c (c=0, 1, and 0 is

benign, 1 is malicious). Thus, the importance of f is directly proportional to chi-square value χ^2, and the corresponding formula is shown in Equation 4.

$$\chi^2(f,c) = \frac{N \times (AD - CB)^2}{(A+C) \times (B+D) \times (A+B) \times (C+D)} \quad (4)$$

where $N = A+B+C+D$, A and B are the number of malware and benign apps with f, C and D are the number of malware and benign apps without f. For each feature we calculate its χ^2, and sort all the features on χ^2 in descending order. Finally, the features on the top are the best for malware detection.

While information gain is a concept of Information Theory. In feature selection, information gain considers the amount of information, $IG(f)$, brought by feature f. Its formula is shown in Equation 5.

$$IG(f) = H(c) - H(c|f), H(x) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \quad (5)$$

In a classification system, we calculate the total information of the system with and without f, $H(c)$ and $H(c|f)$[2] for each feature f. Then, we subtract the two values, and the difference $IG(f)$ is regarded as information gain. Like chi-square, we sort features on $IG(f)$ in descending order, and select the top features for machine learning algorithms.

2) Distribution Bias And Long Tail Effect: We observe that the two feature selection algorithms still have two problems on our feature dataset, which are described as follows.

- *Distribution Bias*: we find that 93.6% of 32,247 features have been used in benign apps and only 35% of them are used in malware, and we call this phenomena as *Distribution Bias*. For example, one of these features named *android.view.ViewGroup.addView* is ranked in the second place of all features in results of the two algorithms, but this feature is used for user interface design (we call this as UI-related feature, which usually contains the keywords such as view, activity and layout, etc) and cannot be used to detect malware. We observe that these kinds of features are frequently used by apps, which make the two algorithms overestimate their importance. However, these features are not helpful to distinguish malware and benign apps. In other words, these algorithms are not suitable for our feature dataset and consequently, we need an algorithm which can select features more frequently used in malware.
- *Long Tail Effect*: Based on our experiments, we find that 75.56% of the features get χ^2 less than 10.0, 88.25% of the features obtain $IG(f)$ under 0.01. Therefore, most of features distributed on the tail are unimportant, and they contribute little to machine learning algorithms. Moreover, the two algorithms only sort all features without indicating which of them should be selected, we have to manually cut off the tail and select the rest. Because of the number of features will significantly influence the accuracy of classification, we expect the feature selection algorithm to be more clever, not only telling the importance of features, but also picking out typical features.

[2] $H(c|f)$ is conditional entropy, which is equal to the system entropy without f.

3) FrequenSel Algorithm: To address the two problems, we propose a novel feature selection algorithm named *FrequenSel*, which is effective in malware detection scenarios. In *FrequenSel*, we assume that T_m and T_b are the number of malware m and benign apps b, then we examine two conditions:

Cond. 1 $r_c \geq \alpha_c$? $r_c = N_c/(N_m + N_b)$, $c \in \{m, b\}$.
Cond. 2 $N_c/T_c \geq \beta_c$?

To better address them, we discuss with $c = m$, $c = b$ is similar, so N_m is the number of malware that contains a certain feature, N_b for benign apps contain the certain feature, α_m and β_m are thresholds, $0.5 \leq \alpha_m \leq 1$ and $0 \leq \beta_m \leq 1$, respectively. Then *Cond.* 1 implies that a feature is used more frequently in malware than benign apps. *Cond.* 2 means that the occurrence times of a feature in all malware exceeds threshold β_m. A feature should be selected as a typical feature once it satisfies the two conditions. In this way, the typical features we collect are not only frequently used by malware, but also have a certain coverage in the feature dataset, *Distribution Bias* and *Long Tail Effect* are well solved.

Algorithm 1 FrequenSel

Input: FrequenSel($F, \alpha_m, \beta_m, \alpha_b, \beta_b$);
Output: F'

```
1:  for i → 1 to F.size() do
2:      N_m ← countInMalware(f_i);
3:      N_b ← countInBenign(f_i);
4:      r_m ← N_m/(N_m + N_b);
5:      if r_m ≥ α_m&&N_m/T_m ≥ β_m then
6:          F' ← f_i;
7:      elseF' ↚ f_i;
8:      end if
9:      r_b ← N_b/(N_m + N_b);
10:     if r_b ≥ α_b&&N_b/T_b ≥ β_b then
11:         F' ← f_i;
12:     elseF' ↚ f_i;
13:     end if
14: end for
15: return(F');
```

Algorithm 1 shows the details. The input has five parameters: F is the original feature set, and $\alpha_m, \beta_m, \alpha_b, \beta_b$ are thresholds from the two conditions. F' is the output containing typical features, the size of F' is determined by the combination of parameters and not manually defined, generally $F' \ll F$. In *FrequenSel*, for each feature f_i in F, N_m, N_b are the number of malware and benign apps which contain f_i, respectively. r_m, r_b are percentages of N_m, N_b for malware and benign apps. If the feature satisfies the two conditions, it will be added into F'. Importantly, features in F' are selected from two ways: the first way focuses on more frequent features in malware (code from line 1 to line 8), and the other one focuses on benign apps (the rest code). The main reason of this selection strategy is that we attempt to keep the balance of recall in malware and benign apps, if we only use features that are used more frequently in malware, the recall of correctly

classified benign apps will be relatively lower than malware according to our experiments.

We combine different values of $\alpha_m, \beta_m, \alpha_b$ and β_b to select five most typical features, Table I shows the results, *p.SEND_SMS* short for *android.permission.SEND_SMS* and *a.SIG_STR* short for *android.intent.action.SIG_STR*.

TABLE I
TOP 5 FEATURES BY FREQUENSEL

Top5 Permissions	Top5 APIs	Top5 Actions
p.SEND_SMS	SmsManager.getDefault	a.SIG_STR
p.READ_SMS	SmsManager.sendTextMessage	a.BATTERY_CHANGED_ACTION
p.RECEIVE_SMS	WifiManager.setWifiEnabled	a.USER_PRESENT
p.WRITE_SMS	SmsMessage.createFromPdu	a.PHONE_STATE
p.INSTALL_PACKAGES	ComponentName.getShortClassName	a.DATA_SMS_RECEIVED

As we have mentioned in Section II, malware usually prefers message-related features, they sniffer the packages installed on user devices, leak information and monitor system events and privacy for their malicious motivation. Importantly, features like *p.INTERNET* are not selected as typical features, because their usages have no significant difference in malware and benign apps. Therefor, some features are not so important as we thought, *FrequenSel* can select indeed typical ones and this is the fundamental difference compared with other algorithms.

IV. EVALUATION

To evaluate performance of FEST, we conduct several experiments to verify its accuracy and effectiveness. Our dataset contains 7972 apps, and 50% of them are malware and the rest 50% are benign apps. Malware is collected from Drebin [3] and several public malware libraries, and benign apps are downloaded from Google Play and scanned with anti-virus softwares, Such as Kaspersky and Bitdefender, to ensure that none of them is malicious. We evaluate *FrequenSel* and malware detection models with 10-fold cross-validation on Weka [10].

A. Results of Detection Model Building

To find suitable algorithms for machine learning in our dataset, we compare performance of several well-known classification algorithms, including Naïve Bayes, J48, KNN and SVM. The experiment results are shown in Figure 2, and Figure 3. Accuracy and recall are defined as $Accuracy = (TP + TN)/(TP + FP + FN + TN)$, $Recall = TP/(TP + FN)$, respectively. TP, FP, TN and FN are the number of True Positive, False Positive, True Negative and False Negative samples.

In this experiment, we use four algorithms to build our models on the features selected by *FrequenSel*. Before learning on Weka, there are several parameters that need to be decided, including the value of k in KNN, and the value of c and g in SVM. According to the greedy search tests, we find that k=1, c=20 and g=0.01 are the optimal values. From Figure 2, with the increasing number of features, the accuracy of four algorithms get a bit risen. Among the four algorithms, SVM can achieve the highest accuracy all the time when the features change, nearly 98%. The performance of

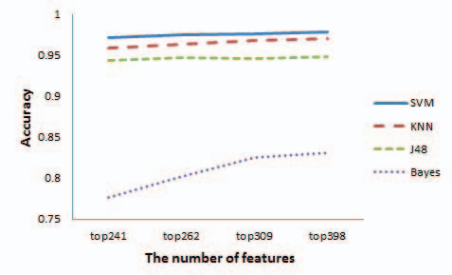

Fig. 2. Accuracy of the four algorithms with different features

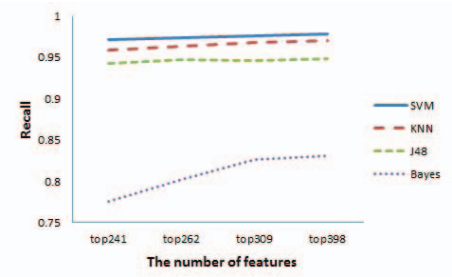

Fig. 3. Recall of the four algorithms with different features

Naïve Bayes is the worst of all algorithms because of its strictly independence limitation. However, features are closely relevant to each other in our feature dataset, for example, the feature *p.READ_CONTACTS* usually works with *p.INTERNET* in some instant messaging apps. Figure 3 shows recall of four algorithms which has nearly the same curve as accuracy, which means features selected by *FrequenSel* can equivalently classify malware and benign apps. Unlike DroidMat [4] with lower recall, the improvement benefits from *FrequenSel*, which selects more frequently used typical features in malware. Moreover the accuracy and recall of SVM are slightly higher than KNN, but the time used for building SVM models is much longer than KNN, 59.9s and 0.02s respectively. As mentioned in Section II, we choose SVM and KNN as the optimal classification algorithms for further experiments.

B. Performance of FEST

To evaluate the performance of *FrequenSel*, we compare three features selection algorithms. First we select features with *FrequenSel* when α_m=0.5, 0.6, 0.7, 0.8, thus we automatically have four subsets containing 398, 309, 262, 241 typical features. For comparison, we sort all the features with chi-square and information gain, and manually select top 398, 309, 262, 241 features in the two ranked list. Finally, we build detection models based on KNN and SVM, the results are shown in Figure 4 to Figure 7.

From the four figures, we have two observations: 1) With the increasing number of features, KNN and SVM classify apps more precisely. 2) *FrequenSel* significantly outperforms the other algorithms in terms of accuracy and recall all the time. The result of recall is as good as accuracy, which means *FrequenSel* can classify malware and benign apps equally well. Compared with the other two algorithms, once the number of features is fixed, *FrequenSel* can achieve higher accuracy and

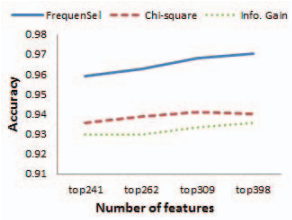

Fig. 4. Accuracy of KNN

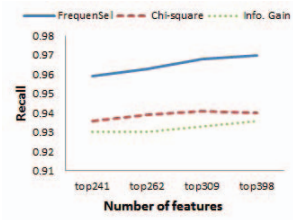

Fig. 5. Recall of KNN

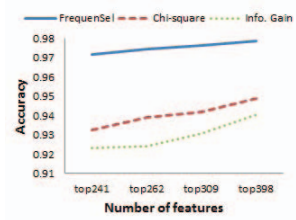

Fig. 6. Accuracy of SVM

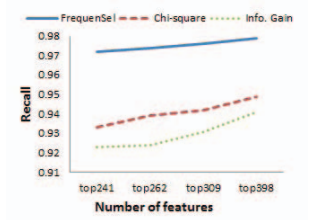

Fig. 7. Recall of SVM

TABLE II
SIMILARITY OF 398 SELECTED FEATURES

Alg.	Same features	Similarity
Chi&Info. Gain	352	88.4%
FrequenSel&Chi	83	20.9%
FrequenSel&Info	62	15.6%

TABLE III
COMPOSITION OF 398 SELECTED FEATURES

Alg.	Permi-ssions	APIs	Others
Chi-square	1.3%	98.5%	0.2%
Info. gain	1.3%	98.5%	0.2%
FrequenSel	6.8%	91.5%	1.7%

TABLE IV
THE COMPARISON OF EVALUATION METRICS

Alg.	Accuracy	FP Rate	Recall	Detect failed
Andro.	76.4%	23.6%	63.1%	34
Fest	97.8%	2.1%	97.8%	0

recall because more typical features are selected. Therefore, *FrequenSel* is more effective for feature selection.

To better explain the advantages of *FrequenSel*, we compare the similarity of the features generated by the three selection algorithms in Table II and their composition in Table III. In Table II, we find that 88.4% of features selected by chi-square and information gain are the same, while *FrequenSel* contains less same features when compared with them, which makes chi-square and information gain get similarly lower accuracy and recall than *FrequenSel*. In Table III, we analyze the composition of these features, and observe that *FrequenSel* selects more features from permission and action[3] category. Specifically, permissions can reflect the behaviours of apps, which can indicate the restricted operations and help detect malware. *FrequenSel* tends to balance the number of features in each category, makes the models built on diversiform features, and obtains higher accuracy and recall.

Moreover, we discover that more than 53% of the features selected by chi-square and information gain are UI-related, and the same ratio of *FrequenSel* is only 32.2%. This result indicates that Chi-square and information gain overestimate the importance of these UI-related features because of their widely usage in benign apps, actually these features are useless for correct classification. *FrequenSel* can address this problem and focuses on the typical features in malware for purpose of malware detection.

Additionally, in Table IV we compare FEST with a popular Android analysis tool, Androguard [11], on the same dataset, including 7972 apps. The results show that 34 apps cannot be correctly detected by Androguard because of *Bad CRC-32* exception[4]. We find its accuracy and recall are significantly lower than FEST, the main cause is that Androguard is not good at classifying malware in our dataset, only 63.1%, meanwhile it correctly classifies 89.9% benign apps. In some works, for example DroidMat [4], the average accuracy of Androguard is higher because the number of malware is less than benign samples. However, FEST outperforms Androguard completely, and the ability of classifying malware and benign apps makes it more precise without any unhandled samples. And In Table V, we compare FEST with four popular anti-virus scanners on malware dataset, FEST gets 97.8% accuracy which is better than most of the scanners.

[3]Actions, IPs and URLs are included in Others together.

[4]Some files in app's package can not be resolved into correct zip file by Androguard.

TABLE V
MALWARE DETECTION ACCURACY VS. SCANNERS

Fest	BitDefender	F-Secure	Kaspersky	McAfee
97.8%	98.3%	96.5%	94.7%	84.2%

C. Results of Detect Unknown Apps

To evaluate performance of classifying unknown apps based on our approach, we build two detection models based on KNN and SVM algorithms, and feed 398 features selected by *FrequenSel* into each model, then we use the two trained models to evaluate 4300 unknown Android apps. Table VI shows the results.

TABLE VI
RESULTS OF UNKNOWN APPS CLASSIFICATION

Alg.	Correctly classified	TP Rate	FP Rate	Accuracy	Recall
KNN	4163	0.968	0.039	0.968	0.968
SVM	4191	0.975	0.032	0.975	0.975

In Table VI, accuracy and recall are nearly the same with Figure 6, only FP Rate increases less than 1%, which is reasonable that FEST misses a little malware when handling unknown samples. The result demonstrates that our malware detection models can achieve high accuracy and recall in real-world app scenario.

V. RELATED WORK

Many studies on android malware detection have been performed in recent years, static and dynamic analysis are two major directions among these works. Enck et al. [12] gives an overview of malware regarding its dangerous behaviors and vulnerabilities.

1) Static Analysis: Decompiling and data flow tracking are two main techniques in most of the static analysis methods. ScanDroid [13] checks the security of apps by detecting suspicious code regions through data flow analysis. AndroidLeaks [14] focuses on finding privacy or sensitive data leaks with decompiled code. FlowDroid [15] detects malware by building precise model of Android's lifecycle. Moreover, LeakMiner [16] is a tool used in market site, like FEST, and detects leakage of sensitive information on Android with static taint analysis.

2) Dynamic Analysis: Dynamic analysis explores on another direction, and most of them monitor the behaviours of apps in terms of accessing private data or using restricted API calls. Android framework modification and apps repackaging are main techniques for dynamic analysis. For example, TaintDroid [17] is one of the popular system-wide dynamic taint tracking tool and it monitors multiple sources of sensitive data, AppsPlayground [18] takes the advantages of TaintDroid for detecting privacy leaks. These works all require framework modifying, which makes them hard to be deployed. On the contrary, Jeon et al. [19] restrains the access to system resources and sensitive information on phones by repackaging apps and attaching extra code or modules.

3) Machine Learning: Generally, machine learning approaches belong to the static analysis category. However, automatic feature extraction and efficient app classification by models without manual operation make it different from other static analysis techniques. Drebin [3], DroidMat and [4] DroidAPIMiner [5] build models with different machine learning algorithms on features, including permissions, API calls, and so on.

VI. Conclusion

In this paper, we present FEST, which aims at extracting and selecting features for building malware detection models. In addition, we introduce *AppExtractor* as a tool for extracting features in apps, due to the lack of effective feature selection algorithms, we propose *FrequenSel*, compare it with other similar algorithms, and discuss its advantages and why it works better. The experiments show that *FrequenSel* is definitely more suitable for our feature dataset, and our models built on the selected features are effective and reliable for malware detection. More specifically, FEST achieves about 98% accuracy and recall at the same time, and its adaptability of unknown samples inherited from machine learning techniques satisfies the detection demand of third-party markets with lots of fast updated apps.

Acknowledgements

This work in supported by the National Basic Research Program of China (973) under Grant No. 2013CB315805, the National Science Foundation of China under Grant No. 61473123 and 61173167, and the Jiangsu Science Project under Grant No. BY2013095-1-05.

References

[1] Mobile threat report, https://www.f-secure.com/documents/996508/1030743/mobile_threat_report_q1_2014.pdf.

[2] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (SP)*, pages 95–109. IEEE, 2012.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. *NDSS*, 2014.

[4] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Seventh Asia Joint Conference on Information Security (Asia JCIS)*, pages 62–69. IEEE, 2012.

[5] Yousra Aafer, Wenliang Du, and Heng Yin. *DroidAPIMiner: Mining API-level features for robust malware detection in android*, pages 86–103. Springer, 2013.

[6] Apktool, http://code.google.com/p/android-apktool/.

[7] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[8] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[9] Android.basebridge, http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99&tabid=2.

[10] Weka, http://www.cs.waikato.ac.nz/ ml/weka/.

[11] Anthony Desnos. Androguard: Reverse engineering, malware and goodware analysis of android applications, 2013.

[12] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.

[13] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android. *IEEE security & privacy*, 2009.

[14] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*, pages 291–307. Springer Berlin Heidelberg, 2012.

[15] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 29. ACM, 2014.

[16] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Third World Congress on Software Engineering (WCSE)*, pages 101–104. IEEE, 2012.

[17] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014.

[18] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

[19] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.