Incorporating Android Code Smells into Java Static Code Metrics for Security Risk Prediction of Android Applications

Ai Gong[1], Yi Zhong[1,4], Weiqin Zou[2,3], Yangyang Shi[1], Chunrong Fang[1]
[1]*State Key Laboratory for Novel Software Technology, Nanjing University, China*
[2]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China*
[3] *Key Laboratory of Safety-Critical Software (NUAA), Ministry of Industry and Information Technology, China*
[4] *College of Mobile Telecommunications Chongqing University of Posts and Telecom, China*
Email: fangchunrong@nju.edu.cn

***Abstract*—With the wide-spread use of Android applications in people's daily life, it becomes more and more important to timely identify the security problems of these applications. To enrich existing studies in guarding the security and privacy of Android applications, we attempted to predict the security risk levels of Android applications. Specifically, we proposed an approach that incorporated Android code smells into traditional Java code metrics to predict how secure an Android application is. With an evaluation of our technique on 3,680 Android applications, we found that: (1) Android code smells could help improve the performance of security risk prediction of Android applications; (2) By building a Random Forest model based on Android code smells and Java code metrics, we could achieve an Area Under Curve (AUC) of 0.97; (3) Android code smells such as *member ignoring method (MIM)* and *leaking inner class (LIC)* have a relatively-large influence on Android security risk prediction, to which developers should pay more attention during their application development.**

***Index Terms*—Android code smells, Java code metrics, Android security**

I. Introduction

Over the past decade, we have seen a tremendous rise in mobile applications. These applications are widely used in almost every aspect of people's daily life, from gaming, studying, to shopping, working, etc. One fatal problem related to such great popularity of mobile applications is that, once a mobile application has security problems, it may make a large number of users' life suffering. Based on our knowledge, however, it is not rare that mobile applications are of potential security problems. As stated by a report from Qihoo[1] (the biggest Internet Security Company in China), 99.5% of 18,000 Android mainstream applications are at risk of threats, with an average of 38.6 threats per application. Thus, it is significant to timely predict/identify and resolve security problems related to mobile applications.

Researchers have done much work to help solve problems related to application permission [1], application security [2], [3], system security [4], malware detection [5], [6], user privacy [7], [8], and malicious attack [9], etc. Our study also focuses on the topic of application security. In the area of application security, many studies have been done on guarding the security and privacy of Android applications [3], [10], [11]. In this paper, we mainly aim to predict the security risk levels of Android applications, to help developers assess and understand how secure their Android applications are [3].

To build a prediction model with good performance, finding useful predictors or features is full of importance. Previous studies mainly utilized Java static code metrics to predict security risks of Android applications, as these applications were primarily developed in Object-oriented (OO) programming languages such as Java or Objective-C. Android-related metrics that reflect Android-specific features, however, are rarely used to help facilitate the performance of security prediction models.

Unlike traditional Java applications, Android applications have their characteristics. They are developed in different programming paradigms, using different libraries and always running under limited resources. For example, GUIs on Android are declared via XML, and Android applications have no particular main methods (Android entry points are handled by event-handlers). For another instance, many APIs are specifically designed for implementing some mobile features (including Contacts, Power Management, Graphics, etc.). Related to these Android-specific features, Reimann et al. [12] summarised a set of peculiarly bad programming practices of Android developers (e.g., a non-static inner class holds a reference to an outer class), namely Android-specific smells in this paper. These Android-specific smells threatened the security, data integrity, and source code quality of mobile apps [12], [13]. Inspired by existing findings of Android-specific code smells, we decided to explore the effect of Android code smells in predicting the security risks of Android applications.

Specifically, we proposed to integrate Android code smells into existing Java code metrics to aid security risk prediction for Android applications. We first investigated the correlation between Java code metrics and Android code smells. We found that most Android code smells were independent with each

Ai Gong and Yi Zhong contributed equally to this work.
[1]https://research.360.cn/2015/reportlist.html?list=1

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00017

other, and they had a moderate correlation with Java code metrics. Then, we built a list of security risk prediction models by applying various popular machine learning methods to those Android code smells together with Java code metrics. A dataset with 3,680 real-world Android applications from Github was constructed to evaluate our models. The results showed that Android code smells could improve security risk prediction; and the learning method Random Forest (RF) [14] could outperform other learning methods by achieving the best Area Under Curve (AUC) of 0.97. Last, we investigated the importance of each Android code smell in helping predict the security risks of Android applications; and highlighted several Android-specific code smells to which developers should pay more attention during their Android application development.

Our major contributions are as follows:

- We are the *first* to consider incorporating Android-specific code smells into Java code metrics for security risk prediction of Android applications.
- We compared various prediction models built by applying different learning methods to these Android code smells and Java code metrics; As a result, we find that Random Forest could obtain the best prediction results, with an AUC of 0.97.
- We studied the importance of individual Android code smells in predicting the security risks of Android applications. The results provided a guide for developers in solving security problems related to Android code smells.

This paper is organized as follows. Section 2 presents our approach. Section 3 depicts the experiments. Section 4 presents results and analysis. Section 5 introduces related work, and Section 6 concludes our work.

II. Methodology

As shown in Figure 1, we adopt a three-step approach to build a security risk level prediction model. First, we extract Java static code metrics and Android code smells from Android applications. Then we calculate a security risk level for each application with manual help. Last, we apply main-stream machine learning algorithms to Android applications (with extracted code metrics and smells as features and security risk levels as class labels) to build a prediction model. The details are as follows.

A. Code Metrics Extraction

This subsection mainly describes the process of extracting code metrics in our work, including Java static code metrics and Android code smells.

1) Java Static Code Metrics: Existing studies have found that some Java static code metrics like bad coding practices, duplications, and OOP (Object Oriented Programming) attributes are promising in predicting the security risk of Android applications [3], [15]. We also use those metrics used in the previous studies [3], [15] to help predict the security risk of Android applications. There are a total of 21 Java static code metrics used in our study (details in Table I). These metrics are retrieved from each Android application with the help of a tool called SonarQube[2] (SonarQube is an open-source tool for continuous analysis and code quality evaluation. It can detect duplicated code, potential bugs, code style problems, and other issues of a project.)

TABLE I
Java static code metrics

Metric	Description
#lines	The number of code lines.
#functions	The number of functions.
#classes	The number of classes.
#files	The number of Java files.
#directories	The number of directories within a project.
function_complexity	The average complexity of all functions.
class_complexity	The average complexity of all classes.
file_complexity	The average complexity of all files.
complexity	Cyclomatic complexity.
#comment_lines	The number of lines containing comments.
#ncloc	The number of non-commenting code lines.
comment_lines_density	The ratio of comment lines.
#blocker_violations	The number of issues with blocker severity.
#critical_violations	The number of issues with critical severity.
#major_violations	The number of issues with major severity.
#minor_violations	The number of issues with minor severity.
#violations	The number of issues of all severity levels.
#duplicated_blocks	The number of duplicated code blocks.
#duplicated_files	The number of duplicated files.
#duplicated_lines	The number of duplicated code lines.
duplicated_lines_density	The ratio of duplicated code lines.

2) Android Code Smells: As mentioned in Section I, Android code smells may threaten non-functional attributes of Android applications and may help improve the effectiveness of Android security and risk prediction [12]. In this paper, we particularly studied the effect of 15 Android code smells (presented in Table II) in predicting the security risk of Android applications. These Android-specific code smells are retrieved by using aDoctor [16], a lightweight code smells detection tool with an average precision of 98% and an average recall of 98%.

B. Risk Levels Calculation

Before we build a machine learning model, we should first construct a dataset with knowing each instance's features and labels. Section II-A has described the way to get features for each instance (i.e., an Android application). In this section, we would detailedly introduce how to decide the class label (i.e., the security risk level) for each instance. Our labeling process includes two parts. First, we would try to obtain an initial risk score; then we use this score to get the final security risk level (Figure 2 describes the details).

1) Calculating Risk Scores: In this part, we first used Androrisk[3] to obtain an initial risk analysis of an application; then we manually checked and adjusted (when necessary) the analysis results to obtain the final risk score for the Android application. Androguard is an open-source tool. We can use it to extract some information from Android applications,

[2]http://www.sonarqube.org
[3]https://code.google.com/p/androguard

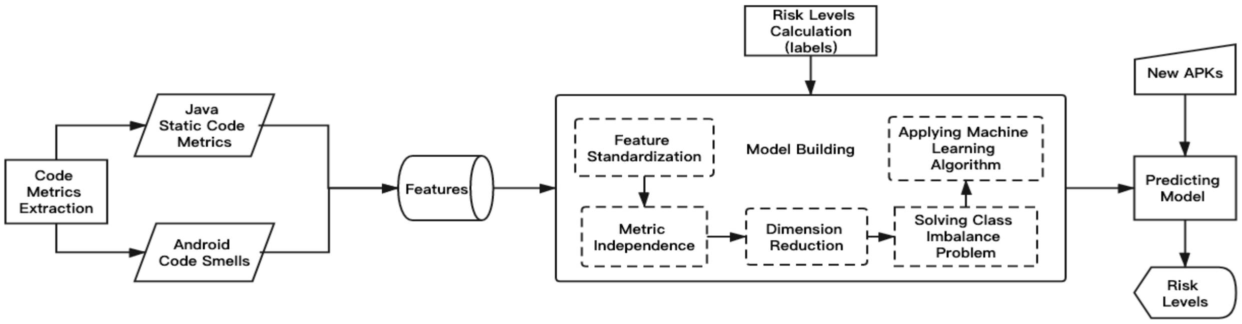


Fig. 1. The framework of security risk level prediction for Android applications.

TABLE II
ANDROID CODE SMELLS

Metric	Description
DTWC	Data transmission without compression.
DR	Debuggable Release: An app is set to be debuggable in Android Manifest.xml.
DW	Durable Wakelock: A *Wakelock* is not released finally.
IDFP	Inefficient Data Format and Parser: Inefficient parsers are used to parse files.
IDS	Inefficient Data Structure: Inefficient data structures are used to map integers.
ISQLQ	Inefficient SQL Query: An inefficient SQL is sent to the server to query the data.
IGS	Internal Getter and Setter: Use getters and setters to access internal fields.
LIC	Leaking Inner Class: A non-static nested class holds a reference to an outer class.
LT	Leaking Thread: A thread is not adequately stopped.
MIM	Member Ignoring Method: Methods not accessing internal properties are not made static.
NLMR	No Low Memory Resolver: No methods are used to clean unnecessary resources.
PD	Public Data: Private data are exposed to other applications.
RAM	Rigid Alarm Manager: An Alarm Manager-triggered operation wakes up the phone.
SL	Slow Loop: An enhanced version of a for loop is not used.
UC	Unclosed Closable: A *Closeable* class do not call the close method.

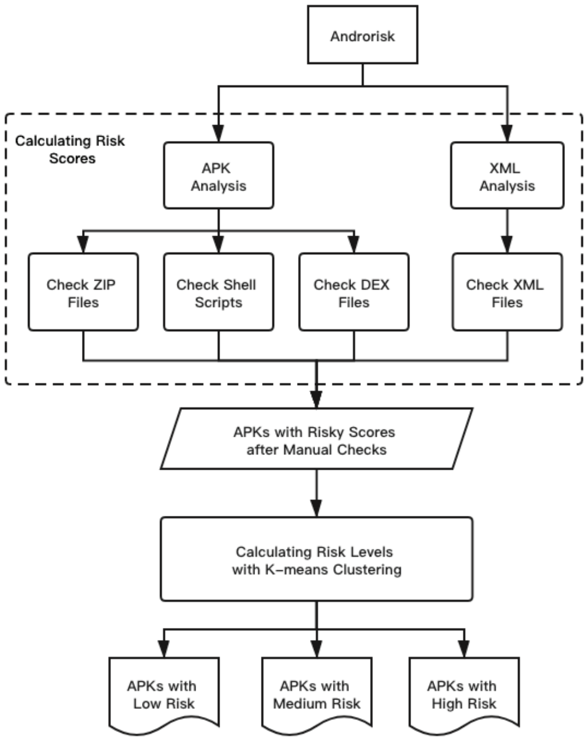


Fig. 2. The process of calculating risk levels

including ZIP files, APK files, DEX files, XML files, and so on. Androrisk is one of the Androguard modules, which is a widely-used risk assessment tool [3], [15]. It has two risk assessment modules. These two modules are used to analyze APK files and XML files, respectively. After we obtained the analysis result of each module, we manually checked the result and modified the code of the Androrisk to get a relatively fair risky score.

For the analysis of APK files, Androrisk calculates the risk score based on the presence of files including ZIP files, shell scripts, and DEX files. Within Androrisk, these files are believed to be easy to be exploited by hackers. However, the presence of these files does not mean an APK is definitely risky. If an APK itself has a pre-coded checking mechanism in handling potential risks of these files, then the APK could also be free of security threats. Hence, to avoid possible false positive warnings from Androrisk, we manually checked each APK to see whether they have corresponding checks. Specifically, for ZIP files, we mainly checked whether an APK checked if the ZIP file names have a special string like "../" ("../" may cause the decompressed files to overwrite files in other directories, eventually leading to arbitrary code execution). For shell scripts, we mainly checked whether an APK conducted checks on external parameters passed to shell scripts. For DEX files, we mainly checked when an APK tried to load a DEX file, whether this APK checked the integrity (through CRC32 or hash code) of this DEX file. Only those APKs which lack necessary checks in our study are considered as risky.

For the analysis of XML files, Androrisk assigns a weight to each permission based on its sensitivity and risk (i.e., access to the Internet, manipulate SMS or manipulate your location). However, it is unreliable to only analyze sensitive permissions of applications to evaluate the security of applications, as many sensitive permissions are used by many benign applications. To reduce false-positive errors, we manually checked Androrisk's analysis of permissions. More specifically, we manually checked whether there is a difference between the application description and its permissions, and took unnecessary permission as a risk.

2) Calculating Risk Levels: After we obtained risk scores of individual applications in Section II-B1, we further discretized these continuous numeric values to get their risk levels as their final class labels. In order to discretize the continuous risk scores, we apply K-means clustering [17] on risk scores of Android applications. The input to the k-means clustering includes the number of clusters k and n data objects. In our study, the risk scores of the 3,680 Android applications are used as the data to create the clusters. The number of clusters k is determined by using the Elbow method [17]. Elbow would run k-means clustering on the dataset for a range of values for k. For each k, Elbow would calculate the sum of squared errors (SSE) of prediction results. SSE is the clustering error of all data points. The lower SSE is, the better the clustering quality is. The way to compute SSE is as follows:

$$SSE = \sum_{i=1}^{K} \sum_{p \in C_i} |p - m_i|^2 \tag{1}$$

C_i represents the ith cluster; p is one sample point in C_i; m_i is the centroid of C_i (the mean value of all samples in C_i).

When k is smaller than the actual number of clusters, increasing k would greatly increase the degree of aggregation of each cluster, and further largely decrease the whole SSE. While k reaches the number of real clusters, the increment of the degree of aggregation obtained by increasing k will rapidly become smaller; the decline of SSE will also become slower and tend to be flat as k continues to increase. That is, the relationship between SSE and k is the shape of an elbow, and the elbow corresponds to the k value is the actual number of clusters of a dataset.

C. Model Building

After we get the original features (i.e., Java code metrics and Android code smells) and class labels (security risk levels) of individual instances (Android applications), we can theoretically build a machine learning model on them. However, given that the values of extracted features in our study are quite different in their scales, and there may exist a multicollinearity problem among those features, we decide to do some preprocessing on those features first, to avoid the relevant negative effect on model building. Besides, we also observe that our dataset is quite imbalanced, i.e., the numbers of instances of different classes are quite different, such an imbalanced-classes problem should also be resolved during model building as it would greatly affect the performance of a machine learning model [18]. After the above steps, we could then apply typical machine learning algorithms to build prediction models. Below are the details.

1) Feature Standardization: In our study, the numerical values of our instance features (i.e., Java code metrics and Android code smells) are quite different in their range scales. For example, some Android code smells metrics are within the range of 0 to 10, while code line metrics are as large as several thousand. Without handling the scale problem, the predicted results might be dominated by some features with large values; and it would also affect the rational comparison of individual features and the training speed. To overcome the scale problem, we decided to standardize features during model building. There are usually two ways to standardize features. One is called Min-max standardization, which limits all features to a scale of 0 to 1. Another one is called z-score standardization. It is based on the mean and standard deviation of the original features for data standardization. In this paper, we decided to adopt the z-score standardization way. By applying z-score standardization, the standard deviation of features of each dimension would be 1 and the mean value would be 0. This could help us avoid the problem of predicted results being dominated by some features with large value ranges.

2) Metric Independence and Dimension Reduction: Generally speaking, it is not uncommon that some instance features may correlate with some other features (so does our case). For example, Android application with a larger number of lines of code tends to have a higher complexity. A prediction model that does not fully deal with the multicollinearity problem among features may increase the variability of dependent variables and thus reduce its performance [19]. In this paper, we used PCA (Principal Component Analysis) [20] to solve the multicollinearity problem. The features generated from PCA are not related to each other. With PCA, we could not only reduce the feature dimension but also achieve competitive prediction performance.

3) Imbalanced classes: In our study, the numbers of instances of risk level classes are imbalanced. Learning algorithms that do not take account of class imbalance are often overwhelmed by the majority class and ignore the minority class. For example, a learning algorithm that minimizes error rates may classify all examples into the majority class. In this way, all examples of minority classes will be misclassified. In order to obtain a well-performed prediction model, the class imbalance problem should be carefully handled.

There are two widely-used solutions to address the class imbalance problem, i.e., oversampling and undersampling. Oversampling tries to balance classes by adding multiple copies of some instances of the minority classes; while undersampling tries to balance classes by discards some instances of the majority classes. Considering that the number of our minority class is relatively small, we decide to use the oversampling approach in our study (By using undersampling, our training dataset may be too small to build a good prediction model). Related to the oversampling approach, we did not use the original random oversampling approach as it is easy to introduce over-fitting problems. Instead, we used an improved oversampling method, i.e., SMOTE [21], together with Wilson's Edited Nearest Neighbor Rule (ENN) [22] to solve the class imbalance problem (SMOTE could help us avoid the over-fitting problems, ENN could remove any example that is misclassified by its three nearest neighbors).

4) Applying Machine Learning Algorithms: Different machine learning methods are very likely to have different performances in predicting security risk levels of Android applications. To understand how far we can go correctly predicting how secure an Android application is by incorporating Android-specific code smells and Java static code metrics, we compared a variety of typical machine learning algorithms in our study. They are Naive Bayes (NB), k-nearest neighbor (KNN), Logistic regression (LR), Random Forest (RF), Decision Tree (DT), SVM (SVM), gradient enhanced Decision Tree (GBDT), multilayer perceptron (MLP), and Convolution neural network (CNN).

III. Experiment

A. Dataset Preparing

Our experimental projects are crawled from GitHub[4]. Specifically, we first searched a list of projects from GitHub through keyword ”Android application” as candidate projects. Then from those candidate projects, we took those which are developed mainly in Java as our experimental projects. 3,680 Android applications are used in our experiments. These applications are from various domains and are of different sizes. For each Android application, we then used the tool SonarQube to collect Java code metrics and the tool aDoctor to collect Android code smells, respectively.

As described in Section II-B, we used the elbow method to determine the number of clusters 3,680 Android applications belonging to. We calculated the SSE values of K ranging from 1 to 9, and found that when K ranged from 1 to 3, the change of the SSE was the greatest, which means 3 is an optimal number for clustering these applications. Hence we set k as 3 and used k-means to divide 3,680 applications into 3 clusters. We assigned three risk labels to these three clusters, i.e., low risk (L), medium risk (M) and high risk (H). The centroids of three clusters were 0.15715467, 50.90121676 and 92.89716516, respectively. The numbers of Android applications with low risk, medium risk, and high risk were 1209, 2118, and 353, respectively.

[4]http://github.com

B. Evaluation Metrics and Settings

We used four metrics namely accuracy, precision, recall and F1 score to evaluate the effectiveness of different classifiers. They are defined as follows.

- **Accuracy** is an intuitive performance metric that is defined as the ratio of samples correctly classified by the classifier over the total number of samples. The formula is as follows.

$$A(M) = \frac{TN + TP}{TN + FP + FN + TP} \tag{2}$$

- **Precision** is defined as the number of true positives (TP) over the number of true positives plus the number of false positives (FP). The formula is as follows.

$$P(M) = \frac{TP}{TP + FP} \tag{3}$$

- **Recall** is defined as the number of true positives (TP) over the number of true positives plus the number of false negatives (FN). The formula is as follows.

$$R(M) = \frac{TP}{TP + FN} \tag{4}$$

- **F1 score** can be interpreted as a harmonic mean of precision and recall. The formula is as follows.

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{5}$$

In the above formulas, FN (False Negative) represents the number of samples which are actually positive samples but are judged as negative samples. FP (False Positive) represents the number of samples which are actually negative samples but are judged as positive samples. TN (True Negative) represents the number of samples which are in fact negative sample and are also judged as negative samples. TP (True Positive) represents the number of samples that are in fact positive samples and are also judged as positive samples. For the multi-classification problem in this paper, while calculating the above-mentioned four evaluation metrics, each risk level is treated as ”positive” alone, and all other risk levels are considered as ”negative”.

In this paper, we also use ROC (Receiver Operating Characteristic) to evaluate the performance of the classifier. ROC is a good measurement to reveal the accuracy of a classifier [23]. In ROC space, the x-axis corresponds to the FPR (False Positive Rate) and the y-axis corresponds to the TPR (True Positive Rate). ROC depicts a trade-off between TP (True Positive) and FP (False Positive). TPR and FPR are defined as follows.

- **True Positive Rate (TPR)** represents the proportion of positive cases that are correctly identified. Its calculation is the same as the way to compute recall in formula (4).
- **False Positive Rate (FPR)** represents the proportion of negative cases incorrectly identified as positive cases. The formula is as follows.

$$FPR = \frac{FP}{FP + TN} \tag{6}$$

TABLE III
SPEARMAN'S CORRELATION AND CORRELATION LEVEL

Correlation Coefficient	Correlation Level
0.0 - 0.1	None
0.1 - 0.3	Small
0.3 - 0.5	Moderate
0.5 - 0.7	High
0.7 - 0.9	Very High
0.9 - 1.0	Perfect

In order to minimize the randomness of experimental results, we repeated 10-fold cross-validation 10 times. During each 10-fold cross-validation process, we randomly partition the data into 10 folds, with 9 folds as the training data and 1 fold as the testing data. After repeating the process 10 times, we aggregate the results and take their average results as the final results to evaluate the performance of the prediction model. All experiments are conducted on an Ubuntu 64-bit system with 32 GB RAM and 2.5 GHz Intel Xeon CPU.

C. Research Questions

In this paper, we mainly try to answer the following three research questions.

RQ1: Are there any correlations among the Java static code metrics and Android code smells?

RQ2: How effective our approach is in predicting the risk level of Android applications?

RQ3: Which features are more important in predicting the security risk level of Android applications?

RQ1 aims to understand whether there is an interactive relationship between Java static code metrics and Android code smells. RQ2 aims to investigate how helpful the proposed metrics are at risk level prediction. To better answer this research question, we tested several typical machine learning methods. RQ3 aims to check whether all metrics especially those Android code smells share the same importance for risk prediction. By knowing this could help developers better avoid some bad development practice.

IV. RESULTS AND ANALYSIS

A. Results for RQ1

We used Spearman's Rank Correlation Coefficient to explore the correlation between hybrid metrics (i.e., Java static code metrics and Android code smells in the paper) [24]. We chose Spearman correlation since it has no strict requirement for data conditions, and it can be used to study both variables regardless of their overall distribution and sample size. Table III presents the correlation coefficients and their corresponding correlation levels according to [25].

Figure 3 shows the heatmap of Spearman rank correlations. From Figure 3, we can find that the Spearman coefficients between Java code metrics range from 0.8 to 1, which indicates a high to perfect correlation (according to Table III). This means some Java static code metrics used in this paper to a large extent are redundant with some other Java static code metrics in building a prediction model, and this is one major reason that we performed PCA processing towards these metrics in model building.

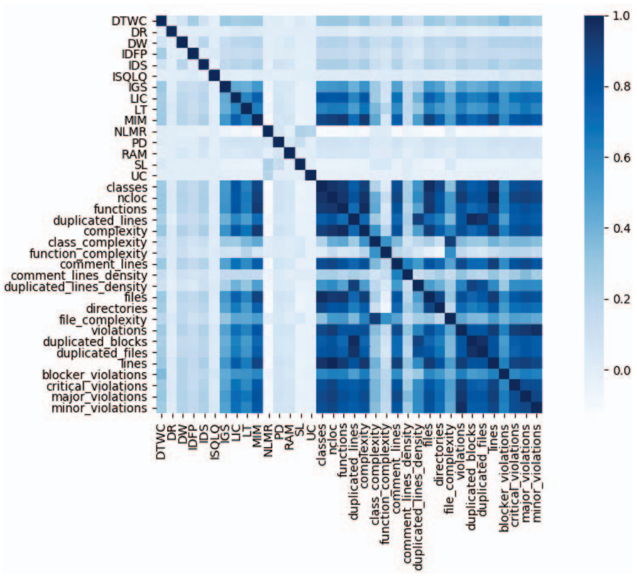


Fig. 3. The correlation between hybrid metrics (i.e., Java static code metrics and Android code smells).

As related to Android code smells, we can find that from Figure 3, the coefficients between the Android code smells range from 0.0 to 0.2, which shows none to small correlation. Further, the correlation coefficients between Java metrics and Android code smells are around 0.4, which means that the correlation between them is moderate. The none-to-small correlation among Android code smells and the moderate correlation between Android code smells and Java static code metrics makes us believe that, by adding Android code smells into Java static code metrics would lead to a better performance in predicting the security risk levels of android applications.

Figure 3 also shows that the total number of smells in an Android application is related to the number of lines, the number of files, and the number of classes, which is also observed in [26], [27]. However, among those code smells (which mostly are independent with each other), only a few smells (i.e., internal getter and setter (IGS), leaking inner class (LIC), leaking thread (LT), and member ignoring method (MIM)) are related to code size.

Answer to RQ1: Most Android code smells are independent with each other; the correlations between Android code smells and Java static code metrics are moderate, and some Java static code metrics are highly correlated with some other Java static code metrics.

B. Results for RQ2

To answer RQ2, we constructed a series of prediction models by applying nine machine learning algorithms (as mentioned in Section II), to a data set with 3,680 Android applications.

The original risk level distribution of 3,680 Android applications is 1209 for L (Low-risk level), 2118 for M (Medium-risk level), and 353 for H (High-risk level). After balancing three classes of risk levels with SMOTEENN, the numbers of applications that belong to L, M, H are 1131, 609, and 1766 respectively. For each application, we applied `z-score` standardization and PCA algorithm to process its features (i.e., Java static code metrics and Android code smells). After using `z-score` standardization, the standard deviation and corresponding mean of each feature are 1 and 0, respectively. To better understand the effects of Android code smells and Java static code metrics in predicting the security risk level of Android applications, we attempted to test each machine learning algorithm on three data sets with different instance features, i.e., only Android code smells, only Java static code metrics, and their combination. All instance features were preprocessed by using PCA during each model's building. Finally, after applying PCA, the original 15 Android code smells were reduced to 10 features, original 21 Java static code metrics were reduced to 12 features, and the 36 hybrid metrics (Java static code metrics plus Android code smells) were reduced to 20 dimensions, respectively.

After the above-mentioned preprocessing steps, we applied relevant machine learning methods to build prediction models. To obtain the performance of each prediction model, we used the functions $precision\_score()$, $recall\_score()$ and $f1\_score()$ in Sklearn library to calculate $precision$, $recall$ and $F1$ score respectively, with setting function parameter *average= "weighted"*. Table IV shows the evaluation results.

According to Table IV, we can find that those machine learning methods based on Java code metrics and Android code smells have similar performance in terms of precision, recall, F1-score, and accuracy. This means that Android code smells are as important as Java code metrics for predicting risk levels. Furthermore, as shown in Table IV, all machine learning methods except Naive Bayes achieved the best performance by combining Java static code metrics and Android code smells. Among them, the average values of precision, recall F1-score, and accuracy are 0.70, 0.75, 0.70, and 0.80 respectively. This indicates that Java code metrics and Android code smells are complementary to each other in predicting risk levels of Android applications.

As explained in Section III-B, to better explore the prediction ability of different classification algorithms, we further examined the ROC curves of each prediction model in Figure 4. There are two kinds of ROC curves for multi-classification problems. In the first case, for each category, we can calculate the probability that the test samples belong to that category. Then, for n categories, we can draw corresponding n ROC curves, and get the average value of n ROC curves. In the second case, for each test sample, the label consists of only 0 or 1, 1 indicating its category, 0 indicating other categories. In order to evaluate ROC curves better, we invoke the *sklearn.metrics.roc_auc_score()* function to calculate the AUC value in Python. The first case corresponds to the parameter *average= "macro"*, and the second case corresponds to the parameter *average= "micro"*. In addition, these Figures also show ROC curves for each category (i.e., low, medium, and high risk level).

As shown in Figure 4, We described the ROC curves corresponding to each risk level and the ROC curves under multi-classification problems. In addition, the area under the ROC curve (AUC) is calculated separately. From Figure 4, we can observe that our proposed techniques have high predictive power for Android applications with high-risk and low-risk levels. The reason might be that the numbers of Android applications at those two levels are relatively larger, which makes the model more fully trained. Another observation is that RF (random forest) and GBDT (gradient boosting decision tree) have better prediction performance compared with other algorithms -- the ROU of RF and GBDT are 0.97 and 0.96, respectively (we used the scikit-learn Python library to build RF and GBDT models with default configuration settings in the paper). Both GBDT and RF are classic ensemble models that produce several decision trees at training time. Compared to a single decision tree, such an ensemble method could greatly reduce the over-fit problem caused by a single decision tree [28]–[30], thus further improved the prediction performance in practice.

Answer to RQ2: Android code smells could achieve competitive prediction performance compared to Java static code metrics; and they are complementary to each other in predicting the security of an Android application (By combining these two kinds of features, we could obtain an effective prediction model with the precision of 0.89, recall of 0.89, accuracy of 0.88, and F1-score of 0.89). Among nine typical machine learning methods, Random Forest (RF) could outperform other classification algorithms by obtaining an AUC of 0.97.

C. Results for RQ3

In this paper, we used a gradient boosting machine to identify important features in predicting the security risk level of an Android application. For a boosted tree, an average reduction of Gini impurity[5] for each feature is calculated and is further used to represent the importance of the feature. Figure 5 shows the importance of the Java static code metrics and Android code smells in security risk level prediction models. From Figure 5, We can find that 30 features contributed as much as 99% of the cumulative importance while 6 features did not contribute to cumulative importance (these 6 features were inefficient data structure (IDS), no low memory resolver (NLMR), slow loop (SL), debuggable release (DR), unclosed closable (UC) and inefficient SQL query (ISQLQ)). Among those features which contributed to cumulative importance, we can find that metrics including bad violations, lines, member ignoring method (MIM), leaking inner class (LIC) and so

[5]https://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity

TABLE IV
EFFECTIVENESS OF JAVA CODE STATIC METRICS AND ANDROID CODE SMELLS IN PREDICTING THE SECURITY RISK LEVEL OF ANDROID APPLICATIONS.

Classifier	**PCA**	**SMOTEENN**	**Metrics**							
			Java code metrics				Android code smells metrics / Hybrid code metrics			
			precision	recall	F1-score	accuracy	precision	recall	F1-score	accuracy
Naive Bayes	NO	NO	0.520	0.562	0.513	0.562	0.394 / 0.519	0.431 / 0.562	0.405 / 0.512	0.431 / 0.562
		YES	0.538	0.261	0.216	0.261	0.392 / 0.537	0.461 / 0.262	0.381 / 0.218	0.461 / 0.262
	YES	NO	/	/	/	/	/	/	/	/
		YES	/	/	/	/	/	/	/	/
KNN	NO	NO	0.597	0.625	0.599	0.625	0.572 / 0.602	0.605 / 0.627	0.576 / 0.602	0.605 / 0.627
		YES	0.798	0.796	0.789	0.796	0.722 / 0.803	0.725 / 0.803	0.719 / 0.795	0.725 / 0.803
	YES	NO	0.583	0.619	0.592	0.619	0.576 / 0.610	0.611 / 0.634	0.582 / 0.607	0.611 / 0.634
		YES	0.780	0.777	0.769	0.777	0.698 / 0.791	0.702 / 0.790	0.698 / 0.783	0.702 / 0.790
Logistic Regression	NO	NO	0.578	0.630	0.577	0.630	0.532 / 0.579	0.583 / 0.630	0.465 / 0.582	0.583 / 0.630
		YES	0.679	0.677	0.657	0.677	0.658 / 0.688	0.639 / 0.685	0.603 / 0.666	0.639 / 0.685
	YES	NO	0.571	0.627	0.563	0.627	0.430 / 0.568	0.573 / 0.620	0.424 / 0.559	0.573 / 0.620
		YES	0.649	0.633	0.610	0.633	0.662 / 0.658	0.638 / 0.647	0.598 / 0.625	0.638 / 0.647
Random Forest	NO	NO	0.645	0.660	0.637	0.660	0.601 / 0.656	0.621 / 0.666	0.605 / 0.640	0.621 / 0.666
		YES	0.880	0.881	0.879	0.881	0.758 / 0.883	0.758 / 0.884	0.758 / 0.882	0.758 / 0.884
	YES	NO	0.654	0.664	0.638	0.664	0.607 / 0.634	0.620 / 0.650	0.610 / 0.626	0.620 / 0.650
		YES	0.838	0.838	0.835	0.838	0.733 / 0.832	0.733 / 0.833	0.732 / 0.831	0.733 / 0.833
Decision Tree	NO	NO	0.599	0.596	0.597	0.596	0.568 / 0.601	0.568 / 0.597	0.567 / 0.599	0.568 / 0.597
		YES	0.833	0.834	0.833	0.834	0.731 / 0.834	0.730 / 0.835	0.730 / 0.834	0.729 / 0.835
	YES	NO	0.589	0.582	0.585	0.582	0.588 / 0.591	0.582 / 0.591	0.584 / 0.590	0.582 / 0.591
		YES	0.780	0.783	0.781	0.783	0.699 / 0.799	0.696 / 0.800	0.696 / 0.799	0.696 / 0.800
SVM	NO	NO	0.574	0.638	0.599	0.638	0.627 / 0.615	0.581 / 0.640	0.445 / 0.600	0.581 / 0.640
		YES	0.719	0.699	0.684	0.699	0.671 / 0.735	0.633 / 0.713	0.605 / 0.700	0.633 / 0.713
	YES	NO	0.595	0.659	0.625	0.659	0.629 / 0.689	0.587 / 0.657	0.462 / 0.622	0.587 / 0.657
		YES	0.734	0.717	0.708	0.717	0.677 / 0.747	0.642 / 0.725	0.614 / 0.715	0.642 / 0.725
Gradient Boosting	NO	NO	0.661	0.677	0.651	0.677	0.613 / 0.662	0.631 / 0.678	0.609 / 0.653	0.631 / 0.677
		YES	0.888	0.887	0.884	0.887	0.749 / 0.891	0.748 / 0.891	0.7447 / 0.889	0.748 / 0.891
	YES	NO	0.645	0.663	0.636	0.663	0.614 / 0.637	0.631 / 0.660	0.609 / 0.632	0.631 / 0.660
		YES	0.810	0.807	0.803	0.807	0.722 / 0.836	0.719 / 0.832	0.711 / 0.828	0.719 / 0.832
MLP	NO	NO	0.614	0.652	0.620	0.652	0.594 / 0.639	0.627 / 0.647	0.596 / 0.620	0.627 / 0.647
		YES	0.766	0.763	0.758	0.763	0.682 / 0.795	0.670 / 0.791	0.652 / 0.787	0.670 / 0.791
	YES	NO	0.675	0.669	0.633	0.669	0.574 / 0.625	0.628 / 0.661	0.592 / 0.631	0.628 / 0.661
		YES	0.724	0.722	0.715	0.722	0.672 / 0.767	0.659 / 0.762	0.637 / 0.754	0.659 / 0.762
CNN	NO	NO	0.588	0.648	0.615	0.648	0.564 / 0.613	0.624 / 0.647	0.592 / 0.615	0.624 / 0.647
		YES	0.698	0.694	0.682	0.694	0.671 / 0.765	0.638 / 0.759	0.597 / 0.752	0.638 / 0.759
	YES	NO	0.579	0.629	0.561	0.629	0.565 / 0.586	0.632 / 0.646	0.590 / 0.604	0.632 / 0.646
		YES	0.681	0.664	0.643	0.664	0.674 / 0.750	0.644 / 0.740	0.609 / 0.733	0.644 / 0.740

on have a relatively large influence on the performance of prediction models.

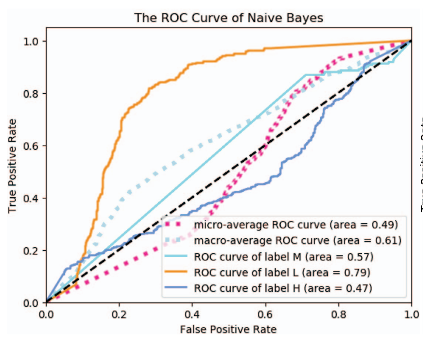
Compared to Java static code metrics, we can observe that although the importance ranks of Android code smells are not highest, the absolute values of the importance of code smells are just slightly smaller than those of Java code static metrics. This indicated the competitive importance of these two kinds of features in helping to predict the security risk level of an Android application.

Figure 5 also shows that not all Android code smells are related to risks in Android applications. Among those risk-related code smells, smells like member ignoring method (MIM), leaking inner class (LIC), leaking thread (LT), internal getter and setter (IGS), data transmission without compression (DTWC), inefficient data structure (IDS), durable wakelock (DW), and rigid alarm manager (RAM) are much more indicative for security risk prediction. This means that developers should pay more attention to avoid these bad practices during Android application development. For example, MIM is introduced if a method that does not access any internal properties of the class is not made static [12]. To avoid the potential security risk MIM brings, developers may have better make the non-static method as a static one. While analyzing code smells, we found that some smells which are naturally related to Android security (e.g., public data (PD)) only showed a weak correlation with application risks in our study. This was mainly because only a few applications contained such smells in our dataset.

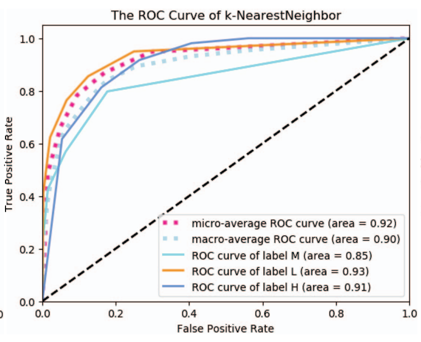
Answer to RQ3: Member ignoring method (MIM) and leaking inner class (LIC) have a relatively large influence on Android security risk prediction; developers should pay more attention to avoid these code smells in their application development.

D. Threats to Validity

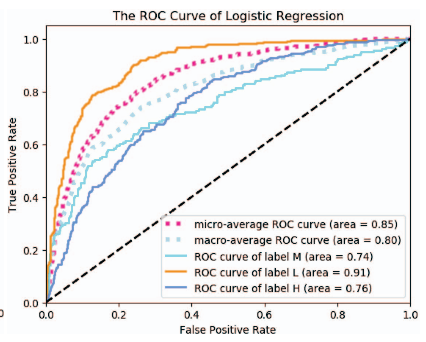
Construct Validity. In order to use Android-specific code smells to help predict the security risk level of an application, we should first define what Android code smells are and what code smells to use. As different practitioners may have different opinions about the definitions of Android code smells, in



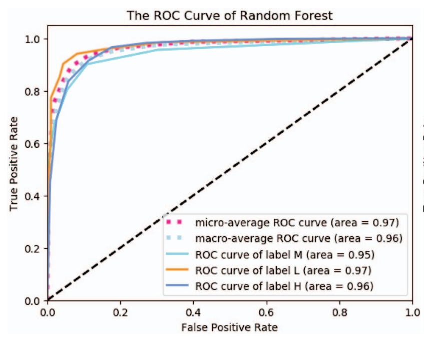
(a) Naive Bayes



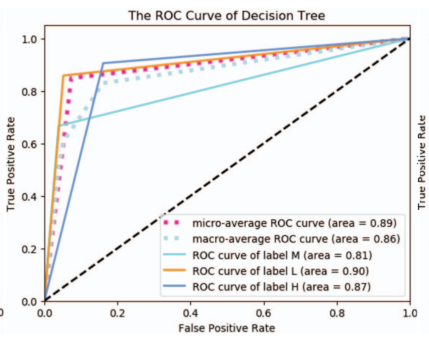
(b) k-NearestNeighbor



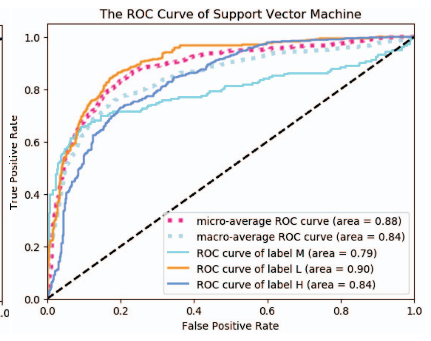
(c) Logistic Regression



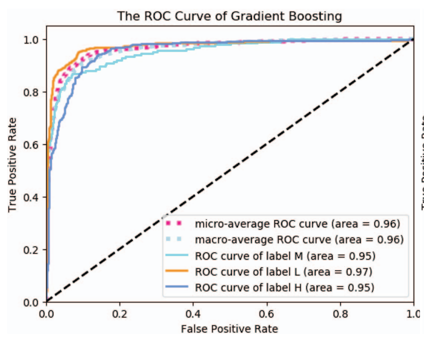
(d) Random Forest



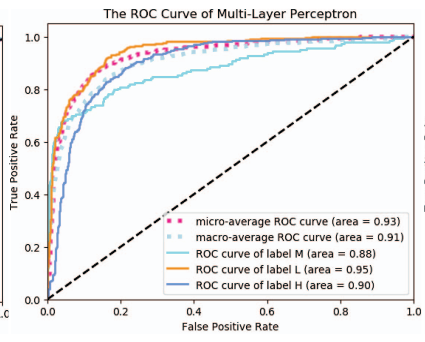
(e) Decision Tree



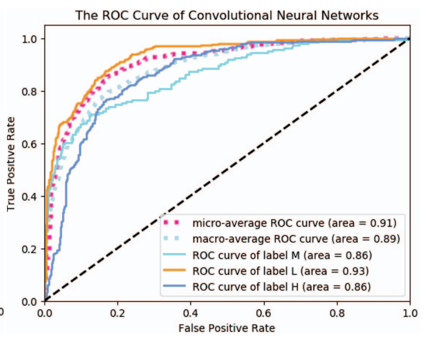
(f) Support Vector Machine



(g) Gradient Boosting



(h) Multi-Layer Perceptron



(i) Convolutional Neural Networks

Fig. 4. ROC Curves of different classification algorithms

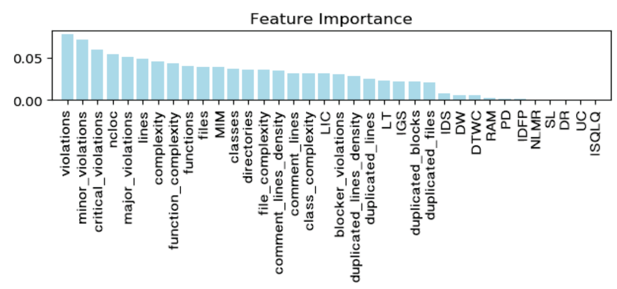


Fig. 5. The importance of Java static metrics and Android code smell metrics

this paper, we only retrieved some code smells which are well defined and examined in [12].

Internal Validity. In this paper, we used two widely-used tools SonarQube and aDoctor to extract Java static code metrics and Android-specific code smells. Some unnoticeable errors introduced by these two tools may to some extent threaten our results. Besides, we used Androrisk to help label an application with a security risk level. To ensure that our labels are reasonable, we manually checked Android files based on the results of Androrisk. However, we were not able to check all Android files and some other unknown factors may also affect the security of an application. We cannot claim that our labels are 100 percent correct.

External Validity. In our study, all applications used are from Github. We cannot guarantee that our conclusions apply to other OSS platforms or industrial communities. However, considering that our applications are from various domains, and are of different scales, we believe that our study still projected some valuable insights about code smells and security risk of applications.

V. Related Work

Many studies have been conducted to explore Android vulnerability, including permission mechanisms, system security, application security, and privacy. Gibler et al. [7] developed a tool called AndroidLeaks to detect security information leakage problems. Chess et al. [31] proposed an approach to reveal security flaws in source code. Wu et al. [32] provided a static analyst paradigm to detect Android malware. Kumar et al. [33] also built a model based on GIST features to detect Android malware. Bose et al. [34] trained Support Vector Machine (SVM) classifiers to distinguish malicious behaviors from normal behaviors of applications. Shabtai et al. [35] presented a behavior-based detection framework for Android-powered mobile devices. The authors continuously monitored various features and events from mobile devices and then applied machine learning algorithms to classify data as normal (benign) or abnormal (malicious). In this paper, our focus is on one aspect of Android vulnerability, i.e., the application security problem. We tried to extract useful features from static analysis to predict the security risk level of an Android application.

Androrisk [36] is most related to our work. Androrisk [36] is a tool that aims to give risk scores to Android applications using fuzzy logic. Within Androrisk, more sensitive permissions (i.e. access to the location, SMS messages, or payment systems) and functionalities which are more dangerous (i.e. shared libraries, use of cryptographic functions, the reflection API) are assigned with higher risk values. One problem with Androrisk is that it could neither provide accurate risk scores nor provide effective suggestions for developers to reduce security risks of applications, since most applications with low-security scores (measured by Androrisk) by using some sensitive permissions and dangerous functionality are actually not malicious. In this paper, we introduced the Android code smells to compensate for the shortcomings of Androrisk. Android code smells are closely related to the security of an application and could be eliminated by code refactoring. Our focus is to provide code modification suggestions for developers when their application is at high risk, so as to help them develop much safer Android applications.

Existing studies have found that various kinds of software metrics and code smells are related to software quality [37]–[39]. For example, software metrics (including object-oriented (OO) metrics [40], change metrics [38], [41], etc.) have been widely used for fault prediction [37]. And code smells (which were first proposed by Holschuh et al. [42]) were found to be effective in defect prediction on Java projects. Related to Android applications, Mannan et al. [39] studied some Android-specific code smells; they found that the distribution of code smells in Android applications were different from that of Java desktop applications. Hecht et al. [37], [43] found that Android code smells occurred more frequently than other code smells. Currently, Android code smells have not been explored in security risk prediction tasks. We are the first to consider exploiting Android code smells to facilitate security risk level prediction of Android applications.

Barrera et al. [44] used 27 function-level metrics to examine the correlation between software internal quality and security vulnerabilities. Rahman et. al. [3] evaluated how static code metrics such as the number of lines, functional complexity, and McCabe's complexity could be used to predict security risk for Android applications. Alenezi et al. [15] did an empirical study on the impact of static code metrics in predicting security vulnerabilities within Android applications. Unlike the above studies, we introduced a new kind of Android-specific code metrics, i.e., Android code smells, to predict security risk for Android applications. We combined Java code metrics and Android code smells to build prediction models, and further addressed some practical problems such as imbalanced classes. Various machine learning methods were also compared to each other during the evaluation process.

VI. Conclusion

In this paper, we proposed to make use of Android-specific code smells to help predict security risk levels for Android applications. More specifically, we incorporated Android code smells into Java code metrics (aka. hybrid metrics) and applied a variety of learning methods upon those hybrid metrics to build relevant prediction models. A data set of 3,680 Android applications from GitHub was constructed to evaluate the effectiveness of our models. We found that the learning algorithm Random Forest outperformed other learning methods, achieving an AUC of 0.97. We further analyzed how helpful individual Android code smells were in identifying security risk levels of Android applications. We found that Android code smells such as *member ignoring method (MIM)* and *leaking inner class (LIC)* were more important in predicting security risk levels of Android applications.

In the future, we plan to further improve our approach by identifying and exploiting more types of Android code smells. We also plan to investigate how Android-specific code smells would affect other software tasks such as software maintenance like code refactoring.

Acknowledgements

This work is supported partially by National Key R&D Program of China (2018YFB1403400), National Natural Science Foundation of China(61802171), Fundamental Research Funds for the Central Universities(14380021), and the Key Project of Natural Science Research in Anhui Higher Education Institutions(KJ2019ZD67).

References

[1] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 73–84.

[2] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri, "A study of android application security." in *USENIX security symposium*, vol. 2, 2011, p. 2.

[3] A. Rahman, P. Pradhan, A. Partho, and L. Williams, "Predicting android application security and privacy risk with static code metrics," in *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*. IEEE, 2017, pp. 149–153.

[4] Y. Li, X. Lu, and J. Fang, "Android system security vulnerability and response measures," in *The 2nd Information Technology and Mechatronics Engineering Conference (ITOEC 2016)*. Atlantis Press, 2016.

[5] 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.

[6] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." in *USENIX security symposium*, 2012, pp. 569–584.

[7] 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.

[8] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1043–1054.

[9] Y. J. Ham, D. Moon, H.-W. Lee, J. D. Lim, and J. N. Kim, "Android mobile application system call event pattern analysis for determination of malicious attack," *International Journal of Security and Its Applications*, vol. 8, no. 1, pp. 231–246, 2014.

[10] M. Ghafari, P. Gadient, and O. Nierstrasz, "Security smells in android," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2017, pp. 121–130.

[11] P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz, "Security code smells in android icc."

[12] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers," in *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*, vol. 2014, 2014.

[13] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, 2016, pp. 59–69.

[14] Vladimir, Svetnik, y, Liaw, Christopher, Tong, J., Christopher, Culberson, and R. and, "Random forest: A classification and regression tool for compound classification and qsar modeling," *Journal of Chemical Information & Modeling*, 2003.

[15] M. Alenezi and I. Almomani, "Empirical analysis of static code metrics for predicting risk scores in android applications," in *5th International Symposium on Data Mining Applications*. Springer, 2018, pp. 84–94.

[16] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 487–491.

[17] M. Syakur, B. Khotimah, E. Rochman, and B. Satoto, "Integration k-means clustering method and elbow method for identification of the best customer profile cluster," in *IOP Conference Series: Materials Science and Engineering*, vol. 336, no. 1. IOP Publishing, 2018, p. 012017.

[18] M. Molinara, M. T. Ricamato, and F. Tortorella, "Facing imbalanced classes through aggregation of classifiers," in *14th international conference on image analysis and processing (ICIAP 2007)*. IEEE, 2007, pp. 43–48.

[19] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on software engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[20] Moore and B., "Principal component analysis in linear systems: Controllability, observability, and model reduction," *IEEE Transactions on Automatic Control*, vol. 26, no. 1, pp. 17–32, 1981.

[21] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[22] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *Acm Sigkdd Explorations Newsletter*, vol. 6, no. 1, pp. 20–29, 2004.

[23] G. Czibula, Z. Marian, and I. G. Czibula, "Software defect prediction using relational association rule mining," *Information Sciences*, vol. 264, pp. 260–278, 2014.

[24] D. C. Montgomery and G. C. Runger, *Applied statistics and probability for engineers*. John Wiley & Sons, 2010.

[25] W. G. Hopkins, *A new view of statistics*. Will G. Hopkins, 1997.

[26] D. I. Sjoberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, no. 8, pp. 1144–1156, 2013.

[27] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[28] C. Leistner, A. Saffari, J. Santner, and H. Bischof, "Semi-supervised random forests," in *2009 IEEE 12th International Conference on Computer Vision*. IEEE, 2009, pp. 506–513.

[29] R. Liu, J. Cheng, and H. Lu, "A robust boosting tracker with minimum error bound in a co-training framework," in *2009 IEEE 12th International Conference on Computer Vision*. IEEE, 2009, pp. 1459–1466.

[30] B. Zhang, G. Ye, Y. Wang, J. Xu, and G. Herman, "Finding shareable informative patterns and optimal coding matrix for multiclass boosting," in *2009 IEEE 12th International Conference on Computer Vision*. IEEE, 2009, pp. 56–63.

[31] B. V. Chess, "Improving computer security using extended static checking," in *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*. IEEE, 2002, pp. 160–173.

[32] 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.

[33] A. Kumar, K. P. Sagar, K. Kuppusamy, and G. Aghila, "Machine learning based malware classification for android applications using multimodal image representations," in *Intelligent Systems and Control (ISCO), 2016 10th International Conference on*. IEEE, 2016, pp. 1–6.

[34] A. Bose, X. Hu, K. G. Shin, and T. Park, "Behavioral detection of malware on mobile handsets," in *Proceedings of the 6th international conference on Mobile systems, applications, and services*. ACM, 2008, pp. 225–238.

[35] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""andromaly": a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2012.

[36] K. Dunham, S. Hartman, M. Quintans, J. A. Morales, and T. Strazzere, *Android Malware and Analysis*. Auerbach Publications, 2014.

[37] G. Hecht, L. Duchien, N. Moha, and R. Rouvoy, "Detection of anti-patterns in mobile applications," in *COMPARCH 2014*, 2014.

[38] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

[39] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*. IEEE, 2016, pp. 225–236.

[40] S. R. Chidamber and C. F. Kemerer, *Towards a metrics suite for object oriented design*. ACM, 1991, vol. 26, no. 11.

[41] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 181–190.

[42] T. Holschuh, M. Pauser, K. Herzig, T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects in sap java code: An experience report," in *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, 2009, pp. 172–181.

[43] G. Hecht, "An approach to detect android antipatterns," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 766–768.

[44] H. Alves, B. Fonseca, and N. Antunes, "Software metrics and security vulnerabilities: Dataset and exploratory study," in *Dependable Computing Conference (EDCC), 2016 12th European*. IEEE, 2016, pp. 37–44.