Comprehensive Behavior Profiling for Proactive Android Malware Detection

Britton Wolfe[1], Karim O. Elish[2], and Danfeng (Daphne) Yao[2]

[1] Information Analytics and Visualization Center,
Indiana Univ.-Purdue Univ. Fort Wayne (IPFW),
2101 E. Coliseum Blvd., Fort Wayne, IN 46825 USA
wolfeb@ipfw.edu
[2] Department of Computer Science,
Virginia Tech., 2202 Kraft Dr., KWII, Blacksburg, VA 24060 USA
danfeng@vt.edu, kelish@vt.edu

Abstract. We present a new method of screening for malicious Android applications that uses two types of information about the application: the permissions that the application requests in its installation manifest and a metric called percentage of valid call sites (PVCS). PVCS measures the riskiness of the application based on a data flow graph. The information is used with machine learning algorithms to classify previously unseen applications as malicious or benign with a high degree of accuracy. Our classifier outperforms the previous state of the art by a significant margin, with particularly low false positive rates. Furthermore, the classifier evaluation is performed on malware families that were not used in the training phase, simulating the accuracy of the classifier on malware yet to be developed. We found that our PVCS metric and the SEND_SMS permission are the specific pieces of information that are most useful to the classifier.

Keywords: android, malware, machine learning, mobile security.

1 Background

The Android operating system continues to gain market share among smart phone users across the world. At the end of 2013, it had reached over 50% market share in the United States and Great Britain and over 70% in Germany and China [22]. In all four countries, Android gained more than 4% market share over the previous year. With an increase in market share also comes an increase in the attention of malware developers. There are hundreds of malicious applications in the official and alternative Android marketplaces [15]. This work presents a new way of detecting malicious Android applications, resulting in higher accuracy than previous methods.

Our technique combines two very different types of information about Android applications. The first one is the set of permissions that the application requests when it is installed (Section 2.2). The second, percentage of valid call

S.S.M. Chow et al. (Eds.): ISC 2014, LNCS 8783, pp. 328–344, 2014.

sites (PVCS) (Section 2.3), is a measure of an application's riskiness, calculated from its data dependence graph. While each kind of information is useful on its own, when combining them, we are able to detect over 83% of malware with only 1% false positive rate, a significant improvement over previous work (Section 3.2).

1.1 Malware Classification

Traditional methods for detecting malware rely upon recognizing a specific signature that has been previously identified as belonging to a specific, known malware. A limitation of this approach is that it cannot recognize previously unknown malware. In contrast, heuristic-based or machine learning methods learn general rules and patterns from examples of malware and clean files, which are then used to automatically recognize previously unseen malware. The capability of identifying unseen new malware is important for realizing proactive defense of the mobile infrastructure.

When using machine learning, each application is represented as a vector of feature values, which can come from dynamic analysis or static analysis. Dynamic analysis involves running the application in a sandbox and recording information about its behavior, such as battery and network usage [3]. Static analysis uses features extracted without running the application, such as the list of permissions that the application requests upon installation or information about the control flow of the program (e.g., 15). Both types of analyses are useful and provide complementary insights about applications' behaviors. Our work uses static analysis.

1.2 Related Work: General Security

Machine learning techniques have been widely adopted in the computer security literature since the work by Lee et al. [12]. Equipped with domain knowledge, the methods extract domain specific features based on empirical observations of malicious programs or traffic patterns.

For example, solutions described by Cova et al. [6] use binary classification techniques to identify malicious Javascript code on the web. The features they extracted from malicious code include the presence of redirection and obfuscation. Xie et al. [24] used a Bayesian network to infer abnormal network traffic patterns. Besides classifying programs and network traffic, learning-based security research also includes database intrusion detection [19] and SMS/social network spam detection [20].

1.3 Related Work: Android Malware

Researchers have applied both static [2, 4, 9] and dynamic [13] approaches to malware detection on Android devices. The approaches differ in the features extracted and the classification algorithms employed, leading to varying degrees of

success. The data sets employed by the researchers were also of different qualities, ranging from just a handful of malware that the researchers created themselves up to data sets with hundreds of examples pulled from live marketplaces.

Schmidt et al. [17] used a data set of ELF files. It consisted of approximately 240 malware which targeted Linux systems (i.e., not specifically designed for the mobile ARM architecture), and less than 100 Linux system commands from an Android device. They used static analysis to construct binary features, one for each function called by any file in the data set. That information was extracted using `readelf`. They applied three classifiers (rule inducer, nearest neighbor, and decision tree) to a few subsets of the features. All of their configurations that achieved 80% or higher detection rate (i.e., true positive rate) also suffered a false positive rate over 10%.

Burguera et al. [5] proposed the CrowdDroid system for identifying a specific type of malware: repackaged malware. Repackaged malware is created by taking a benign application and repackaging it with additional malicious code. The CrowdDroid approach uses dynamic features. A central system collects the frequencies of several system calls from several users running the application on different devices. It then uses k-means clustering with $k = 2$ to cluster the results, with the goal of separating the benign instances of the application from the malicious (repackaged) instances. Their experiments used only four author-created malware and two real malware. While CrowdDroid successfully identified all of the author-created malware, it produced a 20% false positive rate on one of the two real malware (the more substantial application of the two).

Shabtai et al. [18] also used a small number of fabricated malware (i.e., four applications) to test their Andromaly system, due to a lack of real malware at that time. They used 88 hand-designed dynamic features, including memory page activity, CPU load, SMS message events, network usage, touch screen pressure, binder information, and battery information, among others. They pared down the features using the information gain and Fisher scores for each individual feature, selecting the features with the best scores. Then they applied several classifiers: decision trees, naïve Bayes, Bayes nets, histograms, k-means, and logistic regression. Even on a synthetic data set, their best configuration—naïve Bayes after using Fisher score to select 10 features—still had over 10% false positive rate, with approximately 88% accuracy.

Later research had the advantage of access to more actual malware. Like Andromaly, Amos et al. [3] used hand-selected dynamic features (e.g., memory, CPU, binder information), but evaluated performance on a larger data set: 1330 malware and 408 benign applications. They compared random forests, naïve Bayes, multilayer perceptrons, Bayes nets, logistic regression, and decision trees. As with the previously mentioned work, their methods suffer from a high false positive rate: over 15% for all of their configurations. Their accuracy was 95% on new traces from applications included in the training set, but no higher than 82% on traces from applications that were not included in the training set.

Sanz et al. [16] used a simple feature set: the permissions and features of the device that the application requests upon installation. They are listed in the

downloaded application's manifest, so these features are extracted with static analysis. Their data set consisted of 357 benign and 249 malicious applications. They tried several classifiers: logistic regression, naïve Bayes, Bayes nets, support vector machines with polynomial kernel, k-nearest neighbors, decision trees, random trees, and random forests. As with other work, the false positive rate remains stubbornly high: their false positive rate is never below 11%, and even that classifier only detects 45% of the malware. The best overall accuracy was 86%, using random forests.

Sahs and Khan [15] tried a substantially different approach, training a 1-class support vector machine on benign applications in order to detect malware as anomalies. They used a custom kernel that combines permissions information with control flow graph information, both of which come from static analysis. However, their false positive rate is nearly 50%, making their method untenable.

Wu et al. [23] report much better results—false positive rate below 1% and accuracy of 98%—but they only report on the training set error. Without evaluating on a testing set or using cross-validation, the good results are likely due to overfitting[1] instead of a model that generalizes well to unseen malware.

Peng et al. [14] explored the use of different probabilistic generative models for scoring the risk of different Android applications. They used the permissions requested by the application as the binary features (i.e., static analysis). Each model estimates the probability that an application would request those permissions. Each model is trained on several thousand applications from the marketplace, which the authors assume to all be benign. When a new application requests permissions that have a low probability according to the model, it is flagged as unusual or high risk. The probabilistic models range in complexity from simple naïve Bayes through a hierarchical mixture of naïve Bayes models. They used 378 malware applications mixed with different subsets of the benign set to calculate cross-validation error. The hierarchical mixture of naïve Bayes models performs the best, detecting 78% of malware with a false positive rate of 4%. The simpler models also do well, achieving close to the same results.

1.4 Receiver Operating Characteristics (ROC) Curve

For classifiers that produce probability estimates—e.g., there is a 72% chance this application is malware—instead of just a yes/no decision, the aggressiveness of the overall system can be adjusted without modifying the classifier itself. To do this, one simply adjusts the probability threshold at which an application is declared malware. When the threshold is 0.0, everything is declared malware (i.e., the most aggressive classifier). On the other extreme, when the threshold is 1.0, nothing is declared malware. The default threshold is 0.5, picking the

[1] Overfitting is a common problem in machine learning applications where the algorithms memorize characteristics specifically of the training examples instead of general trends. When evaluated on the training data, the algorithm uses those characteristics to re-recognize *the same examples*. This gives a false sense of accuracy, since the real evaluation should be on examples other than the training examples.

Table 1. Summary of related work reporting moderate or low false positive rates. TPR numbers read from a plot are approximate, indicated by ≈.

Citation	AUC	TPR values for FPR $\leq x$					Limitations
		$x = 0.01$	$x = 0.02$	$x = 0.05$	$x = 0.10$	$x = 0.15$	
Schmidt et al. [17]	-	0.77	-	-	0.99	1.00	ELF files only
Shabtai et al. [18]	0.913	-	-	≈ 0.967	-	0.847	author-created malware
Sanz et al. [16]	0.920	-	-	-	-	0.50	
Peng et al. [14]	0.954	< 0.5	≈ 0.59	≈ 0.79	≈ 0.87	≈ 0.90	

most likely category according to the classifier. This ability is important for malware classification because in different situations, different levels of aggressiveness would be appropriate. If one wants very high security, one might pick an aggressive classifier that can detect all of the malware, but also mistakenly flags several benign applications as malware (i.e., high false positives). On the other hand, if the classifier is used as part of a larger security suite, a less aggressive classifier would be preferred, producing fewer false positives.

While there are several ways to measure the quality of a classifier—accuracy, false positive rate, precision, etc.—the receiver operating characteristic curve (ROC curve) illustrates the trade off between false positives and false negatives as one moves from a conservative classifier (i.e., nothing is malware) to an aggressive classifier (i.e., everything is malware). (See Figure 1 for examples of ROC curves.) One can examine the curves in several different ways. The most concise is to calculate the area under the curve (AUC), which summarizes the quality of the classifier at all different levels of aggressiveness. An AUC of 1.0 is optimal, representing a perfect classifier.

One can also examine specific points on the ROC curve to find what fraction of malware can be detected—the true positive rate or TPR—when limiting the false positive rate (FPR) below some threshold. For example, one might want no more than 2% FPR in a particular system, so looking at the TPR value on the ROC curve when FPR=0.02 will estimate the detection rate of such a system.

1.5 Summary of Related Work

Table 1 summarizes the results from previous work. The table lists the TPR for different values of the FPR, along with the AUC. When the ROC is not reported in the given work, the closest FPR column is filled in. The best classifiers from each publication that meet the FPR limit are reported, and the best AUC is reported. Thus, the different columns may represent different classifiers. Publications where all of the FPR values were above 0.15 are omitted. As noted in Section 1.3, there are many factors that influence the results, such as the makeup of the data set and whether or not applications as a whole are classified (static analysis) or execution traces from applications are classified (dynamic analysis). Thus, this table alone is an oversimplification of the results, but it highlights the difficulty in achieving decent detection rates at FPR of 0.02 or less.

2 Methods

Our work utilizes a static analysis feature, called *percentage of valid call sites (PVCS)* [8], described in Section 2.3. We also examine the most common features used in previous static analysis work: the permissions that the Android application requests upon installation (Section 2.2). We compare classifiers' performance when trained using PVCS with classifiers trained using the permissions features, as well as a combination of the two. Adding the PVCS information to the permissions information leads to classifiers that are substantially better than the previous state of the art (Section 3.2).

2.1 Learning Process Overview

For each of the three feature sets—permissions, PVCS, and the combination of both—we use the following learning process. To begin, the vector of feature values for each application in the training set is calculated. Then the set of vectors is given to several classifier learning algorithms. We compared five classifiers: support vector machines (SVMs), random forests, naïve Bayes, k-nearest neighbors (KNN), and boosted decision trees (J48 with Adaboost). We used the Weka implementation of each classifier [10]. These classifiers represent fundamentally different approaches to classification, each of which has its own strengths and weaknesses. Thus, we evaluate all of these classifiers to find the best kind for classifying Android malware.

Three of the classifiers have hyperparameters that the user selects to tune the classifier performance. We used 10-fold cross-validation on the training set to pick these parameters, selecting the values that led to the highest cross-validation AUC. For support vector machines, we explored values of $C \in \{10^{-3}, 10^{-2}, 10^{-1}, ..., 10^{3}, 10^{4}\}$ and $\gamma \in \{2^{-1}, 2^{-2}, 2^{-3}, ..., 2^{-10}\}$. For the random forests, we explored numbers of trees in $\{16, 32, 64, 128, 256\}$. For k-nearest neighbors, we picked the best value of k from 1 through 7.

After picking the hyperparameters, there is one trained classifier of each type for each feature set (15 total). These classifiers are then evaluated on the test set, including the malware from families not present in the training set. Section 3.2 presents the results of this testing evaluation, but first we describe the feature sets in more detail.

2.2 Permissions Bits

Each Android application is required to list in its installation manifest the permissions that it will need at any point during its execution. That list of permissions is used by the Android system to restrict or allow access to system-wide resources like network connections, contact list, boot notifications, sending or receiving SMS text messages, etc. Thus, the permissions list indicates what system resources an application is allowed to use. In addition to the standard Android system permissions, users can define their own permissions. For example, there are permissions specific to particular hardware manufacturers.

For each application, we compute a vector of bits, each of which represents a particular permission. A 1 indicates that the application requests that permission and a 0 indicates that it does not. Sanz et al. [16] use the same encoding of permissions information as their feature set. Sahs and Khan [15] use the same encoding for system permissions, combining with their own features from user-defined permissions and control flow graphs. Peng et al. [14] also use the same encoding, but only for the 20 most frequently requested permissions. We include all the standard Android system permissions as well as any user-defined permissions.

2.3 Percentage of Valid Call Sites

In addition to using permissions bits, we utilize a statistic of Android applications called *percentage of valid call sites (PVCS)* [8]. Android applications are characterized by intensive user interaction. Researchers [7, 8] found that the majority of the benign applications require user interaction in order to initiate sensitive operations like network access. On the other hand, malicious applications require little to no user interaction before executing sensitive operations. Hence, the PVCS metric is designed to capture the dependence relations between user triggers and sensitive operations. This metric represents the degree of sensitive operations that are authorized by the user. It is a fine-grained metric which provides more in-depth behavior information about the applications, as opposed to Android permission information which does not capture the applications' behavior.

To define the PVCS metric, we first define some other terms: operation, call site, and valid call site. An *operation* is defined as a function call related to network operations, file operations, and telephony services in an application. For example, operations include APIs related to sending/receiving network traffic, sending text messages, and accessing private information such as location information. These are sensitive API calls that we want to examine in order to detect malicious behavior.

A *call site* is defined as one instance of an operation. Each API operation may have one or more call sites in an application. Each call site is checked to determine if it is triggered by user actions by constructing a data dependence graph. A *data dependence graph (DDG)* is a well-known program analysis technique which represents data flows through a program [11]. The DDG is a directed graph representing data dependence between program statements, where each node represents a program statement, and an edge represents the data dependence between two nodes.

Android has a special mechanism called *Intent* to provide communication between applications or components (Activity, Service, Receiver). Therefore, the DDG needs to be augmented in order to obtain the complete set of operations that depend on user triggers through Intent. The Android Intent-based dependence analysis tracks the control flow between Intent-sending methods in intra- and inter-application communication. This Intent-specific control flow analysis

helps to bridge disjoint graph components and captures the data dependence relations across multiple Android components.

The DDG is constructed for each application by utilizing the libraries provided by Soot [1], a static analysis toolkit for Java. Furthermore, the constructed DDG is augmented with the Android Intent-based dependence analysis to get one complete, connected DDG. More implementation details can be found in [8].

After building the DDG, each call site is labeled as valid or not valid. The call site is called *valid* if there is a valid path in the data dependence graph from a user trigger to the call site.

The PVCS metric is defined as follows [8]:

Definition 1. Percentage of Valid Call Sites $PVCS \in [0\%, 100\%]$ *of an application is the percentage of valid call sites out of the total number of call sites across all the operations. Let* k_i *be the number of* valid *call sites for operation* i *and let* l_i *be the number of total call sites for operation* i*. Given the* n *operations used in an application,* $PVCS$ *is computed as*

$$PVCS = \frac{\sum_{i=1}^{n} k_i}{\sum_{i=1}^{n} l_i} \tag{1}$$

For example, assume that there are 10 call sites in an application. If 9 out of 10 call sites are triggered by the user, the PVCS value of the application is 90%. A high PVCS is desirable, as it generally indicates that there are not sensitive operations going on without the user's knowledge.

3 Experiments

3.1 Data Set

We used a collection of 3869 Android applications, which consists of 1433 malicious applications and 2436 benign applications. The malicious Android applications were collected from the VirusShare repository[2] and the Android Malware Genome Project[3] [25]. The benign Android applications are free, real-world applications collected from the Google Play market, covering various application categories. These free applications include different levels of popularity, as determined by the user rating scale. We used two existing malware detection tools [7, 21] to scan the collected free applications. Applications that did not trigger any alerts in those tools are kept in the benign set.

The applications were partitioned into training and test sets. For the clean applications, a random 20% were selected for the test set, with the remainder going into the training set. The malicious applications were split based on the malware family. For each family with just one application, that application was randomly assigned to training or testing. For all the other families, at least one application was assigned to the test set. A few families of varying sizes

[2] `http://virusshare.com/`

[3] `http://www.malgenomeproject.org/`

were completely held out of the training set (Table 5), so we could evaluate the algorithm's accuracy on completely unseen malware families. For each of the other malware families, 20% of the applications were selected for the test set, with the remainder going into the training set. In the end, there were 1948 benign and 1066 malicious applications in the training set, and there were 488 benign and 367 malicious applications in the test set.

3.2 Classification Results

After picking the best classifier of each type (Section 2.1), we evaluated their accuracies on our test set (i.e., applications that were not used at all in the training or parameter selection). The experiments answer the following questions:

- Which of the three feature sets (permissions, PVCS, or both) is best for malware screening?
- When using the best feature set, which classifier type is best for malware screening?

We ran two evaluations of the classifiers, using two different subsets of the test data. They each use all the benign applications in the test set, but they differ in which malware from the test set is used. The "unfamiliar" comparison only uses the malware from families that were *not* represented in the training data. The "familiar" comparison (Section 3.4) uses the other malware (i.e., their families were represented in the training data).

Of the two, the unfamiliar is more important. In reality, we want to detect new malware families that have been created after training on existing malware families. Table 2 presents the results, including AUC and the true positive rate (TPR) for different levels of false positive rate (FPR). The corresponding ROC curves for the best performers are plotted in Figure 1.

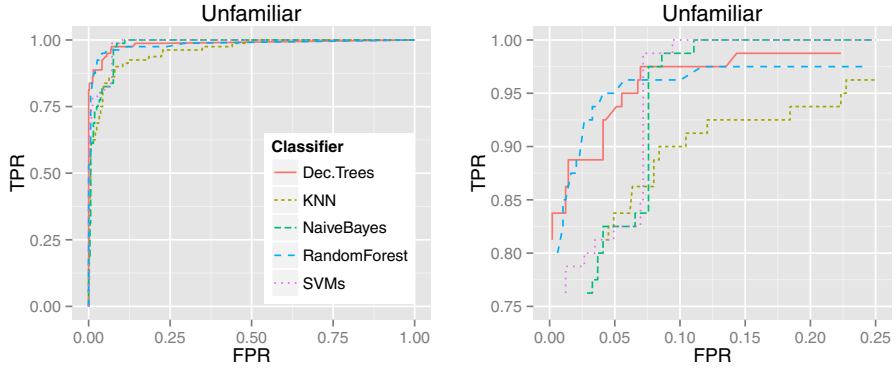
While different classifiers perform the best at different FPR levels, **all of the best performers use both permissions bits and PVCS**. That is, for each FPR level, the model with the best TPR is always one that uses both permissions bits and PVCS. Furthermore, the model with the best AUC also uses the combined feature set.

For practical use, an FPR of even 5% is too high, as it would flag one out of every 20 clean applications as malicious. Thus, the **boosted decision trees classifier trained upon permissions and PVCS is the best option for detecting malware.** It has the highest AUC (0.9850) and the highest TPR for both the FPR=0.01 and FPR=0.02 levels, detecting 83.75% of the malware from unfamiliar families at the FPR=0.01 level. This is in stark contrast to previous work on malware screening of Android APKs, where the TPR at FPR=0.01 is less than 50%, even when testing on malware families used in the training (Table 1, Peng et al. [14]).

Looking at Figure 1 and Table 2, one can see that for FPR values less than 5%, the boosted decision trees and random forest have higher TPR than the other three classifiers by a considerable margin. It is noteworthy that those

Table 2. Evaluation on unfamiliar test data, sorted by the TPR at FPR= 0.01. The best value in each column is highlighted in bold.

Features	Algorithm	AUC	TPR values for FPR= x				
			$x = 0.01$	$x = 0.02$	$x = 0.05$	$x = 0.10$	$x = 0.15$
PVCS	KNN	0.9550	0.1625	0.5875	0.7750	0.9250	0.9750
Permissions	Naïve Bayes	0.9030	0.1875	0.2750	0.5875	0.7125	0.7625
Permissions	SVMs	0.9380	0.2375	0.4750	0.8000	0.8625	0.9000
Permissions	KNN	0.9080	0.4250	0.5125	0.6500	0.8000	0.8125
Permissions	Boosted Dec. Trees	0.9450	0.4750	0.7250	0.8500	0.8750	0.9000
PVCS	Random Forest	0.9580	0.4875	0.6000	0.8125	0.9250	0.9750
PVCS	Boosted Dec. Trees	0.9640	0.5125	0.5125	0.7125	0.8625	0.9875
PVCS	Naïve Bayes	0.9600	0.5375	0.5750	0.6250	0.9000	0.9875
PVCS	SVMs	0.9590	0.5375	0.6250	0.6625	0.7875	0.9875
Permissions	Random Forest	0.9240	0.5500	0.5500	0.8375	0.8750	0.8875
Both	Naïve Bayes	0.9790	0.6125	0.7375	0.8250	0.9875	**1.0000**
Both	KNN	0.9590	0.6250	0.6250	0.8375	0.9000	0.9250
Both	SVMs	0.9840	0.7500	0.7875	0.8250	**1.0000**	**1.0000**
Both	Random Forest	0.9820	0.8125	0.8750	**0.9500**	0.9625	0.9750
Both	Boosted Dec. Trees	**0.9850**	**0.8375**	**0.8875**	0.9250	0.9750	0.9875



(a) Full curves (b) Zoomed curves of the top left corner

Fig. 1. The ROC curves for the classifiers trained on both permissions bits and PVCS, evaluated on the unfamiliar data subset: true positive rate (TPR) versus false positive rate (FPR).

two classification algorithms performed the best, since both are based upon decision trees. These decision tree-based learning algorithms are designed to intelligently select *some* of the features to use for classification, in contrast with KNN, SVMs, and naïve Bayes, which each construct a model using *all* of the features to perform classification. This indicates that PVCS and *some* of the permissions bits are useful for detecting malware, but other permissions bits are not useful. The next section examines this issue in more detail.

3.3 Feature Analysis

Since the boosted decision tree model performed the best, we examined its structure to find insights about what contributed toward its good performance. The boosted decision tree model consists of several decision trees; the overall prediction of the model is a weighted vote of the classifications from the decision trees. Each decision tree consists of several decision nodes, where the value of one particular feature (a permission bit or the PVCS value) is examined. Comparing the feature value to a learned threshold decides which branch of the tree is (recursively) used to make a decision. Leaf nodes are the decision: malware or benign.

In order to determine which features are most useful in screening for malware, we assigned a score to each feature (permission bit or PVCS value) that was used in the model. Simply summing the number of times a feature is used in one of the decision trees would be one option, but it does not account for the fact that nodes higher in the tree are deemed more discriminative by the learning algorithm. Thus, we weight each occurrence of the feature according to its depth in the tree. Furthermore, the scores from each tree are weighted according to that tree's contribution to the overall decision of the classifier; those tree weights come from the boosting learning algorithm. In mathematical form, the score for a feature f is

$$\sum_{x \in N(f)} \frac{1}{d(x)+1} w(x) \tag{2}$$

where $N(f)$ is the set of decision tree nodes that examine feature f, $d(x)$ is the depth of x, and $w(x)$ is the weight of the tree in which x is contained.

The model used 59 different features out of the 387 features in the training data (15%). The features with the top 20 scores are listed in Table 3. The highest scoring feature by far is PVCS. In fact, it was the root node feature (i.e., most informative feature) in half of the decision trees in the model. System permissions, as opposed to user-defined permissions, dominate the list, filling out the top 10 features. The second highest scoring feature, the SEND_SMS permission, scores very high, about 50% higher than the next permission. This is likely indicative of malware that send out text messages without the user's consent.

In addition to its prominence in the boosted decision trees, the importance of PVCS is also seen when considering the classifiers trained only with PVCS information. When comparing them with the classifiers trained only on permissions (Table 2), the PVCS feature generally did better than the permissions features. Specifically, four out of five of the PVCS classifiers have a higher TPR for FPR= 0.01 than the permissions classifiers, with the notable exception of the permissions random forest. Furthermore, all of the PVCS classifiers have higher AUC than any of the permissions classifiers.

Table 3. Top scoring features from the best model (boosted decision trees trained on permissions and PVCS). System permissions have the "android.permission" prefix removed from their names in the table.

Rank	Feature	Score
1	**PVCS**	72.52
2	SEND_SMS	21.07
3	READ_PHONE_STATE	13.79
4	ACCESS_COARSE_LOCATION	11.98
5	RECEIVE_BOOT_COMPLETED	11.45
6	ACCESS_NETWORK_STATE	10.45
7	INTERNET	10.19
8	SET_ORIENTATION	8.91
9	READ_CONTACTS	8.63
10	CAMERA	8.02
11	GET_ACCOUNTS	7.33
12	WAKE_LOCK	7.11
13	com.software.android.install.permission.C2D_MESSAGE	6.84
14	GET_TASKS	6.78
15	READ_SETTINGS	6.65
16	READ_SMS	6.38
17	CHANGE_WIFI_STATE	6.03
18	com.android.browser.permission.READ_HISTORY_BOOKMARKS	5.70
19	INSTALL_PACKAGES	5.59
20	WRITE_EXTERNAL_STORAGE	5.22

3.4 Classification of Known Families

While the ability of the classifier to detect unfamiliar malware families is most important, we also want to verify that the classifier can detect new instances of malware from families upon which it was trained. Table 4 shows the results from evaluating the 15 models on the "familiar" subset of the test data. Figure 2 plots the corresponding ROC curves for the models trained on both permissions and PVCS.

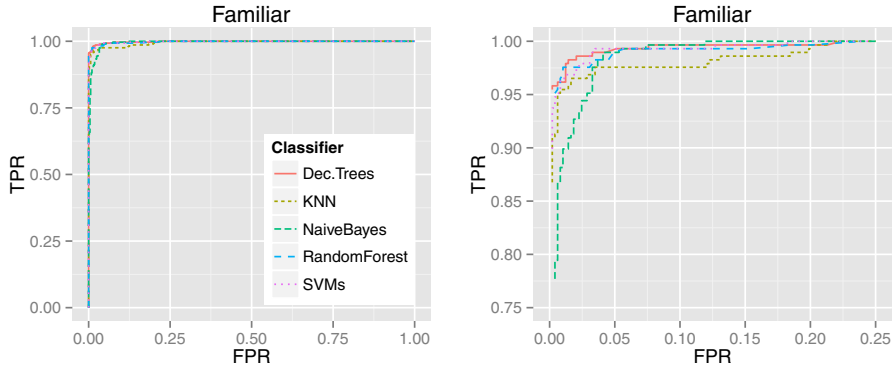
As when testing on the unfamiliar subset, the combination of permissions and PVCS information leads to the best classifiers. Specifically, for each FPR level, the best model uses the combination of feature sets. The best model at screening for unfamiliar malware—the boosted decision trees—also has the best AUC when screening for familiar malware. Furthermore, its TPR for FPR= 0.02 is the best among the 15 models, and the TPR of 0.9617 for FPR= 0.01 is within half of a percent of the best model (0.9652). Thus, that model is not only the best at screening for new malware families, it is also very nearly the best at screening for malware from known families.

3.5 Analyzing the Mistakes

We found 22 applications out of the 855 testing applications (2.6%) are classified incorrectly by the best-performing classifier. This section provides some

Table 4. Evaluation on familiar test data, sorted by the TPR at FPR= 0.01. The best value in each column is highlighted in bold.

Features	Algorithm	AUC	TPR values for FPR= x				
			$x = 0.01$	$x = 0.02$	$x = 0.05$	$x = 0.10$	$x = 0.15$
PVCS	KNN	0.9790	0.1533	0.7875	0.9059	0.9826	0.9930
Permissions	Naïve Bayes	0.9680	0.6411	0.7770	0.8815	0.9373	0.9547
PVCS	Random Forest	0.9860	0.7003	0.7909	0.9129	0.9861	0.9965
PVCS	Naïve Bayes	0.9860	0.7631	0.7909	0.9164	0.9686	0.9965
PVCS	SVMs	0.9820	0.7631	0.7700	0.8223	0.9652	0.9965
PVCS	Boosted Dec. Trees	0.9860	0.7770	0.7770	0.9233	0.9582	0.9965
Both	Naïve Bayes	0.9950	0.8815	0.9268	0.9895	**0.9965**	**1.0000**
Permissions	Boosted Dec. Trees	0.9830	0.9059	0.9164	0.9617	0.9686	0.9756
Permissions	KNN	0.9880	0.9164	0.9443	0.9547	0.9686	0.9721
Permissions	SVMs	0.9840	0.9199	0.9373	0.9512	0.9617	0.9721
Permissions	Random Forest	0.9900	0.9338	0.9443	0.9652	0.9686	0.9721
Both	KNN	0.9950	0.9547	0.9652	0.9756	0.9756	0.9861
Both	Boosted Dec. Trees	**0.9980**	0.9617	**0.9826**	0.9895	**0.9965**	0.9965
Both	Random Forest	**0.9980**	**0.9652**	0.9756	0.9895	0.9930	0.9930
Both	SVMs	**0.9980**	**0.9652**	0.9686	**0.9930**	0.9930	0.9965



(a) Full curves (b) Zoomed curves of the top left corner

Fig. 2. The ROC curves for the classifiers trained on both permissions and PVCS, evaluated on the familiar data subset

insights on the reasons why these applications are misclassified. There are 8 free benign applications misclassified as malware. The main reason behind this is that these applications contain ads/analytics libraries in which sensitive operations have no valid user trigger according to our PVCS calculations. Hence, these benign applications have low PVCS values, resulting in benign applications with profiles similar to malware applications. As an example, `com.jaredshack.androidtimecardfree` is a timesheet application to track the time. It contains the Google ad library in which some sensitive operations, such as `getLatitude()`

and `getLongitude()`, have invalid user triggers. Its PVCS value is 0.3 which is considered low and similar to the values of malware applications.

The reason behind the 14 malware applications misclassified as benign is that these malware applications are repackaged applications. Malware writers bundle malicious code with existing benign applications, producing what is called a repackaged application. Therefore, these malware applications have high PVCS values since most of the sensitive operations inside these applications have valid user triggers according to our calculation of PVCS. As a result, they exhibit similar profiles to benign applications. For example `DroidKungFu` malware is bundled with `com.sniper.awrvvitieetewa` (a game application). Its PVCS value is 0.6 which is considered high, similar to the values of benign applications.

Just as we were able to combine PVCS with permissions information to greatly improve classification accuracy, future work can look for further improvements in accuracy by adding additional information into the feature sets for the classifiers. Our analysis suggests that information about the use of ad libraries or very high similarity with other applications (e.g., repackaged applications) could be important pieces of information to add. In addition, information from dynamic analysis could be added to our static analysis features, providing a more complete picture of the application's behavior. Such work would address limitations of static analysis in general, like code that is dynamically loaded at runtime, the use of native code, or extensive obfuscation. Of course, dynamic analysis has its own weaknesses, such as the difficulty of realistically simulating user behavior in a sandbox environment. Thus, while dynamic analysis could improve detection rates, our work demonstrates that high detection is possible using different kinds of static analysis features.

4 Conclusions and Future Work

We presented a new method for classifying Android applications as malicious or benign that is more accurate than previous work. The method combines two sources of information about the application: the percentage of valid call sites (PVCS) measure and the permissions requested by the application. Both are obtained through static analysis of the application, so there is no need to run the application in order to compute this information. Either set of features alone produced results that were comparable to the previous state of the art.

However, the primary contribution of this work is a demonstration that combining the information from PVCS with the information from permissions results in substantially better performance than previous work. The previous best work detected less than 50% of the malware when limiting false positives to 1% [14], whereas our best classifier detects 83.75% of the malware from unfamiliar families and 96% of the malware from familiar families while maintaining less than 1% false positives.

For future work, we plan to explore other variations of program analysis-based risk features for detecting Android malware, in combination with the permission analysis of applications. We will also perform more extensive evaluation on new applications from the Android Play market.

Appendix: Details of Malware Families in the Data Set

Table 5. Training/testing split for each malware family, listing the number of applications. In addition to the families in the table, the families with only one application were divided as follows: training set included the adrd, andcom, foncy, lovetrap, nickispy, nickyspy, smswatcher, smszombie, youmi, and zsone families; testing set included the airpush, anserverbot, droidcoupon, fakeapp, fakelogo, fakesite, gamblersms, generic, sheridroid, and smsbomber families.

	test	train
bgserv	1	1
crusewin	1	1
gamex	1	1
koogame/koomer	1	1
walkinwat	2	0
wapsx	1	1
asroot	1	2
droiddream	3	0
droidkungfusapp	1	2
ggtracker	1	2
ksapp	1	2
kuguo	1	2
mania	1	2
mobiletx	1	2
opfake	1	2
gingerbreak	1	3
hipposms	1	3
infostealer	1	3
jifake	4	0
imlog	1	4
tapsnake	1	4
wooboo	5	0
adwo	2	4
droidkungfu	2	4

	test	train
smskey	2	4
droidkungfu2	2	5
leadbolt/ropin	2	5
penetho	2	5
yzhc	2	5
fakedoc	2	6
faketimer	2	7
geinimi	2	7
kmin	9	0
pjapps	2	7
smssend	2	7
gingermaster	2	8
fakeplayer	3	8
jsmshider	3	12
zitmo	4	12
droiddreamlight	4	13
droidkungfu4	17	0
golddream	4	13
droidkungfu1	4	15
droidrooter	30	0
droidkungfu3	9	33
plankton	11	41
basebridge	12	45
fakeinst	189	752

References

1. Soot: a Java optimization framework (2012), `http://www.sable.mcgill.ca/soot/`
2. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: Mining API-level features for robust malware detection in Android. In: Zia, T., Zomaya, A., Varadharajan, V., Mao, M. (eds.) SecureComm 2013. LNICST, vol. 127, pp. 86–103. Springer, Heidelberg (2013)
3. Amos, B., Turner, H., White, J.: Applying machine learning classifiers to dynamic android malware detection at scale. In: 2013 9th Int. Wireless Commun. and Mobile Computing Conf. (IWCMC), pp. 1666–1671 (2013)

4. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: Drebin: Efficient and explainable detection of Android malware in your pocket. In: Proc. of 17th Network and Distributed System Security Symposium (NDSS) (2014)
5. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: Behavior-based malware detection system for Android. In: Proc. of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2011, pp. 15–26 (2011)
6. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious JavaScript code. In: Proc. of 19th Int. World Wide Web Conf. (2010)
7. Elish, K.O., Yao, D., Ryder, B.G.: User-centric dependence analysis for identifying malicious mobile apps. In: Proc. of the IEEE Mobile Security Technologies (MoST) Workshop, in conjunction with the IEEE Symposium on Security and Privacy (2012)
8. Elish, K.O., Yao, D., Ryder, B.G., Jiang, X.: A static assurance analysis of Android applications. Technical Report TR-13-03, Virginia Tech (2013)
9. Grace, M.C., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: RiskRanker: scalable and accurate zero-day Android malware detection. In: Proc. of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys), pp. 281–294. ACM (2012)
10. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: An update. SIGKDD Explorations 11 (2009)
11. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12, 26–60 (1990)
12. Lee, W., Stolfo, S.J., Mok, K.W.: A data mining framework for building intrusion detection models. In: Proc. of the 1999 IEEE Symposium on Security and Privacy, pp. 120–132. IEEE (1999)
13. Liu, L., Yan, G., Zhang, X., Chen, S.: VirusMeter: Preventing your cellphone from spies. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) RAID 2009. LNCS, vol. 5758, pp. 244–264. Springer, Heidelberg (2009)
14. Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., Molloy, I.: Using probabilistic generative models for ranking risks of android apps. In: Proc. of the 2012 ACM Conf. on Computer and Commun. Security, CCS 2012, pp. 241–252 (2012)
15. Sahs, J., Khan, L.: A machine learning approach to Android malware detection. In: 2012 European Intelligence and Security Informatics Conf. (EISIC), pp. 141–147 (2012)
16. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P.G., Álvarez, G.: PUMA: Permission usage to detect malware in Android. In: Herrero, Á., et al. (eds.) Int. Joint Conf. CISIS'12-ICEUTE'12-SOCO'12. AISC, vol. 189, pp. 289–298. Springer, Heidelberg (2013)
17. Schmidt, A.D., Bye, R., Schmidt, H.G., Clausen, J., Kiraz, O., Yuksel, K., Camtepe, S., Albayrak, S.: Static analysis of executables for collaborative malware detection on android. In: IEEE Int. Conf. on Commun., pp. 1–5 (2009)
18. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: Andromaly: a behavioral malware detection framework for Android devices. Journal of Intelligent Inform. Syst. 38(1), 161–190 (2012)
19. Srivastava, A., Sural, S., Majumdar, A.K.: Database intrusion detection using weighted sequence mining. Journal of Computers 1(4), 8–17 (2006)
20. Tan, H., Goharian, N., Sherr, M.: $100,000 prize jackpot. Call now!: Identifying the pertinent features of SMS spam. In: Proc. of the 35th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, pp. 1175–1176. ACM (2012)

21. Virustotal: Virus Total (2013), https://www.virustotal.com/
22. Whitney, L.: iPhone market share shrinks as Android, Windows Phone grow (January 2014), http://news.cnet.com/8301-13579_3-57616679-37/iphone-market-share-shrinks-as-android-windows-phone-grow/
23. Wu, D.J., Mao, C.H., Wei, T.E., Lee, H.M., Wu, K.P.: DroidMat: Android malware detection through manifest and API calls tracing. In: 2012 Seventh Asia Joint Conf. on Inform. Security (Asia JCIS), pp. 62–69 (2012)
24. Xie, P., Li, J.H., Ou, X., Liu, P., Levy, R.: Using Bayesian networks for cyber security analysis. In: 2010 IEEE/IFIP Int. Conf. on Dependable Syst. and Networks (DSN), pp. 211–220. IEEE (2010)
25. Zhou, Y., Jiang, X.: Dissecting Android malware: Characterization and evolution. In: Proc. of the IEEE Symposium on Security and Privacy, pp. 95–109 (2012)