DroidScribe: Classifying Android Malware Based on Runtime Behavior

Santanu Kumar Dash*, Guillermo Suarez-Tangil*, Salahuddin Khan*, Kimberly Tam*,
Mansour Ahmadi†, Johannes Kinder*, and Lorenzo Cavallaro*
*Royal Holloway, University of London, United Kingdom
†University of Cagliari, Italy

***Abstract*—The Android ecosystem has witnessed a surge in malware, which not only puts mobile devices at risk but also increases the burden on malware analysts assessing and categorizing threats. In this paper, we show how to use machine learning to automatically classify Android malware samples into families with high accuracy, while observing only their runtime behavior. We focus exclusively on dynamic analysis of runtime behavior to provide a clean point of comparison that is dual to static approaches. Specific challenges in the use of dynamic analysis on Android are the limited information gained from tracking low-level events and the imperfect coverage when testing apps, e.g., due to inactive command and control servers. We observe that on Android, pure system calls do not carry enough semantic content for classification and instead rely on lightweight virtual machine introspection to also reconstruct Android-level inter-process communication. To address the sparsity of data resulting from low coverage, we introduce a novel classification method that fuses Support Vector Machines with Conformal Prediction to generate high-accuracy prediction sets where the information is insufficient to pinpoint a single family.**

I. Introduction

The growth of malware for the Android mobile platform increasingly requires highly scalable methods that can quickly analyze and categorize applications. We can identify two distinct problems: (i) *malware detection*—distinguishing malicious from benign applications and (ii) *malware classification*—classifying malware samples into known families of related malware. While the former immediately benefits users, the latter is a crucial part of forensic analysis, threat assessment, and mitigation planning. In this paper, we focus on the latter—classification of malware into families. We assume that the samples under analysis have been identified as malicious or suspicious by other means.

Machine learning has been successfully applied to both detection and classification. Typically, a classifier is trained on a labeled dataset, using a set of features that describe syntactic or semantic properties of the applications. The structure of Android apps provides rich syntactic information like method and package names, permissions, and configuration files. As a result, statically derived features have allowed to detect and classify Android applications with high accuracy [1, 5, 29].

However, there is evidence that with growing popularity of static malware detection and analysis, Android malware increasingly begins to adopt obfuscation and evasion techniques that break static methods [23, 28]. The same trend has already been witnessed on desktop malware where the use of runtime packers and other types of obfuscation has become the norm. We therefore take the position that Android malware analysis should increasingly focus on runtime behavior, which is independent of any syntactic artifacts and is visible in managed and native code alike. Intuitively, malware samples with similar malicious behavior should be classed as being in the same family. As of yet, it is unknown whether the observation of runtime behavior of Android apps allows to achieve classification results comparable to static analysis.

In this paper, we focus on classifying Android malware into families using runtime behavior derived from system calls observed during dynamic analysis. We exclusively focus on dynamic analysis, to provide a point of comparison that is completely dual to existing static approaches. We face three main challenges specific to this setting: (i) to be resilient to obfuscation and the use of native code, the dynamic analysis should operate at the level of system calls where high-level application semantics are obscured; (ii) malware samples may fail to exhibit sufficient behavior for reliable classification, e.g., because of deactivated command and control infrastructure; (iii) different malware families may exhibit characteristics that are too similar for a classifier to pick up the differences.

We first address the problem of finding the right level of abstraction and context for effective classification based on system calls. In the desktop realm, system calls have been used extensively to detect [20], classify [2], and cluster [15] malware with high accuracy. Intuitively, system calls are well-suited to characterize process behaviors on traditional desktop operating systems [7], since they are required for external effects. For Android, the situation is far from clear, because the system call profile of Android apps is very different from that of desktop applications. In particular, much of the Android-specific behavior is exhibited through the same `ioctl` system call that is dispatched to the *Binder* kernel driver, which implements Android's main mechanism for inter-process and inter-component communication.

We therefore use a dynamic analysis framework for Android that allows lightweight virtual machine introspection [25]. We generate features at different levels, including pure system calls, decoded Binder communication, and abstracted behavioral patterns. We feed this extracted data into a Support Vector Machine [3] (SVM)-based multi-class classifier. After the classifier has been trained on a training set labeled with malware family names, it is able to classify malware in the test set (new malware in an operational setting) into families. We evaluate our approach on malware from the Android Malware Genome Project [31] and the Drebin

†Mansour Ahmadi's work was conducted while visiting Royal Holloway.

dataset [1], which add up to 5,246 malware samples. Our results show that parsing Binder invocations and abstracting the raw system call data into high-level behaviors significantly improves classification accuracy.

The second and third problems are due to observing malware samples with sparse runtime behavior and selecting features that map to multiple families, respectively. Both problems lead to poor classification accuracy. In this paper, we aim at providing a classification system that is robust for even sparse runtime behavior. We consider improvements to the behavior extraction and app stimulation to be an orthogonal problem. We statistically evaluate the decisions made by the SVM-based classifier and show that misclassifications are often a consequence of SVM being forced to make a choice between classes that were not well distinguishable during training. To solve this issue, we suggest to use Conformal Prediction [27] to improve the accuracy of SVM. Conformal Prediction allows to predict a *set* of best matches instead of being forced to decide on a single one, and it therefore can significantly improve the accuracy in the presence of sparse behavior profiles. In an operational setting, we automatically identify the cases where the results for SVM appear statistically unreliable and selectively invoke Conformal Prediction for an optimal classification. Overall, the main contributions of this paper are:

- We present a framework and experimental results for multi-class classification of the runtime behavior of Android malware. To the best of our knowledge, this is the first work to perform multi-class classifications of Android malware using a purely dynamic approach.
- We use a statistical mechanism to evaluate classification *quality* as proposed in [12] and show that single-choice classification algorithms can be unreliable for sparse behavior profiles.
- We introduce a new approach to refine SVM classification by selectively applying Conformal Prediction to compute sets of matches whenever the SVM classifier does not achieve acceptable confidence. Our tunable framework improves classification accuracy from 84% to 94% even on sparse behavior profiles.

The remainder of the paper is organized as follows: We begin with an overview of the system used for gathering behavior profiles from Android applications (§II). We then present three machine learning techniques for classifying behavior (§III) and evaluate them on our system (§IV). Finally, we discuss limitations (§V), review related work (§VI), and conclude (§VII).

II. Tracing Android Behavior

In this section, we discuss how we generate the behavioral data used for classification.

A. System Overview

DroidScribe uses CopperDroid [25] as its dynamic analysis component, which runs apps in a sandbox, records system calls and their arguments, and reconstructs high-level behavior. CopperDroid provides full access to the arguments of all transactions going through the Binder mechanism for inter-process and inter-component communication [6]. The introspection is lightweight, however, and designed to be directly applicable to all Android OS versions without requiring any modifications to the Android system.

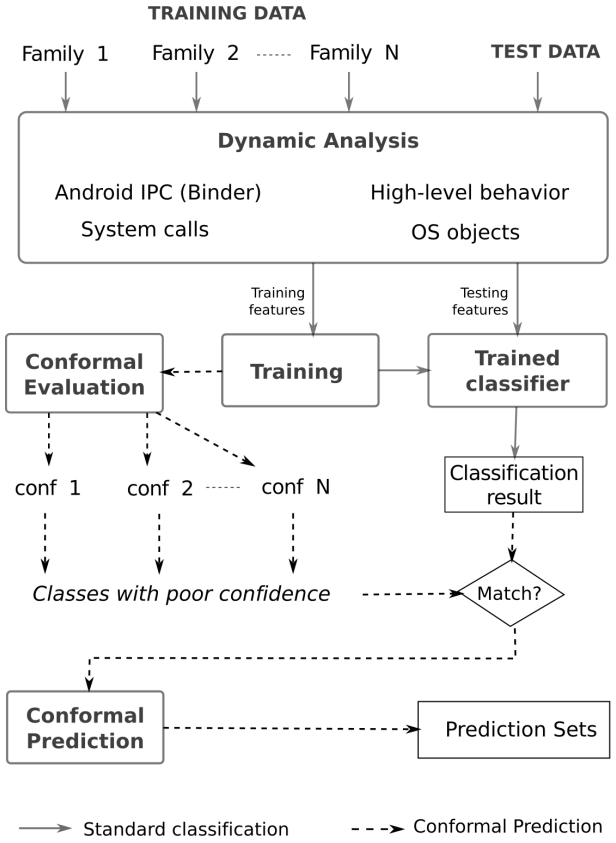


Fig. 1: System overview. Android malware is executed and monitored in an emulator. High-level behaviors are extracted from system call dependencies and Binder methods before being preprocessed and passed to the classifier and the predictor. Classification decisions on the training dataset are evaluated using Conformal Evaluation [12].

Figure 1 gives an overview of DroidScribe and its dynamic analysis and machine learning components (§III). Malware for training the classifier is passed to CopperDroid for dynamic analysis, which captures system calls and abstracts high-level behaviors (details below). During the following preprocessing stage, DroidScribe selects certain behaviors and processes them for classification (see §III-A). It then assesses the classification accuracy, and, depending on the outcome, applies Conformal Prediction (see §III-B) instead of SVM for selected classes (see §III-C).

B. High-level Behavior Extraction

From the trace of system calls and arguments, CopperDroid reconstructs high-level behavior, which encompasses both traditional OS operations (e.g., process creation, file creation) and Android Binder-related methods (e.g., sending SMS, IMEI access, Intent communications). The high-level behaviors specific to Android are reconstructed through deep inspection of Binder transactions, which appear as `ioctl`

Feature Set	Contained Details
$S1$ Network Access	IP, port, and network traffic size
$S2$ File Access	File name/type, name classes
$S3$ Binder Methods	Method name and parameters
$S4$ Execute File	File name/type, user permissions

TABLE I: Behavioral classes and details extracted by CopperDroid. A subset of this information is used as features for our classification framework.

system calls. In addition, CopperDroid conflates sequences of related and interdependent system calls to derive a single high-level behavior, such as file system access or network access. The classes of high-level behaviors include network accesses, file accesses, Binder methods, and file execution (see Table I). When multiple system calls are condensed into single behaviors, the arguments and return values of the individual calls are used to enrich the behaviors with additional details.

For classification, we use the classes of high-level behaviors currently generated by CopperDroid, several of which represent a series of system calls. It is important to note here that although CopperDroid reconstructs the arguments for the high-level classes as well, we do not use them for our classification. We do this to avoid overfitting our model with fine-grained information about the arguments. Having said that, we intend to devise more abstract and meaningful representations of the arguments as a part of future work. Details of the classes, numbered from S1 to S4, are:

S1: Network Access. Malware often establishes network connections to external entities. Each network access behavior represents a sequence of system calls, normally beginning with `connect`, followed by occurrences of `sendto`. By correlating the parameters of these calls, the behavior is enriched with the target IP address and the amount of transferred data.

S2: File Access. File access is reconstructed from system calls using the same file descriptor. Each chain of dependent system calls begins with a new file descriptor returned by `open`, continues with calls like `write`, and terminates with a call to `close` or `unlink` (related system calls, such as `dup`, are considered part of the chain). Using these chains of system calls, CopperDroid is able to fully recreate the actual file for further analysis.

S3: Binder Methods. CopperDroid effectively reconstructs Binder communications from the `ioctl` system calls. Since Binder communications are the principal means of inter-process/inter-component communication, they are the gateway to services from the Android system and enable app-to-app interactions. Consequently, monitoring Binder communications and identifying the invoked method is crucial to modeling the behavior of Android malware.

S4: Execute. There are various files that may be executed within the Android system to run exploits or silently install applications. We differentiate different file executions by breaking down these behaviors by analyzing their parameters. For example, if the parameters include a "pm" followed eventually by an "install" and a file name, this is an indication of an application being installed silently without the user's permission. Furthermore, there a multiple ways to execute the same file, e.g., the same application installation can be done with different arguments; grouping all of them into the same class of behavior with the same outcome makes our method less susceptible to misdirection.

In the remainder of the paper we refer to all four types as high-level behaviors; in particular, with this term we always include Binder methods.

III. Classification of Android Malware Families

We now present three approaches to malware classification. We start with briefly recalling SVM (§III-A), and we explain Conformal Prediction and how it can be used with metrics based on SVM (§III-B). Finally, we present a hybrid approach that combines the speed of SVM with the improved accuracy of Conformal Prediction (§III-C).

A. Standard Classification

Support vector machines (SVM) [3] were initially introduced for the two-class classification problem and later extended for multi-class classification. Given a dataset of samples belonging to different classes, *support vector machines* segregate the given samples using a hyperplane. A hyperplane is the set of points $\mathbf{x}$ that satisfies the relation $\mathbf{x} \cdot \mathbf{w} - b = 0$. Here, $\cdot$ denotes the dot product, $\mathbf{w}$ is the normal to the hyperplane, and $\frac{b}{\|\mathbf{w}\|}$ is offset of the hyperplane from the origin along the normal.

1) Two-Class Support Vector Machines: A training dataset $\mathcal{D}$ consists of a set of tuples $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i$ is a p-dimensional vector of real numbers and $y_i \in \{-1, +1\}$ denotes the class. SVM then separates the two classes by constructing the optimal hyperplane by subjecting $\mathbf{w}$ and b to the following constraints for the two classes:

$$\forall y_i = +1 : \mathbf{x}_i \cdot \mathbf{w} - b \geq +1$$
$$\forall y_i = -1 : \mathbf{x}_i \cdot \mathbf{w} - b \leq -1$$

Complete segregation of the two classes is possible only when the samples are linearly separable. If the samples are not, it is possible to use other separation kernels such as polynomial or radial basis function [17]. Once the hyperplane is established, a decision for the classification of samples from the testing dataset can be obtained by substituting $\mathbf{x}_i$ for the test samples. Since non-linear kernels are computationally expensive and do not scale to the size of our experimental data, we used the linear kernel.

2) Multi–Class Support Vector Machines: There are two main approaches to extend two-class classification to the multi-class case: the one-vs-all approach and the one-vs-one approach. We briefly outline both methods; Hsu and Lin [11] provide an in-depth comparison.

The one-vs-all approach constructs k SVMs for k classes in the dataset—one per class. In particular, the j-th SVM distinguishes the j-th class from a combination of the remaining $k-1$ classes. Labeling samples in class j as +1 and all other samples as -1 gives rise to k decision functions (one for each SVM) of the form $\mathbf{x} \cdot \mathbf{w^1} + b^1, \ldots, \mathbf{x} \cdot \mathbf{w^k} + b^k$. The class of sample i is then chosen according to the decision criterion $class_i = argmax_{j=1\ldots k}(\mathbf{x}_i \cdot \mathbf{w}^j + b^j)$ using the decision functions derived from all k SVMs.

In contrast, the one-vs-one approach constructs SVMs to distinguish each possible pair of classes, i.e., $k(k-1)/2$ SVMs for k classes [13, 8, 14]. After training, the testing is done using a voting system. For each decision function for classes i and j, denoted by $\mathbf{x} \cdot \mathbf{w}^{ij} + b^{ij}$, the sign of the result indicates whether the samples belongs to i or j. If it belongs to i, then the vote for i is increased by 1. Otherwise, the vote for j is increased by 1. At the end of the voting across all $k(k-1)/2$ decision functions, the sample is classified into the class with the highest votes. For our experiments in this paper, we have chosen to apply the one-vs-all approach as it is faster while still giving a better notion of non-conformity scores, which are a crucial part of our statistical classification framework (see §III-B).

3) Classification of Behavior Profiles: Machine learning algorithms normally require data to be presented as vectorial data. Therefore, we must embed the behavioral profiles and system calls from dynamic analysis into a vector space. We construct one feature vector per sample using the overall set of features observed across all samples (represented by S), comprised of reconstructed behaviors. We build a 2D vector space model of size (*number of samples*) $\times$ ($|S|$) which is used as an input to the classifier.

Each malware sample x is mapped to the vector space by constructing its feature vector $f(x)$. The feature vector is constructed by inserting the frequency of each behavioral feature $s \in S$ as observed for x. For a set of malware samples, the mapping f can be formally defined as

$$f : X \to \{0, n\}^{|S|}, f(x) \to (I(x, s))_{s \in S},$$

where the indicator function $I(x, s)$ is defined as

$$I(x, s) = \begin{cases} \sum\limits_{i} [b_i = s] & \text{number of instances } s \text{ in } x \\ 0 & \text{otherwise} \end{cases}$$

and b_i corresponds to the i^{th} behavior observed for sample x.

Therefore the importance, or significance, of a behavior in a sample can be measured by the frequency of its occurrence. For example, after the vector space has been normalized, a frequency of 0 (i.e., $f(x, s) = 0$) shows that behavior has little to no importance in identifying a sample because the sample rarely exhibits the behavior. On the other hand, a behavior with a non-zero feature frequency illustrates that it represents this sample's actions better than the previous one.

B. Classification by Conformal Prediction

In traditional classification, the algorithm must typically choose a single class label per sample. It does not take into account whether the best fit is actually a *good* fit, and it ignores any alternate choices, regardless of their likelihood. If more than one choice of similar high likelihood exists, a traditional classification algorithm is prone to error.

To address these shortcomings, Conformal Prediction (CP) was suggested as a technique to statistically assess how well a sample fits into a family [27]. For the qualitative scoring, CP takes a *non-conformity score* as input. The non-conformity score is a geometric measure of how well a sample fits into a class; e.g., in the case of SVM, this could be derived from the sample's distance to the segregating hyperplane, with samples closer to the hyperplane being less conforming. CP converts this geometric distance to *p-values*, which are a statistical measure of how well a sample fits into a class. This mapping from geometric to statistical space lends CP its flexibility: the user can specify how accurate they want CP to be.

The non-conformity score is typically a real-valued function $A(B, z)$ which measures how different a sample z is from samples of class B. It is used to derive p-values, the proportion of samples in a class with identical, or higher, non-conformity scores. When the p-values for all classes of a sample are written out in descending order, introducing a cut-off level (ρ) selects a potential set of classes ($\mathcal{P}$) to which the sample may belong. All other classes ($\mathcal{P}'$) having p-values less than ρ are treated as rejected options for classification.

CP can give qualitative assessments of how good a prediction set $\mathcal{P}$ identifies a given sample. The first assessment metric is *credibility*, which measures the highest p-value among classes in $\mathcal{P}$. A high credibility score indicates that CP found good matches for the sample and it is unlikely to hail from a new family. The other qualitative metric that CP provides for a prediction set is *confidence*. The confidence level is defined as $1 - p$, where p is the highest p-value for classes in $\mathcal{P}'$ that were filtered out by the cut-off ρ.

As discussed by Jordaney et al. [12], one can interpret the quality of classification by analyzing credibility and confidence scores. For example, if the CP choices have high credibility but the decision has a poor confidence, it implies that other classification options are available and that they have a p-value closer to the chosen options. In such a case, it means that one may need to reconsider either the algorithm or the set of features in order to better discriminate between classes. Also when considering the right set of features, this information could be extremely valuable for security analysts in those cases where two families are related to each other and/or behave alike. Alternatively, if both the credibility and confidence are poor, it demonstrates that the sample does not match any known family and may belong to a new malware family (i.e., zero-day malware).

Another interesting aspect to Conformal Prediction is that the confidence level can be used to obtain a set of predictions, in which case the error rate is (1 − confidence). If we implicitly set a confidence threshold or a *p-value threshold*, we can obtain a set of predicted classes that a sample may belong to. This makes CP a highly desirable proposition: one can always tune p-value thresholds in a bid to achieve perfect classification. The price to be paid is that one must choose the most appropriate option by other means when the p-value threshold returns multiple classes. While this may require manual effort, it obviates the need to consider *all* classes and allows to focus on the few selected by Conformal Prediction. Furthermore, one can reduce or increase the number of classes by changing the p-value threshold based on the desired accuracy level.

SVM also provides probabilities of a sample mapping to a class, and these have been used in the past to detect new malware families [20]. However, the derivation of probabilities is based on Platt's scaling [18], which is a form of logistic regression. Much like other regression techniques, it is sensitive to outliers. It has been shown that the accuracy prediction can be too optimistic or too pessimistic if the probabilities are used

to predict the accuracy—as is done in the case of Conformal Prediction with the confidence threshold [30]. This is because Platt's scaling uses a transformation of the dataset conducted by SVM, whereas CP creates its own transformation based on the actual dataset.

In DroidScribe, we use Conformal Prediction in the following manner. For each sample s that needs to be classified and for each class $\mathcal{C}$ in the training set, the Conformal Prediction trains by including s in the training set as a member of $\mathcal{C}$. Using this procedure, it obtains the distance of all samples in $\mathcal{C}$ (including s) to all the hyperplanes bounding $\mathcal{C}$. These distances are then used as the non-conformity measure for the calculation of p-values as described above. For a given s, this sequence of actions is then repeated by putting s in all classes in the dataset and retraining. Thereby, we obtain a p-value for all classes with s as the test case. The prediction set for s is then the set of classes that are retained after filtering out classes having a p-value lower than the cut-off thresholds.

C. Hybrid Prediction: CP Augmenting SVM

With a flexible confidence level (spanning to 100% with larger prediction sets), CP is a highly desirable algorithm for classifying malware, but generally expensive. For each sample, CP tries to place it in every possible class and computes the non-conformity measure for the sample-class pair by running a standard classification algorithm. To this end, for n samples in a dataset and c possible classes, CP runs a traditional classification algorithm $(n \times c)$ times to obtain the non-conformity measures for all samples for all classes. We use distances from the SVM hyperplane obtained from a one-vs-one multi-class SVM (see previous section) as our non-conformity score. Therefore, in a naive implementation, CP would run the SVM classification $(n \times c)$ times in our case which can be prohibitively expensive for large datasets.

Because of the high cost, we propose to combine CP with SVM and to rely on CP only selectively to improve the accuracy of a standard classification algorithm. We invoke CP only when it is necessary and when SVM does not meet a desired classification quality metric. Interestingly, the classification quality metric for SVM in our case is derived from evaluating SVM decisions with CP itself. This is where the flexibility of the CP comes into play. P-values not only help in classification, but also help evaluate existing classifications by other algorithms.

The non-conformity measure of a sample, with respect to a class, is used as an input to calculate the p-values. If we use the same non-conformity measure for CP as for SVM (i.e., the distance to hyperplane), we can verify whether the decision taken by SVM was reliable. If the decision was unreliable, the class chosen by SVM should have low p-value compared to others. Hence, *confidence* (refer to §III-B) for the SVM decision would be low. During initial training and testing, we evaluate SVM decisions with CP and obtain the average confidence for the true positives for each class. We call this the *class-level confidence*. If a new sample is classified to be in a class that has high class-level confidence during training with SVM, we decide not to refer to the CP for further refinement.

To benchmark class-level confidence scores and use them as a quality metric for invoking the Conformal Prediction, we define a *quality threshold* as the cutoff in class-level confidence below which we invoke CP. The cutoff works in practice in the following manner: if SVM maps a test sample to any class whose class-level confidence score during training is below a chosen *quality threshold*, we invoke Conformal Prediction. We call the classes below the quality threshold *BQL classes* (short for below-quality threshold). While the quality threshold is highly flexible and can be tuned to trade accuracy against performance, for our experiments we used the median of class-level confidence scores for all classes as the quality threshold.

An overview of our approach is shown in Figure 1. As discussed above, we first evaluate the confidence of SVM on the training set using CP as an evaluation framework. By running SVM on the training and testing (TT) set, we obtain a measure of class-level confidence for the right decisions. The class-level confidence for the right decisions is the average confidence of all samples that were rightly classified during testing. It is a measure on how much we can rely on the hyperplanes that SVM constructs. A poor-class level confidence implies an imprecise segregation of classes.

If the SVM decision for a new sample maps it to a class that does not meet our *quality metric*, then the SVM decision is deemed unreliable. In such a case, the CP is invoked and a p-value threshold (or, a confidence level) is provided to CP to determine all possible classes that pass that threshold. By expanding our prediction set with a few top choices returned by the CP, we achieve a better classification accuracy and both fewer false positives and false negatives. For our experiments we use a range of p-value thresholds to demonstrate the seamless relationship between size of the prediction set and the classification accuracy.

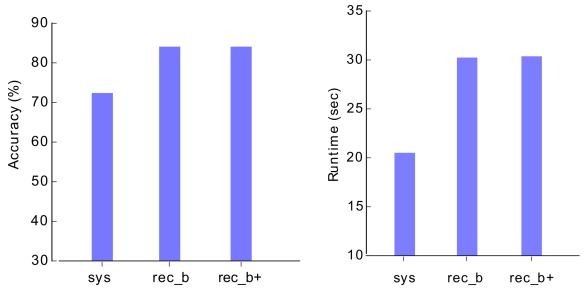
IV. Evaluation

In this section, we experimentally confirm the choices made in designing DroidScribe and evaluate the overall system. We first introduce our experimental setup and the metrics used (§IV-A) before purely evaluating the SVM-based classifier (§IV-B). We then describe our methodology to identify the minimum number of samples in a class so that CP can reliably improve SVM decisions (§IV-C). Finally, we evaluate the hybrid predictor and show that it can achieve near-perfect accuracy by trading off the size of the prediction set (§IV-D).

A. Setup

The work presented in this paper is largely based on a sizable dataset of real-world Android OS malware samples. The dataset was originally collected, characterized, and discussed by Zhou and Jiang [31] and later extended by Arp et al. into the Drebin dataset [1]. It consists of 5,560 samples from which we extract features for 5,246. For the remaining 314 samples, we found that most were not valid APK files and we were unable to run them even in an off-the-shelf Android hardware device (for a discussion of limitations, see §V).

For measuring the classification quality, we use the notion of true positives (TP), false positives (FP), false negatives (FN), precision, and recall for multi-class classification, as discussed by Sokolova and Lapalme [22]. Note that for multi-class classification, accuracy is equivalent to recall, and we



(a) Accuracy (b) Runtime

Fig. 2: (a) Classification accuracy (recall) across SVM modes, and (b) runtime for feature vector extraction (Ext) and classification (SVM).

Mode	Features	# Features
sys	system calls only	110
rec_b	**sys** + Binder	251
rec_b+	**rec_b** + high-level	254

TABLE II: Operational modes for the classifier, the types of features considered and the total number of features.

will use both terms interchangeably. Precision p and recall r are defined per class. For each class i, we have

$$p_i = \frac{TP_i}{TP_i + FP_i} \quad \text{and} \quad r_i = \frac{TP_i}{TP_i + FN_i}$$

where TP_i represents the number of samples in class i that were correctly classified; FP_i is the number of samples incorrectly classified into class i; and FN_i is the number of samples from i classified into other classes. Hence, $TP_i + FN_i$ is the total number of samples in class i in the ground truth. When reporting the results of multi-class classification experiments, we compute the average precision and recall across all samples.

B. SVM-based Classification

In our first set of experiments, we use SVM for multi-class classification (see §III-A). In particular, we aim at evaluating the effect on the resulting accuracy after applying a number of operational modes that represent behaviors with different granularity (see Table II). We begin by establishing a baseline with a configuration that uses raw system calls (**sys**) before reconstructing Binder communication (**rec_b**) and then Binder communication and other system call-based high-level behaviors (**rec_b+**) (see II-B). We report the accuracy for each configuration in Figure 2a and runtime results for SVM in Figure 2b. The relative small number of samples in some classes prevent us from using a three-way split validation set. Instead, we used a leave-k-out cross-validation which has been widely used with unbalanced datasets in the past [19]. Specifically, the experimental results were obtained after 20-fold cross-validation after using random sampling and averaging the results over the 20 folds. We also removed all samples belonging to families with fewer than 20 samples to avoid having folds without samples, which left us with 4,533 samples.

1) System Call-Only Baseline: As the simplest configuration for multi-class SVM classification we use frequencies derived from basic system calls (**sys** in Table II). Overall, CopperDroid collects 205 types of system calls. This configuration achieved 72% accuracy, which we use as a baseline for our subsequent experiments.

2) High-Level Behaviors: Intuitively, abstracting low-level system calls to high-level should be able to reduce noise and improve accuracy. To experimentally validate this intuition and to determine the right level of abstraction, we enriched the system call data by introducing high-level behaviors. While the first configuration for high-level behaviors uses semantically rich binder transactions as features (**rec_b**) the second configurations also includes file system or network accesses (see §II-B) as features (**rec_b+**). System calls that are abstracted to high-level behavior are discarded from the trace. From Figure 2a it is apparent that using high-level behaviors significantly improves accuracy (84% for **rec_b+**).

C. Tuning Conformal Prediction

In order to use CP we need to compute p-values for our base case for all samples. However, it is important to note that the Conformal Prediction is based on statistics rather than on geometrical distances; therefore, the number of samples in a class has a direct bearing on the accuracy of the predictor.

For a class c with cardinality n and a new sample s, the p-value is the fraction of samples in c that have weaker presence of c's than s. If a sample s does not belong to c, then all samples in c will have stronger footprints of the properties of c and their p-value would be $\frac{1}{n+1}$. Each p-value is computed after adding the new sample to the class. Therefore, p-values are sensitive to the cardinality of the class. For a class that has only one sample, the p-value for a new dissimilar sample would be 0.5 ($\frac{1}{1+1}$). On the other hand, if the cardinality is 9, the p-value for a new sample would be 0.1 ($\frac{1}{9+1}$). This skews the comparison across classes in terms of p-values. If the new sample is dissimilar to both the classes, the Conformal Prediction shows a propensity towards picking the class with fewer samples because a lower cardinality has a higher p-value. Therefore, for our experiments, we filter out all families with fewer than 20 samples before applying the hybrid prediction scheme. This corresponds to a minimum p-value of 0.05 and is a reasonable level of granularity to apply Conformal Prediction without skewing the choice families for a sample.

D. Hybrid Prediction

In this section we demonstrate how Conformal Prediction can, together with SVM, provide a highly flexible framework to achieve high accuracy even when dealing with sparse behavior profiles (e.g., due to common code-coverage issues in dynamic analysis). We gave an overview of our framework in §III-C. The key idea in this framework is to use Conformal Prediction to emit a set of predictions for classifying a sample instead of a single predictions. However, CP is computationally expensive, so we apply it selectively to maintain efficiency. Only where the SVM decisions are expected to be of poor quality, we invoke CP during classification.

We present experiments that illustrate how DroidScribe uses the hybrid framework functions in an operational setting.

We first evaluate SVM with Conformal Prediction anddetermine that the decisions of SVM can be prone to errors. Then we show how invoking the Conformal Prediction can help correct such errors committed by SVM.

1) Quality Threshold: The decisions of the SVM are not always reliable. As discussed in Section III-C, we evaluate SVM using CP and plot the distribution of confidence per class for correct SVM decisions (Figure 3), with a sample threshold of 20. The correct classifications for samples in *Adrd*, *DroidKungFu4* and *DroidDream*, for example, show lower confidence during training. The samples in these classes cannot be easily distinguished from others and SVM is forced to pick a class. Therefore, 3, *Adrd*, *Boxer* and *Jifake* are examples of "below-quality threshold" or BQL classes as described in §III-C. Whenever a test sample is mapped one of these BQL classes, CP is used to minimize misclassifications.

2) Tuning CP Towards Perfect Accuracy: A key advantage of CP over SVM is that for a desired confidence score, it can provide a set of top choices such that each sample falls into one of the classes in the set with the desired confidence. Therefore, the desired confidence score is the most important factor in determining how much CP can improve the decisions of SVM. The confidence score has a bearing on the size of prediction set for a classification decision using SVM and larger prediction sets lead to improvements in precision and recall.

We now evaluate how our classifier trades off accuracy against prediction set size. In our experiments, for every CP invocation, we invoke CP with a set of desired confidence scores and observe how accuracy improves as a function of the prediction set size. Figure 4 shows how (i) *recall*, (ii) *overall precision*, and (iii) *prediction set size* change for BQL classes as the confidence level is varied from 0.50 to 1.00 (the maximum possible) in step sizes of 0.05. Note that the overall precision is the same as the overall recall and overall accuracy as we are dealing with multi-class classification. The recall/accuracy for samples mapping to BQL classes (left axis) steadily improves from 91% to 100% as the desired confidence-level is increased from 0.90 to 1.00. The *overall* precision for classification improves as classification decisions for samples mapping to BQL classes are improved through CP and grows steadily from the baseline SVM accuracy of 84% to 94%. The prediction set size also increases (right axis) as the desired confidence levels are increased. It increases steadily for the confidence range of 0.50 to 0.95 (corresponding p-value thresholds of 0.50 down to 0.05). Beyond this, however, it increases rapidly even though the gains in overall precision are modest. For a confidence level of 1.00, which gives us a perfect classification for samples mapping to BQL classes, we have to consider all classes in the dataset even though the gain in overall precision is only 2%. With Conformal Prediction, however, the trade-off between prediction set size and improvement in classification accuracy is seamless. This trade-off can be swung one way or another using the desired confidence level as a parameter which makes DroidScribe a highly tunable system.

3) Analysis of Individual Classes: While Figure 4 shows the improvement in recall for samples mapping to BQL classes when Conformal Predictionis invoked, the responses for individual classes are quite different from the overall picture. In Figure 5, we show how accuracies for individual classes

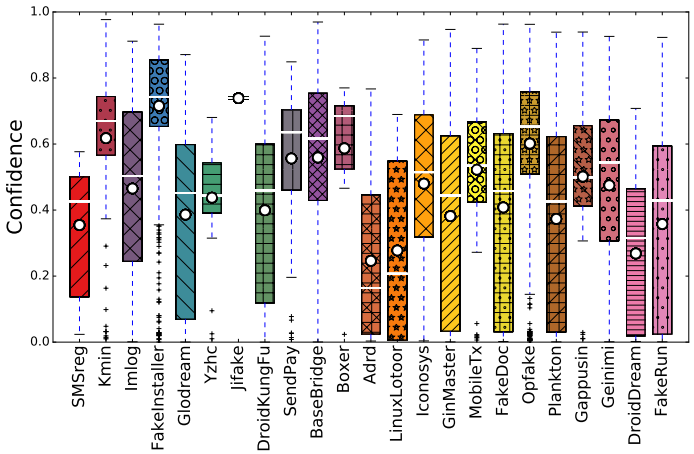


Fig. 3: Confidence of correct SVM decisions for all families with more than 20 samples. The box plot displays the distribution of confidence values per family; circles denote the average confidence; boxes delineate the interquartile range; the segment inside the box shows the median; and the whiskers show the maximum and minimum values and delimit the outliers.

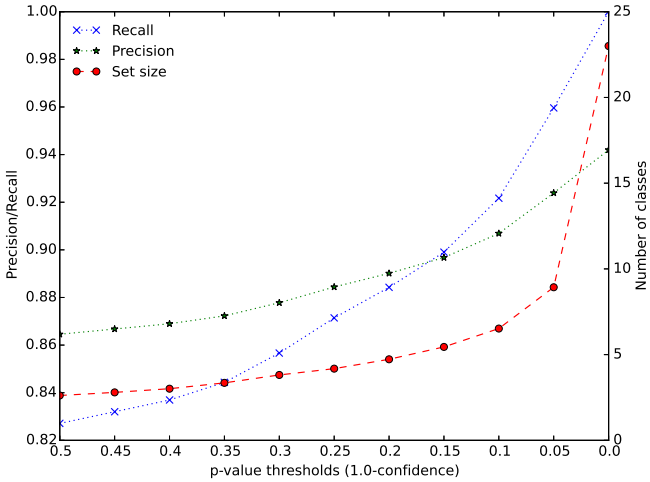


Fig. 4: Recall and precision vs. size of prediction set for p-value cut-offs of 0.50 through 0.00 in steps of -0.05. All classes with fewer than 20 samples are filtered out. At a confidence level of 0.95, the overall precision is 92%; recall for the BQL classes is 96%, and the average size of the prediction set is 9.

change as a function of the confidence level. For the sake of brevity, we only discuss the recall for the classes here.

The recall for samples mapping to *Adrd* increases rapidly as the desired confidence level is increased as shown in Figure 5a. For predictions that map to *Adrd*, we can increase the accuracy from 64% to 93% just by considering an additional three classes (from five classes considered at 64% to eight classes considered at 93%). This is just a subset of the entire set of 23 classes, significantly reducing the burden on a human analyst. In order to correctly classify misclassified samples, the human

analyst would have had to consider all 23 classes. However, with the use of Conformal Prediction, this set is reduced to a small fraction of the original set. This demonstrates the robust statistical foundations of CP; it picks only the top matching classes and does not unnecessarily increase the prediction set size. Beyond confidence level of 0.95, however, the size of the prediction set increases rapidly because a small proportion of samples does not exhibit any meaningful behavior to help CP identify top classification choices.

The improvement in recall for *DroidDream* follows a similar steady increase in recall (as shown in Figure 5b) when the desired confidence-levels are increased. However, there is a marked jump in recall from 68% to 83% when the confidence is increased from a confidence-level of 0.7 (p-value threshold of 0.3) to a confidence level of 0.75 (p-value threshold of 0.25). This jump is likely due to the conformal predictor showing a propensity to picking the correct classes for many samples at those specific confidence levels. Once again, it can be seen that beyond a certain limit (confidence levels of 0.95), it is difficult to boost the recall without considering excessively many classes; even minor improvements in recall require us to consider almost the entire set of classes.

The correlation between the prediction set size and the recall achieved is best demonstrated by *DroidKungFu* samples as shown in Figure 5c. Improvements in recall go hand in hand with the increases in prediction set sizes at all confidence-levels and the recall also improves at a rapid rate as the confidence-levels approach 1.0. This is indicative of the fact that we are unable to pick top matching choices for this class at low-level of confidence thresholds. A likely reason for this could be that we do not see enough behavioral evidence from this sample and that our dynamic analysis needs to be further improved. Another reason could be that we do not pick the right discriminating features for this class of samples or that the two misclassified families behave similarly.

The key takeaway from the above discussion shows that choosing the right confidence level for CP is crucial to effectively improve decisions of standard classification algorithms such as SVM. For our dataset, we were able to achieve considerable improvements in precision and recall between confidence levels of 0.50 and 0.95, where the size of the prediction set ranges from 4 to 9 classes. Beyond this point, the size of the prediction set increases rapidly and, depending on the domain, may outweigh the improvements in recall.

This points to the main advantage of the our hybrid classification approach: with the confidence parameter, DroidScribe allows the user to achieve any desired level of accuracy, at the price of increasing the prediction set size. An example of a setting where the desired confidence level would be high could be a semi-automatic deployment where DroidScribe is used for initial classification of malware. A human analyst may be able to easily disambiguate multiple matches in a larger prediction set, but could be distracted by a misclassification.

V. Limitations

As our classification is built on CopperDroid's extracted behaviors, our methods inherit the framework's limitations. For instance, as CopperDroid dynamically executes apps in an emulated environment, only one execution path is traversed per run. Furthermore, malware can fingerprint emulators and virtualized environments to evade the monitoring systems. Split personality malware, such as *Dendroid* and *Android.HeHe*, are capable of detecting emulated environments with a variety of tests (e.g., the IMEI is set to all zeros on a vanilla Android emulator). In such cases, these specimens will only exhibit benign behaviors, avoiding detection and/or possibly bypassing a screening test for an application market. Furthermore, resources (e.g., a specific network end point) may not be available at the time of the analysis. The issue of code coverage is partially addressed by CopperDroid, which uses a simple triggering mechanism to try and stimulate more behaviors. While this has shown an improvement in observed behaviors, dynamic code coverage is still an open problem.

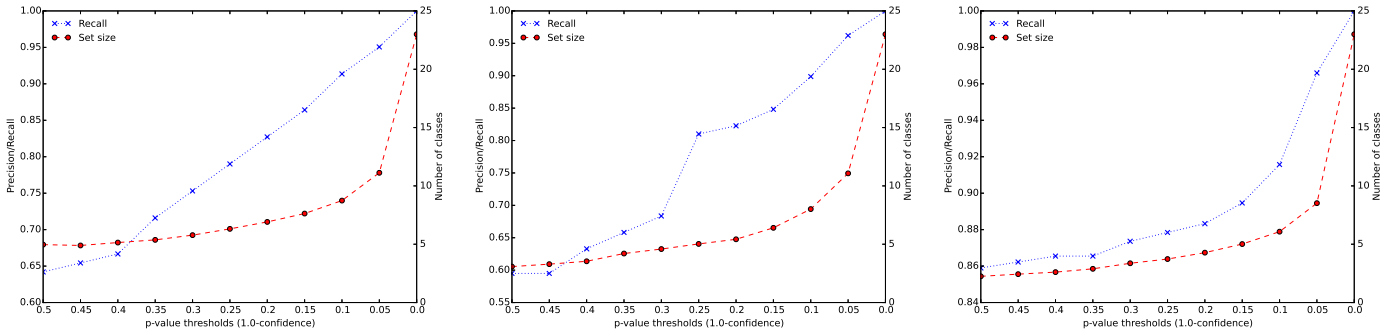
As we are analyzing a stream of system calls, our method might also be vulnerable to mimicry attacks and, in some aspects, randomly added system calls and actions that change the patters in system calls. However, as we rely on behaviors, we are only considering subsets of system calls that cause actual visible change in the Android system. Mimicry attacks decrease the precision of host-based anomaly detection systems by injecting spurious system calls. While our behaviors can also be subject to this attack, this would occur at a higher level of abstraction with visible side effects such as creating a random file. Injecting system calls that correspond to a random high level behavior thus becomes more visible than injecting random sequences of behavior-preserving system calls.

The effectiveness of classification techniques to identify new, previously-unseen, families remains to be tested. Probabilities [20] may introduce bias particularly when multiple classes are considered. Conversely, quality metrics derived from statistics [12] seem to offer insights into stages where machine learning tasks start decaying (e.g., low p-value when a sample x is erroneously classified as belonging to the class y). This may provide evidence in support of the identification of previously-unseen families, which we are currently exploring as part of our ongoing research effort in this direction.

VI. Related Work

Several approaches for automatically *detecting* or *classifying* malware have been presented in the literature, which typically use *static* and/or *dynamic* analysis to extract features from the application under analysis. Once these features are extracted, several techniques can assist the analyst in detecting and classifying the malware, including machine learning [10], data mining [26], expert systems [21], and clustering [4]. Many techniques were first introduced for malware on desktop systems, but for the sake of brevity, we shall focus on the most closely related work only.

Malware Detection. While we do not address malware detection in this paper, the techniques for detection and classification are fundamentally related. Machine learning-based approaches for malware *detection* use binary classification to determine whether a sample is benign or malicious. Drebin [1] uses static analysis to gather features such as permissions, APIs calls, and network addresses declared in clear text and performs binary classification using a linear SVM. Marvin [16] combines static features with additional dynamic features and again uses linear SVMs for binary classification. Ultimately, Marvin aims at



(a) Adrd (b) DroidDream (c) DroidKungFu

Fig. 5: Family-level breakdown of correlation between recall and prediction set size.

returning a risk score for unknown samples. In this case, the combination of static and dynamic analysis improves the detection rate and reduces the number of false positives compared to Drebin [1].

Family Classification. Classifying malicious code into families of related malware is an important step in forensic analysis and threat assessment. It allows analysts to identify malware that is coming from the same source and likely follows similar malicious intents.

Early approaches to classification such as DroidLegacy [5] rely on signature matching to identify malware families. Recent approaches enrich syntactic information with aspects of the program semantics, e.g., program dependency graphs and control flow graphs, to be resilient to some types of obfuscation [29, 24]. DroidSIFT [29] builds dependency graphs for API methods invoked by the app; the graphs are analyzed for similarity with known graphs and then encoded into feature vectors to uncover anomalies and identify families.

Dendroid [24] proposed to use text mining to automatically classify malware samples and analyze families based on control flow structures. RevealDroid [9] uses information flow analysis and sensitive API flow tracking built on top of two machine learning classifiers (C4.5 and 1NN). While RevealDroid's static analysis is shown to be resilient against basic obfuscation schemes, it suffers from the same limitations as the other static approaches against more advanced schemes [23, 28].

In contrast to static analysis, dynamic analysis promises to be resilient against such obfuscation schemes. While dynamic analysis has shown to improve accuracy of malware detection [16], we presented the first dynamic approach to multi-class malware classification on Android. For traditional desktop malware, Bailey et al. [2] proposed a framework to capture effects of system calls, e.g., modified files, network connections made, or changes to the Windows registry. Rieck et al. [20] discussed classification of malware on desktop platforms using SVMs and similar types of dynamic behavior. Similar to our approach, their work uses confidence intervals to express the quality of the fit. However, DroidScribe is more sensitive to outliers as it uses a non-conformity score and reports better accuracy as discussed in §III-B.

Adapting techniques for malware on desktop environments is not straightforward. Apart from the different types of system calls seen on Android (in particular the Binder `ioctl` calls), Android malware typically lacks any discernible propagation behavior. Thus, the existing feature engineering is not directly transferable to the family classification setting on Android. In this regard, DroidScribe is the first work engineering dynamic features at different abstraction layers of Android, i.e., system calls, binder IPCs, and high level behaviors.

VII. Conclusion

We demonstrated how to accurately classify Android malware into families using a purely dynamic approach, thus providing a point of reference for other static or hybrid approaches. We showed that the level of abstraction at which the runtime behavior of Android malware is observed affects the quality of classification. On one hand, the additional detail from reconstructing Binder calls and arguments clearly improves accuracy. On the other hand, aggregating high-level operations such as file system and network accesses reduces noise and similarly improves the achievable results. Overall, we were able to improve the classification accuracy of our SVM classifier from 72% to 84%.

We further used Conformal Prediction as an evaluation framework for our SVM-based classification approach. We demonstrated that datasets with sparse behavioral profiles can lead to error-prone low-confidence classification choices since the classifier lacks the data to disambiguate similar families. In response, we showed how predicting sets of choices using Conformal Prediction leads to an improvement in precision and recall when a sample's mapping to the possible classes was poorly differentiated during the training phase. We proposed a hybrid prediction technique that improves low-confidence SVM decisions using Conformal Prediction and improves the classification accuracy from 84% to 94%.

Acknowledgments

This research has been partially supported by the UK EPSRC grant EP/L022710/1 and by a generous donation from Intel Security (McAfee Labs). We are equally thankful to Roberto Jordaney, the anonymous reviewers, and our shepherd

Drew Davidson for their invaluable inputs, comments, and suggestions to improve the paper.

REFERENCES

[1] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Network and Distributed System Security Symposium, NDSS*, 2014.

[2] M. Bailey, J. Oberheide, J. Andersen, Z. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of Internet malware," in *Research in Attacks, Intrusions and Defenses, RAID*, 2007.

[3] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, 1995.

[4] S. J. Delany, M. Buckley, and D. Greene, "Sms spam filtering: methods and data," *Expert Systems with Applications*, vol. 39, no. 10, pp. 9899–9908, 2012.

[5] L. Deshotels, V. Notani, and A. Lakhotia, "DroidLegacy: Automated familial classification of Android malware," in *ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW*, 2014.

[6] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android security," *IEEE Security Privacy*, vol. 7, 2009.

[7] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *IEEE Symposium on Security and Privacy*, 1996.

[8] J. Friedman, "Another approach to polychotomous classification," Technical report, Department of Statistics, Stanford University, Tech. Rep., 1996.

[9] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek, "Obfuscation-resilient, efficient, and accurate detection and family identification of android malware," Department of Computer Science, George Mason University, Tech. Rep., 2015.

[10] Y.-T. Hou, Y. Chang, T. Chen, C.-S. Laih, and C.-M. Chen, "Malicious web content detection by machine learning," *Expert Systems with Applications*, vol. 37, no. 1, pp. 55–60, 2010.

[11] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE Neural Networks*, vol. 13, 2002.

[12] R. Jordaney, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Misleading metrics: On evaluating machine learning for malware with confidence," http://goo.gl/u2XwTD, Royal Holloway, University of London, Tech. Rep., 2016.

[13] S. Knerr, L. Personnaz, and G. Dreyfus, "Single-layer learning revisited: a stepwise procedure for building and training a neural network," in *Neurocomputing*. Springer, 1990, vol. 68.

[14] U. Kreßel, "Pairwise classification and support vector machines," in *Advances in kernel methods*. MIT Press, 1999.

[15] C. Kruegel, E. Kirda, P. M. Comparetti, U. Bayer, and C. Hlauschek, "Scalable, behavior-based malware clustering," in *Network and Distributed System Security Symposium, NDSS*, 2009.

[16] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Proceedings of the 39th Annual International Computers, Software & Applications Conference*, vol. 2, July 2015, pp. 422–433.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Machine Learning Research*, 2011.

[18] J. C. Platt, "Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods," in *Large Margin Classifiers*. MIT Press, 1999.

[19] P. Refaeilzadeh, L. Tang, and H. Liu, "Cross-validation," in *Encyclopedia of database systems*. Springer, 2009, pp. 532–538.

[20] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Detection of Intrusions and Malware & Vulnerability Assessment, DIMVA*, 2008.

[21] S. Sahin, M. R. Tolun, and R. Hassanpour, "Hybrid expert systems: A survey of current approaches and applications," *Expert Systems with Applications*, vol. 39, no. 4, pp. 4609–4617, 2012.

[22] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing and Management*, 2009.

[23] G. Suarez-Tangil, J. E. Tapiador, and P. Peris-Lopez, "Stegomalware: Playing hide and seek with malicious components in smartphone apps," in *International Conference on Information Security and Cryptology*. Springer, December 2014, pp. 496–515.

[24] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 1, pp. 1104–1117, 2014.

[25] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Network and Distributed System Security Symposium, NDSS*, 2015.

[26] S. Thiruvadi and S. C. Patel, "Survey of data-mining techniques used in fraud detection and prevention," *Information Technology*, vol. 10, no. 4, pp. 710–716, 2011.

[27] A. G. V. Vovk and G. Shafer, *Algorithmic learning in a random world*. Springer-Verlag New York, Inc., 2005.

[28] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "Appspear: Bytecode decrypting and dex reassembling for packed android malware," in *Research in Attacks, Intrusions, and Defenses*. Springer, 2015, pp. 359–381.

[29] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *ACM SIGSAC Computer and Communications Security, CCS*, 2014.

[30] C. Zhou, I. Nouretdinov, Z. Luo, D. Adamskiy, L. Randell, N. Coldham, and A. Gammerman, "A comparison of venn machine with Platt's method in probabilistic outputs," in *Artificial Intelligence Applications and Innovations (AIAI)*, 2011.

[31] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy*, 2012.