Understanding Update of Machine-Learning-Based Malware Detection by Clustering Changes in Feature Attributions

Yun Fan[1], Toshiki Shibahara[2], Yuichi Ohsita[1], Daiki Chiba[2], Mitsuaki Akiyama[2], and Masayuki Murata[1]

[1] Osaka University, Osaka, Japan
{h-un, y-ohsita, murata}@ist.osaka-u.ac.jp
[2] NTT, Tokyo, Japan
toshiki.shibahara.de@hco.ntt.co.jp, {daiki.chiba,akiyama}@ieee.org

Abstract. Machine learning (ML) models are often adopted in malware detection systems. To ensure the detection performance in such ML-based systems, updating ML models with new data is crucial for minimizing the influence of data variation over time. After an update, validating the new model is commonly done using the detection accuracy as a metric. However, the accuracy does not include detailed information, such as changes in the features used for prediction. Such information is beneficial for avoiding unexpected updates, such as overfitting or noneffective updates. We, therefore, propose a method for understanding ML model updates in malware detection systems by using a feature attribution method called Shapley additive explanations (SHAP), which interprets the output of an ML model by assigning an importance value called a SHAP value to each feature. In our method, we identify patterns of feature attribution changes that cause a change in the prediction. In this method, we first obtain the feature attributions for each sample, which change before and after the update. Then, we obtain the patterns of the changes in the feature attributions that are common for multiple samples by clustering the changes in the feature attributions. In this study, we conduct experiments using an open dataset of Android malware and demonstrate that our method can identify the causes of performance changes, such as overfitting or noneffective updates.

Keywords: Malware detection · Machine learning · Feature attribution

1 Introduction

Machine learning (ML) has been used to detect malware. Such ML-based malware detection systems adopt ML models trained on previously collected data to perform predictions on new data. Owing to a phenomenon called concept drift [24], the detection performance in an ML-based system gradually degrades as the statistical characteristics of data change over time [12]. In this situation, updating the ML model using new data can effectively improve the detection performance of the systems.

After an update, the new model is validated using validation data in terms of detection performance [13]. Once the model is successfully validated, it can be deployed in a real detection system. Thus far, the detection accuracy of the validation data has been used as a metric to validate the model after an update. However, the accuracy does not reflect detailed information, such as changes in the features used for prediction. Such information is beneficial for avoiding unexpected updates, such as overfitting or noneffective updates.

To obtain detailed information about model updates, we propose a method for identifying patterns of feature attribution changes that cause a change in the prediction. Feature attributions represent the extent of contribution that features have made to model predictions in a system. When a model is retrained using a dataset updated with newly collected data, important features that were overlooked or did not appear before the update may be found. The attributes of such features change significantly. In other words, by analyzing significant changes in feature attributions, we can identify model changes in detail. In the proposed method, we first obtain the feature attributions for each sample that change before and after the update. Then, we obtain the patterns of the changes in the feature attributions, which are common for multiple samples, by clustering the changes of the feature attributions using the similarities of the features whose attributions changed significantly.

In our experiments, we use Android application samples and build models to detect malicious samples. We evaluate the effectiveness of the proposed method by analyzing model changes while the training dataset is updated with different biased data, and as time goes by, we demonstrate that our method can identify the unexpected model changes caused by the biased data. The experimental results show that updates with severely biased data can lead to an overfitting or noneffective update, causing the performance to deteriorate or remain unchanged. The results also indicate that our method can identify the important features relevant to the performance change, which are difficult to find by using a method that calculates only the feature attributions. Some important features found by our method cannot be found unless by checking more than 100 features if the features are checked in the order of the feature attributions.

The remainder of this paper is organized as follows: Section 2 introduces related works, especially the feature attribution method. Section 3 presents the proposed method. Section 4 introduces the experimental setup and Section 5 presents our experimental results. Finally, Section 6 discusses our observations and Section 7 concludes the paper.

2 Background and Related Work

We propose a method for analyzing updates to determine the cause of the performance changes. Before presenting our method, in this section, we introduce other methods to evaluate the appropriateness of models. We also introduce a method for determining features that contribute to classification.

2.1 Evaluation Methods

Model Evaluation Metric There are several common metrics, such as accuracy, precision, recall, F-measure, true positive rate (TPR), and false positive rate (FPR), for evaluating the classification performance of ML models. These are used to calculate a value that indicates the model performance. In binary classification—distinguishing between positive and negative classes—samples are divided into four different categories based on their predicted and true classes: true positive (TP), true negative (TN), false positive (FP), and false negative (FN). TPs and TNs are samples *correctly* predicted as positive and negative, respectively. FPs and FNs are samples *incorrectly* predicted as positive and negative, respectively. In malware detection, positive and negative samples refer to malicious and benign samples, respectively. For example, the FPs are benign samples which are incorrectly predicted as malicious.

The accuracy metric simply computes the ratio of the correct prediction number to the total sample number, $\frac{TP+TN}{TP+TN+FP+FN}$. Precision is the ratio of the correct positive prediction number to the total positive prediction number, $\frac{TP}{TP+FP}$. The recall (also known as the TPR) is the ratio of the correct positive prediction number to the total positive sample number, $\frac{TP}{TP+FN}$. The FPR is the ratio of the incorrect positive prediction number to the total negative sample number, $\frac{FP}{FP+TN}$. The F-measure (or F1-score) is the harmonic mean of precision and recall: $2 \times \frac{precision \times recall}{precision+recall}$.

The model performance is also shown in the receiver operating characteristic (ROC) curves. ROC curves have true and false positive rates as the vertical and horizontal axes, respectively. ROC curves and the area under the curve (AUC) are commonly used to evaluate the performance of the ML model in cybersecurity.

In addition to these metrics, there are also some criteria to evaluate the model from other perspectives. Typically, the Akaike information criterion (AIC) [2] and the Bayesian information criterion (BIC) [22], are are widely used to avoid overfitting. They are defined as

$$AIC = -2\ln(\mathcal{L}) + 2K, \tag{1}$$

$$BIC = -2\ln(\mathcal{L}) + K\ln(n), \tag{2}$$

where K is the number of learnable parameters in the model, $\mathcal{L}$ is the maximum likelihood of the model, and n is the number of samples.

Cross-validation Cross-validation evaluates ML models by dividing a dataset into several subsets. To estimate the model classification performance, one subset is used for validation and the others are used for training. In k-fold cross-validation, a dataset D is randomly split into k mutually exclusive subsets D_1, D_2, ..., D_k. The model is then trained and tested over k rounds. In each round $i \in \{1, 2, ..., k\}$, training is performed on subset $D \setminus D_i$ and testing on subset D_i. In validation, evaluation metrics such as accuracy and AUC score are typically

used to estimate the classification performance. To reduce variability, the validation results are combined or averaged over all rounds to obtain a final estimate of the classification performance. In stratified cross-validation, subsets are stratified such that they contain approximately the same proportions of labels as the original dataset.

Although these evaluation methods can compute indicators reflecting model performance, they cannot provide sufficient details of the model updates.

2.2 Feature Attribution Methods

To explain predictions by ML models, importance values are typically attributed to each feature to show its impact on predictions. The importance values of features can be output by some popular ML packages, such as scikit-learn [19], wherein permutation importance is frequently used. Permutation importance randomly permutes the values of a feature in the test dataset and observes a change in error. If a feature is important, then permuting it should significantly increase the model error [15].

Another method for interpreting ML models is the partial dependence plots (PDPs) [9]. A PDP can show how a feature affects model predictions by the relation between the target prediction and features (e.g., linear, monotonic, or more complex). However, a PDP can compute two features at most, and it assumes that these features are not correlated with other features. Thus, it is unrealistic to use PDP for models trained on data containing numerous features.

Another popular approach is the local interpretable model-agnostic explanations (LIME) [21]. LIME explains a given prediction by learning a model around that prediction. By computing the feature importance values of a single prediction, we can easily analyze what made the classifier output that prediction. Instead of explaining the whole model, LIME explains only a single sample prediction result. However, LIME still uses permutation to compute feature importance values, making LIME an inconsistent method.

Although these methods are intended to provide insight into how features affect model predictions, the feature attribution methods described above are all inconsistent, meaning that when the model has changed and a feature impact on the model's output has increased, the importance of that feature can actually be lower. Inconsistency makes comparison of attribution values across models meaningless because it implies that a feature with a large attribution value might be less important than another feature with a smaller attribution.

2.3 SHAP

The inconsistency of the methods in Section 2.2 makes it meaningless to compare feature attributions across models, which necessitates a consistent method for analyzing feature attribution changes in different models.

SHAP [16] is a method that explains individual predictions based on Shapley values from game theory. The Shapley value method is represented as an additive

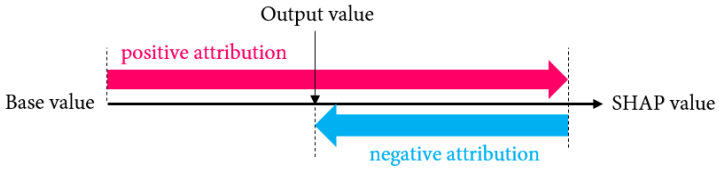


Fig. 1. SHAP values explaining model output as a sum of the attributions of each feature

feature–attribution method (demonstrated in Fig. 1) with a linear explanation model g, described as

$$g(z) = \phi_0 + \sum_{i=1}^{M} \phi_i z_i, \tag{3}$$

where $z \epsilon (0,1)^M$, M is the number of input features, and $\phi_i \epsilon \mathbb{R}$. z_i is a binary decision variable that represents a feature being observed or unknown and ϕ_i is the feature attribution value.

Currently, SHAP is the only consistent and locally accurate individualized feature attribution method. According to Ref. [16], SHAP has three desirable properties: local accuracy, missingness, and consistency. Local accuracy means that the sum of feature attributions equals the output of the model that we want to explain. Missingness means that missing features are assigned no importance, i.e., 0. Consistency means that the attribution assigned to a feature will not be decreased when we change a model such that the feature has a larger impact on the model. Consistency enables comparison of attribution values across models.

When explaining a model f, SHAP assigns ϕ_i values to each feature [15] as

$$\phi_i = \sum_{S \subseteq A \setminus \{i\}} \frac{|S|!(M - |S| - 1)!}{M!} [f_x(S \cup \{i\}) - f_x(S)], \tag{4}$$

where $f_x(S) = f(h_x(z)) = E[f(x)|x_S]$, $E[f(x)|x_S]$ is the expected value of a function conditioned on a subset S of the input features, S is the set of nonzero indices in z, and A is the set of all input features. h_x maps the relationship between the pattern of binary features z and the input vector space.

Because SHAP is the only consistent, locally accurate method for measuring missingness, there is a strong motivation to use SHAP values for feature attribution. However, there are two practical problems remaining to be solved, namely,

1. efficiently estimating $E[f(x)|x_S]$, and
2. the exponential complexity of Eq. (4).

When estimating the predictions of tree models, Lundberg and Lee [15] designed a fast SHAP value estimation algorithm specific to trees and tree ensembles. This algorithm runs in polynomial time instead of exponential time, reducing the computational complexity of exact SHAP value computations for trees and tree ensembles.

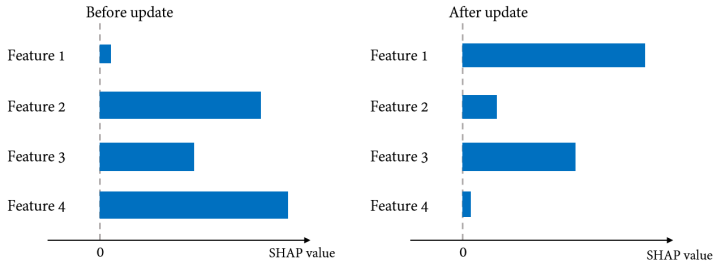


Fig. 2. Changes in SHAP values of features after update

3 Proposed Method

When updating an ML model for real-world deployment, detailed information about model updates is beneficial for preventing unexpected predictions. To obtain detailed information, we propose a method to identify common patterns of feature attribution changes that cause prediction changes. More precisely, the pattern is a combination of features whose attributions changed drastically after the update. Using such information, the operators of the ML model can understand common reasons for prediction changes. Our method consists of two steps. The first step is to calculate feature attribution changes based on SHAP. The second is to identify typical change patterns by clustering samples based on their features whose attributions have drastically changed.

3.1 Calculating Feature Attribution Changes

Because SHAP is a consistent attribution method—meaning that SHAP values are invariant regardless of models—we use SHAP values to measure the attribution changes of features across different models. We investigate changes in the models in detail by analyzing changes in the SHAP values of the features.

Figure 2 shows an example of the changes in SHAP values before and after an update regarding predictions of the same sample. A SHAP value is assigned to each feature to show how important it is. A high SHAP value means that the corresponding feature has a large effect on the prediction and a SHAP value close to 0 means that the corresponding feature has almost no effect on the prediction. The SHAP values for Features 2 and 4 decreased to near 0, and the SHAP value of Feature 1 increased greatly from a value near 0 after the update, indicating that the model significantly changed with respect to these features. On the other hand, the SHAP values of Feature 3 showed no significant change, indicating that the model did not change with respect to this feature. By analyzing features whose SHAP values have significantly changed, we can infer the cause of model updates and their effect on classification performance.

Our method defines an increasing rate that indicates the significance of changes in feature attributions after a model update. Specifically, we compute

the SHAP values for different models and then calculate the significance of the increase in each feature's SHAP value due to the update. This increasing rate also indicates whether changes in SHAP values increase or decrease. As shown in Fig. 2, Feature 1 exhibits a significant increase, whereas Features 2 and 4 show significant decreases after updating. Unlike these features, the increasing rate of Feature 3 is close to 0 because its SHAP value has no significant change after the update.

The following describes our definition of the increasing rate. Let D_1 be the dataset on which the model was trained before the update and D_2 be the dataset after the update. Then, let the model be trained on D_1 and D_2 be f_1 and f_2, respectively. When predicting a label for data $\mathbf{x}$ with model f_m, we denote the SHAP value of the i-th feature x_i as v_{mx_i}.

We define the rate of increase I_{x_i} of a feature x_i as the ratio of the SHAP value increase to the smallest absolute SHAP value. Let v_{1x_i} be the SHAP value of feature x_i in the old model and let v_{2x_i} be the SHAP value of feature x_i in the new model. The rate of increase is high only if the absolute value of one SHAP value (v_{1x_i} or v_{2x_i}) is large and the other is close to zero. In other words, if the absolute values of both SHAP values are either large or small, the rate of increase is small. We add constant terms c_1 and c_2 to make the increasing rate small when both SHAP values are close to zero.

The increasing rate for feature x_i is defined as

$$\begin{aligned} I_{x_i} &= \frac{v_{2x_i} - v_{1x_i} + c_1}{\min(|v_{1x_i}|, |v_{2x_i}|) + c_2}, \\ where\ \ c_2 > 0,\ c_1 &= \begin{cases} c_2, & when\ v_{2x_i} - v_{1x_i} \geq 0, \\ -c_2, & when\ v_{2x_i} - v_{1x_i} < 0. \end{cases} \end{aligned} \tag{5}$$

In this paper, we set the constant term $c_2 = 0.01$.

The SHAP value of a sample $\mathbf{x}$ is an array of size N, where N is the number of features.

$$\mathbf{v}_{m\mathbf{x}} = [v_{mx_1}, v_{mx_2}, ..., v_{mx_i}, ..., v_{mx_N}].$$

The increasing rate of a sample is also an array of size N:

$$\mathbf{I}_{\mathbf{x}} = [I_{x_1}, I_{x_2}, ..., I_{x_i}, ..., I_{x_N}].$$

3.2 Clustering based on Feature Attribution Changes

To make the output more concise and clearer for the operators, we divide the samples into clusters based on their feature attribution changes. By analyzing samples in each cluster in terms of prediction changes and feature attribution changes, the operators can understand common reasons for prediction changes and infer the performance change in real-world deployment.

We use Jaccard similarity [11] to measure the similarity based on feature attribution changes. Specifically, we define the set A as the set of features whose SHAP rate of increase exceeds k or under $-k$ in sample $\mathbf{x}_A$. In this way, we can represent the sample $\mathbf{x}_A$ based on features that significantly changed after

the update. If A is empty, we do not use sample $\mathbf{x}_A$ for clustering. The Jaccard similarity between samples $\mathbf{x}_A$ and $\mathbf{x}_B$ is defined as the size of the intersection divided by the size of the union of sets A and B:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}. \tag{6}$$

Note that $0 \leq J(A,B) \leq 1$.

Based on the Jaccard similarity matrix, we conducted clustering via density-based spatial clustering of applications with noise (DBSCAN) [19]. The maximum distance between two samples to be considered as the same cluster was 0.5. In other words, samples in which half of the significantly changed features are common are considered similar and assigned to the same cluster. We used the default set of the software for other parameters, which means the minimum number of samples in a neighborhood for a point to be considered as a core point was five.

After clustering, we calculated the average prediction of each cluster and selected the clusters whose average predictions changed after the update. Then, we calculated each feature's average rate of increase in each cluster and output the top 10 features in terms of the rate of increase (for the cluster having less than 10 features, output all). Based on the output, the operators can understand which features cause prediction changes and infer the performance change in real-world deployment.

4 Experimental Setup

In this section, we introduce the experimental setup for our evaluation using Android applications.

4.1 Dataset

We used samples from AndroZoo [3] to conduct the experiments. AndroZoo is a collection of Android applications from several sources, including the official Google Play app market and VirusShare. It contains over ten million Android application package (APK) files. Each file was analyzed by over 70 antivirus software packages, providing knowledge of malware. We selected files that were not detected as malware by any antivirus software for use as benign samples. For malicious samples, we selected files that were detected as malware using at least four antivirus software packages.

We collected over 1,000 samples per month from AndroZoo between 2016 and 2018. In total, we gathered 61,724 benign samples and 11,160 malicious samples. We used applications collected from 2016 to 2018 because Miller etal. [18] empirically showed that antivirus detection became stable after approximately one year. We followed Ref. [20] when adjusting the ratio of malicious samples to benign ones. Specifically, we set the percentage of malicious samples to 10% and benign samples to 90% in the dataset.

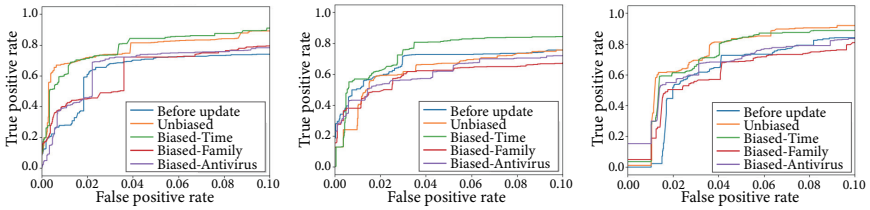
Fig. 3. Sliding window setup. *a* represents the first half of the year. *b* represents the second half of the year.

4.2 Model Update

In real-world applications, ML models are often updated based on a sliding window setup [5], which means that new data are added to the pre-update dataset, and old data are removed. Thus, we evaluated our method thrice under this setup, as shown in Fig. 3. In each update, the data from the next period of six months were added to the pre-update training dataset, and the older half in the pre-update training dataset was removed. Each training dataset had a similar size of approximately 3,800 benign samples and 420 malicious samples. The number of samples in each dataset is listed in Table 9 in Appendix A. We used the data from the following period of the post-update training dataset as the test dataset to evaluate the model ROC curve and AUC in real-world deployment. Each test dataset contained approximately 5,000 benign samples and 550 malicious samples. The number of samples is also shown in Table 9.

To simulate successful and failed updates, we used biased and unbiased post-update training datasets. For pre-update training datasets, we always used unbiased datasets because we assumed that the pre-update models are successfully trained and validated in real-world deployment. Consequently, post-update training datasets were composed of the first half of the unbiased dataset and the second half of the differently biased dataset. For example, the post-update training dataset of update 1 consisted of the *unbiased* dataset of 2016b and the *biased* dataset of 2017b. Using such datasets, the models updated differently depending on the bias. We evaluated whether operators can distinguish different updates based on the output of our method. The unbiased and biased datasets were prepared as follows:

1. *Unbiased:* We randomly selected an equal number of samples from every month.
2. *Biased-Time:* We randomly selected all samples from the latest month of the period.



(a) Update 1 (b) Update 2 (c) Update 3

Fig. 4. ROC curves of models trained on different biased datasets

3. *Biased-Family:* We randomly selected malicious samples from 3 major families; there were more than 40 small families in total. The benign samples were the same as those in the *unbiased dataset*.
4. *Biased-Antivirus:* We randomly selected malicious samples from samples easily detected. Samples were determined as easily detected if they were detected by more than 20 antivirus software in VirusTotal [23]. The benign samples were the same as those in the *unbiased dataset*.

4.3 Features and Models

We use Drebin [4], which is a lightweight method for detecting malicious APK files based on broad static analyses, to extract features from APK files. Features were extracted from the manifest and disassembled dex codes of the APK file. From these, Drebin collected discriminative strings, such as permissions, API calls, and network addresses. To build a machine learning model, we used random forest [6], a method well known for its excellent classification performance and applicability to many tasks, including malware detection. For a detailed setup of the features, models, and hyper parameters, please refer to Appendix A.

5 Experimental Results

We conducted experiments with three updates using the four biased datasets described in Section 4. First, we show which models are successfully updated by using test datasets. Then we describe results of quantitative and qualitative evaluation. In these evaluations, we investigate whether the output of our method is beneficial to ML system operators in understanding model updates. As a quantitative evaluation, we investigate the extent to which our method can reduce the number of features that operators must consider to understand model updates. This evaluation shows that operators can easily understand model updates using our method. As a qualitative evaluation, we investigate whether operators can infer the classification performance of the updated models in real-world deployment by using the output of our method with a post-update training dataset. This evaluation shows how useful our method is.

Table 1. AUC in cross-validation and AUC on test dataset

			Update 1	Update 2	Update 3
Pre-update	Unbiased	CV	0.9673	0.9722	0.9522
		Test	0.9095	0.8932	0.9163
Post-update	Unbiased	CV	0.9722	0.9522	0.9573
		Test	0.9425	0.9273	0.9493
	Biased-Time	CV	0.9729	0.9538	0.9693
		Test	0.9509	0.9513	0.9439
	Biased-Family	CV	0.9809	0.9687	0.9739
		Test	0.9313	0.8976	0.9296
	Biased-Antivirus	CV	0.9812	0.9727	0.9696
		Test	0.9157	0.8731	0.9320

5.1 Classification Performance of Updated Models

We show which models are successfully updated by using test datasets. In addition, we show that inferring the classification performance on test datasets and understanding changes in models are difficult based on the conventional model validation method, i.e., cross-validation (CV) with post-update training datasets. We use AUCs and ROC curves to evaluate classification performance. The AUC on test datasets and in CV are shown in Table 1. The AUC in CV were much better than the AUC on test datasets, indicating that the cross-validation is inappropriate because its result may be over-optimistic under concept drift. Moreover, operators cannot understand why model updates cause prediction changes and infer whether the updates are reasonable.

To investigate the classification performance more precisely, we show the ROC curve of each model in each update in Fig. 4). In general, the performance improved after updates of "Unbiased" and "Biased-Time" datasets, and deteriorated or almost stayed unchanged after updates of "Biased-Family" and "Biased-Antivirus" datasets.

5.2 Quantitative Evaluation

We investigate the number of features that operators must analyze to understand model updates. The smaller the number, the less effort the operators need to make for the analysis. Without our method, operators analyze features important to classification. In other words, operators look into features in descending order of SHAP values, i.e., from the most important to the least important. For this reason, we investigate the maximum order of the SHAP value (the least important) in each cluster's features. Table 2 shows the number of clusters, the number of features in each cluster, and the maximum order of SHAP values. The number of features with our method is much smaller than that without our method.

More detailed results are shown in Table 3; it shows the output features of the two clusters when using the Unbias dataset in Update 1. In cluster 1, the

Table 2. Number of cluster/features and maximum order of SHAP

		# clusters	# features in each cluster	Max. order of SHAP
Update 1	Unbiased	5	7–10	39–487
	Biased-Time	4	1–10	39–142
	Biased-Family	3	2–8	22–110
	Biased-Antivirus	3	3–9	53–218
Update 2	Unbiased	1	6	64
	Biased-Time	6	3–8	24–190
	Biased-Family	3	4–10	24–428
	Biased-Antivirus	0	-	-
Update 3	Unbiased	5	2–10	31–371
	Biased-Time	6	3–10	55–371
	Biased-Famliy	2	3–10	371
	Biased-Antivirus	1	2	198

Table 3. Output features of two clusters using Unbiased dataset in Update 1

	Features	Order of increasing rate	Order of SHAP
Cluster 1	android.location.locationmanager.getproviders	1	49
	android.nfc.tech.ndefformatable.format	2	86
	android.nfc.tech.ndefformatable.connect	3	109
	android.nfc.tech.ndef.connect	4	81
	android.nfc.tech.ndef.writendefmessage	5	100
Cluster 2	android.permission.vibrate	1	30
	android.widget.videoview.setvideopath	2	7
	android.widget.videoview.pause	3	13
	android.widget.videoview.stopplayback	4	6
	android.widget.videoview.start	5	14

features causing prediction change are mainly relevant to `nfc.tech` because four out of five features are relevant to `nfc.tech`. Similarly, in cluster 2, the features relevant to `widget.videoview` mainly caused the prediction changes. However, these features are difficult to identify if only SHAP values are used because the maximum order of SHAP is 109.

5.3 Qualitative Evaluation

As shown above, the classification performance on the test dataset depends on the bias in the training dataset, and our method can reduce the number of features that operators must analyze to understand a model update. Here, we investigate whether operators can infer that a model update is successful or failed using the output of our method with a post-update training dataset. Specifically, the main cause of a failed model update is overfitting and noneffective update. Overfitting involves learning the training dataset too much and not generalizing

Table 4. Example of similar clusters

	Features	Mean rate
Cluster 1	android.permission.write_external_storage	4.77
	android.permission.read_external_storage	4.11
Cluster 2	android.intent.action.main	6.5
	android.permission.read_external_storage	4.81
	android.widget.videoview.setvideopath	4.76
	android.permission.internet	4.47
	android.permission.write_external_storage	4.12
	android.permission.internet	3.29
	android.permission.access_network_state	3.2

it to the test dataset. More precisely, an overfitted model learns only a few families or overlooks some families. The noneffective update is that a model update does not change predictions, even though a model update is expected to change some predictions under concept drift. The noneffective update is mainly caused by noninformative newly added data.

We describe how to analyze the output of our method from the aforementioned three perspectives: learning a few families, overlooking some families, and noneffective updates. Note that our method does not output any cluster of benign data, which means that the performance changes are mainly caused by malicious data. For this reason, we only show results of malicious data.

Learning a Few Families To confirm whether a model learns only a few families, we focus on a variety of changes caused by the model update. Using our method, data with similar attribution changes are assigned to the same cluster. The number of clusters reflects the variety of changes. The lack of variety can result in an overfitting model because the model can only learn features related to some types of data, causing a performance degradation after the update. The number of clusters can be used to evaluate whether the model is overfitted.

Table 2 shows the number of clusters in each update. When the dataset is biased, for example, it only contains major families with a large number of malicious samples, and the lack of variety may cause the model to only learn features related to certain families, resulting in overfitting. As can be seen in Table 2, the cluster numbers of "Biased-Family" are always less than the results of other updates, and the results of "Biased-Antivirus" are also low in some cases, which indicates that the bias of the dataset causes a lack of variety and influences the update as a result.

We can more precisely identify the lack of variety by investigating the similarity of features between clusters. For example, Table 4 shows the features of two of the clusters using "Biased-Family" in Update 1. All features in Cluster 1 are included in Cluster 2, meaning that the variety of data is low.

Table 5. The number of clusters whose predictions become false

	Unbiased	Biased-Time	Biased-Family	Biase-Antivirus
Update 1	0	0	0	0
Update 2	0	0	1	0
Update 3	0	0	2	0

Table 6. Cluster with false prediction change in Update 2. "None" means SHAP value is 0 after update and ranked in the last.

Features	Mean rate	Order of SHAP
com.qihoo.util.appupdate.appupdateactivity	-25.66	None
com.qihoo.util.startactivity	-25.06	None
com.switfpass.pay.activity.qqwappaywebview	-17.20	None
com.alipay.sdk.auth.authactivity	-15.54	None
blue.sky.vn.api	-14.66	None
landroid/telephony/smsmanager.sendtextmessage	-14.23	33
blue.sky.vn.mainactivity	-11.85	None
blue.sky.vn.webviewactivity	-10.42	None
blue.sky.vn.gamehdactivity	-10.33	None
com.qihoo.util.commonactivity	-8.09	None

Table 7. Difference values used to measure the extent of improvement

	Unbiased	Biased-Time	Biased-Family	Biased-Antivirus
Update 1	103	122	31	25
Update 2	70	104	-17	0
Update 3	78	131	-27	12

Overlooking Some Families To confirm whether a model overlooks some families, we focus on clusters with predictions going false from true. For example, the shortage of certain data, such as minor malware families, can prevent the model from learning features related to those data. When the model is unable to learn some data after the update, the predictions of such data become false. Thus, we investigate the clusters with predictions changing from true to false.

Table 5 shows the number of clusters whose predictions change from true to false. As we can see, only the results of "Biased-Family" have such clusters. We can also obtain further information about the failure related to these clusters by showing their features of a high rate of increase. For example, Table 6 shows the cluster whose prediction changes from true to false in update 2. The false predictions are mainly caused by the lack of features `com.qihoo.util` and `blue.sky.vn`, because their SHAP values decreased to zero after the update.

Noneffective Update To identify noneffective updates mainly caused by non-informative newly added data, we focus on cluster size, i.e., the number of samples in each cluster. The cluster size shows how many samples have a different

prediction after the update. The more the prediction results change from false to true, the better the model performance will improve. If the performance does not improve sufficiently after the update, the update is ineffective. In a malware detection system, a change in the prediction results from false to true means that the prediction of malicious samples becomes positive, or the prediction of benign samples becomes negative. Therefore, the size of the output clusters indicates the extent of the performance change during the update. We use the number of samples whose prediction changes from false to true to evaluate whether the model has been updated effectively. Specifically, we measure the extent of performance improvement by the difference value between the number of samples whose predictions become true and the number of samples whose predictions become false. Table 7 presents the results for each difference value. The minus number in Table 7 indicates that the samples whose predictions change from true to false are more than those whose predictions change from false to true.

Table 7 indicates that for a dataset of approximately 220 samples, the performance of "Unbiased" and "Biased-Time" improve after update, whereas the performance of "Biased-Family" and "Biased-Antivirus" have very limited change or no change after update, which is consistent with the results of the ROC and AUC but more clear. When the difference value of the data becomes true and the data becoming false is large, we can conclude that the model performance has improved after the update, and the update is effective. When the difference value is relatively small, the performance remains almost unchanged, and the update is ineffective.

Summary In our experiment, we used data with four different types of bias and three different periods of time to conduct 36 updates to demonstrate our method. Table 8 shows the evaluation result of each update. As we can see, in most cases, the results of "Biased-Family" and "Biased-Antivirus" appear to be overfitted or noneffective, which explains the performance not improving after updates by those dataset. All the results of "Biased-Time" are neither overfitted nor noneffective, explaining the performance improvement after updates by "Biased-Time," as shown in the ROC curves.

6 Discussion

Application of the Proposed Method Although this study focuses mainly on malware detection systems, our method should be applied to all types of machine learning tasks. The SHAP method provides algorithms for estimating SHAP values for any ML model, i.e., our method can be applied to any ML model, regardless of the dataset or model. For example, our method can be applied to suspicious URL detection [17], malicious website detection [8], and malware family classification [1]. In multiclass classifications, we can identify changes in important features by analyzing feature attribution changes by focusing on each class.

Table 8. Summary of qualitative evaluation. ✓represents that an undesirable update is observed, and × represents that an undesirable update is not observed.

		Learning a few families	Overlooking some families	Noneffective update
Update 1	Unbiased	×	×	×
	Biased-Time	×	×	×
	Biased-Family	✓	×	✓
	Biased-Antivirus	×	×	✓
Update 2	Unbiased	✓	×	×
	Biased-Time	×	×	×
	Biased-Family	✓	✓	✓
	Biased-Antivirus	✓	×	✓
Update 3	Unbiased	×	×	×
	Biased-Time	×	×	×
	Biased-Family	✓	✓	✓
	Biased-Antivirus	✓	×	✓

Limitation In this paper, we demonstrated that our method outputs detailed information about model updates, such as the important features that are relevant to the performance change during the update. Though we discussed the importance of the outputted information, we need a user study, which is one of our future works.

7 Conclusion

ML methods have been widely applied to many tasks. In practical use, it is necessary to regularly update the model to maintain its classification performance. AUC and accuracy are generally used to validate models to confirm their performance after updates. However, it is difficult to gain sufficiently detailed information for understanding model updates, such as what causes performance changes and the influence on a certain type of data.

Therefore, we propose a method for determining samples in which the features important for classification have significant changes. By selecting those samples and clustering them by feature attribution changes, we can know more about why performance changes or how an update influences a certain type of data. For the feature attribution computation, we used a consistent importance value called the SHAP value because SHAP values are comparable across different models. Our proposed method calculates the rates of increase in SHAP values after updates to reflect changes in feature importance, and clustering the samples by their feature attribution changes to generate the information given to the operator.

We conducted experiments using an open dataset of Android malware. We investigated model changes while the training dataset is updated with different biased data, and demonstrated that our method can identify the unexpected model changes such as overfitting or noneffective update caused by the biased

data. The results also indicate that our method can identify the important features relevant to the performance change, which are difficult to find by using a method that calculates only the feature attributions.

Though we discussed the importance of the outputted information, we need a user study, which is one of our future works.

A Detailed Experimental Setup

Dataset Each training dataset had a similar size of approximately 3,800 benign samples and 420 malicious samples, and each test dataset contained approximately 5,000 benign samples and 550 malicious samples. The number of samples is shown in Table 9.

Feature To extract features in our experiments, we used Drebin [4], a lightweight method for detecting malicious APK files based on broad static analyses. Features are extracted from the manifest and disassembled dex codes of the APK file. From these, Drebin collects discriminative strings, such as permissions, API calls, and network addresses. Drebin extracts eight sets of strings: four from manifests and four from dex code.

1. Hardware components
2. Requested permissions
3. App components
4. Filtered intents
5. Restricted API calls
6. Used permissions
7. Suspicious API calls
8. Network addresses

The features are embedded into an N-dimensional vector space, where each element is either 0 or 1. Each element corresponds to a string, with 1 representing the presence of the string and 0 representing its absence. The extracted feature vector $\mathbf{x}$ is denoted as

$$\mathbf{x} = (\cdots\ 0\ 1\ \cdots\ 0\ 1\ \cdots)\,.$$

The feature vector can be used as input for a machine-learning model.

Classification Models Our experiments use random forest [6], which is well known for its excellent classification performance and can be applied to many tasks, including malware detection. Random forest is an ensemble of decision trees. Each decision tree is built using a randomly sampled subset of data and features. By creating an ensemble of many decision trees, random forest achieves high classification performance even when the dimensions of feature vectors exceed the dataset size. Furthermore, the SHAP package [14] associated with Ref. [15] provides a high-speed algorithm called TreeExplainer for tree ensemble methods, including random forests.

Table 9. Number of samples in each dataset

			Malicious	Benign
Update 1	Pre-update	Unbiased	416	3,732
	Post-update	Unbiased	416	3,809
		Biased-Time	425	3,847
		Biased-Family	424	3,809
		Biased-Antivirus	417	3,841
	Test	Unbiased	595	5,322
Update 2	Pre-update	Unbiased	416	3,809
	Post-update	Unbiased	423	3,816
		Biased-Time	423	3,850
		Biased-Family	423	3,854
		Biased-Antivirus	421	3,837
	Test	Unbiased	598	5,302
Update 3	Pre-update	Unbiased	423	3,816
	Post-update	Unbiased	429	3,814
		Biased-Time	432	3,854
		Biased-Family	431	3,814
		Biased-Antivirus	431	3,843
	Test	Unbiased	532	4,628

Hyperparameter Optimization When training random forest models, we conduct a grid search for each model to determine the best combination of parameters among the following candidates:

1. Number of trees: 10, 100, 200, 300, 400.
2. Maximum depth of each tree: 10, 100, 300, 500.
3. Ratio of features used for each tree: 0.02, 0.05, 0.07, 0.1, 0.2.
4. Minimum number of samples required at a leaf node: 5, 7, 10, 20.

Each candidate combination is validated using five-fold cross validation. Specifically, we calculated an average of five AUC scores for each combination and selected the best combination in terms of the average AUC score as the result of the grid search.

References

1. Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., Giacinto, G.: Novel feature extraction, selection and fusion for effective malware family classification. In: Proceedings of the 6th ACM Conference on Data and Application Security and Privacy. pp. 183–194 (2016)
2. Akaike, H.: Information theory and an extension of the maximum likelihood principle. In: Selected Papers of Hirotugu Akaike, pp. 199–213. Springer (1998)

3. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: Collecting millions of android apps for the research community. In: Proceedings of the 13th IEEE/ACM Working Conference on Mining Software Repositories. pp. 468–471 (2016)
4. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: Drebin: Effective and explainable detection of android malware in your pocket. In: Proceedings of the 2014 Network and Distributed System Security Symposium (2014)
5. Bouktif, S., Fiaz, A., Ouni, A., Serhani, M.A.: Single and multi-sequence deep learning models for short and medium term electric load forecasting. Energies **12**(1), 149 (2019)
6. Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (2001)
7. Burnham, K.P., Anderson, D.R.: Multimodel inference: understanding aic and bic in model selection. Sociological Methods & Research **33**(2), 261–304 (2004)
8. Canali, D., Cova, M., Vigna, G., Kruegel, C.: Prophiler: a fast filter for the large-scale detection of malicious web pages. In: Proceedings of the 20th International Conference on World Wide Web. pp. 197–206 (2011)
9. Friedman, J.H., Meulman, J.J.: Multiple additive regression trees with application in epidemiology. Statistics in Medicine **22**(9), 1365–1381 (2003)
10. Google Cloud: Mlops: Continuous delivery and automation pipelines in machine learning (2020), https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning
11. Jaccard, P.: The distribution of the flora in the alpine zone. 1. New phytologist **11**(2), 37–50 (1912)
12. Jordaney, R., Sharad, K., Dash, S.K., Wang, Z., Papini, D., Nouretdinov, I., Cavallaro, L.: Transcend: Detecting concept drift in malware classification models. In: Proceedings of the 26th USENIX Security Symposium. pp. 625–642 (2017)
13. Karlaš, B., Interlandi, M., Renggli, C., Wu, W., Zhang, C., Mukunthu Iyappan Babu, D., Edwards, J., Lauren, C., Xu, A., Weimer, M.: Building continuous integration services for machine learning. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 2407–2415 (2020)
14. Lundberg, S.M., Erion, G., Chen, H., DeGrave, A., Prutkin, J.M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N., Lee, S.I.: From local explanations to global understanding with explainable ai for trees. Nature Machine Intelligence **2**(1), 2522–5839 (2020)
15. Lundberg, S.M., Erion, G.G., Lee, S.I.: Consistent individualized feature attribution for tree ensembles. arXiv preprint arXiv:1802.03888 (2018)
16. Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. In: Proceedings of the 31st Advances in Neural Information Processing Systems. pp. 4765–4774 (2017)
17. Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Beyond blacklists: learning to detect malicious web sites from suspicious urls. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1245–1254 (2009)
18. Miller, B., Kantchelian, A., Tschantz, M.C., Afroz, S., Bachwani, R., Faizullabhoy, R., Huang, L., Shankar, V., Wu, T., Yiu, G.: Reviewer integration and performance measurement for malware detection. In: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 122–141 (2016)
19. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V.: Scikit-learn: Machine learning in python. Journal of Machine Learning Research **12**, 2825–2830 (2011)

20. Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L.: TESSERACT: Eliminating experimental bias in malware classification across space and time. In: Proceedings of the 28th USENIX Security Symposium. pp. 729–746 (2019)
21. Ribeiro, M.T., Singh, S., Guestrin, C.: "why should I trust you?": Explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1135–1144 (2016)
22. Schwarz, G.: Estimating the dimension of a model. The Annals of Statistics **6**(2), 461–464 (1978)
23. Sood, G.: virustotal: R Client for the virustotal API (2017), r package version 0.2.1
24. Tsymbal, A.: The problem of concept drift: definitions and related work. Computer Science Department, Trinity College Dublin **106**(2), 58 (2004)