EAGLE: Evasion Attacks Guided by Local Explanations against Android Malware Classification

Zhan Shu, Guanhua Yan, *Member, IEEE*

Abstract—With machine learning techniques widely used to automate Android malware detection, it is important to investigate the robustness of these methods against evasion attacks. A recent work has proposed a novel problem-space attack on Android malware classifiers, where adversarial examples are generated by transforming Android malware samples while satisfying practical constraints. Aimed to address its limitations, we propose a new attack called EAGLE (**E**vasion **A**ttacks **G**uided by **L**ocal **E**xplanations), whose key idea is to leverage local explanations to guide the search for adversarial examples. We present a generic algorithmic framework for EAGLE attacks, which can be customized with specific feature increase and decrease operations to evade Android malware classifiers trained on different types of count features. We overcome practical challenges in implementing these operations for four different types of Android malware classifiers. Using two Android malware datasets, our results show that EAGLE attacks can be highly effective at finding functionable adversarial examples. We study the attack transferrability of malware variants created by EAGLE attacks across classifiers built with different classification models or trained on different types of count features. Our research further demonstrates that ensemble classifiers trained from multiple types of count features are not immune to EAGLE attacks. We also discuss possible defense mechanisms against EAGLE attacks.

Keywords—Malware, mobile security, adversarial machine learning

1 Introduction

The popularity of Android devices has made them favorable targets by 98% of mobile malware attacks according to a recent cyber threat report [46]. The numerous Android malware attacks have motivated an abundance of research efforts that apply machine learning to detect malicious Android applications [31], [44], [33], [51], [34], [41], [10], [59], [57], [24], [29], [37]. Among the plethora of efforts investigating the robustness of machine learning in an adversarial environment [32], [40], [17], some focused on generation of adversarial samples for evading Android malware classifiers [27], [21], [62]. Particularly, the seminal work by Pierazzi *et al.* [45] proposed a novel problem-space attack on Android malware classifiers, where adversarial examples are generated by transforming Android malware samples while satisfying practical constraints such as available transformations and preserved semantics.

Although the method in [45] addresses the feature-mapping problem by generating practical samples capable of evading the DREBIN malware classifier [20] and its hardened variant [21], it has the following three limitations. First, as it attacks only malware classifiers trained on *binary* features, when the target malware classifier is trained on non-binary features (e.g., those considered in [51], [35], [39], [34]), the attack method becomes ineffective. Second, the method requires the attacker to know the internal parameters of the target classifier in order to identify the top benign features for the evasion attacks. Last, it poses inconvenience to the attacker as a significant number of benign Android applications are needed to extract gadgets with the top benign features through program slicing operations.

Inspired by these challenges, the goal of this work is to develop a generic approach called EAGLE (**E**vasion **A**ttacks **G**uided by **L**ocal **E**xplanations) to generate practical samples that can evade Android malware classifiers trained on arbitrary count features. Count features, which can be deemed as an extension of binary features, capture not only whether a feature exists in an Android sample but also how many times it appears in it. EAGLE attacks apply local explanation tools such as LIME (Local Interpretable Model-agnostic Explanations) [47] to interpret why a malware sample is classified as malicious in its local neighborhood. Given the important count features which are found by local explanation tools to have positively contributed to the target classifier's detection of the original sample, new variants are created by perturbing their values, in hopes that some of them may cross the decision boundary used by the classifier. As EAGLE attacks apply local perturbation to search for adversarial examples, they do not require the attacker to transplant gadgets with good features that can separate benign Android applications from malicious ones.

In this work we present a generic algorithmic framework for EAGLE attacks against Android malware classifiers trained on arbitrary count features. For each type of count features used by the target classifier, EAGLE

Z. Shu and G. Yan are with the Department of Computer Science, Binghamton University, State University of New York, Binghamton, NY, 13902, USA. Email: {zshu1, ghyan}@binghamton.edu.

attacks can be customized with two types of operations to increase and decrease the values of those important features identified by local explanations, respectively. Our framework also adopts a hill-climbing strategy to search for variants mutated from the original malware sample whose malicious scores are gradually increased until an evasive example is found. As the attack strategy is decoupled from the specific machine learning model used by the target Android malware classifier, our attack framework is model agnostic. To demonstrate the feasibility of EAGLE attacks, we construct four types of count features which have been well received for Android malware detection in the literature. They include permissions included in Android manifest files [31], n-gram opcode sequences extracted from disassembled Android applications [34], API calls which are highly informative of malware behaviors [10], and function call graphs (FCGs) containing structural information [24]. For each of these count features, we design and implement operations to increase and decrease selected feature values based on LIME's explanation results.

To evaluate the performance of EAGLE attacks, we use two datasets obtained from different sources, each containing thousands of benign/malicious Android applications. We also consider two types of classification models, Random Forest (RF) and Multilayer Perceptron (MP). Based on these datasets and classification models, we conduct multiple sets of experiments. In the first set of experiments, we apply EAGLE attacks against different combinations of classification models and count features. The experimental results show that their successful rates are high for both evaluation datasets, irrespective of the classification model or the type of count features used. Further comparison experiments show that EAGLE attacks consistently outperform two other alternative attack methods in generating adversarial examples against both the RF and MP classifiers.

In the second set of experiments, we investigate the transferability of EAGLE attacks across different classification models and count feature types. Particularly, we observe that attack transferability across classification models is *asymmetric*: adversarial examples generated from attacking the RF models can be transferred to attack the corresponding MP models with an average successful rate of 66.3% while those obtained from attacking the MP models succeed in attacking the corresponding RF models with a miserably successful rate of 8.38%.

In the third set of experiments, we study the robustness of ensemble classifiers which combine the prediction results from multiple classification models, each trained on a particular type of count features. Our results show that in direct attacks where surrogate and target classifiers are trained with the same model and the same type of features, the successful rate can as high as between 78.7% and 100%, suggesting that ensemble classifiers trained on a set of count features are not immune to successful EAGLE attacks.

We perform the final set of experiments to study the effectiveness of the adversarial training defense mechanism against EAGLE attacks. We observe that malware classifiers built upon permission and API call features can be robust against EAGLE attacks if they are trained with additional adversarial examples. As these features can be easily obfuscated, they may not seem useful to train robust Android malware detectors [55]. However, our experimental results show that malware classifiers trained on such features can still help improve the overall robustness of an ensemble detection approach after they are hardened by adversarial training.

In a nutshell, our main contributions can be summarized as follows: **(1)** We develop a generic algorithmic framework for EAGLE attacks against Android malware classifiers, which can be customized with specific feature increase and decrease operations based on what count features are used by the target classifier. **(2)** We implement EAGLE attacks against four Android malware classifiers trained on different types of count features. For each of these attacks, we overcome the practical challenges in generating practical samples with targeted feature values. **(3)** Using two Android malware datasets, we demonstrate the feasibility of EAGLE attacks in creating adversarial samples against the four count feature-based Android malware classifiers, all with high successful evasion rates. We also compare the performance of EAGLE attacks against those of two alternative attack methods. **(4)** We investigate attack transferrability of adversarial samples produced from EAGLE attacks across classifiers built with different classification models or feature types. **(5)** We demonstrate that ensemble classifiers may still be vulnerable to EAGLE attacks. **(6)** We investigate the effectiveness of adversarial training in defending against EAGLE attacks.

The remaining of this paper is organized as follows. Section 2 surveys related work. Section 3 introduces background knowledge about Android malware classifiers and LIME. Section 4 presents the problem formulation. Section 5 describes the algorithmic framework of EAGLE attacks and Section 6 elaborates on their implementation details. We show experimental results in Sections 7-9. We discuss the defense implications in Section 10. Section 11 makes concluding remarks and discusses the limitations of this work.

2 Related Work

2.1 Machine learning for Android malware detection

Previous works have explored a variety of features extracted from Android applications, including permissions [31], [44], [33], opcodes [34], [41], API calls [10], [59], [57], and FCGs [24], [29], [37], to detect Android malware. The features used in these works for Android malware detection are either binary or non-binary. Although some works have shown that the binary features can be used to build highly predictive Android malware detection models (e.g., [20], [55]), others have demonstrated that non-binary ones, such as counts of

N-gram API call sequences [51], occurrences of string literals [35], numbers of different loops sharing the same semantic tags [39], and frequencies of N-gram opcode sequences [34], also possess high discriminatory power. Some of these malware classifiers can be potentially targeted by EAGLE attacks as they use count features.

Noticing that some malware features can be easily obfuscated, the work in [55] advocates inclusion of different application characteristics features which indicate either structural inconsistencies or logical inconsistencies for detection of obfuscated Android malware. Admittedly, malware variants created by EAGLE attacks in this work may be still detectable by the classifiers proposed in [55]. However, if the attacker knows that the application characteristics features proposed in [55] have been used by the target malware classifier, EAGLE attacks can still be performed to identify which of them have played the most important roles in malware detection and then perturb their values to search for evasive variants.

A few works apply ensemble learning to classifying Android applications for either malware detection or family identification. The *EC2* algorithm combines supervised learning and unsupervised clustering to classify a malware sample into both large and small families [18]. The malware features considered in their work include both static ones (e.g., permissions, authors and application components) and dynamic ones (i.e., n-gram count features belonging to four groups: read/write operations, system-related operations, network-related operations, and SMS-related operations). As these are all count features, in principle the supervised malware classification module in EC2 is not immune to EAGLE attacks. Ficco's approach [23] stacks multiple malware detectors, including both specialized ones that detect individual malware families and generic ones that distinguish malware from benign applications to optimize malware detection accuracy. These malware detectors use diverse feature types, some of which are count features (e.g., API call frequency and network traffic characteristics) while others are not. Therefore, EAGLE attacks can be launched against some individual classifiers used in the ensemble detector but not all of them.

2.2 Evasion attacks against malware classifiers

The robustness of machine learning methods against evasion attacks has been studied for PDF malware [16], [54], [60], PE malware [12], [14], [36], IoT malware [11], and Android malware [27], [62]. Among these efforts, a few have considered the practical challenges in generating adversarial examples against PE malware classifiers in the problem space [14], [36], [38], [53]. However, the technical challenges involved evading machine learning-based PE malware detectors differ from those in evasion attacks against Android malware classifiers, which are the primary focus of this work.

To evade Android malware classifiers. the work in [27] applies the attack method proposed in [43] on classifiers trained on binary features while keeping the malware program's malicious functionality. Its reported misclassification rate ranges between 63% and 69% on the Drebin dataset [9]. Compared with the work in [27], ours has made the following improvements. First, EAGLE attacks are model agnostic but the method proposed in [27] is not because it relies on the target model's Jacobian matrix to identify what features should be modified. Due to this constraint, it cannot be directly applied to evade the Random Forest classifier considered in this work. Second, EAGLE attacks can be performed on Android malware classifiers trained on arbitrary count features, but the method proposed in [27] works only on those with binary features. Third, the successful misclassification rates seen from EAGLE attacks are significantly higher than those results achieved in [43].

Closely related to our work is the recent effort by Pierazzi *et al.* [45]. To overcome the so-called inverse feature-mapping problem, their approach identifies practical constraints in constructing functionable Android malware variants for evading Android malware classifiers. The key idea of their proposed method is to extract gadgets with benign features from legitimate Android applications and then transplant them into malware programs, allowing the modified ones to bypass the detection of a target malware classifier. The gadgets are carefully added to the malware program to preserve its original semantics while ensuring that the modified programs are robust against preprocessing techniques. EAGLE attacks address the three limitations of Pierazzi's approach. First, they can be used to evade any Android malware classifier which is trained on arbitrary count features, as long as it is plausible to construct methods that can increase and/or decrease their occurrences in the problem space. Second, EAGLE attacks allow the attacker to treat the target classifier as a blackbox and thus do not need him to know its internal parameters. In contrast, the work in [45] assumes that the attacker knows the top benign features based on the negative weights in the target classifier. Finally, EAGLE attacks do not need a significant number of benign Android applications to extract gadgets with benign features.

2.3 XAI applications in malware classification

Explainable Artifical Intelligence (XAI) methods have been used to explain predictions made by malware classifiers [28], [42], [30]. The work in [20] investigated various binary feature sets extracted from Android manifest files and disassembled code, which are used by a linear SVM model to detect Android malware with high accuracy. The features with the largest weights in the linear SVM model are used to explain why an application is detected as malicious. This explanation method is neither model-agnostic nor a local explanation approach needed by EAGLE attacks.

Closely related to our work are two recent efforts which have also applied XAI methods in creation of adversarial examples against malware classifiers. The

work in [48] uses an XAI-based algorithm to select important features in PE malware classification by a substitute model and then modifies "easily modifiable" features with predefined feature values to create evasive samples; their successful evasion rates however seem to be poor (below 38%). In comparison, our work targets evasion of Android malware classifiers with a much higher successful rate. The work in [50] applies XAI methods to find a greedy combination of important features for perturbation in poisoning attacks against malware classifiers. Different from our work, it focuses on poisoning attacks instead of evasion attacks. In terms of XAI methods, common to both works in [48], [50] is the use of SHAP (SHapley Additive exPlanations), which can also be used for local explanations. As SHAP requires global permutations to achieve high local explanation accuracy, its high computational overhead makes it a less appropriate method for EAGLE attacks than LIME (see Section 7.5).

3 BACKGROUND

3.1 Android malware classifiers

In this work we consider the following Android malware classifiers, each trained on a type of count features.

Permission classifier. Permissions requested by an Android application can be extracted from its manifest file. These permissions have been used as Boolean features to train Android malware classifiers [31], [44], [33].

N-gram opcode classifier. Opcodes disassembled from malware executable files have been widely used in machine learning-based malware detection [61], [34], [41], [63]. Particularly, n-gram opcode classifiers train prediction models from counts or frequencies of any n-gram opcode sequences contained in consecutive instructions.

API call classifier. API calls are useful for malware detection because they can reveal malicious behaviors of suspicious Android applications [10], [59], [49], [57]. In this work, we consider an API call classifier similar to DroidAPIMiner [10], which is trained from dangerous API calls extracted through static analysis.

Function call graph classifier. Structural information contained within malware's FCGs has also been used for Android malware classification [24], [29], [37]. Particularly, the pioneering Android malware classifier proposed by Gascon *et al.* works as follows. Through static analysis, each Android application is represented as a labelled FCG, denoted as a 4-tuple $G = (V, E, L, l)$, where V includes all functions in the application and E call relationships among these functions. With L representing a multiset of labels for the FCG, $l : V \to L$ is a labelling function which assigns a label to each node in the graph. The label of each function node in G is a m-bit vector, where m is the number of instruction categories considered (e.g., branch, move, and invoke). For each of these categories, if a function contains any instruction that belongs to this category, the corresponding bit in the vector is set to 1; otherwise it is 0.

The structural information contained within an FCG is considered when computing the hash for each node $v \in V$ and its neighboring nodes V_v as follows:

$$h(v) = r(l(v)) \oplus \left(\bigoplus_{z \in V_v} l(z) \right), \tag{1}$$

where r denotes a single-bit left rotation. We call $l(v)$ and $h(v)$ the *original label* and the *hash label* of v, respectively.

The *neighborhood hash* of graph G, denoted by $G_h = (V, E, L_h, h(\cdot))$, is derived by replacing the original label of each node $v \in V$ with its hash value calculated according to Eq. (1). L_h is the multiset of labels in G_h. Given two FCGs G_h and G'_h, the graph kernel $K(G_h, G'_h)$ can be defined as the size of the intersections of their label multisets, L_h and L'_h. That is to say, $K(G_h, G'_h) = |L_h \cap L'_h|$.

Let $H = \{a_1, ..., a_N\}$ be the histogram of multiset L_h with where a_i is the occurrence of the i-th hash in G_h. Given that N is the number of bins in histogram H and M is the maximum number among all bins, histogram H can be mapped to an NM-dimensional vector $\phi(H)$:

$$\phi(H) = (\mathbf{1}_{a_1}, \mathbf{0}_{M-a_1}, \mathbf{1}_{a_2}, \mathbf{0}_{M-a_2}, ..., \mathbf{1}_{a_N}, \mathbf{0}_{M-a_N}), \tag{2}$$

where $\mathbf{1}_k$ and $\mathbf{0}_k$ denote k consecutive 1's and 0's, respectively. Assuming that H and H' are the histogram of multiset L_h and L'_h, respectively, it is easy to see that $K(G_h, G'_h)$ is equal to the dot product of $\phi(H)$ and $\phi(H')$, i.e., $K(G_h, G'_h) =< \phi(H), \phi(H') >$.

With the above kernel function, a linear SVC (Support Vector Classifier) classifies G_h as malicious or benign.

3.2 LIME

Consider a binary classification model like malware detection, $f : R^d \to R$, where $f(x)$ gives the probability that x belongs to the relevant class. Explanation of its predictions is also defined as a model $g \in G$, where G is a class of potentially interpretable models (e.g., decision trees). The domain of g is $\{0, 1\}^{d'}$, so explanations are presented as the absence or presence of the d' interpretable components.

For any data point x, LIME finds the optimal local explanation for the prediction by model f by solving:

$$\xi(x) = \text{argmin}_{g \in G} \mathcal{L}(f, g, \Pi_x) + \Omega(g), \tag{3}$$

where $\Pi_x(z)$ measures the distance between z and x, $\mathcal{L}(f, g, \Pi_x)$ measures how closely the explanation model g approximates the original model f in the neighborhood defined by Π_x, and $\Omega(g)$ measures the complexity of the explanation model g.

LIME uniformly samples the area around the binary representation of x, perturbs it to get $z' \in \{0, 1\}^{d'}$, recovers the sample in the original representation space $z \in R^d$, and uses $f(z)$ as the label for the explanation model. Using a perturbed dataset $\mathcal{Z}'$ constructed as above, LIME solves the optimization problem in Eq. (3) to get an explanation $\xi(x)$.

LIME uses sparse linear models for local explanations with $g(z') = w_g \cdot z'$. A locally weighted square loss function is used for $\mathcal{L}$:

$$\mathcal{L}(f, g, \Pi_x) = \sum_{z, z' \in \mathcal{Z}} \Pi_x(z)(f(z) - g(z'))^2, \quad (4)$$

where $\Pi_x(z) = exp(-D(x,z)^2/\sigma^2)$ is an exponential smoothing kernel for some distance function D.

LIME solves the optimization problem in Eq. (3) with the K-LASSO algorithm, which selects K interpretable components for local explanation with their weights indicating their relative relevance.

The continuous data for each feature in the perturbed dataset $\mathcal{Z}'$ are first discretized into bins, each containing a range of values. By default a quartile discretizer is used. Next a corresponding binary interpretable component is defined for each feature to indicate whether an instance's feature value falls into the same bin as instance x whose prediction result is to be explained.

Given model f, the set of features extracted from instance x, which is denoted by Q_x, and explanation length K, the LIME tool produces a set of K weighted feature rules, each in the form of $(t_i, [\theta_i^{min}, \theta_i^{max}), w_i)$ where t_i gives the feature identity selected at the i-th place (i.e., $1 \leq t_i \leq |Q_x|$), $[\theta_i^{min}, \theta_i^{max})$ is the value range that this feature falls into for instance x after discretization, and w_i is this feature's weight learned by the K-LASSO algorithm. Moreover, having weight $w_i > 0$ means that the i-th feature's value supports the model's classification result while weight $w_i < 0$ suggests otherwise. For ease of presentation, we use notation $LIME(f, Q_x, K)$ to represent the set of weighted feature rules produced by LIME.

4 Problem Formulation

Consider an Android malware detector, $f : R^d \rightarrow R$, which, given a d-dimensional count feature vector extracted from an Android application, predicts the probability of its being malicious (i.e, *malicious probability score*). Let Q_x denote the count feature vector extracted from any instance x. Without loss of generality, x is detected to be malicious if $f(Q_x) > 0.5$ or benign otherwise.

Given an Android malware sample x_0 that can be successfully detected by model f as malicious (i.e., $f(Q_{x_0}) > 0.5$), the attacker's goal is to obfuscate x_0 into a different instance x' such that $f(Q_{x'}) \leq 0.5$, where x' should be functionally equivalent to x_0. Informally speaking, functional equivalence means that when x' and x_0 are executed in the same environment, their behaviors perceived by the environment should be almost always the same. Precise functional equivalence may be hard to achieve due to non-determinism of execution environments or side effects of obfuscation (e.g., changes of file sizes), but from the attacker's perspective, as long as the new instance x' can cause the same damage to the environment as x_0 while evading detection of model f, it is deemed as successful.

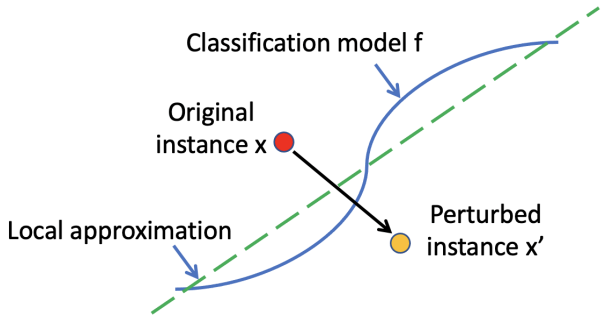


Fig. 1. Intuition behind an EAGLE attack

Threat model. We make the following assumptions about the attacker: (1) *Source code unavailable*: Without the source code of the original sample x_0, the attacker needs to disassemble/decompile it first, make appropriate modifications, and then reassemble/recompile the code to generate a new variant which hopefully can bypass the detection of model f. (2) *Soft-label blackbox attack*: We assume that the internal details of the model are not revealed to the attacker, but the attacker can make queries to acquire the corresponding soft-label decisions (i.e., output probabilities) from the model [19]. *(3) Ability to evade query-based anomaly detection*: As the attacker needs to query the target malware classifier for soft labels, it is possible that an anomaly detection module can be deployed to catch and thereby suspend suspicious queries. We assume that the attacker is capable of deploying countermeasures to evade such query-based anomaly detection. For example, the attacker can spread these queries over multiple attack machines to avoid excessive queries originating from a single one.

5 Algorithm Description

In this section, we explain the intuition behind EAGLE attacks, their workflow, and their algorithmic details.

5.1 Intuition

EAGLE attacks use local explanations to guide mutations of the original Android malware instance x_0 within its neighboring feature space into variants that are likely to evade malware classifier f. The attack algorithm is built upon the LIME tool, although other similar methods are also applicable as long as they can identify the most important K features in classifying any individual instance in its neighborhood by model f.

Our key intuition, which is illustrated in Figure 1, is that we can use the local approximation model as the surrogate to perturb the features of the original instance x for evasion attacks. As the explanation provided by LIME includes the top K features and their value ranges, if we perturb those that have positively contributed to classification of x as malicious and perturb their values *outside* their respective ranges, the new variant x' may cross the decision boundary of the local approximation

model, thus flipping its prediction result. If this surrogate model can approximate the decision boundary of classification model f closely in the locality of instance x, it is likely that the perturbed instance x' can evade the detection by classification model f.

5.2 Workflow

The workflow of an EAGLE attack is illustrated in Figure 2. It repeatedly performs mutation operations on variants of the original sample x_0 until one is found to evade classification model f or the maximum number of iterations is reached. A working instance x is first initialized as the original sample x_0. Also, parameter K, the number of top features selected by the LIME tool, is initialized to be K_0.

In each iteration, the EAGLE attack first extracts the count feature vector Q_x from APK x as the input for classifier f. Given the explanation length K, count feature vector Q_x, and model f, the LIME tool outputs the top K count features along with their weights and value ranges (see Section 3.2). Based on these outputs, the mutation module searches for a best variant x' whose probability of being malicious (i.e., $f(Q_{x'})$) is the lowest. If the mutation module fails to find a better candidate x' than the current instance x, the original one x is still used but parameter K is increased by δ for the next iteration. For the best candidate x' returned by the mutation module, if the classification model f still detects it as malicious (i.e., $f(Q_{x'} > 0.5)$, the working sample x is replaced by x' and the above process is repeated; otherwise, the algorithm simply returns x'.

The single DEX (Dalvik Executable) file called *classes.dex* contained in APK x includes the original malware's application logic compiled into Dalvik bytecode, which is executable within a Dalvik Virtual Machine (VM). Due to lack of source code, the mutation module in Figure 2 first disassembles this DEX file in APK x into Smali code using Apktool [1], a popular reverse engineering tool for Android APKs. Based on the type of features used by classification model f, the mutation module further modifies the Smali code disassembled from APK x appropriately, reassembles the changed Smali code into an APK using Apktool again, and signs this new APK file with the attacker's own key (or any other fake key) using the jarsigner tool. Finally, this signed APK file is executed within an emulated Android device to test whether it is functionable or not.

5.3 Mutation algorithm

Although the technique used by the mutation module to modify Smali code varies with the type of features needed by model f, it follows a similar skeleton as described in Algorithm 1. The mutation algorithm takes a list of Inc-Dec function pairs, denoted by Γ, as input. For each Inc-Dec function pair in Γ, the algorithm searches for a better variant whose malicious probability score is lower than the current one x with increasing perturbation ranges (Lines 3-20).

Within its inner loop (Lines 6-16), n variants, each denoted by x'', are generated from the working instance x within the current perturbation range controlled by variable z, which is initialized to be α (Line 4). If any of these n variants has a lower malicious probability score output by model f than the working instance x, it is returned by the mutation module (Line 18); otherwise, the current perturbation range controlled by parameter z is enlarged by a factor of α (Line 20) and the process is repeated until parameter z becomes greater than a predefined value z_{max}. If no better candidate is found than instance x after z exceeds z_{max}, the next Inc-Dec function pair in Γ is used to repeat the above process. If no better candidate is found than instance x after all Inc-Dec function pairs in Γ have been tried, the original sample x is returned but parameter K, the number of top features found by LIME, is increased by δ (Line 21).

Algorithm 1: Mutation algorithm

Input: x: (working instance), Q_x: feature vector, R: LIME output (i.e., $R = LIME(f, Q_x, K)$), f: classification model, n: mutation count, α: perturbation scale factor, z_{max}: maximum perturbation scale, Γ: list of Inc-Dec function pairs to be applied in mutation, δ: increase step size for parameter K used by LIME

Output: x': best variant found, Δ_K: increase in K

```
1  function mutate(x, Q_x, R, f, n, α, z_max, Γ, δ)
2      x' ← x, s_min ← f(Q_x)
3      for l ← 1, |Γ| do
4          z ← α, Inc ← Γ[l][1], Dec ← Γ[l][2]
5          while z ≤ z_max do
6              for i ← 1, n do
7                  x'' ← x
8                  for j ← 1, |R| do
9                      if R[j][3] > 0 then
10                         r ← perturb(R[j][2], z)
11                         if Q_x[R[j][1]] < r then
12                             x'' ← Inc(x'', R[j][1]], r −
                                   Q_x[R[j][1]])
13                         else
14                             x'' ←
                                   Dec(x'', R[j][1]], Q_x[R[j][1]]−
                                   r)
15                 if f(Q_x'') < s_min then
16                     s_min ← f(Q_x''), x' ← x''
17             if s_min < f(Q_x) then
18                 return (x', Δ_K = 0)
19             else
20                 z ← αz
21     return (x', Δ_K = δ)
```

For each variant x'', it is modified from the current working sample x by perturbing selected features in R,

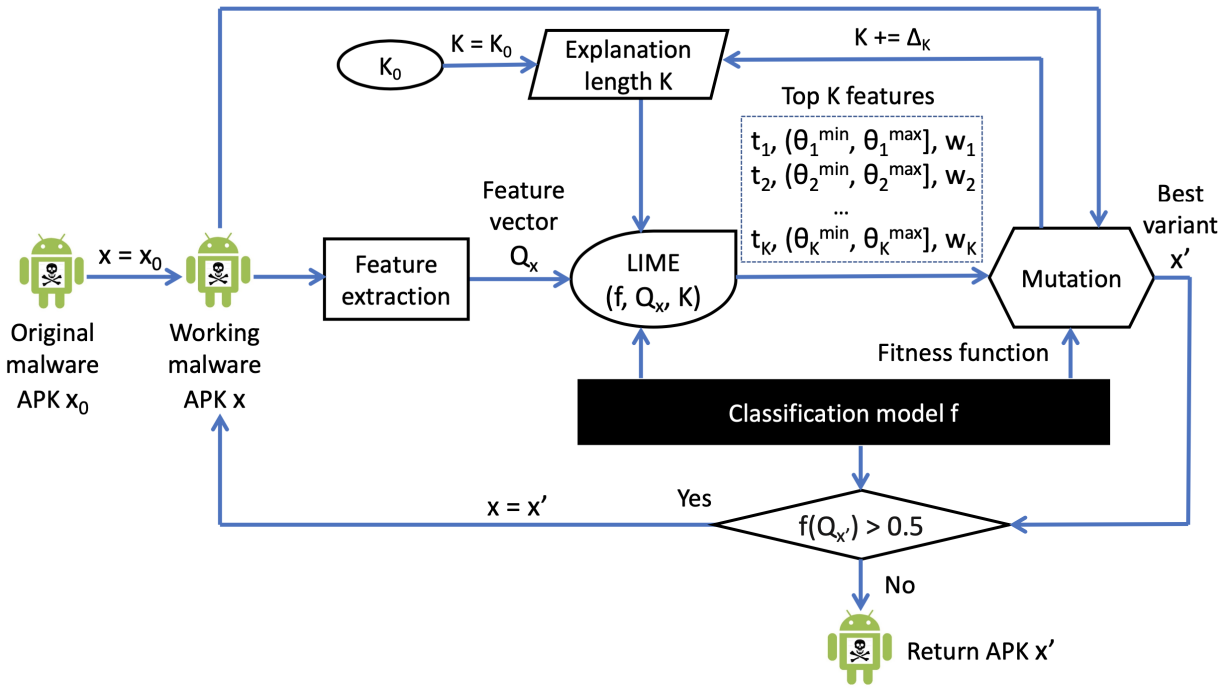


Fig. 2. Workflow of a EAGLE attack against classification model f

which is returned from the LIME tool (Lines 8-14). Recall that the LIME tool returns a set of triples including the selected feature id, its feature value range after discretization, and its weight (see Section 3.2); if a feature has a positive (negative) weight, it supports (contradicts) the classification of sample x into the malicious category. As the goal of mutation is to flip classification model f's prediction from *malicious* to *benign*, the mutation module randomly perturbs the value of each of those top K features with only positive weights (Line 9) to ensure that its new value falls outside the original range. This is done by calling the `perturb` function (Line 10).

The pseudocode of the `perturb` function is given in Algorithm 2. It takes an integer value range $[v_{min}, v_{max})$, where v_{min} and v_{max} can be ∞, and perturbation scale factor z as input, and returns a non-negative integer value. If v_{max} is $+\infty$, then the perturbed value is uniformly chosen from $[0, v_{min}-1]$ (Lines 2-3). Perturbation considers only non-negative integer values to simplify creation of real samples. Hence, v_{min}, if negative, is reassigned to be 0 (Lines 4-5). Within a perturbation range whose size is $z \cdot (v_{max} - v_{min})$, a random number, r, is uniformly chosen (Line 6) to indicate the deviation from the original range $[v_{min}, v_{max})$. Depending on the relationships among r, v_{min}, and $v_{max} - v_{min}$, the perturbed value can be chosen from either side of the original value range (Lines 7-12).

For each of those count features whose values are perturbed, we define two types of procedures, $Inc(x, t, d)$ and $Dec(x, t, d)$, which produce a variant of instance x whose t-th feature value in Q_x is increased and decreased by d, respectively. As the implementations of these two procedures vary with the feature types used by classification model f, we leave their details to Section 6.

Algorithm 2: Perturbation algorithm

Input: Integer value range $V = [v_{min}, v_{max})$, perturbation scale factor z
Output: A perturbed non-negative integer value

```
1  function perturb(V, z)
2      if v_max = +∞ then
3          return uniform([0, v_min − 1])
4      if v_min < 0 then
5          v_min ← 0
6      r ← uniform([0, z · (v_max − v_min) − 1])
7      if (v_max − v_min)/2 ≤ r < v_min then
8          return v_min − r − 1
9      else if r ≥ v_min then
10         return v_max + r − v_min
11     else
12         return v_max + (v_max − v_min) − r
```

Complexity analysis. Assuming that a random number can be generated within $O(1)$ time, the time complexity of the `perturb` function shown in Algorithm 2 is $O(1)$. Therefore, the time complexity of the `mutate` function in Algorithm 1 is $O(n \cdot |\Gamma| \cdot log_{\alpha} z_{max} \cdot |R|)$.

5.4 Search strategy

Even the best variant x' found by the mutation module may not evade the detection of classification model f due to the following reasons. *First*, perturbing the top K features may not be sufficient to sway the decision by classification model f. We explain this with a simple linear classification model $y = w_1x_1 + w_2x_2 + ... + w_nx_n$ with $x_i \in \{-1, 1\}$ for each $i \in [1, ..., n]$ and $w_1 > w_2 >$

$... > w_n$. The model predicts *malicious* if $y > 0$ or *benign* otherwise. Considering $K = 1$, obviously feature x_1 is the most important one because its weight is the largest. Suppose that a malware sample has all of its features to be 1. Perturbing the value of its feature x_1 from 1 to -1 does not change the model's prediction result if $w_1 < \sum_{i=2}^{n} w_i$.

Second, LIME uses a sparse linear model to approximate the prediction by model f in the locality of an individual instance x. It is possible that a perturbed instance x' can evade the local approximate model but not the classification model f.

Third, each iteration of the mutation algorithm aims to perturb the working sample x to achieve a targeted feature value r for one of those top K count features. As we shall see in Section 6, there are practical constraints when modifying disassembled Smali code to match the targeted feature value r. Hence, it is possible that the features extracted from the best variant x' returned do not match their targeted values.

When variant x' fails to flip the prediction by classification model f, our method replaces the working sample x with x' and then repeats the mutation process. As the mutation algorithm always finds a variant x' whose fitness score is at least as good as that of the current working sample x, it applies a hill-climbing strategy to search for an optimal evasive sample. As this strategy may get stuck at a local optimum, the workflow can be repeated multiple times to circumvent this issue.

6 Practical Implementations

This section presents the implementation details of EAGLE attacks against four Android malware classifiers.

6.1 Permission classifier

For the permission classifier, we extract the list of permissions requested by each Android application in its manifest file. Let $P = \{P_t\}_{1 \le t \le |P|}$ denote an ordered set of Android permissions that can be requested by an Android app. The *binary feature vector* $Q_x^{p,b}$ is constructed in such way that $Q_x^{p,b}[t] = 1$ if permission P_t is present in APK instance x's manifest file or $Q_x^{p,b}[t] = 0$ otherwise.

To evade the permission classifier, we define list Γ in Algorithm 1 to be $[(Inc, Null)]$, where the $Null$ function does nothing. We consider only additions of new but unnecessary permissions into the manifest file for evasion attacks as removing those permissions requested by the original malware may break its functionality.

Implementation of $Inc(x, t, d)$: When evading the permission classifier, d must be one because $Q_x^{p,b}$ is a binary feature vector. Calling $Inc(x, t, d)$ simply adds permission P_t to x's manifest file if it is not included.

6.2 2-gram opcode classifier

We consider feature vectors constructed as follows for APK x. Let O be the entire set of opcodes used by Dalvik bytecode [2]. For each method in APK x's disassembled Smali code, we count the occurrence of each 2-gram opcode sequence. The *count feature vector* $Q_x^{o,c}$ is constructed by aggregating these counts over all the methods in the Smali code for each 2-gram opcode sequence. Let the t-th feature of $Q_x^{o,c}$ be (a_t, b_t), where $a_t \in O$ and $b_t \in O$. To evade the 2-gram opcode classifier, we define list Γ in Algorithm 1 to be $[(Inc, Dec), (Inc, Null)]$, where function $Null$ does nothing.

Implementation of $Inc(x, t, d)$: We add d new methods with different names into the Smali code disassembled from instance x. Each of these d new methods contains two dummy instructions with opcodes a_t and b_t, respectively. For either of these dummy instructions, its dependant data are also provided within the same method if necessary.

Implementation of $Dec(x, t, d)$: We search for d consecutive instructions with opcodes a_t and b_t in the Smali code of instance x and for each one found, a NOP instruction is added between them. If either a_t or b_t is already NOP, this procedure is ignored.

The Dec procedure described above has the side effect of increasing the occurrences of (a_t, NOP) and (NOP, b_t) in the new variant, suggesting that the feature vector of the new variant may not match exactly the intended value. To address this issue, the mutation algorithm performs another round of search with the same Inc function but replacing *Dec* with the $Null$ function.

6.3 API call classifier

For the API call classifier, we consider those types of APIs providing significant semantic information about Android applications' behaviors, which include application-specific resources APIs, Android framework resources APIs, DVM related resources APIs, system resources APIs, and utilities APIs [10]. For each of these APIs, the *count feature vector* $Q_x^{a,c}$ extracted from instance x includes the number of times it has been called in x's disassembled Smali code. Let $A = \{A_t\}_{1 \le t \le |A|}$ denote an ordered set of significant APIs considered for training the API call classifier. $Q_x^{a,c}[t]$ then gives the number of times API A_t has been called by instance x in its Smali code.

To evade the API call classifier, we define list Γ in Algorithm 1 to be $[(Inc, Dec)]$.

Implementation of $Inc(x, t, d)$: We add a new method, which makes d calls to API A_t.

Implementation of $Dec(x, t, d)$: We search for d calls to API A_t in instance x's Smali code and for each one discovered, we apply the reflection obfuscation technique to obscure the call. According to the study in [22], reflection has been applied by 48.3% of Android applications in Google Play, 49.7% of those downloaded from third-party markets, and 51.0% of Android malware, suggesting that its use is not a strong malware indicator.

Reflection can be performed through a sequence of `Class.ForName()`, `getMethod()`, and `invoke()` operations, each of which is called by `invoke-static` or

invoke-virtual in the added Smali code [22]. However, using invoke-*kind* directly for these operations, where *kind* indicates the kind of invocation (i.e., virtual, static, super, direct, or interface), is constrained because the numeric register indices for their arguments must fit within four bits. For example, consider the following simple Smali code (mut is the class name):

```
.method public static a(I)V
    .locals 100
    invoke-static{p0}, Lmut;->b(I)V
```

The above Smali code cannot be assembled because register $p0$ storing the input argument provided to method a is equivalent to register $v100$, whose register index cannot fit within four bits.

Therefore, when we construct arguments for the invoke-*kind* calls needed by reflection, the use of extra registers to store them may violate the aforementioned constraint. To circumvent this issue our implementation uses invoke-*kind*/range, which allows the register index to be as large as 65535 (i.e., 16 bits). A call to invoke-*kind*/range takes registers with a continuous range of indices as its input arguments. When preparing the arguments for getMethod() and invoke() in reflection, we use new-array, which requires one extra register whose index can be fit within four bits. Due to this, our current implementation can obfuscate any calls to API A_t in the original Smali code that are done through invoke-static or invoke-virtual and made by the methods using at most 15 registers. It is thus better than obfuscapk, which requires four extra 4-bit registers to reflect an API call [15].

6.4 FCG classifier

Given the FCG of instance x, the hash label of each node is computed according to Eq. (1). The *count feature vector* $Q_x^{g,c}$ includes the occurrence of each hash label derived from the FCG of instance x. To evade the FCG classifier, we define list Γ in Algorithm 1 to be $[(Inc, Dec)]$.

Implementation of $Inc(x, t, d)$: This procedure increases the occurrences of the t-th hash label by d. Examination of the FCG classifier's source code [25] reveals that the neighboring nodes of v, V_v, includes only those following outgoing edges in the FCG (i.e., the methods called by v). Our implementation of the evasion attack is faithful to this definition.

For ease of presentation, let I_k denote a label represented by a bit vector with all 0's except its k-th element being 1. The original work [24] defined 15 label categories. For each category, we define a separate *label method*. For example, for the *move* category (bit 2), we add the following label method whose method name *label2* indicates bit 2 used for this category:

```
.method public static label2()V
    .locals 50
    .prologue
    move v0, v1
    .end method
```

As it does not call any other methods, its label is I_2.

The following method has an empty hash label:

```
.method static public emptylabel()V
    .locals 2
    invoke-static {}, Lmut;->dup14()V
.end method
```

As it calls another method with invoke-static, which belongs to category invoke (bit 13), its original label is I_{13}. Method dup14 is a duplicate of label method label14 with a different name. According to Eq. (1) its new label after hashing is given by: $r(I_{13}) \oplus I_{14} = 0$.

We can adapt method emptylabel to achieve any target label h_t after the hashing operation given in Eq. (1): if its i-th bit is 1, meaning presence of instructions falling into the i-th label category, we add the following code after the invoke-static instruction in the code body of method emptylabel:

```
invoke-static {}, Lmut;->label[i]()V
```

where [i] is substituted with the value of i ($1 \leq i \leq 15$).

The purpose of using dup14 in method emptylabel is to deal with the complication when bit 13 (invoke) is set in target label h_t: using dup14 ensures that the new method adapted from emptylabel has one outgoing edge to dup14 and another one to label14. Otherwise, if label14 is used at both places, only one outgoing edge is added to label14, which does not achieve the effect of an empty original label. It is easy to verify that the hash label of the new method adapted from emptylabel must be h_t. To increase the occurrence of h_t by d, we can duplicate this method with different names.

Implementation of $Dec(x, t, d)$: Let h_t be the t-th hash label for H. We first search the FCG of instance x to find d methods whose hash labels are all h_t. For each of these methods discovered, we inject some dummy code to change its hash label to an insignificant hash label. An insignificant hash label means any one that does not belong to the top K ones found by the LIME tool.

Consider method v whose hash label is h_t. Suppose that our goal is to change its hash label to a target insignificant label, $h_{t'}$. Given the way in which the hash label is calculated in Eq. (1), we make up for the difference between h_t and $h_{t'}$, which is $h_t \oplus h_{t'}$ as follows: for each bit in $h_t \oplus h_{t'}$ that is set, we invoke a corresponding label function as discussed earlier. For instance, if the eighth bit in $h_t \oplus h_{t'}$ is 1, we invoke a label method containing an instruction that falls into category *jump* in method v.

However, we need to deal with a couple of complications. First, the added code should never be executed so it is unable to change the behavior of the original malware. This can be achieved by adding a check before the new code block whose precondition is never satisfied. Second, the original label of method v is changed because it calls extra label methods and adds the check to ensure that these label methods are never executed.

In our implementation, we use the following code snippet to address these two issues:

```
const/4 v0, 0x5
if-eqz v0, :cond_mutation
:cond_mutation
invoke-static {}, Lmut;->dup3()V
invoke-static {}, Lmut;->dup10()V
invoke-static {}, Lmut;->dup14()V
```

The code after the `if-eqz` instruction is never executed because the value stored in register $v0$ is not zero. The label category bits set for the above code block include 2 (move), 9 (branch), and 13 (invoke). If the original code of method v does not include any instructions falling into these categories, adding the above code changes the original label of method v. After the rotate operation in Eq. (1), these bits become 3, 10, and 14. Therefore, if adding the above code flips a bit in the original label of method v, we invoke a corresponding label method (i.e., `label3`, `label10`, or `label14`) to cancel out the side effect with the XOR operation.

Following the above code snippet, we can invoke the other label methods based on $h_t \oplus h_{t'}$ as discussed earlier. As the precondition for checking is never satisfied, these added label methods are not executed at run time. Note that in the above code snippet methods `dup3`, `dup10`, and `dup14` are duplicates of label methods `label3`, `label10`, and `label14`, respectively, with different names. This ensures that separate edges are added from method v to these newly inserted label methods in the FCG of the new variant.

7 Evasion of Individual Classifiers

To evaluate the effectiveness of EAGLE attacks, we use two Android APK datasets, whose statistics are summarized in Table 1. The Android malware in Dataset I are obtained from work [62], and those in Dataset II from work [58]. The details about how the malware samples were collected can be found in the original papers, respectively. Benign samples were randomly chosen from a dataset downloaded from Google Play. We train two types of classifiers: (1) *Random Forest (RF)*: We use scikit-learn's implementation with its default settings [3]; (2) *Multilayer Perceptron (MP)*: We use PyTorch [4] to build a multilayer perceptron neural network with one input layer, four hidden layers, and one output layer. The four hidden layers have 512, 256, 128, and 64 neurons, respectively. The loss function is `BCEWithLogitsLoss` and the optimizer is the Adam algorithm. Each model is trained with five epochs.

The classification performances of both RF and MP models on the two datasets are presented in Table 2. From the results, we can see that both models lead to F1 scores higher than 0.91 in all scenarios with only one exception (the F1 score of the MP model on the 2-gram opcode features for Dataset II is 0.8873). Moreover, the RF model outperforms the MP model in all cases. The classification results for Dataset I are better than those for Dataset II, probably because the malware samples in the latter (71 families) are more diverse than those in the former (47 families).

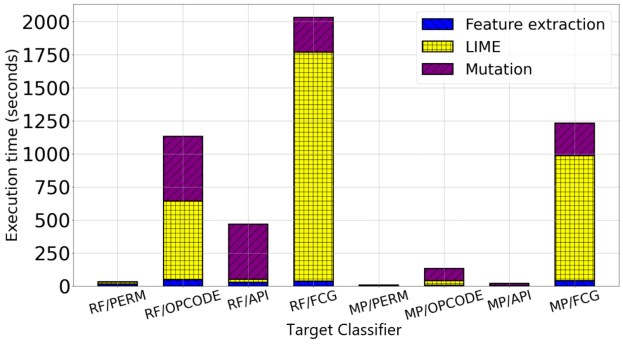


Fig. 3. Execution time breakdown in EAGLE attacks ($Q^{p,b}$: PERM, $Q^{o,c}$: OPCODE, $Q^{a,c}$: API, $Q^{g,c}$: FCG)

In all our experiments, the default settings of the parameters in a EAGLE attack (see Figure 2 and Algorithm 1) are given as follows: $n = 10$, $K_0 = 100$, $\delta = 100$, $\alpha = 2$, and $z_{max} = 16$.

7.1 Evasion effectiveness

For each malware family we randomly choose 10 samples for EAGLE attacks. If there are fewer than 10 samples in a family all of them are chosen. Hence, for Dataset I and II, 253 and 630 malware samples are chosen to perform the attacks, respectively. Table 3 shows the evasion results of EAGLE attacks against the RF and MP models for the four types of count features extracted. We observe that for both datasets, the successful evasion rates are high, ranging from 92.4% to 100%. For all classifier types except FCG, the evasion rate on Dataset I is higher than that on Dataset II, possibly because the latter contains more malware families than the former.

Recall that the mutation module is invoked iteratively to search for evasive variants (see Figure 2). Table 3 includes the mean number of iterations needed to find an evasive variant as well as its standard deviation. In most cases it takes more iterations to evade a RF classifier than its MP counterpart, regardless of the type of features used for classification. Also in most cases it takes more iterations to evade a malware classifier trained from 2-gram opcode features than those from other features, regardless of the classification model used.

7.2 Execution time

We next study how much time is spent on each of the three modules in an EAGLE attack shown in Figure 2: feature extraction, LIME, and mutation. We randomly choose a malware sample from Dataset I and the evasion attack is launched against eight classifiers trained with two classification models (RF and MP) and four feature types. Each EAGLE attack is performed five times using a commodity workstation that has an AMD Ryzen 7 2700 8-Core Processor and 16GB RAM.

Dataset	Number of samples		Malware	Number of Features			
	Benign	Malicious	Families	Permission ($Q^{p,b}$)	2-gram opcode ($Q^{o,c}$)	API Call ($Q^{a,c}$)	FCG ($Q^{g,c}$)
I	2000	1260	47	457	20192	961	32768
II	2000	1530	71	457	21773	1142	32768

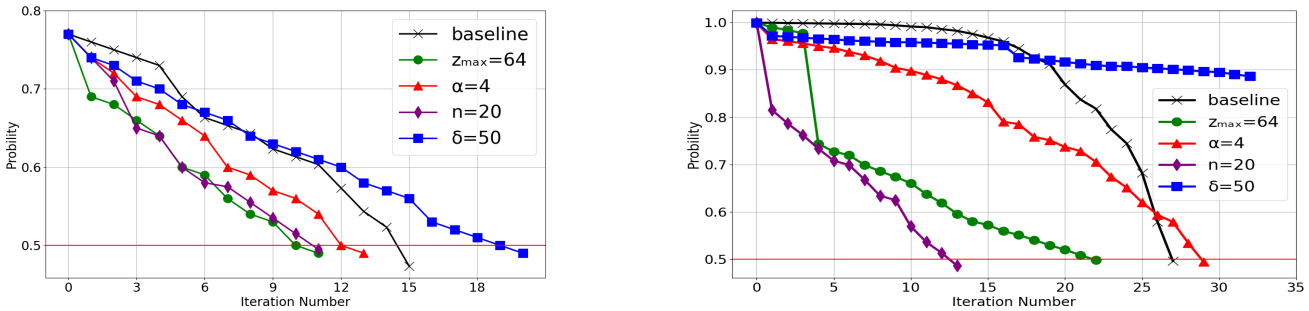
TABLE 1
Statistics of the two datasets used in our experiments

	Model	Permission ($Q^{p,b}$)			2-Gram Opcode ($Q^{o,c}$)			API Call ($Q^{a,c}$)			Function Call Graph ($Q^{g,c}$)		
		Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
I	RF	0.9773	0.9563	0.9667	0.9974	0.9546	0.9755	0.9975	0.9694	0.9832	0.9965	0.9190	0.9561
	MP	0.9085	0.9456	0.9267	0.8886	0.9493	0.9136	0.8828	0.9515	0.9129	0.9261	0.9468	0.9360
II	RF	0.9350	0.9348	0.9348	0.9257	0.9684	0.9464	0.9623	0.9700	0.9661	0.8675	0.9750	0.9166
	MP	0.9203	0.9280	0.9241	0.8385	0.9562	0.8873	0.9199	0.9475	0.9329	0.8845	0.9478	0.9113

TABLE 2
Classification performances of individual classifiers trained from both datasets

	Model	Permission ($Q^{p,b}$)		2-Gram Opcode ($Q^{o,c}$)		API Call ($Q^{a,c}$)		FCG ($Q^{g,c}$)	
		EvasionRate	#iterations	EvasionRate	#iterations	EvasionRate	#iterations	EvasionRate	#iterations
I	RF	100%	7.94/4.00	100%	16.00/6.84	100%	3.92/1.46	93.2%	10.52/7.42
	MP	100%	2.64/0.99	98.0%	4.04/3.41	96.0%	2.79/0.49	92.4%	3.71/1.63
II	RF	97.6%	10.50/3.29	94.7%	2.90/1.30	95.3%	13.07/6.13	95.3%	7.52/4.87
	MP	97.6%	2.73/1.04	95.2%	13.29/9.26	94.2%	2.93/0.42	94.7%	7.65/7.76

TABLE 3
Evasion results of individual classifiers. For "#iterations", x/y gives mean x and standard deviation y.



(1) Random Forest (2) Multilayer Perceptron

Fig. 4. Effects of algorithm parameters on search for evasive samples. Iteration 0 means the original malware sample.

The average execution time taken by each of these EAGLE attacks is depicted in Figure 3. Clearly, even using the same malware instance, the execution time of each attack varies significantly with the target classifier. The attack against the RF model trained from FCG features takes an average execution time that is two orders of magnitude longer than that taken against the MP model trained from API call features. Generally speaking, attacking the RF model takes longer time than doing it against the MP model, when the same type of malware features is used to train the model. Moreover, the order of average execution time in attacking the target classifier trained from the four types of malware features is consistent over the two classification models.

Regarding the execution time used by each component, we make the following observations. First, the time spent on feature extraction, which varies from about 2.35 seconds (MP/API) to 47.22 seconds (RF/OPCODE), is short with respect to the times spent on the other two modules in all cases. Second, in the three longest EAGLE attacks (i.e., RF/OPCODE, RF/FCG, and MP/FCG), LIME takes the majority of the execution time. Particularly in the attack against the RF/FCG classifier, 66.4% of the execution time is used by the LIME tool to find the top count features in explaining the prediction results of the current working instances.

7.3 Parameter effects

To study the effects of algorithm parameters, we randomly pick a malware sample from Dataset II to evade both RF and MP classifiers trained from 2-gram opcode features. Table 3 tells us that on average it takes a relatively large number of iterations to find an evasive variant in both cases. In each experiment, we vary only one parameter in the baseline case while keeping the others intact. Figure 4 shows how the malicious proba-

bility score of the best variant found in an EAGLE attack evolves with the iteration number.

From the results we make the following observations. *First*, increasing parameter n in the mutation algorithm from 10 to 20 helps reduce the number of iterations needed to find successful evasive variants, irrespective of the target classification model. This agrees with our intuition because testing more variants in each iteration improves the chance of finding a best one with a lower malicious probability score by the target model. *Second*, increasing parameter z_{max} from 16 to 64 also helps reduce the number of iterations needed to find a successful evasive sample. The reason is similar as in the previous case: a larger z_{max} allows each iteration of the mutation algorithm to test more variants whose features are perturbed from a larger range. *Third*, the effects of increasing parameter α can be mixed. On one hand, a larger α perturbs the selected features of the current instance within a larger range at each round of the while loop at Line 5 of Algorithm 1, suggesting that it is possible to find a better variant with a lower malicious probability score, which is returned if it is better than the current working instance x (Lines 17-18). On the other hand, a larger α means that fewer ranges are considered in each iteration of the mutation algorithm given the same z_{max}. Figure 4(1) shows that for the RF model, increasing α from 2 to 4 helps reduce the malicious probability score of the best variant found in each iteration more quickly until an evasive sample is discovered successfully, but for the MP model, doing so only helps for the first 25 iterations. *Fourth*, when an iteration fails to find a better variant with a lower malicious probability score than the current working instance, the mutation module indicates that the next iteration should increase the number of top features returned by the LIME tool by δ so that more features can be perturbed on the same working instance. For both RF and MP, reducing δ from 100 to 50 slows down the search for a variant that can bypass the detection.

7.4 Comparison of attack strategies

We next perform experiments to compare the performances of EAGLE attacks against those of two alternative approaches, *Random* and *Top-N*.

Random: In each iteration, we randomly choose 100 features. For each of these features, we randomly set their values between 0 and 100. We then mutate the current malware sample to match the targeted feature values, using the same implementation methods as discussed in Section 6 to generate practical malware variants. As in the EAGLE attacks, each iteration generates 10 such variants randomly and picks the one with the lowest malicious probability score. If the current iteration fails to generate a practical sample for a successful evasion attack, the aforementioned process repeats by continuing to mutate this best variant.

Top-N: We first rank all the features based on their importance. More specifically, for the Random Forest classifier, we rank features based on mean decrease in impurity [5] and for the MLP classifier, we rank features based on their permutation importance [6]. In each iteration, we select the top 100 features that have not been mutated yet. When a feature is chosen for mutation, its value is randomly chosen between 0 and 100. As in the EAGLE attacks, each iteration generates 10 such variants randomly and picks the one with the lowest malicious probability score. If the current iteration fails to find a successful evasion attack, the next top 100 features will be considered in the new iteration, which continues to mutate the best variant found from the previous round.

It is noted that for the Top-N mutation strategy, the attacker needs to access malware and benign application datasets for ranking the features. In our experiment we consider the best case where the attacker uses exactly the same dataset from which the classification models are trained. For both the Random mutation strategy and the EAGLE attacks, there is no such restriction.

For performance comparison, we use the same two datasets shown in Table 1. To reduce the effects of random noise, for each malware family considered, we randomly select five samples for evasion attacks (if a malware family contains fewer than five samples in the dataset, we use all of them). In total, we have used 161 original malware samples from Dataset I and 350 ones from Dataset II for evasion attacks.

For each malware sample, if an attack method considered fails to find a variant to evade the target classifier successfully within 24 hours, we terminate the experiment and treat the attack as unsuccessful. Moreover, as shown in Section 6, our implementation adds new methods to the original sample. However, the Dalvik Executable specification limits the total number of methods that can be referenced within a single DEX file to be at most 65,536 [7]. Therefore, when searching for a practical adversarial example for evasion attacks, if the total number of methods in the mutated variant exceeds this limit, we also treat it as an unsuccessful attack.

Table 4 summarizes the performance results of the three attack strategies on both datasets. We make the following observations from Table 4. *First*, for Dataset II, none of the attack methods can achieve a successful evasion rate of 100%, regardless of the target classifiers. Close examination reveals that there are 10 malware samples whose Smali code cannot be recompiled to generate practical variants. Hence, in the best case where successful evasion attacks can be carried out for all other samples, the successful evasion rate is at most 97.1% (i.e., 340 out of 350). *Second*, for some feature types their corresponding classifiers are easier to evade than others. For permission features, each attack method can always find a successful evasion attack, except for those samples in Dataset II whose Smali code cannot be recompiled. Classifiers trained on API call features are also relatively easy to evade: in all the cases the successful evasion rates are higher than 90%, except one exception where the Top-N attack method is applied on the Random Forest

	Model	Permission ($Q^{p,b}$)			2-Gram Opcode ($Q^{o,c}$)			API Call ($Q^{a,c}$)			Function Call Graph ($Q^{g,c}$)		
		Random	Top-N	EAGLE	Random	Top-N	EAGLE	Random	Top-N	EAGLE	Random	Top-N	EAGLE
I	RF	100%	100%	100%	**15.5%**	**80.1%**	100%	99.4%	98.8%	100%	**6.8%**	**5.0%**	95.0%
	MP	100%	100%	100%	100%	100%	100%	100%	100%	100%	95.0%	**19.3%**	95.0%
II	RF	97.1%	97.1%	97.1%	**13.4%**	**86.0%**	94.6%	94.6%	**52.9%**	96.2%	**8.9%**	90.6%	96.3%
	MP	97.1%	97.1%	97.1%	90.3%	92.9%	93.1%	94.9%	91.1%	97.1%	**43.4%**	**22.0%**	94.6%

TABLE 4
Successful evasion rates of EAGLE attacks and two alternative attack strategies.

classifier for Dataset II and the successful evasion rate is achieved at only 52.9%. In contrast, classifiers trained on opcode features and FCG features can be difficult to evade; in some cases the successful evasion rate can be even lower than 10.0%. *Third*, in most cases, the MP classifier is not harder to evade than the RF classifier, regardless of the attack method or the feature type. One exception is the Top-N attack method, which has a much higher successful evasion rate against the RF classifier (90.6%) than against the MP classifier (22.0%) when applied on Dataset II.

More importantly, the EAGLE attack can achieve high successful evasion rates consistently, regardless of the target classifier or the dataset used. In all cases, its successful evasion rate is always higher than 90%. In comparison, both the other two attack methods have poor performances in some cases. For easy explanation, those under-performing cases with successful evasion rates lower than 90% are highlighted in bold in Table 4. Among all 16 cases for each attack method (i.e., four feature types, two datasets, and two classification models), both the Random and the Top-N attack methods have five under-performing scenarios. In some of them, the successful evasion rate can be as low as 6.8% and 5.0% for the Random and the Top-N attack methods, respectively. These results demonstrate that the EAGLE attack is indeed a more effective method for evasion attacks than the other two.

7.5 LIME vs. SHAP

The EAGLE attacks leverage local explanation results to perturb malware samples in search for evasive variants. Although LIME is used in this work, in principle other local explanation tools can also be used for EAGLE attacks. In a new set of experiments we replace LIME with SHAP [8] in our implementation of EAGLE attacks. When calling SHAP for local explanations, we use its default permutation explainer and set the maximum number of evaluations to be $2 \times n + 1$, where n is the number of features, as recommended by the tool's manual. As the explanation result returned by SHAP does not include a range for each feature, we randomly perturb the value of a feature between 0 and 100 if it is ranked among the top K important ones by the tool.

The experiments are set up in the same manner as in Section 7.4. From the experimental results, we make the following observations. First, when the number of features is large, the execution performance of SHAP is poor, which is consistent with the observations made in [28]. Even using a machine with 132G RAM we are unable to run the SHAP tool when the target malware classifier uses 2-gram opcode or FCG features due to lack of memory. Second, when target malware classifier uses the permission or the API call features, the comparison results on successful evasion rates using SHAP and LIME are reported in Table 5. We observe that while LIME performs consistently well in all the four cases, SHAP performs poorly in two of them (the MP classifier trained on API call features for Dataset I and the RF classifier trained on permission features for Dataset II). The poor performance of SHAP may result from the following two key differences. First, unlike LIME, it does not include the range of feature values in its explanation results, which thus lacks guidance on how an EAGLE attack should perturb an important feature. Second, the SHAP explainer is trained globally through averaging the features' marginal contributions across all permutations and then applied for local explanations at individual samples. However, perturbing the most important features globally may not be the most effective way in flipping the classifier's decision on detecting the malware sample in its local neighborhood.

	Tool	Random Forest		Multilayer Perceptron	
		Permission	API Call	Permission	API Call
I	LIME	100%	100%	100%	100%
	SHAP	100%	98.1%	100%	40.1%
II	LIME	97.1%	96.2%	97.1%	97.1%
	SHAP	39.7%	97.1%	97.1%	93.7%

TABLE 5
Comparison of successful evasion rates with different local explanation tools. The results of using LIME are copied from the EAGLE column in Table 4.

8 Attack Transferability

In the previous section we have evaluated the robustness of RF or MP classifiers trained from $Q^{p,b}$, $Q^{o,c}$, $Q^{a,c}$, and $Q^{g,c}$ against EAGLE attacks. For the latter three, we can derive the following feature types with dependencies among individual features:

- For each counter feature type $O^{t,c}$, where $t \in \{o, a, g\}$, we can also construct a corresponding *frequency feature vector* $Q_x^{t,f}$ for APK instance x, where $Q_x^{t,f}[i] = O_x^{t,c}[i] / \sum_j O_x^{t,c}[j]$. Hence, the frequency feature vectors introduce dependencies among individual feature values with a common normalization factor, which is their sum.
- Given the FCG of instance x, the hash label of each of its nodes is computed according to Eq. (1). The

	Surrogate Classifier	Target classifier												
		Random Forest						Multilayer Perceptron						SVC
		$Q^{o,c}$	$Q^{o,f}$	$Q^{a,c}$	$Q^{a,f}$	$Q^{g,c}$	$Q^{g,f}$	$Q^{o,c}$	$Q^{o,f}$	$Q^{a,c}$	$Q^{a,f}$	$Q^{g,c}$	$Q^{g,f}$	$Q^{g,b}$
I	RF/$Q^{o,c}$	–	**(27/47**	5/47	1/47	32/47	45/47	**[47/47**	47/47	2/47	2/47	38/47	47/47	12/47
	RF/$Q^{a,c}$	1/47	11/47	–	**(25/47**	3/47	18/47	20/47	23/47	**[44/47**	43/47	22/47	15/47	0/47
	RF/$Q^{g,c}$	1/47	16/47	5/47	1/47	–	**(33/47**	34/47	40/47	2/47	2/47	**[33/47**	47/47	13/47
	MP/$Q^{o,c}$	**{1/47**	4/47	5/47	1/47	8/47	11/47	–	**(35/47**	2/47	2/47	30/47	29/47	2/47
	MP/$Q^{a,c}$	1/47	11/47	**{12/47**	28/47	2/47	20/47	20/47	21/47	–	**(46/47**	21/47	11/47	0/47
	MP/$Q^{g,c}$	1/47	4/47	5/47	1/47	**{6/47**	16/47	36/47	33/47	2/47	2/47	–	**(34/47**	0/47
II	RF/$Q^{o,c}$	–	**(19/71**	2/71	0/71	49/71	65/71	**[8/71**	1/71	8/71	4/71	57/71	44/71	26/71
	RF/$Q^{a,c}$	2/71	4/71	–	**(32/71**	1/71	12/71	7/71	2/71	**[61/71**	1/71	14/71	7/71	0/71
	RF/$Q^{g,c}$	2/71	4/71	1/71	0/71	–	**(25/71**	8/71	2/71	7/71	3/71	**[26/71**	36/71	13/71
	MP/$Q^{o,c}$	**{4/71**	5/71	2/71	0/71	11/71	19/71	–	**(14/71**	7/71	4/71	30/71	27/71	4/71
	MP/$Q^{a,c}$	2/71	0/71	**{2/71**	0/71	6/71	1/71	8/71	3/71	–	**(64/71**	13/71	1/71	0/71
	MP/$Q^{g,c}$	2/71	4/71	2/71	0/71	**{1/71**	9/71	8/71	2/71	7/71	4/71	–	**(53/71**	8/71

TABLE 6
Number of successful cases in transferred EAGLE attacks. The bracket symbol indicates the attack type as follows. '(': count-to-frequency attacks; '[': RF-to-MP attacks; '{': MP-to-RF attacks.

	Random Forest			Multilayer Perceptron			SVC
	$Q^{o,f}$	$Q^{a,f}$	$Q^{g,f}$	$Q^{o,f}$	$Q^{a,f}$	$Q^{g,f}$	$Q^{g,b}$
I	0.9583	0.9708	0.9532	0.9558	0.9399	0.9310	0.9481
II	0.9257	0.9556	0.8531	0.9057	0.9129	0.8907	0.9199

TABLE 7
F-1 scores of classifiers trained from dependent features

original classifier in [24] uses a binary representation of vector H as shown in Eq. (2). Let $Q_x^{g,b}$ be this *binary feature vector* used by the original approach. The unique coding scheme for vector H allows dependencies among feature values in $Q_x^{g,b}$.

Table 7 presents the F-1 scores of different classifiers trained from dependent feature types discussed above. We observe that these classification performances are comparable to those models trained from count feature vectors shown in Table 2. Compared with counter feature vectors, dependent feature vectors make it difficult to perturb the feature values within targeted ranges through *Inc* or *Dec* operations. Consider an example where the LIME result suggests that it is necessary to increase the frequencies of both API calls A and B above θ_A and θ_B, respectively. Suppose that we first add more API calls of A to make its frequency higher than θ_A. If we next add more API calls of B to make its frequency higher than θ_B, these added calls increase the total number of API calls and may bring down the frequency of API call A below θ_A.

We next study the transferability of attacks across different malware classifiers. Although it is difficult to attack classifiers trained from dependent features, we can use one trained from counter feature vectors as a surrogate model to generate evasive samples, which are further used to attack another classifier. Due to the large number of experiments, for both datasets, we randomly choose one malware sample from each family to perform EAGLE attacks. Table 6 shows the transferability of evasion attacks across different malware classifiers.

Count-to-frequency attacks: These attacks use a surrogate model trained from count feature vectors to find evasive samples, which are further used to attack the classifier trained from the corresponding frequency feature vectors. We observe that the transferability of count-to-frequency attacks is modest with an average successful rate of 58.6% over all cases. These results show that although classifiers trained from frequency feature vectors are hard to attack directly by our proposed scheme, it is possible to evade them indirectly with a modest successful rate through count-to-frequency attacks.

Cross-model attacks. We consider malware classifiers trained from counter feature vectors with two different models, RF and MP. In *RF-to-MP attacks*, the evasion attack is performed against a surrogate RF model to find evasive samples, which are further used to attack the corresponding MP model. The evasion results of RF-to-MP attacks shown in Table 6 have an average successful rate of 66.3% over all cases. On the other side, in *MP-to-RF attacks*, the evasion attack uses a surrogate MP model to find evasive samples and then use these samples to attack the corresponding RF model. The average successful rate of MP-to-RF attacks, which is miserably 8.4%, is much lower than that of RF-to-MP attacks. Our results demonstrate that RF is superior to MP for Android malware classification: RF has a slight classification performance advantage over MP (Table 2), it takes longer time and more iterations to attack an RF classifier than its MP counterpart (Table 3 and Figure 3), and RF is more robust than MP against cross-model attacks (Table 6).

Cross-feature-type attacks. We consider a target malware classifier trained from feature type $Q^{t,c}$ where $t \in \{o, a, g\}$. The evasive samples are obtained from attacking a surrogate model trained from feature type $Q^{t',c}$ where $t' \neq t$. When the target classifier uses the RF model, such cross-feature-type attacks mostly have low successful rates except the case with $t = g$, where evasive samples found to attack a surrogate RF model trained from $Q^{o,c}$ can evade the target RF model trained from $Q^{g,c}$ with a relatively high successful rate (68.1% for Dataset I and 69.0% for Dataset II).

When the target classifier uses the MP model, such cross-feature-type attacks can have high successful rates for both $t = o$ and $t = g$. For example, a target MP classifier trained from $Q^{o,c}$ can be indirectly attacked

	Surrogate Classifier	Target Classifier: Random Forest		Target Classifier: Multilayer Perceptron	
		$Q^{p,b}$, $Q^{o,c}$, $Q^{a,c}$, $Q^{g,c}$	$Q^{p,b}$, $Q^{o,f}$, $Q^{a,f}$, $Q^{g,f}$	$Q^{p,b}$, $Q^{o,c}$, $Q^{a,c}$, $Q^{g,c}$	$Q^{p,b}$, $Q^{o,f}$, $Q^{a,f}$, $Q^{g,f}$
I	RF/$Q^{p,b}$-$Q^{o,c}$-$Q^{a,c}$-$Q^{g,c}$	**37/41/47**	**(20/41/47**	**[35/41/47**	37/41/47
	RF/$Q^{g,c}$-$Q^{a,c}$-$Q^{o,c}$-$Q^{p,b}$	**44/44/47**	**(21/44/47**	**[41/44/47**	40/44/47
	MP/$Q^{p,b}$-$Q^{o,c}$-$Q^{a,c}$-$Q^{g,c}$	**{0/47/47**	8/47/47	**47/47/47**	**(38/47/47**
	MP/$Q^{g,c}$-$Q^{a,c}$-$Q^{o,c}$-$Q^{p,b}$	**{0/47/47**	10/47/47	**47/47/47**	**(36/47/47**
II	RF/$Q^{p,b}$-$Q^{o,c}$-$Q^{a,c}$-$Q^{g,c}$	**62/67/71**	**(8/67/71**	**[5/67/71**	0/67/71
	RF/$Q^{g,c}$-$Q^{a,c}$-$Q^{o,c}$-$Q^{p,b}$	**65/67/71**	**(10/67/71**	**[6/67/71**	0/67/71
	MP/$Q^{p,b}$-$Q^{o,c}$-$Q^{a,c}$-$Q^{g,c}$	**{0/66/71**	0/66/71	**50/66/71**	**(3/66/71**
	MP/$Q^{g,c}$-$Q^{a,c}$-$Q^{o,c}$-$Q^{p,b}$	**{0/64/71**	0/64/71	**53/64/71**	**(3/64/71**

TABLE 8

Evasion attacks against ensemble classifiers. $x/y/z$ in each entry means: $x =$ number of successful evasion attacks against the target ensemble classifier; $y =$ number of successful sequential attacks against the surrogate ensemble classifier; $z =$ number of original malware instances. The bracket symbol indicates the attack type as follows. None: direct attacks; '(': count-to-frequency attacks; '[': RF-to-MP attacks; '{': MP-to-RF attacks.

with evasive samples found to attack a surrogate MP model trained from $Q^{g,c}$ with a successful rate of 76.60% for Dataset I; a target MP classifier trained from $Q^{g,c}$ can be indirectly attacked by evasive samples found to attack a surrogate MP model trained from $Q^{o,c}$ with a successful rate of 63.8% for Dataset I and with a successful rate of 42.3% for Dataset II.

These results suggest that it is difficult to evade API call classifiers through cross-feature-type attacks, which agrees well with our intuition. This also holds for permission classifiers because permission features are extracted separately from manifest files. In contrast, 2-gram opcode classifiers and FCG classifiers both rely on instruction-level features, which may lead to strong correlations among their prediction results.

Table 6 reveals that the original classifier [24], which is a linear SVC classifier trained on binary feature vector $Q^{g,b}$, is relatively robust against attack samples transferred from other models. In best cases, the successful rate is 27.7% when a surrogate RF model trained from $Q^{g,c}$ is used for Dataset I and 36.6% when a surrogate RF model trained from $Q^{o,c}$ is used for Dataset II.

9 Evasion of Ensemble Classifier

In this section we study the robustness of ensemble classifiers against EAGLE attacks. We consider two sets of feature vectors, $\mathcal{Q}_{pc} = \{Q^{p,b}, Q^{o,c}, Q^{a,c}, Q^{g,c}\}$ (permissions and count features) and $\mathcal{Q}_{pf} = \{Q^{p,b}, Q^{o,f}, Q^{a,f}, Q^{g,f}\}$ (permissions and frequency features). For each $\mathcal{Q}$ where $Q \in \{\mathcal{Q}_{pc}, \mathcal{Q}_{pc}\}$, the ensemble classifier is trained with model f as follows: for each feature type in $\mathcal{Q}$, we train an individual classifier of model f; given APK instance x, if and only if any of these individual classifiers detects it to be malicious, the ensemble classifier reports it as malicious. Hence, in order to fly under the radar against such an ensemble classifier, the adversarial sample has to evade the detection of every individual classifier involved.

To study the robustness of ensemble classifiers as constructed, we consider evasion attacks performed in a sequential manner against the individual classifiers trained from $\mathcal{Q}_{pc}$ as follows. We use EAGLE attacks to mutate the original sample x_0 to find an adversarial sample x_1 that can evade the first classifier in the sequence, mutate x_1 to find an adversarial sample x_2 that can evade the second classifier in the sequence, and so on. In our experiments, we consider the following two sequences: $Q^{p,b}$-$Q^{o,c}$-$Q^{a,c}$-$Q^{g,c}$ and $Q^{g,c}$-$Q^{a,c}$-$Q^{o,c}$-$Q^{p,b}$.

For both datasets, we randomly choose a malware sample from each family to perform EAGLE attacks against ensemble classifiers. Table 8 summarizes our results of evasion attacks.

Direct attacks. In a direct attack the surrogate and target classifiers are trained with the same model and the same set of features. The results of direct attacks are shown in green in Table 8. Clearly, the successful rates of direct attacks are high in all cases, varying from 78.7% to 100%. It is also noted that using the attack sequence $Q^{g,c}$-$Q^{a,c}$-$Q^{o,c}$-$Q^{p,b}$ leads to a slightly higher successful rate than using its reverse sequence.

Count-to-frequency attacks. In a count-to-frequency attack the surrogate classifier is trained from permission/count features while the target classifier uses the same model but is trained from permission/frequency features. The results of count-to-frequency attacks are shown in blue in Table 8. We observe that the successful rates with Dataset I, varying from 42.6% to 80.9%, are much higher than those with Dataset II, which vary from 4.2% to 14.1%. These results are consistent with what we have observed from Table 5: the successful rates of for count-to-frequency attacks are higher with Dataset I than those with Dataset II. When the attack transferability over individual classifiers is poor (e.g., 19/71 and 14/71 for the RF and MP classifier, respectively, with Dataset II), their accumulative effects further reduce the successful rate of evading the ensemble classifier.

Cross-model attacks. In a cross-model attack, the surrogate and target classifiers are trained from the same set of features but with different models. In RF-to-MP attacks, the surrogate classifier uses RF while the target classifiers uses MP. The attack effects of RF-to-MP attacks are shown in purple in Table 8. Clearly, Dataset I has a much higher successful rate of RF-to-MP attacks than Dataset II. This is unsurprising because from Table 5, we observe that Dataset II has a poor successful rate of RF-to-MP attacks for $O^{o,c}$ (11.3%) and $O^{g,c}$ (36.6%), while

	Random Forest				Multilayer Perceptron			
	$Q^{p,b}$	$Q^{o,c}$	$Q^{a,c}$	$Q^{g,c}$	$Q^{p,b}$	$Q^{o,c}$	$Q^{a,c}$	$Q^{g,c}$
I	1/47 (2.1%)	43/47 (91.5%)	6/47 (12.8%)	45/47 (95.7%)	19/47 (40.4%)	24/47 (51.1%)	12/47 (25.5%)	43/47 (91.5%)
II	5/71 (7.0%)	18/71 (25.4)	0/71 (0.0%)	34/71 (47.9%)	4/71 (5.6%)	53/71 (74.6%)	5/71 (7.0%)	53/71 (74.6%)

TABLE 9
Successful evasions against classifiers enhanced with adversarial training

Dataset I has a high successful rate of RF-to-MP attacks in all cases, varying from 72.2% to 100%.

The results of MP-to-RF attacks, where the surrogate classifiers use MP but the target classifiers use RF, are shown in orange in Table 8. Our observation that none of these MP-to-RF attacks have been successful is consistent with poor transferability of MP-to-RF attacks with individual classifiers as seen in Table 6.

10 Defense Implications

EAGLE attacks focus on count features that can be extracted from Android applications and use customized *Inc*/*Dec* operations to modify original malware samples in evasion attacks. Hence, one defense strategy against EAGLE attacks is to use non-count features, or count features whose values are hard to manipulate through *Inc*/*Dec* operations. For instance, our experimental results in Section 8 show that simply using frequency features can reduce the effectiveness of EAGLE attacks significantly as the count-to-frequency attack transferability is only modest at an average successful evasion rate of 58.6% in all cases. Moreover, the original FCG classifier proposed in [24] does not use count features due to its coding scheme for histogram H in Eq. (2). The results shown in the last column of Table 6 suggests that the performance of evading this classifier using EAGLE attacks is poor with an average successful rate of only 12.3% among all cases. In addition to using dependent feature values, another way of avoiding straightforward count features is to increase uncertainty in extracted feature values through randomization [13].

As shown in Section 6, EAGLE attacks rely upon various obfuscation techniques to generate practical adversarial examples. Previous work [55] has revealed that obfuscations can leave ample traces useful for accurate detection and family identification of Android malware. It is thus possible to detect EAGLE attacks through their obfuscation artifacts. For instance, when applying EAGLE attacks against permission-based Android malware classifiers, they may lead to an issue of bloated permissions due to addition of irrelevant permissions. Similarly, as discussed in Section 6.4, the implementation of EAGLE attacks against the FCG classifier introduces extra labelling methods, which may be used as signatures to detect the resulting adversarial examples.

Aside from enhanced features for defenses against EAGLE attacks, we can also improve the robustness of the machine learning models used in Android malware classification. Our results presented in Sections 7 and 8 have shown that different classifiers do not behave equally well when faced with EAGLE attacks: the RF classifier requires more time and iterations for a successful EAGLE attack than the MP classifier *and* it is also more robust against cross-model attacks than the latter.

Many defense methods against adversarial machine learning attacks have been proposed previously. Due to limited space we consider only adversarial training [56], [26] as it can be applied directly on both the RF and MP classifiers. In a new set of experiments, for each type of malware classifiers trained, we add all the evasive samples discovered by the EAGLE attacks against it from the experiments done in Section 7.4 (i.e., Table 4) to the corresponding training dataset and then retrain the classification model. For each of these retrained classifiers, we randomly choose a malware sample from each family and perform the EAGLE attack against it. Each experiment lasts at most 24 hours. Therefore if the EAGLE attack cannot find an adversarial example within 24 hours, it is deemed as unsuccessful.

Table 9 depicts the effectiveness of adversarial training in defending against EAGLE attacks. Clearly, for both datasets, the adversarial training defense mechanism is more effective against the permission and API call classifiers than it against the 2-gram opcode and FCG classifiers, regardless of the classification model used. In contrast, the choices of classification model (RF or MP) and dataset used for evaluation (I or II) have mixed effects on the effectiveness of adversarial training in defending against EAGLE attacks. From a defense perspective, the results shown in Table 9 confirm the importance of an ensemble approach to malware detection: although permission and API call features are easy to obfuscate, the classifiers trained from such features can still improve the overall robustness of the ensemble classifier when hardened with adversarial training.

11 Conclusions and Future Work

In this work we explore how EAGLE attacks leverage local explanations to generate adversarial examples capable of evading Android malware classifiers trained on count features. Using two Android malware datasets, we extract four types of count features and train multiple Android malware classifiers based on RF and MP models. Our results show that EAGLE attacks can be highly effective at finding adversarial examples to evade these classifiers. We also find that some of these evasive variants can be transferred to attack other malware classifiers trained with different features or models successfully. We further demonstrate that ensemble classifiers may still be vulnerable to EAGLE attacks. We finally discuss the defense implications against EAGLE attacks.

In the future we plan to address the limitations of this work as follows. First, this study has considered only four types of count features extracted from Android applications. The rich information included with Android applications suggest that a variety of other count features, such as API sequences representing ComplexFlows [51], can also be used for malware detection. We will continue to study the effectiveness of EAGLE attacks in evading these other malware classifiers. Second, due to constant evolution of Android malware attacks, predictive power of old malware classifiers may deteriorate over time, leading to a so-called concept drift phenomenon [52]. This work has evaluated the effectiveness of EAGLE attacks with only two Android malware datasets [62], [58]. We plan to investigate the performance of such attacks with new Android malware samples collected in the wild. Third, we have discussed some defense strategies against EAGLE attacks and shown the effectiveness of adversarial training. We plan to conduct a systematic study on the performances of other possible defense strategies in our future work. Fourth, this work considers only EAGLE attacks against malware classifiers trained from arbitrary count features. For malware classifiers trained on feature embeddings (e.g., node embeddings in Graph Neural Network models), although local explanation methods can also be used to explain the importance of feature embeddings in individual malware classifications, it is not straightforward to map the suggested changes in feature embeddings by local explanations to the practical modifications of the original malware sample in evasion attacks. We plan to explore this issue in depth in our future work. Fifth, this work assumes that the attacker is able to evade any anomaly detection method aimed to catch excessive soft label queries generated by the local explanation tool in an EAGLE attack. In our future work, we plan to investigate how to make EAGLE attacks effective while limiting the number of soft label queries requested by each attack machine. Potential solutions include reuse of soft label query results across different local explanations, distribution of soft label queries to multiple attack machines, and training of local surrogate models to replace the targeted malware classifier. Last but not least, this study has adopted LIME for EAGLE attacks. As explained in Section 3.2, LIME needs to sample the neighborhood of the data point to create a perturbed dataset, from which a sparse linear model is trained for local explanation. Such sampling-based local explanations can incur significant overhead, as shown in Figure 3. In principle other local explanation tools can also be used in EAGLE attacks. For example, using LEMNA [28] instead of LIME may provide more accurate guidance in sample mutation as it groups adjacent features and trains a non-linear mixture regression model. However, its local approximation model is more complex and thus can take even longer time to train. In the future we plan to study other local explanation methods to further improve the efficiency of EAGLE attacks.

Acknowledgments

We thank the editors and the anonymous reviewers for their constructive feedback. This work is partially supported by the US National Science Foundation under awards CNS-1943079 and CNS-1618631.

References

[1] *https://ibotpeaches.github.io/Apktool/*.
[2] *https://source.android.com/devices/tech/dalvik/dalvik-bytecode*.
[3] *https://scikit-learn.org/*.
[4] *https://pytorch.org*.
[5] *https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html*.
[6] *https://eli5.readthedocs.io/en/latest/blackbox/permutation_importance.html*.
[7] *https://developer.android.com/studio/build/multidex*.
[8] *https://shap.readthedocs.io/en/latest/index.html*.
[9] The Drebin Dataset. *https://purplesec.us/resources/cyber-security-statistics/*.
[10] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer, 2013.
[11] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen. Adversarial learning attacks on graph-based iot malware detection systems. In *International Conference on Distributed Computing Systems (ICDCS'19)*. IEEE, 2019.
[12] A. Al-Dujaili, A. Huang, E. Hemberg, and U.-M. O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *IEEE Security and Privacy Workshops (SPW'18)*, 2018.
[13] H. Alasmary, A. Abusnaina, R. Jang, M. Abuhamad, A. Anwar, D. Nyang, and D. Mohaisen. Soteria: Detecting adversarial examples in control flow graph-based malware classifiers. In *International Conference on Distributed Computing Systems*, 2020.
[14] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth. Learning to evade static PE machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, 2018.
[15] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX*, 11:100403, 2020.
[16] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 387–402. Springer, 2013.
[17] B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84, 2018.
[18] T. Chakraborty, F. Pierazzi, and V. Subrahmanian. Ec2: Ensemble clustering and classification for predicting android malware families. *IEEE Transactions on Dependable and Secure Computing*, 17(2):262–277, 2017.
[19] M. Cheng, T. Le, P.-Y. Chen, J. Yi, H. Zhang, and C.-J. Hsieh. Query-efficient hard-label black-box attack: An optimization-based approach. *arXiv preprint arXiv:1807.04457*, 2018.
[20] A. Daniel, S. Michael, G. Hugo, and R. Konrad. Drebin: Efficient and explainable detection of Android malware in your pocket". In *Network and Distributed System Security Symposium*, 2014.
[21] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on Android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 16(4), 2017.
[22] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang. Understanding Android obfuscation techniques: A large-scale investigation in the wild. In *International Conference on Security and Privacy in Communication Systems*. Springer, 2018.
[23] M. Ficco. Malware analysis by combining multiple detectors and observation windows. *IEEE Transactions on Computers*, 71(6), 2022.
[24] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In *ACM Workshop on Artificial Intelligence and Security*, 2013.
[25] Github. *Source code for paper "Structural Analysis and Detection of Android Malware"*. https://github.com/hgascon/adagio.
[26] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[27] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*. Springer, 2017.

[28] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing. LEMNA: Explaining deep learning based security applications. In *ACM Conference on Computer and Communications Security*, 2018.

[29] M. Hassen and P. K. Chan. Scalable function call graph-based malware classification. In *ACM Conference on Data and Application Security and Privacy*, 2017.

[30] J. D. Herath, P. P. Wakodikar, P. Yang, and G. Yan. CFGExplainer: Explaining Graph Neural Network-based malware classification from control flow graphs. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2022.

[31] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu. Performance evaluation on permission-based detection for Android malware. In *Advances in Intelligent Systems and Applications: Volume II*. Springer, 2013.

[32] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, 2011.

[33] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan. Pindroid: A novel Android malware detection system using ensemble learning methods. *Computers & Security*, 68:36–46, 2017.

[34] B. Kang, S. Y. Yerima, K. McLaughlin, and S. Sezer. N-opcode analysis for android malware classification and categorization. In *International Conference on Cyber Security and Protection of Digital Services*, pages 1–7. IEEE, 2016.

[35] R. Killam, P. Cook, and N. Stakhanova. Android malware classification through analysis of string literals. In *Workshop Programme*, page 27, 2016.

[36] Y. Kucuk and G. Yan. Deceiving portable executable malware classifiers into targeted misclassification with practical adversarial examples. In *ACM Conference on Data and Application Security and Privacy*, 2020.

[37] P. Liu, W. Wang, X. Luo, H. Wang, and C. Liu. Nsdroid: efficient multi-classification of Android malware using neighborhood signature in local function call graphs. *International Journal of Information Security*, 20(1), 2021.

[38] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre. Malware makeover: breaking ml-based static analysis by modifying executable bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 744–758, 2021.

[39] A. Machiry, N. Redini, E. Gustafson, Y. Fratantonio, Y. R. Choe, C. Kruegel, and G. Vigna. Using loops for malware classification resilient to feature-unaware perturbations. In *Annual Computer Security Applications Conference*, 2018.

[40] P. McDaniel, N. Papernot, and Z. B. Celik. Machine learning in adversarial settings. *IEEE Security & Privacy*, 14(3):68–72, 2016.

[41] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. J. Ahn. Deep android malware detection. In *ACM Conference on Data and Application Security and Privacy*, 2017.

[42] M. Melis, D. Maiorca, B. Biggio, G. Giacinto, and F. Roli. Explaining black-box Android malware detection. In *European Signal Processing Conference*. IEEE, 2018.

[43] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *European Symposium on Security and Privacy*. IEEE, 2016.

[44] N. Peiravian and X. Zhu. Machine learning for Android malware detection using permission and API calls. In *IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 2013.

[45] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *IEEE Symposium on Security and Privacy*. IEEE, 2020.

[46] PurpleSec. 2021 Cyber Security Statistics. *https://purplesec.us/resources/cyber-security-statistics/*, 2021.

[47] M. T. Ribeiro, S. Singh, and C. Guestrin. "why should I trust you?" Explaining the predictions of any classifier. In *ACM Conference on Knowledge Discovery and Data Mining (KDD'16)*, 2016.

[48] I. Rosenberg, S. Meir, J. Berrebi, I. Gordon, G. Sicard, and E. O. David. Generating end-to-end adversarial examples for malware classifiers using explainability. In *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020.

[49] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze. Malware detection based on mining api calls. In *ACM Symposium on Applied Computing*, 2010.

[50] G. Severi, J. Meyer, S. Coull, and A. Oprea. Explanation-guided backdoor poisoning attacks against malware classifiers. In *USENIX Security Symposium*, 2021.

[51] F. Shen, J. Del Vecchio, A. Mohaisen, S. Y. Ko, and L. Ziarek. Android malware detection using complexflows. *IEEE Transactions on Mobile Computing*, 18(6):1231–1245, 2018.

[52] A. Singh, A. Walenstein, and A. Lakhotia. Tracking concept drift in malware families. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, pages 81–92, 2012.

[53] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin. Mab-malware: A reinforcement learning framework for attacking static malware classifiers. *ACM Asia Conference on Computer and Communications Security*, 2022.

[54] N. Šrndić and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *IEEE Symposium on Security and Privacy*, pages 197–211. IEEE, 2014.

[55] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *ACM Conference on Data and Application Security and Privacy*, 2017.

[56] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[57] L. Taheri, A. F. A. Kadir, and A. H. Lashkari. Extensible Android malware detection and family classification using network-flows and api-calls. In *International Carnahan Conference on Security Technology*, pages 1–8. IEEE, 2019.

[58] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou. Deep ground truth analysis of current Android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276. Springer, 2017.

[59] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and API calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69. IEEE, 2012.

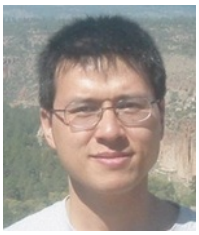
[60] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *Network and Distributed Systems Symposium*, 2016.

[61] G. Yan, N. Brown, and D. Kong. Exploring discriminatory features for automated malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer, 2013.

[62] W. Yang, D. Kong, T. Xie, and C. A. Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in Android apps. In *Annual Computer Security Applications Conference*, pages 288–302, 2017.

[63] H. Zhang, X. Xiao, F. Mercaldo, S. Ni, F. Martinelli, and A. K. Sangaiah. Classification of ransomware families with machine learning based on n-gram of opcodes. *Future Generation Computer Systems*, 90:211–221, 2019.

Zhan Shu received the BE degree in Network Engineering from Jinan University, Shandong Province, in China in 2006 and the MS degree in Computer Science from Binghamton University, State University of New York, in 2016. He earned the PhD degree in Computer Science from Binghamton University in 2022. His research focuses on proactive cybersecurity for large-scale distributed and networked systems.



Guanhua Yan received the PhD degree in computer science from Dartmouth College, Hanover, New Hampshire, in 2005. After working at Los Alamos National Laboratory in New Mexico for nine years, he joined Binghamton University, State University of New York as a faculty member in 2014. His research interests span cybersecurity, networking, and large-scale modeling and simulation. He has made contributions to about 80 articles in these fields.