Abstract—Strengthening the robustness of machine learning-based Android malware detectors in the real world requires incorporating realizable adversarial examples (RealAEs), i.e., AEs that satisfy the domain constraints of Android malware. However, existing work focuses on generating RealAEs in the problem space, which is known to be time-consuming and impractical for adversarial training. In this paper, we propose to generate RealAEs in the feature space, leading to a simpler and more efficient solution. Our approach is driven by a novel interpretation of Android malware properties in the feature space. More concretely, we extract feature-space domain constraints by learning meaningful feature dependencies from data and applying them by constructing a robust feature space. Our experiments on DREBIN, a well-known Android malware detector, demonstrate that our approach outperforms the state-of-the-art defense, Sec-SVM, against realistic gradient- and query-based attacks. Additionally, we demonstrate that generating feature-space RealAEs is faster than generating problem-space RealAEs, indicating its high applicability in adversarial training. We further validate the ability of our learned feature-space domain constraints in representing the Android malware properties by showing that (i) re-training detectors with our feature-space RealAEs largely improves model performance on similar problem-space RealAEs and (ii) using our feature-space domain constraints can help distinguish RealAEs from unrealizable AEs (unRealAEs).

1. Introduction

Due to the on-going proliferation of Android malware, machine learning (ML)-based malware detection solutions continue to remain a well-studied topic by security researchers. However, despite their superior performance [1]–[9], ML-based solutions are vulnerable to adversarial examples [10], i.e., malware instances that are crafted to carry out evasion attacks. In such attacks, adversaries modify malware applications (apps) such that the resulting manipulated samples can successfully mislead malware classifiers.

Over the last years, various defense strategies (e.g., adversarial training [10] and defensive distillation [11]) have been presented to strengthen ML models against adversarial examples. However, in constrained domains (e.g., malware, botnet, and credit risk detection), the defenders must improve the adversarial robustness by mitigating the vulnerability of detection systems against realistic evasion attacks. This is because evasion attacks in constrained domains must meet domain constraints [12], [13] (e.g., malicious functionality in the malware context) to generate adversarial examples. In the Android domain, which is a constrained domain [13], a key requirement of the adversarial examples is that they must be realizable, i.e., satisfying domain constraints in order to meet all Android malware properties (e.g., malware functionality) [14]. Accordingly, strengthening the robustness of Android malware detectors should be also based on realizable adversarial examples (RealAEs) [15]. Therefore, access to RealAEs is essential to properly evaluate ML-based malware detectors because they operate on the feature space of a constrained domain [12].

Most prior defenses are based on unrealizable adversarial examples (unRealAEs) [16]–[25], thus leading to an unfair assessment of model robustness. Such examples are commonly generated in the feature space by arbitrarily changing certain features determined based on the sensitivity of ML models against adversarial perturbations [16], [19], [21]–[24]. These examples ignore Android malware properties, which are primarily feature dependencies stemming from real-world objects. Moreover, using unRealAEs have been shown to be not useful for mitigating the vulnerability of detectors against realistic evasion attacks [26], [27]. For instance, an extensive analysis carried out by Berger et al. [27] shows that retraining ML-based Android malware detectors with unRealAEs (generated in the feature space) is ineffective against realistic evasion attacks, which are performed in the problem space.

Although defenses that have been proposed recently have relied on RealAEs, they are, however, limited to generating them in the problem space [14], [15], [28]–[31]. We argue that problem-space RealAEs are not good candidates for improving the adversarial robustness of ML models mainly for two reasons. First, manipulating problem-space malware through available realizable transformations is more costly in terms of time and space than directly perturbing feature representations of the malware [12]. Second, finding
effective transformations that meet domain constraints in the problem space is not trivial [20], [28], [32].

1.1. Research Challenges

In this paper, we address the aforementioned limitations by investigating a simpler and more efficient defense based on feature-space RealAEs. Our defense is enabled by our novel interpretation of the Android malware domain constraints in the feature space. Specifically, our defense is based on learning domain constraints from data by exploiting meaningful feature dependencies. We mainly address the following three challenges:

- **Interpreting domain constraints.** Exploiting all feature dependencies is not only time-consuming and difficult, but also unnecessary [33]. Thus, we need to identify the meaningful feature dependencies, which can sufficiently guarantee the domain constraints.

- **Extracting domain constraints.** Due to the high dimensionality of the feature space and a large number of data, an appropriate learning method should be adopted to extract the meaningful feature dependencies.

- **Applying domain constraints.** After obtaining the meaningful feature dependencies, we then need to develop an approach to apply them in order to eventually improve adversarial robustness.

1.2. Contributions

This study investigates the effects of Android malware properties on adversarial robustness by not only interpreting domain constraints [14] in the feature space, but also learning and applying them in the problem space.

Our contributions\(^1\) can be summarized as follows:

- **We propose a new feature-space interpretation of Android malware domain constraints.** This new interpretation considers key characteristics of domain constraints to achieve realizable adversarial examples (RealAEs).

- **We learn our feature-space domain constraints by exploiting meaningful feature dependencies of existing data, based on their statistical correlations and a graph-based clustering algorithm, known as Optimum-Path Forest (OPF) [34].** We then apply these learned domain constraints to constructing a new robust feature space, which leads to our final defense approach.

- **We empirically evaluate our approach to show that our new defense outperforms the state-of-the-art approach, Sec-SVM [16], in defeating both gradient-based and query-based realistic attacks in the limited-knowledge (transfer) and perfect-knowledge settings.** We also show that generating feature-space RealAEs is faster than generating problem-space ones.

- **Finally, we demonstrate that our learned domain constraints can successfully fulfill the properties of Android malware.** First, re-training malware detectors by incorporating our feature-space RealAEs largely improves model accuracy on similar, problem-space RealAEs. Second, our feature-space domain constraints are shown to be useful in distinguishing RealAEs from unRealAEs.

2. Related Work

Despite the significant advances in adversarial machine learning, this research area is faced with a multitude of challenges in constrained domains, such as malware detection. In constrained domains, adversarial examples that do not meet the domain constraints are not realizable. Over the last years, different studies have been conducted to achieve adversarial robustness against RealAEs. Such realizable adversarial examples (RealAEs) can be generated by considering the domain constraints either in the problem space or feature space. Table 1 presents an overview of all the related work in the Android malware context.

The majority of studies [17], [18], [20]–[24], [35], [36] have focused on the feature-space adversarial examples. Croce et al. [35] proposed a query-based evasion attack based on a random search, which was evaluated in different contexts, especially in Android malware detection. Xu et al. [24] developed a semi-black-box framework based on the Simulated Annealing method to perturb features of Android apps by querying the target malware detector. Li et al. [17], [18] generated adversarial examples in the feature space based on different gradient-based and gradient-free evasion attacks. Rathore et al. [19] proposed two Android attack techniques based on Reinforcement Learning to generate feature-space adversarial examples. Liu et al. [23] used Genetic Algorithm (GA) to present a framework that can create feature-space adversarial examples for improving the robustness of Android malware detection. Chen et al. [21], [22] considered different feature-space evasion attacks (e.g., anonymous attacks and well-crafted attacks) to bypass Android malware detection. Demontis et al. [16] proposed a feature-space evasion attack to make Android adversarial examples by changing the features that seem important for the SVM classifier. Grosse et al. [20] generated adversarial examples based on a forward derivative approach by modifying the features extracted from Android Manifest files of Android malware apps.

Despite their effective evasion attacks, none of the above studies can achieve RealAEs, which satisfy the malware domain constraints (i.e., preserved-semantic, plausibility and robustness-to-preprocessing constraints [14]). Specifically, the adversarial examples discussed in [16]–[18], [20]–[24], [35] do not satisfy the robustness-to-preprocessing constraint because the proposed attacks considered adding features to manifest files in order to generate adversarial exam-

---

\(^1\) We make our code publicly available at https://anonymous.4open.science/r/robust-Android-malware-detector-C5D8 to allow reproducibility.
TABLE 1. Overview of related work on domain constraints. Our study is the first to explore feature-space domain constraints.

| Study                  | Year | Problem-Space Domain Constraints | Feature-Space Domain Constraints |
|-----------------------|------|----------------------------------|----------------------------------|
| Croce et al. [35]     | 2022 | N/A                              | x                                |
| Labaca-Castro et al. [15] | 2021 | ✓                                | x                                |
| Bostani and Moonsamy [29] | 2021 | ✓                                | ✓                                |
| Xu et al. [24]        | 2021 | N/A                              | x                                |
| Li et al. [18]        | 2021 | ✓                                | N/A                              |
| Rathore et al. [19]   | 2021 | ✓                                | ✓                                |
| Cara et al. [31]      | 2020 | ✓                                | N/A                              |
| Pierazzi et al. [14]  | 2020 | ✓                                | N/A                              |
| Li et al. [17]        | 2020 | N/A                              | ✓                                |
| Chen et al. [30]      | 2019 | ✓                                | N/A                              |
| Liu et al. [23]       | 2019 | N/A                              | x                                |
| Chen et al. [22]      | 2018 | N/A                              | x                                |
| Chen et al. [21]      | 2017 | N/A                              | ✓                                |
| Yang et al. [28]      | 2017 | ✓                                | N/A                              |
| Grosse et al. [20]    | 2017 | N/A                              | x                                |
| Demontis et al. [16]  | 2017 | ✓                                | x                                |
| Our work              | 2022 | N/A                              | ✓                                |

Domain Constraints

3. Interpreting Feature-Space Domain Constraints

In the problem space, the domain constraints of Android malware apps are defined as: (a) available transformations, (b) preserved semantics, (c) robustness to preprocessing, and (d) plausibility [14]; however, we aim to interpret these constraints into a set of new constraints over the feature values in the feature space. This section introduces our novel feature-space interpretation of domain constraints for Android malware apps. Before going into the detailed definitions (Section 3.2), we first provide a mathematical background of RealAEs in both the problem and feature spaces (Section 3.1).

3.1. Problem-Space and Feature-Space RealAEs

Problem-Space RealAEs. Suppose $\psi : Z \to \mathcal{X}$ is a mapping function that transforms each Android app in the problem space $Z$ into a $d$–dimensional feature vector in the feature space $\mathcal{X}$. A malware detector is an ML-based binary classifier $f : \mathcal{X} \to \mathcal{Y}$ with a discriminant function $g : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ where $f(x) = \arg \max_{y \in \mathcal{Y}} g_i(x)$ determines the label of $x \in \mathcal{X}$. Specifically, $\mathcal{Y} = \{0, 1\}$ is the label set with $y = 0$ indicating benign labels and $y = 1$ indicating...
malicious labels. Each element in the feature vector $x \in \mathcal{X}$ is usually binary [9], [45], [46], where 0 indicates the absence and 1, the presence of a specific feature.

Generally, adversarial examples can be generated by modifying $z \in \mathcal{Z}$ through problem-space transformations or modifying $x \in \mathcal{X}$ through feature-space perturbations. To be specific, in the problem space, the following optimization is solved [14]:

$$
\arg \min_{\mathcal{T}} \quad g_1(\psi'(z) = \mathcal{T}(z)) \quad \text{s.t.} \quad \mathcal{T}(z) \models \Gamma_Z,
$$

where $\mathcal{T}$ is a sequence of transformations that satisfy the domain constraints defined in the problem space, $\Gamma_Z$, such as preserving the malicious functionality of the malware [18]. In the feature space, the following optimization is solved [14], [16]:

$$
\arg \min_{\delta} \quad g_1(x' = x + \delta) \quad \text{s.t.} \quad \delta \models \Omega,
$$

where the perturbation vector $\delta$ must satisfy the domain constraints defined in the feature space, $\Omega$. Note that most existing work on generating feature-space adversarial examples do not consider domain constraints but instead adopt the naive norm bound [14]. In this paper, we only consider feature-addition perturbations since feature-removal perturbations may change the functionalities of the original malicious apps [16].

**Feature-Space RealAEs.** In the feature space, $x' = x + \delta$ is a RealAE if there exists at least one corresponding malware app $z'$ in the problem space (i.e., $\psi(z') = x'$) that not only bypasses malware detection but also satisfies problem-space constraints $\Gamma_Z$. Figure 1 illustrates how adversarial transformations in the problem space make adversarial perturbations in the feature space. Reconstructing $z'$ from $x'$ is not possible since $\psi$ (i.e., mapping function from $\mathcal{Z}$ to $\mathcal{X}$) is neither invertible nor differentiable [14]. However, to verify the realizability of $x'$, there is no need to directly reconstruct $z'$ from $x'$ to see if $z'$ meets the domain constraints in the problem space because satisfying the domain constraints in the feature space is sufficient. In other words, $x'$ is realizable if it meets the domain constraints in the feature space because they demonstrate Android malware properties in the feature space (cf. Section 5.3).

3.2. Domain Constraints in the Feature Space

Here we introduce our new feature-space interpretation for the four aspects of domain constraints defined in the problem space.

(a) **Available perturbations** are all adversarial perturbations $\Delta = \{\delta_1, \delta_2, \ldots, \delta_n\}$ in the feature space that ensures $x' = x + \delta_i$ meets domain constraints. Using these perturbations makes $x'$ corresponds to at least one problem-space RealAE $z'$.

Generally, an Android app contains different units (e.g., statements, functions, and classes) that provide various functionalities. As shown in Figure 2, the presence of a specific feature in the feature vector depends on the existence of the corresponding unit in the app. Moreover, the dependencies between multiple units (e.g., sendSMS API...
and \text{SEND_SMS} permission) also indicate that they offer a particular functionality (e.g., sending messages in Android apps). In the problem space, practical transformations are the ones that consider these sorts of dependencies during app modification. For instance, in the code transplantation technique used to manipulate Android apps [14], [15], [28], [29], an organ (i.e., a transformation) is extracted from a donor based on the code dependencies because an organ must include all codes associated with a certain functionality [47]. In problem space, the dependencies between units can be clarified by the System Dependency Graph [47]; however, these dependencies can be extracted from samples in the feature space. Therefore, we argue that using feature dependencies is sufficient for interpreting the domain constraints in the feature space. Specifically, according to the domain constraints defined in the problem space (i.e., preserved-semantic, robustness-to-preprocessing, and plausibility constraints), we introduce the following two sets of dependencies over the feature values in the feature space in (b) and (c).

(b) Perfect feature dependencies. Given a feature-space adversarial example \( x' = x + \delta_i \), the perturbation \( \delta_i \in \Delta \) might not satisfy domain constraints if \( \delta_i \) does not guarantee all perfect feature dependencies.

The semantic equivalence of two programs (e.g., Android apps) is undecidable [47], therefore, in the problem space, adversaries satisfy the preserved-semantics constraint by installing and running the manipulated app \( z' \) on an Android emulator and performing smoke testing to make sure that \( z' \) can be executed without crashing [14], [17], [28], [48]. Similarly, in the feature space, we should ensure that all perfect feature dependencies are extracted from an executable app also appear in \( x' \). Otherwise, if only one of the perfectly dependent features exists in \( x' \), \( z' \) might not be executable due to the lack of other dependent units.

(c) Relatively strong feature dependencies. Given a feature-space adversarial example \( x' = x + \delta_i \), the perturbation \( \delta_i \in \Delta \) might not satisfy domain constraints if each feature in \( \delta_i \) does not guarantee all relatively strong (including perfect) feature dependencies. These dependencies indicate the most dependent features for each feature.

Generally, beyond just preserving the semantics, robustness-to-preprocessing and plausibility constraints further require the adversarial example \( z'/\delta_i \) to be similar to a realizable program in the problem/feature space. For this reason, in the feature space, we should ensure that \( x' \) keeps all features relatively strongly dependent in order to achieve a similar feature representation to that of an executable Android app. Considering relatively strong feature dependencies is sufficient because they capture the most important feature dependencies. Specifically, to satisfy the robustness-to-preprocessing constraint in the problem space, it is ensured that preprocessing operators cannot remove unnecessary content (e.g., unused permissions) that has been added to \( z \) during generating \( z' \) [14]. Similarly, in the feature space, we ensure that there are no removable added features that appeared in \( x' \) by keeping the features that have a relatively strong dependency on each added feature. In other words, a specific feature \( f_i \in x' \) is considered to represent an unused unit \( u_i \in z' \) when none of its dependent features appears in \( x' \). Moreover, to satisfy the plausibility constraint in the problem space, \( z' \) is ensured to be plausible under manual inspection [14]. Similarly, in the feature space, we ensure that \( x' \) looks plausible when the feature representation is inspected.

4. Extracting Feature-Space Domain Constraints

This section introduces our learning-based method for extracting the above-defined domain constraints in the feature space. Note that the learned constraints may only approximate the true distribution of the actual data. Specifically, we simply rely on feature correlations to identify perfect feature dependencies, and a graph-based algorithm called Optimum-Path Forest (OPF) to further identify the rest of the relatively strong dependencies. OPF is a parameter-independent algorithm [34] that essentially considers feature dependencies in our problem to partition dependent features into a cluster.

4.1. Preliminaries of Optimum-Path Forest

Optimum-Path Forest (OPF) is an efficient pattern recognition algorithm based on graph theory [34]. This algorithm reduces a pattern recognition problem to the partitioning of a graph \( G = (V, E) \) derived from input dataset [49]. \( G \) is a complete weighted graph wherein the vertices \( V \) are the feature vectors in the input dataset and the edges \( E = V \times V \) are undirected arcs that connect vertices. Moreover, each edge \( e_{i,j} \in E \) is weighted based on the distance between the feature vectors of corresponding vertices \( v_i \) and \( v_j \) (i.e., \( d(v_i, v_j) \)). OPF algorithm works based on a simple hypothesis called transitive property in which the vertices belonging to the same partition are connected by a chain of adjacent vertices [49]. This algorithm requires several key vertices \( P \subset V \) called prototypes that have been found from \( V \) based on various approaches such as probability density function [50]. The OPF algorithm partitions \( G \) into different Optimum-Path Trees (OPTs), with each OPT rooted at one of the prototypes, through a competitive process among the prototypes to conquer the rest of vertices [49].

Suppose \( \pi_{v_j, v_i} = \{v_j, ..., v_k, v_i\} \) is a path from \( v_j \) to \( v_i \). In OPF algorithm, a connectivity function \( f_{\max} \), which is a smooth function, assigns a path cost to each path as follows [34]:

\[
f_{\max}(\{v_j\}) = \begin{cases} 0 & \text{if } v_j \in P \\ +\infty & \text{otherwise} \end{cases}
\]

\[
f_{\max}(\pi_{v_j, v_i}, \{v_k, v_i\}) = \max \{f_{\max}(\pi_{v_j, v_k}), d(v_k, v_i)\},
\]

(3)
where \( \pi_{v_j,v_k}(v_k,v_i) \) shows the connection of the edge \( (v_k,v_i) \) to the path \( \pi_{v_j,v_k} \). As shown in Equation (3), the path cost of \( \pi_{v_j,v_k} \) is the maximum weight of edges along the path. OPF algorithm aims to find an optimal path for each \( v_i \in V \) by minimizing \( f_{\text{max}} \) through the following cost function:

\[
C_{v_i} = \min_{\forall v_j \in P, \pi_{v_j,v_i}} \{ f_{\text{max}}(\pi_{v_j,v_i}) \}. \quad (4)
\]

In general, the complete graph \( G \) is partitioned into several OPTs by finding a path from each \( v_i \in G \) to the best prototype \( p \in P \), which provides an optimal path with the minimum path cost for \( v_i \).

4.2. Proposed Method

As depicted in Figure 3, the OPF-based method consists of the following major components to extract domain constraints \( \Gamma' = \{ \Upsilon, \Lambda \} \) where \( \Upsilon \) and \( \Lambda \) show perfect and relatively strong feature dependencies, respectively.

(i) Identification of perfect feature dependencies. We use phi coefficient [51] to measure the correlation between every pair of features because our feature space \( X \) contains binary features. The correlation between a pair of features is perfect if the coefficient between them equals 1. We create \( \Upsilon \), the set of all perfect feature dependencies based on the identified perfect correlations. Note that each \( B_i \in \Upsilon \) includes all features in \( F \) that are perfectly correlated.

(ii) Identification of relatively strong feature dependencies. Based on our explanation in Section 3.2, considering only the perfect correlations is not sufficient to satisfy all the domain constraints. For this reason, we adopt OPF to further learn other relatively strong dependencies beyond the perfect ones. The proposed version of OPF partitions \( F \) into the different groups \( A_i \) where the features that are more interdependent belong to the same cluster. As shown in Figure 3, to construct OPF, we first create a complete weighted graph \( G = (V, E) \) where \( G = F \), and \( E \) includes the edges between each pair of features \( (f_i,f_j) \) weighted by phi coefficient. Then, from each set of very strongly correlated features (i.e., \( \varphi > 0.9 \)) indicating a dense area, we randomly select one feature as a prototype. Finally, \( G \) is partitioned based on the typical method used in OPF algorithm which is slightly modified because here, the weights of edges are specified based on the phi coefficient instead of distance as in the original algorithm. More concretely, we change the path-cost function from \( f_{\text{max}} \) to \( f_{\text{min}} \), and for each \( f_a \in F \), the cost function of OPF (i.e., Equation (4)) is modified to:

\[
C_{f_a} = \max_{\forall f_p \in P, \pi_{f_p,f_a}} \{ f_{\text{min}}(\pi_{f_p,f_a}) \}, \quad (5)
\]

where \( F \) is the feature set of \( X \), \( \pi_{f_p,f_a} \) is a path from \( f_p \) to \( f_a \), and \( P \) shows the prototype set. Optimum-path trees constructed by the OPF algorithm let us determine other relatively strong correlations because an OPT includes a subset of all features in the feature space (i.e., \( A \subset F \)) where each feature \( f_a \in A \) is more dependent to other features in \( A \) as compared to the rest of features \( F \setminus A \). According to the specified OPTs, we create \( \Lambda \) which is
the set of all relatively strong feature dependencies. Each \( A_i \in \Lambda \) contains all the features in \( F \) that are relatively strongly correlated.

Note that we also demonstrate that our OPF-based identification method is better than a simple baseline that uses a fixed threshold to keep highly correlated features (see results in Section 7.4).

5. Applying Feature-Space Domain Constraints

This section demonstrates how our learned domain constraints can be applied in our defense approach to improving the robustness of ML-based Android malware detection against RealAEs. We propose a new transformation function (i.e., \( \lambda : X \rightarrow H \)) that maps each \( x \in X \) to an \( h \in H \). Suppose \( L \) and \( A_i \subset F \) indicate the dimensions (i.e., features) of the feature space \( H \) and the feature set of \( i \)-th OPT in the constructed OPF, respectively. As can be seen from Figure 4, in the proposed new feature space \( H \), for each \( A_i \), there exists a feature \( l_i \in L \) as the representative of all features in \( A_i \) to guarantee the accuracy of the detection model. This is because \( A_i \) contains interdependent features that have similar information about the target class; therefore, confirming that a feature in \( L \) is enough for representing these relevant features.

In order to determine whether a feature \( l_i \in L \) should appear in \( H \) for a sample \( x \in X \), we need an activation function that transfers the effect of input features in \( A_i \) to the output feature \( l_i \). The desired activation function should uniformly increase the probability of a feature appearing in the output as the number of features in the input increases. Moreover, this function must make sure that changing a feature \( f_j \in A_i \) cannot simply change \( l_i \) because we aim to increase the evasion costs of adversaries. In this work, we use a sigmoid activation function, which is a monotonic function, to change \( l_i \) based on the features in \( A_i \). Since the sigmoid function has \( S-Shaped Curve \) with slow growth in its initial phase [52], we expect that it can smooth the high effect of adversarial perturbations that may happen in \( A_i \) on \( l_i \). According to the proposed transformation method, the value of feature \( l_i \) in \( h = \lambda(x) \) is computed as follows:

\[
l_i = \begin{cases} 
 1 & \sigma(s) = \sum_{f_j \in A_i} c_j \cdot x_j) > 0.7 \\
 0 & \text{otherwise,} 
\end{cases}
\]  

where \( \sigma(\cdot) = \frac{1}{1 + e^{-\cdot}} \) is the sigmoid function. The \( c_j \) is the path cost of feature \( f_j \) in the constructed OPF, and \( x_j \) is the value of feature \( f_j \) in \( x \). We consider \( \sigma(c_p : x_p) \simeq 0.7 \) as a proper threshold for our activation function because we must ensure that solely modifying the key feature \( f_p \) is sufficient for changing \( l_i \), i.e., \( \sigma(c_p : x_p) > 0.7 \) with \( x_{fp} = 1 \) ensures \( c_{fp} = 1 \) without considering the rest of features in \( A_i \). Otherwise, when \( f_p \) is not changed, the adversary may have to change many other features to make \( l_i \) appear in \( h \) because of the nature of the sigmoid function. Note that although changing \( f_p \) is sufficient, an adversary has to change the features in \( A_i \) that are dependent on \( f_p \) to satisfy the domain constraints. More generally, using a moderate threshold of 0.7 is desired because a very low threshold enables the output feature to appear even with small perturbations in the input features. Alternatively, a very high threshold prevents the effect of input features from being transferred to the output.

6. Learned vs. Actual Domain Constraints

In this section, we introduce two ways to validate how our learned feature-space domain constraints can represent the domain constraints that Android malware apps should satisfy in the problem space. First, we test if incorporating feature-space RealAEs into the model training process can help mitigate problem-space RealAEs that are generated by a similar attack. Second, we test if RealAEs and unRealAEs can be distinguished based on the learned feature-space domain constraints.
6.2. Distinguishing RealAEs from unRealAEs based on learned domain constraints.

We introduce a new metric called Constraints Satisfaction Rate (CSR) to measure the ratio of the features that satisfy our learned domain constraints to all the features of a certain sample. By satisfying our learned domain constraints, we mean that one specific feature appears simultaneously with at least one of its relatively strong dependent features and all its perfectly dependent features specified in $\Lambda$ and $\Upsilon$, respectively. Note that we do not expect the feature representation of real-world apps to fully satisfy our domain constraints because as shown in Figure 5, our learned domain constraints are indeed a subset of true feature-space domain constraints. This is mainly because our domain constraints are learned from a finite set of samples that might not represent the true distribution of all Android actual apps.

7. Experimental Results

In this section, we empirically evaluate the performance of our learned domain constraints in improving the robustness of Android malware detectors against different realistic problem-space evasion attacks. All the experiments have been performed on a Debian Linux workstation with an Intel (R) Core (TM) i7-4770K, CPU 3.50 GHz, and 32 GB RAM.

7.1. Experimental Setup

Dataset. We use a public Android dataset [14] including $\approx 170K$ Android apps collected from AndroZoo [53]. In this dataset, an app is defined as malware if it is detected by 4+ VirusTotal AVs, and as a benign sample, if no AVs detect it. We randomly select $50K$ samples as the training set and $30K$ samples as the test set for Android malware detectors. Specifically, the training set, which is also used to measure the correlation of features, contains $45K$ clean and $5K$ malware samples, and the test set contains $25K$ clean and $5K$ malware samples. All the samples are represented based on a binary feature space, DREBIN [4] feature space, before being fed into the malware detectors. DREBIN feature space is the most common binary feature set in the area of Android malware and is still widely used in recent studies [14], [15], [18], [24], [29], [35]. Since DREBIN is a very high-dimensional but sparse feature space, we select 10,000 features that are present most frequently following recommendations from previous studies [16], [17].

For evaluating the adversarial robustness of different Android malware detectors, we use $1K$ malware samples considered in [29] to generate the adversarial examples.

Threat Models and Attacks. The evasion attacks considered in our experiments generate adversarial examples in various threat models. Generally, a threat model is described with three attributes:

- **Adversary’s Goal.** The goal of the adversary is to cause the Android malware detector to misclassify the adversarial (malware) example as benign.

6.1. Incorporating feature-space RealAEs to help mitigate (similar) problem-space RealAEs.

Suppose $D_m = \{(x_i, y_i)|x_i \in \mathcal{X}, y_i = 1, i = 1, \ldots, k\}$ shows a fraction of all the malware samples in the input training set $D$. We first construct an adversarial set $D'_m = \{(x'_i, y'_i)|x'_i \in \mathcal{X}, y'_i = 1, i = 1, \ldots, k\}$ where each $x'_i$ is the adversarial example of $x_i$ that is generated by using an evasion attack. Then, we use $D' = D \cup D'_m$, the mixed set that is augmented with adversarial examples, to train the ML models.

In this study, we consider the extracted meaningful feature dependencies in the evasion attack that is supposed to prepare $D'_m$ because we aim to augment $D$ with RealAEs. In fact, every adversarial perturbation $\delta$ generated by the evasion attack must satisfy not initial feature space constraints (i.e., $\delta \perp \Omega$) but also our learned feature-space domain constraints (i.e., $\delta \perp \Gamma'_\Upsilon$) because as shown in Figure 5, satisfying $\Gamma'_\Upsilon$ can make an unRealAE becomes a RealAE. Lines 3 - 11 in Algorithm 1 show how a successful adversarial example becomes realizable by adding dependent features.

---

**Algorithm 1:** Generating feature-space RealAEs.

**Input:** $x$, the feature vector of the malware app; $\text{Attack}$, the feature-space evasion attack; $\mathcal{M}$, target malware detector; $\Gamma'_\Upsilon = \{\Upsilon, \Lambda\}$, our learned domain constraints;

**Output:** $x'$, a feature-space RealAE; none, there is no adversarial example.

1. $x' \leftarrow x$.

2. if $\mathcal{M}$ classifies $x'$ as a benign sample then

3. foreach new feature $f_j$ added into $x'$ do

4. Find $\mathcal{A}_i \in \Lambda$ containing $f_j$;

5. if none of features of $\mathcal{A}_i \setminus \{f_j\}$ exists in $x'$ then

6. Randomly add one of the features of $\mathcal{A}_i \setminus \{f_j\}$ into $x'$

7. end

8. if there exists $\mathcal{B}_k \in \Upsilon$ including $f_j$ then

9. Add all features of $\mathcal{B}_k$ into $x'$

10. end

11. end

12. if $\mathcal{M}$ classifies $x'$ as a benign sample then

13. return $x'$

14. else

15. return none

16. end

17. else

18. return none

19. end

20. end
• **Adversary’s Knowledge.** The adversary may have perfect knowledge (PK), limited knowledge (LK), or zero knowledge (ZK) about the target model, including its training data, feature space, and parameters. In other words, in PK, LK, and ZK attacks, the target model is a white-, gray-, and black-box model for the adversaries, respectively.

• **Adversary’s Capability.** The Adversary can generate adversarial examples either in the feature space by perturbing feature representations of Android malware apps under feature-space constraints, or in the problem space by applying a sequence of transformations to the Android malware apps under domain constraints [14].

There are only a few studies (i.e., [14], [29], [31], [35]) that have succeeded in proposing realistic evasion attacks in the Android domain. This paper considers two realistic problem-space attacks, *PiAttack* [14] and *EvadeDroid* [29], to measure the adversarial robustness of malware detectors used in Sections 7.2 and 7.3. *PiAttack* and *EvadeDroid* transform Android apps into adversarial ones by attacking white- and black-box target malware detectors, respectively. In addition to the above attacks, a white-box feature-space attack, *PK-Feature* [16], and a gray-box feature-space attack, *Sparse-RS* [54], are considered to generate adversarial examples used in Sections 7.3 and 7.4. The details of these attacks are described as follows.

• **PiAttack** [14] generates problem-space RealAEs by applying effective transformations (i.e., code snippets called gadgets extracted from donor apps) specified by feature-space perturbations on the target model. This attack adds not only primary features to bypass malware detection but also side-effect features to meet the domain constraints. The attack was originally tested in the PK setting, but here we also test it in an LK setting where the adversarial examples transfer from a surrogate model to a target model.

• **EvadeDroid** [29] also generates problem-space realizable adversarial examples through a sequence of transformations; however, it does so by querying the target model in a ZK setting. In addition to the original ZK setting, here we also consider a more restricted setting where EvadeDroid is only allowed to query a surrogate model and then transfer the adversarial examples to the target model. Specifically, we set the query budget $Q = 10$, and $\alpha = 50\%$ (i.e., the percentage of the relative increase in the size of a malware sample after manipulation).

• **PK-Feature** [16] attack is a white-box attack that perturbs the influential features of a malware sample in each iteration until it reaches the maximum allowable perturbations (i.e., 10 in our settings). Note that the influential features are specified based on the weight parameters of the target classifier learned during training.

• **Sparse-RS** [54] attack is a gray-box attack that perturbs malware samples into AEs based on a random search. This attack queries the target model in each iteration to see the impact of the perturbed features on the classification. Note that in our settings, the maximum number of allowed perturbations is 10. Moreover, the query budget and initial decay factor (i.e., $\alpha_{init}$) are 100 and 1.6, respectively.

**Evaluation Metrics.** For evaluating the malware detectors, we consider Accuracy (Acc). Specifically, we calculate clean Acc on benign malware examples for model utility and robust Acc on adversarial malware examples for robustness. We also report the Training Time to compare the efficiency of different (robust) detectors and the average number of added features required for achieving successful adversarial examples.

7.2. Evaluation of Proposed Defense

This section aims to evaluate our robust feature space introduced in Section 5. We consider the following four ML-based Android malware detectors:

• **DREBIN-Original** [4], a well-known Android malware detector that is based on the linear Support Vector Machine (SVM). It is trained with the original DREBIN feature space.

• **Sec-SVM** [16] is the secure version of DREBIN-Original for strengthening the robustness of linear SVM against adversarial examples. Sec-SVM relies on more features, and this increases the evasion cost.

• **DREBIN-Robust** is our robust DREBIN detector trained with our new robust feature space.

• **DREBIN-FeatureSelect** is similar to DREBIN-Original but trained with a lower-dimensional feature space with the most influential features selected by the Linear SVC. It is considered to ensure the selected original features lead to the same clean accuracy as our DREBIN-Robust.

In the first experiment, we measure the robustness of malware detectors against transferable adversarial examples generated by the two attacks using DREBIN-Original as the surrogate model. We make sure that the adversarial examples are generated from the malware samples correctly detected by all four malware detectors, and the results are calculated on the successful adversarial examples for DREBIN-Original. As can be seen from Table 2, all detectors achieve similar clean accuracy. For robustness, although DREBIN-FeatureSelect achieves substantial robustness compared to DREBIN-Original with the help of feature selection [55], both DREBIN-Robust and Sec-SVM are much more robust. Specifically, our DREBIN-Robust outperforms Sec-SVM, especially for EvadeDroid. We also notice that EvadeDroid is more transferable than PiAttack. This is consistent with the previous finding that Perfect-knowledge attacks, which exploit rich gradient information, lead to low transferability due to overfitting to the surrogate model [56], [57].
of added features required by EvadeDroid for bypassing DREBIN-Robust is lower than bypassing the other detectors due to the property of EvadeDroid. Specifically, in EvadeDroid, a transformation is applied to a malware app only if it can increase the chance of generating successful adversarial examples. This leads to the difference between the number of transformations applied to the detectors (e.g., 1.00 for DREBIN-Robust vs. 2.54 for Sec-SVM). This difference indicates that most transformations are not good enough for attacking DREBIN-Robust, and consequently leads to a difference in the number of added features.

7.3. Evaluating Our Learned Domain Constraints

In this subsection, we validate the utility of our learned domain constraints for representing Android malware properties. Specifically, we conduct two experiments, with the first one to test if incorporating our feature-space RealAEs can help mitigate (similar) problem-space RealAEs and the second one to distinguish realizable AEs from unrealizable AEs based on our learned domain constraints.

Incorporating our feature-space RealAEs can help mitigate (similar) problem-space RealAEs. We augment our initial training data with adversarial examples generated by different problem-space and feature-space attacks. Specifically, we use two feature-space attacks, PK-Feature and Sparse-RS, which are similar to the two problem-space attacks, PiAttack and EvadeDroid, respectively, under our domain constraints. As a comparison, we also consider AT-PK-Feature-un and AT-Sparse-RS-un, which are respectively similar to AT-PK-Feature and AT-Sparse-RS but without the use of our domain constraints. Table 4 illustrates the results. We make sure that the adversarial examples are generated from the malware samples correctly detected by all eight augmented malware detectors, and the robust accuracy is calculated only on the successful transferable adversarial examples.

As shown in Table 4, AT-PK-Feature/AT-Sparse-RS outperforms AT-PK-Feature-un/AT-Sparse-RS-un in mitigating PiAttack/EvadeDroid. We can also observe that when the attacks for augmentation and evaluation are not similar, the data augmentation may not work. In addition, we show that using more examples for augmentation yields higher mitigation performance. This might be explained by the fact that using more examples increases the chance of having more similar adversarial examples to the ones generated by the PiAttack and EvadeDroid. Note that the ~10K adversarial examples are generated from 500 randomly selected malware samples. We can also observe that retraining our AT-PK-Feature model is 492× faster than retraining AT-PiAttack. This is due to the fact that generating a feature-space RealAE costs 0.22 seconds while generating a problem-space RealAE costs 110.53 seconds. This efficiency suggests the great potential of using feature-space RealAEs in improving adversarial robustness through adversarial training.

Table 5 further reports the setting of directly attacking each detector (instead of transferring from DREBIN-

---

**Figure 6.** The evasion success rates of PiAttack [14] against different DREBIN detectors when varying the number of added features.
TABLE 4. The performance of different DREBIN detectors in terms of re-training time (s), and clean and robust accuracy (%). RealAEs are transferred from DREBIN-Original. “*” means both the surrogate and target models are DREBIN-Original. The best performance of three trials is reported.

| Model                | Number of AEs | Re-training Time (s) | Clean Acc | Robust Acc (PiAttack) | Robust Acc (EvadeDroid) |
|----------------------|---------------|----------------------|-----------|-----------------------|-------------------------|
| DREBIN-Original      | N/A           | N/A                  | 96.71     | 0*                    | 26.84*                  |
| AT-PiAttacks         | 500           | 110,543.78           | 96.64     | 100                   | 13.86                   |
| AT-PK-Feature (ours) | 500           | 224.38               | 96.68     | 32.98                 | 2.45                    |
| AT-PK-Feature-un     | 500           | 187.76               | 96.65     | 20.57                 | 5.78                    |
| AT-PK-Feature (ours) | ≈ 10K         | 1,559.25             | 96.70     | 78.84                 | 2.46                    |
| AT-PK-Feature-un     | ≈ 10K         | 952.34               | 96.62     | 30.81                 | 3.16                    |
| AT-EvadeDroid        | 500           | 141,820.14           | 96.66     | 0                     | 100                     |
| AT-Sparse-RS (ours)  | 500           | 900.99               | 96.61     | 0                     | 19.65                   |
| AT-Sparse-RS-un      | 500           | 834.22               | 96.59     | 0                     | 17.37                   |
| AT-Sparse-RS (ours)  | ≈ 10K         | 9,674.14             | 96.52     | 2.60                  | 48.95                   |
| AT-Sparse-RS-un      | ≈ 10K         | 8,794.15             | 96.54     | 0                     | 38.77                   |

TABLE 5. The robustness of different augmented DREBINs against adversarial examples that generated directly on themselves in terms of Accuracy and Number of added Features (NoF). AT-PiAttack, AT-PK-Feature, and AT-PK-Feature-un are attacked by PiAttack, while AT-EvadeDroid, AT-Sparse-RS, and AT-Sparse-RS-un are attacked by EvadeDroid. The best performance of three trials is reported.

| Model             | Robust Acc (%) | NoF |
|-------------------|----------------|-----|
| AT-PiAttacks      | 3.40           | 32.78 |
| AT-PK-Feature (ours) | 80.8          | 127.12 |
| AT-PK-Feature-un  | 50.2           | 111.44 |
| AT-EvadeDroid     | 100            | N/A |
| AT-Sparse-RS (ours) | 45.73         | 66.19 |
| AT-Sparse-RS-un   | 39.84          | 65.15 |

TABLE 6. The specifications of 5 randomly selected Android malware apps in terms of Number of added Features (NoF).

| Model             | NoF by PiAttack | NoF by EvadeDroid | NoF by PK-Feature | NoF by Sparse-RS |
|-------------------|-----------------|-------------------|-------------------|------------------|
| App 1             | 5               | 38                | 3                 | 8                |
| App 2             | 4               | 45                | 4                 | 8                |
| App 3             | 4               | 96                | 4                 | 8                |
| App 4             | 4               | 118               | 4                 | 8                |
| App 5             | 4               | 52                | 4                 | 8                |

Our learned domain constraints are potentially useful to distinguish RealAEs from unRealAEs. We first demonstrate how the added features in adversarial examples generated by different attacks on the original DREBIN can satisfy our learned domain constraints. Specifically, we consider four attacks, i.e., the domain-constraint-aware attacks, Pi-Attack and EvadeDroid, and the domain-constraint-agnostic attacks, PK-Feature and Sparse-RS [54]. We calculate the constraint satisfaction rate (CSR) for 5 randomly selected malware apps and Figure 7 reports the results. As can be seen, the adversarial features added by PiAttack and EvadeDroid can better satisfy our learned domain constraints than the PK-Feature attack (on average, 66% and 75.94% vs. 5% and 2.5%). The high CSR results confirm that the feature dependencies learned by our new method can well represent the (problem-space) domain constraints. We expect that learning with a larger amount of data would further improve the results. It can also be observed that CSR of PiAttack is sometimes lower than EvadeDroid. This might be explained by the fact that the number of features added by PiAttack is considerably lower than EvadeDroid, and so each new feature has a higher impact on CSR than EvadeDroid (cf. Table 6).

The above observation suggests that it is possible to differentiate RealAEs from unRealAEs based on a CSR threshold. Specifically, we consider the AEs above this CSR threshold as RealAEs and otherwise unRealAEs. Here we calculate the CSR for each example based on all its features.

Figure 7. Comparison of 5 randomly selected adversarial examples generated by the domain-constraint-aware attacks, PiAttack and EvadeDroid, and the domain-constraint-agnostic attacks, PK-Feature and Sparse-RS, in satisfying our learned domain constraints.
Figure 8. Differentiating RealAEs (PiAttack/EvadeDroid) from unRealAEs (PK-Feature/Sparse-RS) based on our learned domain constraints.

rather than only the added adversarial features because, in practice, it is not known which of all features are added by an attack. Figure 8 shows the results with varied CSR thresholds. As can be seen, the CSR threshold should be high enough to differentiate RealAEs from unRealAEs. However, the CSR should not be set too high because our learned domain constraints cannot perfectly represent the actual domain constraints. It is worth noting that in our case, the number of added adversarial features is small for all attacks compared to the total features in the samples. However, the more added features, the larger the difference between RealAEs and unRealAEs. This is because only features added in unRealAEs do not satisfy domain constraints but both the original features and features added in RealAEs satisfy domain constraints.

7.4. Evaluating Our OPF-based Method

In this experiment, we aim to validate the ability of OPF in extracting meaningful feature dependencies. We compare the proposed OPF-based approach with a straightforward baseline method that is based on threshold clustering (TC). For a specific feature \( f_n \), TC only maintains its top-N dependent features. Here we specifically compare the adversarial robustness of DREBIN augmented with adversarial examples generated by a new attack called PK-Feature-Ideal, when OPF- or TC-based approaches is used to make adversarial examples realizable. Note that PK-Feature-Ideal is indeed PK-Feature attack under our learned domain constraints and with additional knowledge about the primary features added by PiAttack to generate adversarial examples. This ideal case leads to a better comparison of the performances of OPF- and TC-based methods as the generated adversarial examples will very similar to those generated by PiAttack. Table 7 shows that the clean accuracy for the two detectors is similar. For robustness, although using the TC method (with Top-80) also leads to a substantial robust accuracy (69.39%), using our OPF is 13.28% better. It is also worth noting that the TC method requires careful tuning (e.g., by linear search) of the hyperparameter N to achieve the best performance but our OPF does not.

7.5. Discussion

In order to improve the adversarial robustness of malware detection, it is necessary to provide a realistic view of the vulnerabilities of ML models against RealAEs, which are generated under domain constraints of malware programs [15]. The experimental results have demonstrated the general effectiveness of our features-space solution in strengthening the robustness of the detector. Here we further discuss its advantage from two main aspects: practicality and generalizability.

Practicality. The results reported in Section 7.3 clearly show the practicality of our approach in real scenarios where defenders need to retrain large ML models. For instance, as shown in Table 4, the training time of AT-PK-Feature is \( 492 \times \) faster than AT-PiAttack. This remarkable improvement is due to the fact that our approach directly works in the feature space without the time-consuming generation of problem-space adversarial examples. Moreover, the inefficiency of AT-PiAttack and AT-EvadeDroid also limits the quantity of effective adversarial examples generated in practice.

Generalizability. Our feature-space RealAEs also lead to high generalizability. For instance, in our adversarially-retrained detector, AT-PK-Attack, we can simply generate \( \approx 10K \) different variants of adversarial examples from only 500 malware apps to cover more adversarial perturbations that might be generated by the adversaries. In contrast, problem-space realizable adversarial examples have limitations because they rely on specific problem-space transformations [20], [28], [32].


8. Limitations and Future Work

While extensive experiments have demonstrated the effectiveness of our new approach, there are limitations that need further investigation. First, it is worth exploring how the proposed approach can be applied to more complex feature spaces beyond the current binary space. This paper demonstrates the performance of our approach in understanding and learning Android malware properties in a binary feature space, DREBIN; however, our defined feature dependencies might not be able to apply for complex feature spaces considered in some malware detectors (e.g., [3], [31]). For instance, MaMaDroid [3] is an Android malware detector that uses continuous features derived from a Markov chain modeling built on Android API calls. The introduced types of feature dependencies that are used to capture Android malware properties seem ineffective in specifying the problem-space constraints for MaMaDroid. Working on complex mathematical feature dependencies (e.g., range, minimum, and maximum of feature values) that might be relevant to the domain constraints can be a proper way for addressing this limitation.

Second, like any data-driven technique, our approach may also be potentially biased to the specific training data. The proposed problem-space learning technique follows a purely data-driven technique for determining domain constraints in the feature space. Obviously, only working with data is risky because data might be inaccurate and incomplete. For instance, the purely data-driven approaches might intensify the concept drift issue [58], which is a common challenge in ML context. Here, we empirically illustrate this limitation by testing the DREBIN-Original, Sec-SVM, and DREBIN-Robust on 15K (13K clean and 2K malware) newly collected Android apps from AndroZoo. These new samples have been released in the recent three years (i.e., 2020-2022). As can be seen from Table 8, all new results change by a certain degree compared to the old results, indicating the existence of concept drift. Specifically, in terms of accuracy, DREBIN-Original, Sec-SVM, and DREBIN-Robust are degraded but DREBIN-FeatureSelect is improved.

Third, incorporating domain knowledge could further improve our approach. In this study, not only the quality of data but also the effectiveness of the proposed problem-space learning method affect the identification of the meaningful feature dependencies. Domain knowledge is a rich resource that can improve the proposed approach; however, a key question is how the knowledge of domain experts can be used for specifying problem-space constraints in the feature space. It seems domain knowledge can make feature-dependency searching more precise; therefore, an interesting avenue for future study is employing the domain knowledge to complement the data-driven approaches to finding feature dependencies.

9. Conclusion

In this paper, we have provided a new approach to strengthening the robustness of ML-based Android malware detection based on crafting feature-space RealAEs. Specifically, we present a new interpretation of domain constraints in the feature space by exploiting meaningful feature dependencies. We not only consider statistical correlations but also adopt OPF to extract such dependencies and apply them to a novel defense technique, which is based on constructing a robust feature representation. The empirical results show the general effectiveness of our new approach in strengthening the robustness of Android malware detection. In particular, we demonstrate that generating our novel feature-space RealAEs are more efficient than problem-space RealAEs.

References

[1] Hui-Juan Zhu, Zhu-Hong You, Ze-Xuan Zhu, Wei-Lei Shi, Xing Chen, and Li Cheng. Droiddet: effective and robust detection of Android malware using static analysis along with rotation forest model. Neurocomputing, 272:638–646, 2018.
[2] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. Droiddetector: Android malware characterization and detection using deep learning. Tsinghua Science and Technology, 21(1):114–123, 2016.
[3] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting Android malware by building markov chains of behavioral models (extended version). ACM Transactions on Privacy and Security (TOPS), 22(2):1–34, 2019.
[4] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In Network and Distributed System Security Symposium (NDSS), 2014.
[5] Win Zaw Zarni Aung. Permission-based android malware detection. International Journal of Scientific & Technology Research, 2(3):228–234, 2013.
[6] Yousra Aafer, Wenliang Du, and Heng 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.
[7] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In Proceedings of the 2013 ACM workshop on Artificial intelligence and security, pages 45–54, 2013.
[8] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping 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.
[9] Bo Li and Yevgeniy Vorobeychik. Evasion-robust classification on binary domains. ACM Transactions on Knowledge Discovery from Data (TKDD), 12(4):1–32, 2018.
Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In Proceedings of the International Conference on Learning Representations (ICLR), 2014.

Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Anantharam Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In 2016 IEEE symposium on security and privacy (SP), pages 582–597. IEEE, 2016.

Thibault Simonetto, Salijona Dyrmsih, Salah Ghamizi, Maxime Cordy, and Yves Le Traon. A unified framework for adversarial attack and defense in constrained feature space. arXiv preprint arXiv:2112.01156, 2021.

Ryan Sheatsley, Nicolas Papernot, Michael Weisman, Gunjan Verma, and Patrick McDaniel. Adversarial examples in constrained domains. arXiv preprint arXiv:2011.01183, 2020.

Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1332–1349. IEEE, 2020.

Raphael Labaca-Castro, Luis Muñoz-González, Feargus Pendlebury, Gabi Dreo Rodosek, Fabio Pierazzi, and Lorenzo Cavallaro. Universal adversarial perturbations for malware. arXiv preprint arXiv:2102.06747, 2021.

Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Ignio Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! a case study on android malware detection. IEEE Transactions on Dependable and Secure Computing, 16(4):711–724, 2017.

Deqiang Li and Qianmu Li. Adversarial deep ensemble: Evasion attacks and defenses for malware detection. IEEE Transactions on Information Forensics and Security, 15:3886–3900, 2020.

Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhui Xu. A framework for enhancing deep neural networks against adversarial attacks. IEEE Transactions on Network Science and Engineering, 8(1):736–750, 2021.

Hemant Rathore, Sanjay K Sahay, Piyush Nikam, and Mohit Sewak. Robust android malware detection system against adversarial attacks using q-learning. Information Systems Frontiers, 23(4):867–882, 2021.

Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In European symposium on research in computer security, pages 62–79. Springer, 2017.

Lingwei Chen, Shifu Hou, and Yanfang Ye. Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks. In Proceedings of the 33rd Annual Computer Security Applications Conference, pages 362–372, 2017.

Lingwei Chen, Shifu Hou, Yanfang Ye, and Shouhui Xu. Droideye: Fortifying security of learning-based classifier against adversarial android malware attacks. In 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pages 782–789. IEEE, 2018.

Xiaoleti Liu, Xiaojian Du, Xiaosong Zhang, Qingxin Zhu, Hao Wang, and Mohsen Guizani. Adversarial samples on android malware detection systems for iot systems. Sensors, 19(4):974, 2019.

Guangquan Xu, GuoHua Xin, Liting Jiao, Lian Liu, Shaoying Liu, Meiqi Feng, and Xi Zheng. Ofei: A semi-black-box android adversarial sample attack framework against dlass. arXiv preprint arXiv:2105.11593, 2021.

Harel Berger, Chen Hajaj, and Amit Dvir. When the guard failed the droid: A case study of android malware. arXiv preprint arXiv:2003.14123, 2020.

Salijona Dyrmsih, Salah Ghamizi, Thibault Simonetto, Yves Le Traon, and Maxime Cordy. On the empirical effectiveness of unrealistic adversarial hardening against realistic adversarial attacks. arXiv preprint arXiv:2202.03277, 2022.

Harel Berger, Amit Dvir, Chen Hajaj, and Rony Ronen. Do you think you can hold me? the real challenge of problem-space evasion attacks. arXiv preprint arXiv:2205.04293, 2022.

Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC), pages 288–302, 2017.

Hamid Bostani and Veelasha Moonsamy. Evadedroid: A practical evasion attack on machine learning for black-box android malware detection. arXiv preprint arXiv:2110.03301, 2021.

Xiao Chen, Chaooran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android hiv: A study of repackaging malware for evading machine-learning detection. IEEE Transactions on Information Forensics and Security, 15:987–1001, 2019.

Fabrizio Cara, Michele Scalas, Giorgio Giacinto, and Davide Maiorca. On the feasibility of adversarial sample creation using the android system api. Information, 11(9):433, 2020.

Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elowici. Generic black-box end-to-end attack against state of the art api call based malware classifiers. In International Symposium on Research in Attacks, Intrusions, and Defenses, pages 490–510. Springer, 2018.

Yunzhe Tian, Yingdi Wang, Endong Tong, Wenjia Niu, Liang Chang, Qi Alfred Chen, Gang Li, and Jiqiang Liu. Exploring data correlation between feature pairs for generating constraint-based adversarial examples. In 2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS), pages 430–437. IEEE, 2020.

Joao P Papa, Alexandre X Falcão, and Celso TN Suzuki. Supervised pattern classification based on optimum-path forest. International Journal of Imaging Systems and Technology, 19(2):120–131, 2009.

Francesco Croce, Maksym Andriushchenko, Naman D Singh, Nicolas Flammarion, and Matthias Hein. Sparse-rs: a versatile framework for query-efficient sparse-black-box adversarial attacks. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 36, pages 6437–6445, 2022.

Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In 2018 26th European signal processing conference (EUSIPCO), pages 533–537. IEEE, 2018.

Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In 2017 ieee symposium on security and privacy (SP), pages 39–57. IEEE, 2017.

Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, minced, and Sergul Aydore. Generic black-box end-to-end attack against state of the art api call based malware classifiers. In International Symposium on Research in Attacks, Intrusions, and Defenses, pages 490–510. Springer, 2018.

Francesco Croce, Maksym Andriushchenko, Naman D Singh, Nicolas Flammarion, and Matthias Hein. Sparse-rs: a versatile framework for query-efficient sparse-black-box adversarial attacks. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 36, pages 6437–6445, 2022.

Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In 2018 26th European signal processing conference (EUSIPCO), pages 533–537. IEEE, 2018.

Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In 2017 ieee symposium on security and privacy (SP), pages 39–57. IEEE, 2017.

Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, minced, and Sergul Aydore. Adversarial robustness with non-uniform perturbations. Advances in Neural Information Processing Systems (NeurIPS), 34:19147–19159, 2021.
[43] Liang Tong, Bo Li, Chen Hajaj, Chaowei Xiao, Ning Zhang, and Yevgeniy Vorobeychik. Improving robustness of {ML} classifiers against realizable evasion attacks using conserved features. In 28th USENIX Security Symposium (USENIX Security 19), pages 285–302, 2019.

[44] Alesia Chernikova and Alina Oprea. Fence: Feasible evasion attacks on neural networks in constrained environments. arXiv preprint arXiv:1909.10480, 2019.

[45] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O’Reilly. Adversarial deep learning for robust detection of binary encoded malware. In 2018 IEEE Security and Privacy Workshops (SPW), pages 76–82. IEEE, 2018.

[46] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. arXiv preprint arXiv:1702.05983, 2017.

[47] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, pages 257–269, 2015.

[48] Harel Berger, Chen Hajaj, and Amit Dvir. Evasion is not enough: A case study of android malware. In International Symposium on Cyber Security Cryptography and Machine Learning, pages 167–174. Springer, 2020.

[49] Hamid Bostani, Mansour Sheikhan, and Behrad Mahboobi. A strong coreset algorithm to accelerate opf as a graph-based machine learning in large-scale problems. Information Sciences, 555:424–441, 2021.

[50] Leonardo Marques Rocha, Fábio AM Cappabianco, and Alexandre Xavier Falcão. Data clustering as an optimum-path forest problem with applications in image analysis. International Journal of Imaging Systems and Technology, 19(2):50–68, 2009.

[51] David Freedman, Robert Pisani, and Roger Purves. Statistics (international student edition). Pisani, R. Purves, 4th edn. WW Norton & Company, New York, 2007.

[52] Ewa Lechman. Ict diffusion trajectories and economic development: Empirical evidence for 46 developing countries. In ICTs and the millennium development goals, pages 19–39. Springer, 2014.

[53] Kevin Allix, Tegawendé Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pages 468–471. IEEE, 2016.

[54] Francesco Croce, Maksym Andriushchenko, Naman D Singh, Nicolas Flammarion, and Matthias Hein. Sparse-rs: a versatile framework for query-efficient sparse black-box adversarial attacks. arXiv preprint arXiv:2006.12834, 2020.

[55] Hamid Bostani and Mansour Sheikhan. Hybrid of binary gravitational search algorithm and mutual information for feature selection in intrusion detection systems. Soft computing, 21(9):2307–2324, 2017.

[56] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In International Conference on Learning Representations (ICLR), 2017.

[57] Zhengyu Zhao, Hanwei Zhang, Renjue Li, Ronan Sicre, Laurent Amsaleg, and Michael Backes. Towards good practices in evaluating transfer adversarial attacks. arXiv preprint arXiv:2211.09565, 2022.

[58] Geoffrey I Webb, Roy Hyde, Hong Cao, Hai Long Nguyen, and Francois Petitjean. Characterizing concept drift. Data Mining and Knowledge Discovery, 30(4):964–994, 2016.