# CADE: Detecting and Explaining Concept Drift Samples for Security Applications


Limin Yang*, Wenbo Guo†, Qingying Hao*, Arridhana Ciptadi‡
Ali Ahmadzadeh‡, Xinyu Xing†, Gang Wang*
*University of Illinois at Urbana-Champaign †The Pennsylvania State University ‡Blue Hexagon
liminy2@illinois.edu, wzg13@ist.psu.edu, qhao2@illinois.edu, {arri, ali}@bluehexagon.ai, xxing@ist.psu.edu, gangw@illinois.edu

## Abstract

Concept drift poses a critical challenge to deploy machine learning models to solve practical security problems. Due to the dynamic behavior changes of attackers (and/or the benign counterparts), the testing data distribution is often shifting from the original training data over time, causing major failures to the deployed model.

To combat concept drift, we present a novel system CADE aiming to 1) *detect* drifting samples that deviate from existing classes, and 2) *provide explanations* to reason the detected drift. Unlike traditional approaches (that require a large number of new labels to determine concept drift statistically), we aim to identify individual drifting samples as they arrive. Recognizing the challenges introduced by the high-dimensional outlier space, we propose to map the data samples into a low-dimensional space and automatically learn a distance function to measure the dissimilarity between samples. Using contrastive learning, we can take full advantage of existing labels in the training dataset to learn how to compare and contrast pairs of samples. To reason the meaning of the detected drift, we develop a distance-based explanation method. We show that explaining "distance" is much more effective than traditional methods that focus on explaining a "decision boundary" in this problem context. We evaluate CADE with two case studies: Android malware classification and network intrusion detection. We further work with a security company to test CADE on its malware database. Our results show that CADE can effectively detect drifting samples and provide semantically meaningful explanations.

## 1 Introduction

Deploying machine learning based security applications can be very challenging due to *concept drift*. Whether it is malware classification, intrusion detection, or online abuse detection [6, 12, 17, 42, 48], learning-based models work under a "closed-world" assumption, expecting the testing data distribution to roughly match that of the training data. However, the environments in which the models are deployed are usually dynamically changing over time. Such changes may include both organic behavior changes of benign players and malicious mutations and adaptations of attackers. As a result, the testing data distribution is shifting from the original training data, which can cause serious failures to the models [23].

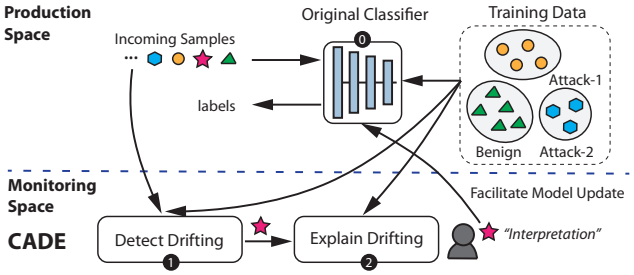



Figure 1: Drifting sample detection and explanation.

To address concept drift, most learning-based models require periodical re-training [36, 39, 52]. However, retraining often needs labeling a large number of new samples (expensive). More importantly, it is also difficult to determine *when* the model should be retrained. Delayed retraining can leave the outdated model vulnerable to new attacks.

We envision that combating concept drift requires establishing a monitoring system to examine the relationship between the incoming data streams and the training data (and/or the current classifier). The high-level idea is illustrated in Figure 1. While the original classifier is working in the *production space*, another system should periodically check how qualified the classifier is to make decisions on the incoming data samples. A detection module (❶) can filter drifting samples that are moving away from the training space. More importantly, to reason the causes of the drifting (*e.g.*, attacker mutation, organic behavior changes, previous unknown system bugs), we need an explanation method (❷) to link the detection decision to semantically meaningful features. These two capabilities are essential to preparing a learning-based security application for the open-world environment.

Prior works have explored the detection of drifting samples by directly checking the prediction confidence of the original classifier (❶) [32]. A low confidence score could indicate that the incoming sample is a drifting sample. However, this confidence score is a probability (sum up to 1.0) calculated based on the assumption that all the classes are known (closed-world). A drifting sample that does not belong to any existing classes might be assigned to a wrong class with high confidence (validated by existing works [25, 32, 37]). A more recent work presents the idea to compute a *non-conformity measure* between the incoming sample and each of the existing classes to determine fitness [38]. This non-conformity measure is calculated based on a distance function to quantify the dissimilarity between samples. However, we find that such distance functions could easily lose effectiveness, especially when the data is sparse with high dimensionality.

**Our Method.** In this paper, we present a new method for detecting drifting samples, coupled with a novel method to explain the detection decisions. Collectively, we build a system called CADE, which is short for "Contrastive Autoencoder for Drifting detection and Explanation." The key challenge is to derive an effective distance function to measure the dissimilarity of samples. Instead of arbitrarily picking the distance function, we leverage the idea of *contrastive learning* [29] to learn the distance function from existing training data, based on existing labels. Given the training data (multiple classes) of the original classifier, we map the training samples into a low-dimensional latent space. The map function is learned by contrasting samples to enlarge the distances between samples of different classes, while reducing the distance between samples in the same class. We show the resulting distance function in the latent space can effectively detect and rank drifting samples.

To explain a drifting sample, we identify a small set of important features that differentiate this sample from its nearest class. A key observation is that traditional (supervised) explanation methods do not work well [22, 28, 53, 62]. The insight is that supervised explanation methods require both classes (drifting samples and existing class) to have sufficient samples to estimate their distributions. However, this requirement is difficult to meet, given the drifting sample is located in a sparse space outside of training distribution. Instead, we find it is more effective to derive explanations based on *distance changes*, *i.e.*, features that cause the largest changes to the distance between the drifting sample and its nearest class.

**Evaluation.** We evaluate our methods with two datasets, including an Android malware dataset [7] and an intrusion detection dataset released in 2018 [57]. Our evaluation shows that our drifting detection method is highly accurate, with an average $F_1$ score of 0.96 or higher, which outperforms various baselines and existing methods. Our analysis also demonstrates the benefit of using contrastive learning to reduce the ambiguity of detection decisions. For the explanation model, we perform both quantitative and qualitative evaluations. Case studies also show that the selected features match the semantic behaviors of the drifting samples.

Furthermore, we worked with our collaborators in a security company to test CADE on their internal malware database. As an initial test, we obtained a sample of 20,613 Windows PE malware that appeared from August 2019 to February 2020 from 395 families. This allows us to test the system performance with more malware families and in a diverse setting. The results are promising. For example, CADE achieves an $F_1$ score of 0.95 when trained on 10 families and tested on 160 previously unseen families. This leads to the interest to further test and deploy CADE in a production system.

**Contributions.** This paper has three main contributions.

- We propose CADE to complement existing supervised learning based security applications to combat concept drift. We introduce an effective method to detect drifting samples based on contrastive representation learning.
- We illustrate the limitation of supervised explanation methods in explaining outlier samples and introduce a distance-based explanation method for this context.
- We extensively evaluate the proposed methods with two applications. Our initial tests with a security company show that CADE is effective. We have released the code of CADE here[1] to support future research.

## 2 Background and Problem Scope

In this section, we introduce the background for concept drift under the contexts of security applications, and discuss the limitations of some possible solutions.

**Concept Drift.** Supervised machine learning has been used in many security contexts to train detection models. Concept drift is a major challenge to these models when deployed in practice. Concept drift occurs as the testing data distribution deviates from the original training data, causing a shift in the true decision boundary [23]. This often leads to major errors in the original model over time.

To detect concept drift, researchers propose various techniques, which mostly involve the collection of new sets of data to statistically assess model behaviors [9, 10, 20, 31]. For some of these works, they also require the effort of data labeling. In security applications, knowing the existence of new attacks and collecting data about them are challenging in the first place. Besides, labeling data is time-consuming and requires substantial expertise. As such, it is impractical to assume that most incoming data can be sufficiently labeled.

Besides supervised models, semi-supervised anomaly detection systems are not necessarily immune to concept drift. For example, most network intrusion detection systems are

[1] https://github.com/whyisyoung/CADE

learned on "normal" traffic, and then used to detect incoming traffic that deviates from the learned "norm" as attacks [24, 34, 48]. For such systems, they might detect previously unknown attacks; however, concept drift, especially in benign traffic, could easily cause model failures. Essentially, intrusion detection is still a classification problem, *i.e.*, to distinguish normal traffic from abnormal traffic. Its training is performed only with one category of data. This, to some extent, weakens the learning outcome. The systems still rely on the assumption that the normal data has *covered all possible cases* – which is often violated in the testing phase [60].

**Our Problem Scope.** Instead of detecting concept drift with well-prepared and fully labeled data, we focus on a more practical scenario. As shown in Figure 1, we investigate individual samples to detect those that are shifted away from the original training data. This allows us to detect drifting samples and labels (a subset of) them as they arrive. Once we accumulate drifting samples sufficiently, we can assess the need for model re-training.

In a multi-class classification setting, there are two major types of concept drift. *Type A: the introduction of a new class:* drifting samples come from a new class that does not exist in the training dataset. As such, the originally trained classifier is not qualified to classify the drifting samples; *Type B: in-class evolution*: the drifting samples are still from the existing classes, but their behavior patterns are significantly different from those in the training dataset. In this case, the original classifier can easily make mistakes on these drifting samples.

In this paper, we primarily focus on Type A concept drift, *i.e.*, the introduction of a new class in a multi-class setting. Taking malware classification for example (Figure 1), our goal is to *detect* and *interpret* drifting samples from previously unseen malware families. Essentially, the drifting samples are out-of-distribution samples with respect to all of the existing classes in the training data. In Section 6, we explore adapting our solution to address Type B concept drift (in-class evolution) and examine the generalizability of our methods.

**Possible Solutions & Limitations.** We briefly discuss the possible directions to address this problem and the limitations.

The first direction is to use the *prediction probability* of the original classifier. More specifically, a supervised classifier typically outputs a prediction probability (or confidence) as a side product of the prediction label [32]. For example, in deep neural networks, a softmax function is often used to produce a prediction probability which indicates the likelihood that a given sample belongs to each of the existing classes (with a sum of 1). As such, a low prediction probability might indicate the incoming sample is different from the existing training data. However, we argue that prediction probability is unlikely to be effective in our problem context. The reason is this probability reflects the *relative fitness* to the existing classes (*e.g.*, the sample fits in class `A` better than class `B`). If the sample comes from an entirely new class (neither class `A` nor `B`), the prediction probability could be vastly misleading. Many previous studies [25, 32, 37] have demonstrated that a testing sample from a new class can lead to a misleading probability assignment (*e.g.*, associating a wrong class with a high probability). Fundamentally, the prediction probability still inherits the "closed-world assumption" of the classifier, and thus is not suitable to detect drifting samples.

Compared to prediction probability, a more promising direction is to assess a sample's fitness to a given class directly. The idea is, instead of assessing whether the sample fits in class `A` better than class `B`, we assess how well this sample fits in class `A` *compared to other training samples in class* `A`. For example, autoencoder [33] can be used to assess a sample's fitness to a given distribution based on a reconstruction error. However, as an unsupervised method, it is difficult for an autoencoder to learn an accurate representation of the training distribution when ignoring the labels (see Section 4). In a recent work, Jordaney et al. introduced a system called Transcend [38]. It defines a "non-conformity measure" as the fitness assessment. Transcend uses a credibility $p$-value to quantify how similar the testing sample $\boldsymbol{x}$ is to training samples that share the same class. $p$ is the proportion of samples in this class that are at least as dissimilar to other samples in the same class as $\boldsymbol{x}$. While this metric can pinpoint drifting samples, such a system is highly dependent on a good definition of "dissimilarity". As we will show in Section 4, an arbitrary dissimilarity measure (especially when data dimensionality is high) can lead to bad performance.

## 3 Designing CADE

We propose a system called `CADE` for drift sample detection and explanation. We start by describing the intuitions and insights behind our designs, followed by the technical details for each component.

### 3.1 Insights Behind Our Design

As shown in Figure 1, our system has two components to (❶) detect drifting samples that are out of the training distribution; and (❷) explain the drifting samples to help analysts understand the meaning of the drift. Through initial analysis, we find both tasks face a common challenge: the drifting samples are located in a sparse outlier space, which makes it difficult to derive meaningful distance functions needed for both tasks.

First, detecting drifting samples requires learning a good distance function to measure how "drifting samples" are different from existing distributions. However, the outlier space is unboundedly large and sparse. For high-dimensional data, the notion of distance starts to lose effectiveness due to the "curse of dimensionality" [74]. Second, the goal of explanation is to identify a small subset of important features that most effectively differentiate the drifting sample from the

training data. As such, we also need an effective distance function to measure the differences.

In the following, we design a drifting detection module and an explanation module to jointly address these challenges. At the high-level, we first use *contrastive learning* to learn a compressed representation of the training data. A key benefit of contrastive learning is that it can take advantage of existing labels to achieve much-improved performance compared to unsupervised methods such as autoencoders [33] and Principal Component Analysis (PCA) [2]. This allows us to learn a distance function from the training data to detect drifting samples (Section 3.2). For the explanation module, we will describe a distance-based explanation formulation to address the aforementioned challenges (Section 3.3).

## 3.2 Drifting Sample Detection

The drifting detection model monitors the incoming data samples to detect incoming samples that are out of the distribution of the training data.

**Contrastive Learning for Latent Representations.** We explore the idea of contrastive learning to learn a good representation of the training data. Contrastive learning takes advantage of the *existing labels* in the training data to learn an effective distance function to measure the similarity (or contrast) of different samples [16]. Unlike supervised classifier, the goal of contrastive learning is not classifying samples to known classes. It is learning how to compare two samples.

As shown in Figure 2, given the input samples (high dimensional feature vectors), the contrastive learning model aims to map them into a low-dimensional latent space. The model is optimized such that, in the latent space, pairs of samples in the same class have a smaller distance, and pairs of samples from different classes have a larger distance. As such, the distance metric in the latent space can reflect the differences in pairs of samples. Any new samples that exhibit a large distance to all existing classes are candidate drifting samples.

To implement this idea, we use an autoencoder augmented with contrastive loss. Autoencoder is a useful tool to learn a compressed representation (with a reduced dimensionality) of a given input distribution [33]. Formally, let $\boldsymbol{x} \in \mathbb{R}^{q\times 1}$ be a sample from the given training set. We train an autoencoder that contains an encoder $f$ and a decoder $h$. Note that $f$ is parameterized by $\boldsymbol{\theta}$; $h$ is parameterized by $\boldsymbol{\phi}$. We construct the loss function as the following:

$$\min_{\boldsymbol{\theta},\boldsymbol{\phi}} \mathbb{E}_{\boldsymbol{x}} \|\boldsymbol{x}-\hat{\boldsymbol{x}}\|_2^2 + \lambda \mathbb{E}_{\boldsymbol{x}_i,\boldsymbol{x}_j}\left[(1-y_{ij})d_{ij}^2 + y_{ij}(m-d_{ij})_+^2\right]. \quad (1)$$

Here, the first term is the reconstruction loss of the autoencoder. More specifically, the goal of the encoder $f$ is to learn a good representation of the original input. Given an input $\boldsymbol{x}$, encoder $f$ maps the original input $\boldsymbol{x}$ to a lower-dimensional representation $\boldsymbol{z} = f(\boldsymbol{x};\boldsymbol{\theta})$. Autoencoder ensures this latent representation $\boldsymbol{z}$ can be decoded to reconstruct the original input with minimal reconstruction loss. Here, $\hat{\boldsymbol{x}} \in \mathbb{R}^{q\times 1}$ is the reconstruction of this original input, *i.e.*, $\hat{\boldsymbol{x}} = h(\boldsymbol{z})$. This loss term represents the mean squared error between $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$.

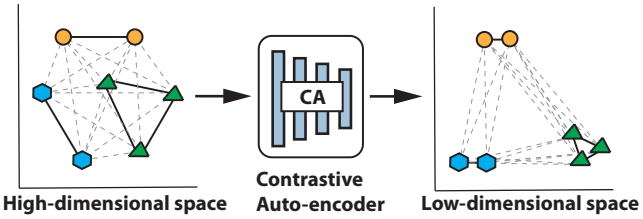


Figure 2: The high-level idea of contrastive learning.

The second term of Eqn. (1) refers to the contrastive loss, which takes a pair of samples $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and their relationship $y_{ij}$ as input. $y_{ij} = 1$, if the two samples are from the different classes; $y_{ij} = 0$, if the two samples are from the same class. $(\cdot)_+$ is a short notation for $\max(0,\cdot)$, and $d_{ij}$ is the Euclidean distance between the latent space representations $\boldsymbol{z}_i = f(\boldsymbol{x}_i;\theta)$ and $\boldsymbol{z}_j = f(\boldsymbol{x}_j;\theta)$, where $\boldsymbol{z} \in \mathbb{R}^{d\times 1}$ $(d \ll p)$. This loss term minimizes the distance of $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ in the latent space if they are from the same class, and maximizes their distance up to a radius defined by $m > 0$, such that the dissimilar pairs contribute to the loss function only when their distance is within this radius. $\lambda$ is a hyper-parameter controlling the weight of the second term in the loss function.

After contrastive learning, encoder $f$ can map the input samples to a low-dimensional latent space where each class forms *tight groups* (as shown in Figure 2). In this latent space, the distance function can effectively identify new samples drifting away from these groups.

**MAD-based Drifting Sample Detection.** After training the contrastive autoencoder, we can use it to detect drifting samples. Given a set of $K$ testing samples $\{\boldsymbol{x}_t^{(k)}\}$ $(k = 1,\ldots,K)$, we seek to determine whether each sample $\boldsymbol{x}_t^{(k)}$ is a drifting sample with respect to existing classes in the training data. The detection method is shown in Algorithm 1.

Suppose the training set has $N$ classes, and each class has $n_i$ training samples, for $i = 1,2,...,N$. We first use the encoder to map all the training samples into the latent space (line 2–4). For each class $i$, we calculate its centroid $\boldsymbol{c}_i$ (by taking the mean value for each dimension in a Euclidean space in line 5). Given a testing sample $\boldsymbol{x}_t^{(k)}$, we also use the encoder to map it to the latent space representation $\boldsymbol{z}_t^{(k)}$ (line 14). Then, we calculate the Euclidean distance between the testing sample and each of the centroids: $d_i^{(k)} = \|\boldsymbol{z}_t^{(k)} - \boldsymbol{c}_i\|_2$ (line 16). Based on its distance to centroids, we determine if this testing sample is out of distribution for each of the $N$ classes. Here, we make decisions based on the sample's distance to the centroids instead of the sample's distance to the nearest training samples. This is because the latter option can be easily affected by the outliers in the training data.

**Algorithm 1** Drift Detection with Contrastive Autoencoder.

**Input:** Training data $\boldsymbol{x}_i^{(j)}, i = 1,\ldots,N, j = 1,\ldots,n_i$, $N$ is the number of classes, $n_i$ is the number of training samples in class $i$; testing data $\boldsymbol{x}_t^{(k)}$, $t$ refers to the testing set, $k = 1, \ldots, K$, $K$ is the total number of testing samples; encoder $f$; a constant $b$.

**Output:** Drifting score for each testing sample $A^{(k)}$, the closest class $y_t^{(k)}$, centroid of each class $\boldsymbol{c}_i$, $\text{MAD}_i$ to each class.

1: **for** class $i = 1$ to $N$ **do**
2: **for** $j = 1$ to $n_i$ **do**
3: $\boldsymbol{z}_i^{(j)} = f(\boldsymbol{x}_i^{(j)}; \boldsymbol{\theta})$ ▷ The latent representation of $\boldsymbol{x}_i^{(j)}$.
4: **end for**
5: $\boldsymbol{c}_i = \frac{1}{n_i}\sum_{j=1}^{n_i} \boldsymbol{z}_i^{(j)}$ ▷ The centroid of class $i$.
6: **for** $j = 1$ to $n_i$ **do**
7: $d_i^{(j)} = ||\boldsymbol{z}_i^{(j)} - \boldsymbol{c}_i||_2$ ▷ The distance between sample and centroid.
8: **end for**
9: $\tilde{d}_i = \text{median}(d_i^{(j)}), j = 1,\ldots,n_i$
10: $\text{MAD}_i = b * \text{median}(|d_i^{(j)} - \tilde{d}_i|), j = 1,\ldots,n_i$
11: **end for**
12:
13: **for** $k = 1$ to $K$ **do**
14: $\boldsymbol{z}_t^{(k)} = f(\boldsymbol{x}_t^{(k)}; \boldsymbol{\theta})$
15: **for** class $i = 1$ to $N$ **do**
16: $d_i^{(k)} = ||\boldsymbol{z}_t^{(k)} - \boldsymbol{c}_i||_2$
17: $A_i^{(k)} = \frac{|d_i^{(k)} - \tilde{d}_i|}{\text{MAD}_i}$
18: **end for**
19: $A^{(k)} = \min(A_i^{(k)}), i = 1,\ldots,N$
20: **if** $A^{(k)} > T_{\text{MAD}}$ **then** ▷ $T_{\text{MAD}}$ is set to 3.5 empirically [40].
21: $\boldsymbol{x}_t^{(k)}$ is a potential drifting sample.
22: **else**
23: $\boldsymbol{x}_t^{(k)}$ is a non-drifting sample.
24: **end if**
25:
26: $y_t^{(k)} = \underset{i}{\text{argmin}}\, d_i^{(k)}, i = 1,\ldots,N$ ▷ The closest class for $\boldsymbol{x}_t^{(k)}$.
27: **end for**

To determine outliers based on $d_i^{(k)}$, the challenge is that different classes might have different levels of tightness, and thus require different distance thresholds. Instead of manually setting the absolute distance threshold for each class, we use a method called Median Absolute Deviation (MAD) [40]. The idea is to estimate the data distribution within each class $i$ by calculating $\text{MAD}_i$ (line 6–10), which is the median of the absolute deviation from the median of distance $d_i^{(j)}$ $(j = 1,\ldots,n_i)$. Here $d_i^{(j)}$ depicts the latent distance between each sample in class $i$ to its centroid, and $n_i$ is the number of samples in class $i$ (line 7). Then based on $\text{MAD}_i$, we can determine if $d_i^{(k)}$ is large enough to make the testing sample $\boldsymbol{x}_t^{(k)}$ an outlier of class $i$ (line 15–24). If the testing sample is an outlier for *all of the N classes*, then it is determined as a drifting sample. Otherwise, we determine it is an in-distribution sample and its closest class is determined by the closest centroid (line 26). The advantage of MAD is that every class has its own distance threshold to determine outliers based on its in-class distribution. For instance, if a cluster is more spread out, the threshold would be larger.

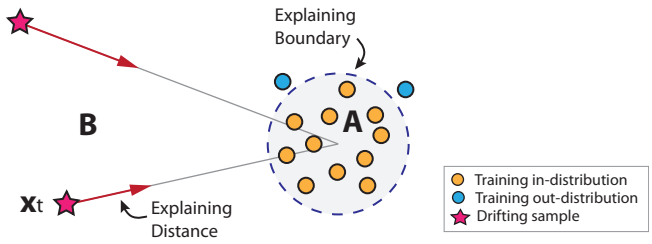



Figure 3: Illustration of the boundary-based explanation and the distance-based explanation in our setup.

Note that MAD might suffer when a class does not have enough samples as its median can be noisy. In our design, contrastive learning can help to mitigate this issue since each of the classes is mapped to a compact region in the latent space which helps to stabilize the median.

**Ranking Drifting Samples.** As shown in Figure 1, drifting samples might need further investigations by analysts to interpret the meaning of the drifting. Given the limited time of analysts, it is important to rank the drifting samples so that analysts can focus on the most novel variants. We use a simple approach to rank drifting samples based on their distance to the nearest centroid (calculated in line 26). This allows us to prioritize the investigation of drifting samples that are furthest away from their nearest centroid.

## 3.3 Explaining Drifting Samples

The explanation module aims to identify the most important features that drive a testing sample away from existing classes. To be specific, given a drifting sample $\boldsymbol{x}_t$, and its nearest class $y_t$ in the training set, we want to identify a small set of features that make $\boldsymbol{x}_t$ an outlier of class $y_t$. To achieve this goal, one instinctive reaction is to convert it to the problem of *explaining a supervised learning model*, which is a well-studied area. For example, we can approximate our drifting detector (❶) as a classifier, and derive explanations using existing explaining methods developed for classifiers [28, 35, 53, 58, 62]. However, due to the high sparsity of the outlier space, we find it difficult to move a drifting sample to cross the decision boundary, and thus fail to derive meaningful explanations. Motivated by this, we design a new explanation method customized for drift detection, which explains the *distance* between a drifting sample and the in-class samples rather than the *decision boundary*. Below, we first analyze the "straightforward approach" and then describe our method.

**Baseline Method: Boundary-based Explanation.** Given the rich literature on explaining supervised classifiers, a straightforward approach is to convert the drifting detection module into a supervised learning model, and then run existing explanation algorithms. Supervised explanation methods are to explain the decision boundary between two classes (*e.g.*, classes A and B). The goal is to identify a minimal set of features within $\boldsymbol{x}_t$, such that perturbing these features will let

$\boldsymbol{x}_t$ cross the decision boundary. As is shown in Figure 3, class A represents the in-distribution training samples from $y_t$, and class B represents the detected drifting sample in the testing set. The decision boundary is illustrated by the blue dashed line (the decision boundary is shown in the form of a norm ball since it is based on distance threshold). Given a drifting sample $\boldsymbol{x}_t$ (denoted by a star in Figure 3), the explanation method pulls the sample into the in-distribution class (*i.e.* the region with gray canvas) by perturbing a small set of important features.[2] We implemented this idea using existing perturbation-based supervised explanation methods [13, 18, 21, 22] (implementation details in Appendix A).

The evaluation result later in Section 5 shows that this approach is fundamentally limited. We believe the reasons are two-fold. First, given the limited number of drifting samples, it is difficult to derive an accurate approximation model for the decision boundary. Second and more importantly, the outlier space is much bigger than the in-distribution region. Given the drifting samples are far away from the decision boundary, it is difficult to find a small set of feature perturbations to take the drifting sample to cross the decision boundary and enter the in-distribution region. Without the ability to cross the boundary, the explanation methods do not have the necessary gradients (or feedback) to compute feature importance.

**Our Method: Distance-based Explanation.** Motivated by this observation, we propose a new approach that identifies important features by explaining the *distance* (*i.e.* the red arrow in Figure 3). Unlike supervised classifiers that make decisions based on the decision boundary, the drift detection model makes decisions based on the sample's distance to centroids. As such, we aim to find a set of original features that help to move the drifting sample $\boldsymbol{x}_t$ toward the nearest centroid $\boldsymbol{c}_{y_t}$. With this design, we no longer need to force $\boldsymbol{x}_t$ to cross the boundary, which is hard to achieve. Instead, we perturb the original features and observe the *distance changes* in the latent space.

To realize this idea, we need to first design a feature perturbation mechanism. Most existing perturbation methods are designed exclusively for images [18], the features of which are numerical values. In our case, features in $\boldsymbol{x}_t$ can be either numerical or categorical, and thus directly applying existing methods will produce ill-defined feature values. To ensure the perturbations are meaningful for both numerical and categorical features, we propose to perturb $\boldsymbol{x}_t$ by replacing its feature value with the value of the corresponding feature in a reference training sample $\boldsymbol{x}_{y_t}^{(c)}$. This $\boldsymbol{x}_{y_t}^{(c)}$ is the training sample that has the shortest latent distance to the centroid $\boldsymbol{c}_{y_t}$. As such, our explanation goal is to identify a set of features, such that substituting them with those in $\boldsymbol{x}_{y_t}^{(c)}$ will impose the highest influence upon the distance between $f(\boldsymbol{x}_t)$ and $\boldsymbol{c}_{y_t}$. Replacing the feature values with those of $\boldsymbol{x}_{y_t}^{(c)}$ also helps to ensure the perturbed sample is moving towards the rough direction of the centroid. As before, the perturbation is done in the original feature space where features have semantic meanings.

We use an $\boldsymbol{m} \in \mathbb{R}^{q\times 1}$ to represent the important features, in which $\boldsymbol{m}_i = 1$ means $(\boldsymbol{x}_t)_i$ is replaced by the value of $(\boldsymbol{x}_{y_t}^{(c)})_i$ and $\boldsymbol{m}_i = 0$ means we keep the value of $(\boldsymbol{x}_t)_i$ unchanged. In other words, $\boldsymbol{m}_i = 1$ indicates the $i^{\text{th}}$ feature is selected as the important one. Each element in this feature mask $\boldsymbol{m}_i$ can be sampled from a Bernoulli distribution with probability $p_i$. As such, we could guarantee that $\boldsymbol{m}_i$ equals to either 1 and 0. Then, our goal is transformed into solving the $p_i$ for $i = 1, 2, ..., q$. Technically, this can be achieved by minimizing the following objective function with respect to $p_{1:q}$.

$$\begin{aligned} &\mathbb{E}_{\boldsymbol{m}\sim Q(\boldsymbol{p})} \|\hat{\boldsymbol{z}}_t - \boldsymbol{c}_{y_t}\|_2 + \lambda_1 R(\boldsymbol{m}, \boldsymbol{b}), \\ &\hat{\boldsymbol{z}}_t = f(\boldsymbol{x}_t \odot (1 - \boldsymbol{m} \odot \boldsymbol{b}) + \boldsymbol{x}_{y_t}^{(c)} \odot (\boldsymbol{m} \odot \boldsymbol{b})), \\ &R(\boldsymbol{m}, \boldsymbol{b}) = \|\boldsymbol{m} \odot \boldsymbol{b}\|_1 + \|\boldsymbol{m} \odot \boldsymbol{b}\|_2, \quad Q(\boldsymbol{p}) = \prod_{i=1}^{q} p(\boldsymbol{m}_i | p_i). \end{aligned} \tag{2}$$

Note that $\odot$ denotes the element-wise multiplication; $\hat{\boldsymbol{z}}_t$ represents the latent vector of the perturbed sample. Given the equation above, directly computing $\boldsymbol{m}$ is difficult due to its high dimensionality. To speed up the search, we introduce a filter $\boldsymbol{b}$ to pre-filter out features that are not worth considering. We set $(\boldsymbol{b})_i = 0$, if $(\boldsymbol{x}_t)_i$ and $(\boldsymbol{x}_{y_t}^{(c)})_i$ are the same. In other words, if a feature value of $\boldsymbol{x}_t$ is already the same as that of the reference sample $\boldsymbol{x}_{y_t}^{(c)}$, then this feature is ruled out in the optimization (since it has no impact on distance change). In this way, $\hat{\boldsymbol{z}}_t = f(\boldsymbol{x}_t \odot (1 - \boldsymbol{m} \odot \boldsymbol{b}) + \boldsymbol{x}_{y_t}^{(c)} \odot (\boldsymbol{m} \odot \boldsymbol{b}))$ represents the latent vector of the perturbed sample.

In Eqn. (2), the first term in the loss function aims to minimize the *latent-space distance* between the perturbed sample $\hat{\boldsymbol{z}}_t$ and the centroid $\boldsymbol{c}_{y_t}$ of the $y_t$ class. Each element in $\boldsymbol{m}$ is sampled from a Bernoulli distribution parameterized by $p_i$. Here, we use $Q(\boldsymbol{p})$ to represent their joint distribution.[3] For the second term, $\lambda$ is a hyper-parameter that controls the strength of the elastic-net regularization $R(\cdot)$, which restricts the number of non-zero elements in $\boldsymbol{m}$. By minimizing $R(\boldsymbol{m}, \boldsymbol{b})$, the optimization procedure selects a minimum subset of important features.

Note that Bernoulli distribution is discrete, which means the gradient of $\boldsymbol{m}_i$ with respect to $p_i$ (*i.e.* $\frac{\partial \boldsymbol{m}_i}{\partial p_i}$) is not well defined. We cannot solve the optimization problem in Eqn. 2 by using a gradient-based optimization method. To tackle this challenge, we apply the change-of-variable trick introduced in [45]. We enable the gradient computation by replacing the Bernoulli distribution with its continuous approximation (*i.e.* concrete distribution) parameterized by $p_i$. Then we can solve the parameters $p_{1:q}$ through a gradient-based optimization method (we use Adam optimizer in this paper).

[2] Note that we do not perform feature perturbation in the latent space, because the latent features do not carry semantic meanings. Instead, we select features in the original input space.

[3] We assume each feature is independently drawn from a distinct Bernoulli distribution.

| Id | Family | # of Samples |
|---|---|---|
| 0 | FakeInstaller | 925 |
| 1 | DroidKungFu | 667 |
| 2 | Plankton | 625 |
| 3 | GingerMaster | 339 |
| 4 | BaseBridge | 330 |
| 5 | Iconosys | 152 |
| 6 | Kmin | 147 |
| 7 | FakeDoc | 132 |
| Total: | | 3,317 |

Table 1: Android malware samples from the Drebin dataset.

## 4 Evaluation: Drifting Detection

In this section, we evaluate our system using two security applications: Android malware family attribution, and network intrusion detection. In this current section (Section 4), we focus on the evaluation of the drifting detection module. We will evaluate the explanation module in Section 5. After these controlled experiments, we tested our system with a security company on their malware database (Section 7).

### 4.1 Experimental Setup and Datasets

**Android Malware Attribution.** We use the Drebin dataset [7] to explore the malware family attribution problem. The original classifier (module ❼ in Figure 1) is a multilayer perceptron (MLP) classifier. It identifies which family a malware sample belongs to. The Drebin dataset contains 5,560 Android malware samples. For this evaluation, we select 8 families [4] where each family has at least 100 malware samples (3,317 samples in total) as shown in Table 1.

To evaluate the drifting sample detection module, for each experiment, we pick one of the 8 families as the *previously unseen family*. For example, suppose we pick FakeDoc (family 7) as the previous unseen family. We split the other seven families into training and testing sets, and add FakeDoc *only to the testing set*. In this way, FakeDoc is not available during training. Our goal is to correctly identify samples from FakeDoc as drifting samples in the testing time.

We split the training-testing sets with a ratio of 80:20. The split is based on the timestamp (malware creation time), which is recommended by several works [52, 65] to simulate a realistic setting. Time-based split also means we cannot use any new features that only appear in the testing set for model training. This leaves us with 7,218 features. We then use scikit-learn's VarianceThreshold function [51] to remove features with very low variance (*i.e.*, $<0.003$), which creates a final set of 1,340 features.

[4] Two families FakeInstaller and Opfake are very similar in terms of their nature of attacks. There is strong disagreement among AV-engines regarding their family labels, *i.e.*, the samples are labeled as one family by some engines but are labeled as the other family by other engines. As such, we only included FakeInstaller (Table 1).

| Id | Family | # of Flows |
|---|---|---|
| 0 | Benign | 66,245 |
| 1 | SSH-Bruteforce | 11,732 |
| 2 | DoS-Hulk | 43,487 |
| 3 | Infiltration | 9,238 |
| Total: | | 130,702 |

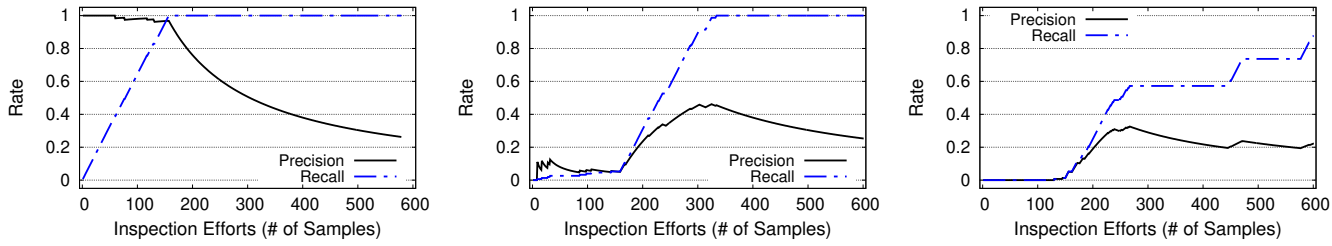
Table 2: Network intrusion dataset: 3 network intrusion classes and 1 benign class from the IDS2018 dataset.

To demonstrate the generalizability of results, we iteratively select each of the malware families to be the "unseen family" and repeat the experiments.

**Network Intrusion Detection.** We use a network intrusion dataset [57], which we refer to as IDS2018. The dataset contains different types of network traces generated by known attacks. For our evaluation, we select the benign class (one day's traffic) and 3 different attack classes: SSH-Bruteforce, Dos-Hulk, and Infiltration. SSH-Bruteforce is a brute-force attack to guess the SSH login password. DoS-Hulk attack aims to flood the targeted machine with superfluous requests in an attempt to make the machine temporally unavailable. Infiltration attack first sends an email with a malicious attachment to exploit an on-host application's vulnerability, and then leverages the backdoor to run port-scan to discover more vulnerabilities. We refer interested readers to [57] for more details about the attacks. To speed up the experiments and test different setups, we use 10% of their traffic for the experimental dataset (Table 2). In Appendix D, we show that more traffic only increases the computational overhead and has a negligible influence on the performance of the selected methods.

We iteratively pick one of the attack families as the previously unseen family and only include this family in the testing set. We repeat the experiments to report the average performance. We split the train-test sets with a ratio of 80:20. Note that features in the IDS2018 dataset need to be further normalized and encoded. To be realistic, we only use the training data to build the feature encoding scheme. At the high-level, each sample represents a network flow. Categorical features such as "destination port" and "network protocol" are encoded with one-hot encoding. The other 77 statistical features are normalized between 0 and 1 with a MinMaxScaler. Each network flow has 83 features. The detailed feature engineering steps are available in the documentation of our released code.

**Evaluation Metric.** For the drifting detection module (module ❶ in Figure 1), the positive samples are samples in the unseen family in the testing set. The negative samples are the rest of the testing samples from the known families. Given a ranked list of detected samples, we simulate an analyst inspecting samples from the top of the list. As we go down the list, we calculate three evaluation metrics: precision, recall, and $F_1$ score. *Precision* measures the ratio of true




(a) CADE (b) Transcend (c) Vanilla AE

Figure 4: Precision and recall vs. number of inspected samples (detected drifting samples are ranked by the respective method).

| Method | Drebin (Avg±Std) | | | | IDS2018 (Avg±Std) | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | $F_1$ | Norm. Effort | Precision | Recall | $F_1$ | Norm. Effort |
| Vanilla AE | $0.63 \pm 0.17$ | $0.88 \pm 0.13$ | $0.72 \pm 0.15$ | $1.48 \pm 0.31$ | $0.61 \pm 0.16$ | $0.99 \pm 0.00$ | $0.74 \pm 0.12$ | $1.74 \pm 0.40$ |
| Transcend | $0.76 \pm 0.19$ | $0.90 \pm 0.14$ | $0.80 \pm 0.12$ | $1.29 \pm 0.45$ | $0.64 \pm 0.45$ | $0.67 \pm 0.47$ | $0.65 \pm 0.46$ | $1.45 \pm 0.57$ |
| CADE | $0.96 \pm 0.05$ | $0.96 \pm 0.04$ | $0.96 \pm 0.03$ | $1.00 \pm 0.09$ | $0.98 \pm 0.02$ | $0.93 \pm 0.09$ | $0.96 \pm 0.06$ | $0.95 \pm 0.07$ |

Table 3: Drifting detection results for Drebin and IDS2018 datasets. We compare CADE with two baselines Transcend [38] and Vanilla AE. For each evaluation metric, we report the mean value and the standard deviation across all the settings.

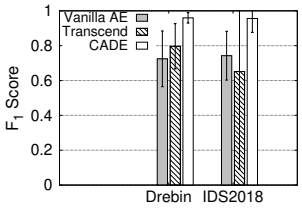



Figure 5: $F_1$ scores of drifting detection.

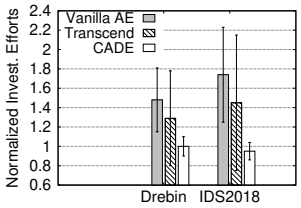



Figure 6: Normalized investigation efforts.

unseen-family samples out of the inspected samples. *Recall* measures the ratio of unseen-family samples that are successfully discovered by the detection module out of all the unseen-family samples. $F_1$ *score* is the harmonic mean of precision and recall: $F_1 = 2 \times \frac{precision \times recall}{precision + recall}$. Finally, to quantify the efforts of inspection, we define a metric called *inspecting effort*, which is the total number of inspected samples, normalized by the number of true unseen family samples in the testing set.

**Baseline Methods.** We include two main baselines. The *first* baseline is a standard Vanilla autoencoder [33], which is used to illustrate the benefit of contrastive learning. We set the Vanilla autoencoder (AE) to have the same number of layers and output dimensionality as CADE. We use it to perform dimension reduction to map the inputs into a latent space where we use the same MAD method to detect and rank drifting samples. The difference between this baseline and CADE is that the baseline does not perform contrastive learning. The hyperparameter setting is in Appendix B.

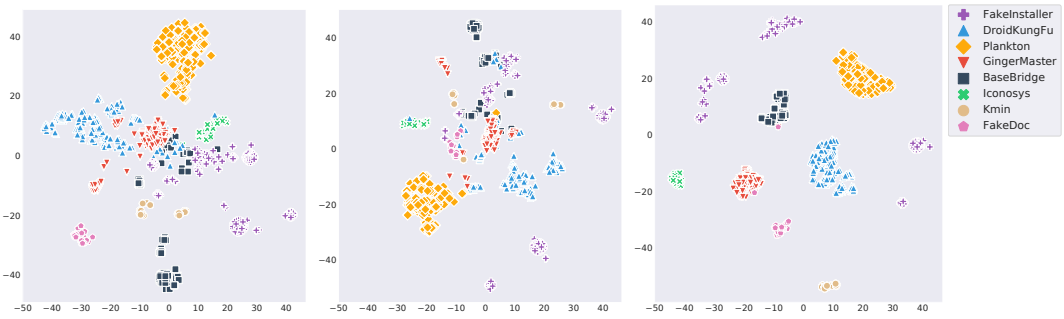
The *second* baseline is Transcend [38]. As described in Section 2, Transcend defines a "non-conformity measure" to quantify how well the incoming sample fits into the predicted class, and calculate a credibility $p$-value to determine if the incoming sample is a drifting sample. We obtain the source code of Transcend from the authors and follow the paper to adapt the implementation to support multi-class classification (the original code only supports binary classification). Specifically, we initialize the non-conformity measure with $-p$ where $p$ is the softmax output probability indicating the likelihood that a testing sample belongs to a given family. Then we calculate the credibility $p$-value for a testing sample. If the $p$-value is near zero for all existing families, we consider it as a drifting sample. We rank drifting samples based on the maximum credibility $p$-value. Note that we did not use other OOD detection methods [14, 41, 49] as our baseline mainly because they work in a different setup compared with CADE and Transcend. More specifically, these methods require *an auxiliary OOD dataset* in the training process and/or *modifying the original classifier*. These requirements are difficult to meet for malware classifiers in a production environment (detailed discussions are in Section 9).

## 4.2 Evaluation Results

In the following, we first compare the drifting detection performance of CADE with baselines and evaluate the impact of contrastive learning. Then, we perform case studies to investigate the potential reasons for detection errors.

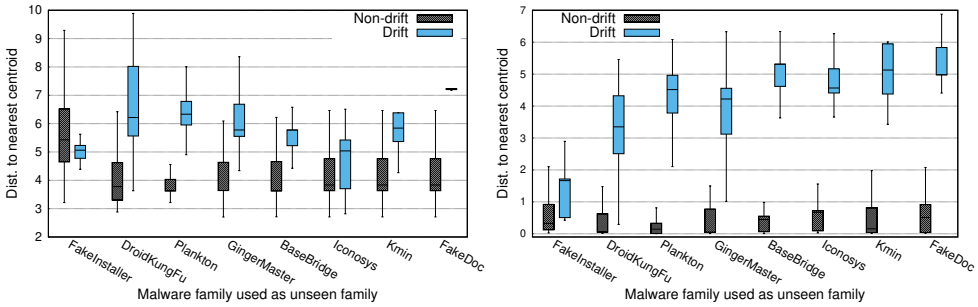
**Drifting Sample Detection Performance.** We first use one experiment setting to explain our evaluation process. Take the Drebin dataset for example. Suppose we use family Iconosys as the previously unseen family in the testing set. After training the detection model (without any samples of Iconosys), we use it to detect and rank the drifting samples. To evaluate the quality of the ranked list, we simulate an analyst inspecting samples from the top of the list.

Figure 4a shows that, as we inspect more drifting samples (up to 150 samples), the precision maintains at a high level (over 0.97) while the recall gradually reaches 100%. Combin-




(a) Original Space (b) Latent space (Vanilla AE) (c) Latent space (CADE)

Figure 7: T-SNE visualization for the original space, and latent spaces of Vanilla AE and CADE (unseen family: FakeDoc).




(a) Original space (b) Latent space (CADE)

Figure 8: Boxplots of the distances between testing samples and their nearest centroids in both the *original space* and the *latent space* for the Drebin dataset. Samples from previously unseen family are regarded as drifting samples.

ing precision and recall, the highest $F_1$ score is 0.98. After 150 samples, the precision will drop since there are no more unseen family samples in the remaining set. This confirms the high-quality of the ranked list, meaning almost all the samples from the unseen family are ranked at the top.

As a comparison, the ranked lists of Transcend and Vanilla AE are not as satisfying. For Transcend (Figure 4b), the first 150 samples return low precision and recall, indicating the top-ranked samples are not from the unseen family. After inspecting 150 samples, we begin to see more samples from the unseen family. After inspecting 350 samples, Transcend has covered most of the samples from the unseen family (*i.e.*, with a recall near 1.0) but the precision is only 0.46. This means more than half of the inspected samples by the analysts are irrelevant. The best $F_1$ score is 0.63. As shown in Figure 4c, the performance of Vanilla AE is worse. The recall is only slightly above 0.8, even after inspecting 600 samples.

To generalize the observation, we iteratively take each family as the unseen family and compute the average statistics across different settings for $F_1$ score (in Figure 5) and normalized inspecting efforts (in Figure 6). Table 3 further presents the corresponding precision and recall. For each experiment setting, we report the *highest* $F_1$ score for each model. This $F_1$ score is achieved as the analysts go down the ranked list and stop the inspection when they start to get a lot of false positives. The "inspecting effort" refers to the total number of inspected samples to reach the reported $F_1$ score, *normalized* by the number of true drifting samples in the testing set.

Table 3 confirms that CADE can detect drifting samples accurately and outperforms both baselines. On Drebin, the average $F_1$ score of CADE is 0.96, while the $F_1$ scores for baselines are 0.80 and 0.72. A similar conclusion can be drawn for the IDS2018 dataset. In addition, the standard deviation of CADE is much smaller than that of baselines, indicating a more consistent performance across different experiment settings. Finally, we show that CADE has lower normalized inspecting efforts, which confirms the high quality of the ranking.

Note that the Transcend baseline actually performs well in certain cases. For example, its $F_1$ score is 99.69% (similar to our system) when DoS-Hulk is set as the unseen family in the IDS2018 dataset. However, the issue is Transcend's performance is not stable in different settings, which is reflected in the high standard deviations in Table 3.

**Impact of Contrastive Learning.** To understand the source of the performance gain, we examine the impact of contrastive learning. First, we present a visualization in Figure 7 which shows the t-SNE plot of the training samples of the Drebin dataset and the testing samples from the chosen unseen family (FakeDoc). T-SNE [66] performs its own

non-linear dimensionality reduction to project data samples into a 2-d plot. To visualize our data samples, we map the samples from the original space (1,340 dimensions) to a 2-d space (Figure 7a). We also map the samples from the latent space (7 dimensions) to the 2-d space as a comparison (Figure 7b and Figure 7c). We can observe that samples in CADE's latent space have formed tighter clusters, making it easier to distance existing samples from the unseen family.

To provide a statistical view of different experiment settings, we plot Figure 8. Like before, we iteratively take one family as the unseen family in Drebin. Then we measure the distance of the testing samples to their nearest centroid in the original feature space (Figure 8a) and the latent space produced by CADE (Figure 8b). The results for the IDS2018 dataset have the same conclusion, and thus are omitted for brevity. We show that drifting samples and non-drifting samples are more difficult to separate in the original space. After contrastive learning, the separation is more distinctive in the latent space. The reason is that contrastive learning has learned a suitable distance function that can stretch the samples from different classes further apart, making it easier to detect unseen family.

**Case Study: Limits of CADE.** CADE performs well in most of the settings. However, we find that in certain cases, CADE's performance suffers. For example, when using FakeInstaller as the unseen family, our detection precision is only 82% when the recall gets to 100%. We notice that many testing samples from GingerMaster and Plankton families were detected as drifting samples. After a closer inspection, we find that, when FakeInstaller is treated as the unseen family, in order to maintain the overall 80:20 training-testing ratio, we need to split the dataset at the time when there were not enough training samples from GingerMaster and Plankton yet. Therefore, many of the testing samples from GingerMaster and Plankton families look very different from the small number of training samples in the two families (based on the latent distance). External evidence also suggests that the two families had many variants [5, 70]. While these malware variants are not from a new family (false positives under our definition), they could also have values for an investigation to understand malware mutation within the same family.

# 5 Evaluation: Explaining Drifting Samples

To evaluate the explanation module, we randomly select one family from each dataset (*i.e.* FakeDoc for Drebin and Infiltration for IDS2018) as drifting samples. Results from other settings have the same conclusion and thus are omitted for brevity. Given this setting, we generate explanations for the detected drifting samples and evaluate the explanation results, both quantitatively and qualitatively.

| Method | Drebin-FakeDoc Avg $\pm$ Std | IDS2018-Infiltration Avg $\pm$ Std |
|---|---|---|
| Original distance | 5.363 $\pm$ 0.568 | 11.715 $\pm$ 2.321 |
| Random | 5.422 $\pm$ 1.773 | 11.546 $\pm$ 3.169 |
| Boundary-based | 3.960 $\pm$ 2.963 | 6.184 $\pm$ 3.359 |
| COIN [43] | 6.219 $\pm$ 3.962 | 8.921 $\pm$ 2.234 |
| CADE | **0.065** $\pm$ **0.035** | **2.349** $\pm$ **3.238** |

Table 4: Comparison of explanation fidelity based on the average distance between the *perturbed* sample and the nearest centroid. A shorter distance is better. "Original distance" is the distance between the drift sample and nearest centroid.

## 5.1 Experimental Setup

**Baseline Method.** We consider three baseline methods: (1) a *random* baseline that randomly selects features as important features; (2) the boundary-based explanation method described in Section 3, and (3) an unsupervised explanation method called COIN [43]. Due to space limit, we only briefly describe how COIN works. COIN builds a set of local LinearSVM classifiers to separate an individual outlier from its in-distribution neighborhood samples. Since the LinearSVM classifiers are self-explainable, they can pinpoint important features that contribute to the outlier classification. For a fair comparison, we select the same number of top features for baselines as our method. The implementation and hyperparameters of these baselines can be found in Appendix B. Note that we did not select existing black-box explanation methods (*e.g.*, LIME [53] and SHAP [44]) as our comparison baselines. This is because white-box methods usually perform better than black-box methods thanks to their access to the original model [67].

**Evaluation Metrics.** Quantitatively, we directly evaluate the impact of selected features on the distance changes. Given a testing sample $\boldsymbol{x}_t$ and an explanation method, we obtain the selected features $\boldsymbol{m}_t$, where $(\boldsymbol{m}_t)_i = 1$, if the $i^{th}$ feature is selected as important, We quantify the fidelity of this explanation result by this metric: $d'_{\boldsymbol{x}_t} = \|f(\boldsymbol{x}_t \odot (1-\boldsymbol{m}_t) + \boldsymbol{x}^{(c)}_{y_t} \odot \boldsymbol{m}_t) - \boldsymbol{c}_{y_t}\|_2$ where $f$, $\boldsymbol{c}_{y_t}$, and $\boldsymbol{x}^{(c)}_{y_t}$ have the same definition as the ones in Eqn. (2). $d'_{\boldsymbol{x}_t}$ represents the latent distance between a perturbed sample of $\boldsymbol{x}_t$ and its closet centroid $\boldsymbol{c}_{y_t}$. The perturbed sample is generated by replacing the values of the important features in $\boldsymbol{x}_t$ with those of the training sample closest to the centroid (*i.e.* $\boldsymbol{x}^{(c)}_{y_t}$). If the selected features are truly important, then substituting them with the corresponding features in the training sample from class $y_t$ will reduce the distance between the perturbed sample and the centroid of $\boldsymbol{c}_{y_t}$. In this case, a lower distance $d'_{\boldsymbol{x}_t}$ is better.

In addition to this $d'_{\boldsymbol{x}_t}$ metric, we also use a traditional metric (Section 5.2) to examine the ratio of perturbed samples that can cross the decision boundary.

| **Drebin Case-A:** Drifting Sample Family: **FakeDoc**; Closest Family: **GingerMaster** |
|---|
| [api_call::android/telephony/SmsManager;->sendTextMessage] , [call::readSMS] , [permission::android.permission.DISABLE_KEYGUARD] , [permission::android.permission.RECEIVE_SMS] , [permission::android.permission.SEND_SMS] , [permission::android.permission.WRITE_SMS] , [real_permission::android.permission.SEND_SMS] , [permission::android.permission.READ_SMS] , [feature::android.hardware.telephony] , [permission::android.permission.READ_CONTACTS] , [real_permission::android.permission.READ_CONTACTS] , [api_call::android/location/LocationManager;->isProviderEnabled], [api_call::android/accounts/AccountManager;->getAccounts], [intent::android.intent.category.HOME], [feature::android.hardware.location.network], [real_permission::android.permission.RESTART_PACKAGES] , [real_permission::android.permission.WRITE_SETTINGS] , [api_call::android/net/ConnectivityManager;->getAllNetworkInfo], [api_call::android/net/wifi/WifiManager;->setWifiEnabled], [api_call::org/apache/http/impl/client/DefaultHttpClient], [url::https://ws.tapjoyads.com/] , [url::https://ws.tapjoyads.com/set_publisher_user_id?] , [permission::android.permission.CHANGE_WIFI_STATE], [real_permission::android.permission.ACCESS_WIFI_STATE], [real_permission::android.permission.BLUETOOTH], [real_permission::android.permission.BLUETOOTH_ADMIN], [call::setWifiEnabled]. |

Table 5: Case study of explaining why a given sample a drifting sample. The highlighted features represent those that match the semantic characteristics that differentiate the drifting sample with the closest family.

| Method | Drebin-FakeDoc | IDS2018-Infiltration |
|---|---|---|
| Random | 0% | 0% |
| Boundary-based | 0% | 0.41% |
| COIN [43] | 0% | 0% |
| CADE | **97.64%** | **1.41%** |

Table 6: Comparison of explanation fidelity based on the ratio of perturbed samples that cross the decision boundary. A higher ratio means the perturbed features are more important.

## 5.2 Fidelity Evaluation Results

**Feature Impact on Distance.** Table 4 shows the mean and standard deviation for $d'_{\boldsymbol{x}_t}$ of all the drifting samples (*i.e.*, the distance between the perturbed samples to the nearest centroid). We have four key observations. *First*, perturbing the drifting samples based on the randomly selected features almost does not influence the latent space distance (comparing Row 2 and 3). *Second*, the boundary-based explanation method could lower the distance by 26%–47% across two datasets (comparing Row 2 and 4). This suggests this strategy has some effectiveness. However, the absolute distance values are still high. *Third*, COIN reduces the latent space distance on the IDS2018 dataset (comparing Row 2 and 5), but it somehow increases the average distance in the Drebin dataset. Essentially, COIN is a specialized boundary-based method that uses a set of LinearSVM classifiers to approximate the decision boundary. We find COIN does not work well on the high-dimensional space, and it is difficult to drag the drifting sample to cross the boundary (will be discussed in Section 5.3). *Finally*, our explanation module in CADE has the lowest mean and standard deviation for the distance metric. The distance has been reduced significantly from the original distance (*i.e.* 98.8% on Drebin and 79.9% on IDS2018, comparing Row 2 and 6). In particular, CADE significantly outperforms the boundary-based explanation method. Since our method overcomes the sample sparsity and imbalance issues, it pinpoints more effective features that have a larger impact on the distance (which affects the drift detection decision).

**Number of Selected Features.** Overall, the number of selected features is small, which makes it possible for manual interpretation. As mentioned, we configure all the methods to select the same number of important features (as CADE). For the Drebin dataset, on average the number of selected features is 44.7 with a standard deviation of 6.2. This is considered a very small portion (3%) out of 1000+ features. Similarly, the average number of selected features for the IDS2018 dataset is 16.2, which is about 20% of all the features.

## 5.3 Crossing the Decision Boundary

The above evaluation confirms the impact of the selected features on the *distance* metric, which is what CADE is designed to optimize. To provide another perspective, we further examine the impact of the selected features on *crossing the boundary*. More specifically, we calculate the ratio of perturbed samples that successfully cross the decision boundary. As shown in Table 6, we confirm that crossing the boundary in the drifting detection context is difficult for most of the settings. In particular, CADE can push 97.64% of the perturbed samples to cross the detection boundary for the Drebin dataset, but only have 1.41% of the samples cross the boundary for the IDS2018 dataset. In comparison, the baseline methods can rarely successfully perturb the drifting samples in the original feature space to make them cross the boundary. By loosing up this condition and focusing on distance changes, our method is more effective in identifying important features.

## 5.4 Case Studies

To demonstrate our method indeed captures meaningful features, we present some case studies. In Table 5, we present a case study for the Drebin dataset. We take the setting when *FakeDoc* is the unseen family and randomly pick a drifting sample to run the explanation module. Out of 1000+ features, our explanation module pinpointed 42 important features, among which 27 features have a value of "1" (meaning this

sample contains these features). As shown in Table 5, the closest family is *GingerMaster*.

We manually examine these features to determine if the features carry the correct semantic meanings. While it is difficult to obtain the "ground-truth" explanations, we gather external analysis reports about FakeDoc malware and GingerMaster [68, 70]. Based on these reports, a key difference from GingerMaster is that FakeDoc malware usually subscribes to premium services via SMS and bill the victim users. As shown in Table 5, many of the selected features are related to permissions and APIs calls for reading, writing, and sending SMS. We highlight these features that match SMS related functionality. Other related features are highlighted too. For example, the permission of "RESTART_PACKAGES" allows the malware to end the background processes (*e,g.*, that displays incoming SMS) to avoid alerting the users. The permission of "DISABLE_KEYGUARD" allows the malware to send premium SMS messages without unlocking the screen. "WRITE_SETTINGS" is also helpful to write system settings for sending SMS stealthily. "url::https://ws.tapjoyads.com/" is an advertisement library usually used by FakeDoc. Again, this small set of features is selected from over 1000 features. We conclude that these features are highly indicative of how this sample is different from the nearest known family.

## 6 Evaluation: In-class Evolution

So far, our evaluation has been focused on one type of concept drift (Type A) where the drifting samples come from a previously unseen family. Next, we explore to adapt our solution to address a different type of concept drift (Type B) where the drifting samples come from existing classes. We conduct a brief experiment in a binary classification setting, following a similar setup with that of [38].

More specifically, we first use the Drebin dataset to train a *binary* SVM classifier to classify malware samples from benign samples. The classifier is highly accurate on Drebin with a training $F_1$ score of 0.99. We want to test how well this classifier works on a different Android malware dataset Marvin [42]. Marvin is a slightly newer dataset (from 2010 to 2014) compared with Drebin (from 2010 to 2012). We first remove Marvin's samples that are overlapped with those in Drebin, to make sure the Marvin samples are truly previously unseen. This left us 9,592 benign samples and 9,179 malware samples in Marvin.

For this experiment, we randomly split the Marvin dataset into a validation set and a testing set (50:50). For both sets, we keep a balanced ratio of malware and benign samples. We apply the original classifier (trained on Drebin data) on this Marvin testing set. We find that the testing accuracy is no longer high ($F_1$ score 0.70) due to potential in-class evaluation in the malware class and/or the benign class.

To address the in-class evolution, we apply CADE and Transcend on the Marvin validation set to identify a small number of drifting samples (they could be either benign or malicious). We simulate to label them by using their "ground-truth" labels and then add these labeled drifting samples back to the Drebin training data to retrain the binary classifier. Finally, we test the retrained classifier on the Marvin testing set.

| # Selected Samples | $F_1$ of Retrained Classifier | |
|---|---|---|
| | CADE | Transcend |
| 0 | 0.70 | 0.70 |
| 100 | 0.91 | 0.71 |
| 150 | 0.92 | 0.76 |
| 200 | 0.93 | 0.74 |
| 250 | 0.94 | 0.71 |

Table 7: Performance of the retrained classifier on the Marvin testing set. We used CADE and Transcend to select the drifting samples to be labeled for retraining.

As shown in Table 7, we find that CADE still significantly outperforms Transcend. For example, by adding only 150 drifting samples (1.7% of Marvin validation set) for retraining, CADE boosts the binary classifier's $F_1$ score back to 0.92. For Transcend, the same number of samples only gets the $F_1$ score back to 0.74. In addition, we find that CADE is also faster: the running time for CADE is 1.2 hours (compared to the 10 hours of Transcend). This experiment confirms CADE can be adapted to handle in-class evolution for a binary malware classifier.

## 7 Real-world Test on PE Malware

We have worked with the security company Blue Hexagon Inc. to test CADE on their proprietary sample set. More specifically, we run an initial test on Blue Hexagon's Windows malware database. In this test, we got access to a set of samples collected from August 29, 2019, to February 10, 2020. This set includes 20,613 unique Windows PE malware samples from 395 families. We use this dataset to test CADE in a more diverse setup (*i.e.*, the drifting samples come from a larger number of families).

**PE Malware Dataset.** For each sample, we have the raw binary file and the metadata provided by Blue Hexagon, including the timestamp when the samples were first observed, and the family name (labeled by security analysts). We follow the feature engineering method of Ember [6], and use LIEF [63] to parse the binary files and extract the feature vectors. Each feature vector has 2,381 dimensions. These features include the frequency histogram of bytes and the entropy of different bytes, printable strings and special patterns, features about file size, header information, section information, imported libraries and functions, exported functions, and the size and virtual addresses of data directories.

**Family Attribution Experiments.** The original classifier is a multi-class classifier to attribute malware families. Our goal is to use CADE to detect unseen families that should not be attributed to existing families. We split the dataset based on

| $N$ | Precision | Recall | $F_1$ | Norm. Effort | Detected Families |
|---|---|---|---|---|---|
| 5 | 0.96 | 0.98 | 0.97 | 1.02 | 161/165 |
| 10 | 0.96 | 0.94 | 0.95 | 0.98 | 153/160 |
| 15 | 0.95 | 0.80 | 0.87 | 0.84 | 140/155 |

Table 8: Drifting detection results for the PE malware dataset. $N$ is the number of known families in the training set. "Detected Families" indicates the number of new families CADE detected out of all the new families.

time. The training set contains the malware samples collected from August 29 in 2019, to January 10 in 2020. The testing set contains samples collected in the following month, from January 10 to February 10, 2020. For training, we need to make sure the malware families have enough samples to train the original classifier. So we focus on the top $N$ families. We test three settings with $N = 5$, 10, and 15, respectively. This makes sure the training families contain at least 298 samples per family in all the settings. Samples that are not in the top $N$ families are excluded from the *training* set. Such a minimal number of samples is necessary for the original classifier to have reasonable accuracy. For example, the accuracy for $N = 15$ is 96.5%. The classifier can potentially support more families if the dataset is larger. For the testing set, all the families are kept. In addition, based on the suggestion from Blue Hexagon's analytics team, we add two families (Tinba and Smokeloader) to the testing set because they have observed that these families have more success in evading existing ML-based malware detection engines. As shown in Table 8, the testing set has 155 to 165 previously unseen families, *i.e.*, the target of CADE.

**Results and Case Studies.** Table 8 shows that CADE still performs well under this diverse set of samples with more than 155 previously unseen families. CADE achieves an $F_1$ score of 95% when the number of training families $N = 10$. The $F_1$ score is still 0.87 when $N = 15$. Most of the previously unseen families are successfully identified. Indeed, a larger number of families has made the problem more challenging. The reason is not necessarily because existing families and unseen families are difficult to separate. Instead, with more training families, we observe more testing samples *within the existing families* that drift even further away compared to those in the unseen families. These in-family variants become the main contributor to false positives under our definition. The observation is similar to our case study in Section 4.2. As a quick comparison, we also run Transcend on this $N = 15$ setting. We find CADE still outperforms Transcend on the more diverse unseen families (Transcend's $F_1$ score is only 0.76).

We did a quick feature analysis using the explanation module on Tinba and Smokeloader which are proven to be challenging examples for the underlying classifier. Tinba (tiny banker trojan) targets financial websites with man-in-the-browser attacks and network sniffing. Smokeloader is a trojan that downloads other malware. It is an old malware family but evolves rapidly. In particular, we find the new samples in Tinba are closest to an existing family Wabot. CADE pinpoints 45 features to offer explanations. For example, we find Tinba enables the "LARGE_ADDRESS_AWARE" option, which tells the linker that the program can handle addresses larger than 2 gigabytes. This option is enabled by default on 64-bit compilers. This provides some explanation on why Tinba has the success in evading existing malware detection engines, given that the vast majority of PE malware files are 32-bit based. Based on features about "sections," we notice that the Tinba sample uses "UPX" as the packer. Based on selected features of imported libraries and functions, we find Tinba imports "crypt32.dll" for encrypting strings. Tinba samples are different from Wabot samples on these features.

## 8 Discussion

**Computational Complexity.** CADE's computational overhead is smaller than existing methods. The complexity of the detection module contains two parts: contrastive learning and drifting detection. The complexity of contrastive learning is $O(IB^2|\theta|)$, where $I$, $B$, and $|\theta|$ represent the number of training iterations, batch size, and model parameters of the autoencoder. The complexity of drifting detection (Algorithm 1) is $O(N\tilde{n}_i + NK)$, where $N$, $\tilde{n}_i$, and $K$ are the number of classes, the maximum number of training samples in each class, and the number of testing samples, respectively. The overall complexity of CADE detection module is $O(IB^2|\theta| + N\tilde{n}_i + NK)$. Our training overhead is acceptable since it is only quadratic to the batch size $B$. Our detection runtime overhead is significantly lower than that of Transcend (which is $O(N\tilde{n}_i K)$). Empirically, we have recorded the average runtime for the detection experiments (Section 4), and confirms that CADE is faster than Transcend. For example, on the larger IDS2018 dataset, the average run time for CADE and Transcend are 1,422.7s and 4,289.3s. Regarding the explanation module, CADE is comparable with boundary-based explanation methods and COIN. For example, for the IDS2018 dataset, the average runtime of CADE, COIN, and boundary-based explanation for explaining one drifting sample are 3.2s, 8.2s, and 3.7s respectively. The boundary-based explanation also requires an additional 76.5s on average to build the approximation model for the explanation.

**Explanation vs. Adversarial Attacks.** We notice that the explanation module in CADE shares some similarities with the adversarial example generation process, *e.g.*, both involve perturbing the given input for a specific objective. However, we think they are different for two reasons. First, they have different outputs. Adversarial attack (with the goal of evasion) directly outputs the perturbation needed to cross the decision boundary; Our explanation method (with the goal of understanding the drift) outputs the important features that affect the distance. Second, they have different constraints on the

perturbations. Our explanation method only tries to minimize the number of perturbed features, while the adversarial attack constrains the magnitude of the perturbation too. More importantly, adversarial samples need to be *valid* for the respective applications (*i.e.*, valid malware samples that can be executed and maintain the malicious behavior, valid network flows that can carry out the original attack). To these ends, generating adversarial samples can be more difficult than deriving explanations in our context. That said, the adversarial attack is out of the scope of this paper. We leave adversarial attacks against CADE to future work (*i.e.*, creating non-perceptible perturbation to convert a drifting sample to an in-distribution sample).

**Limitations and Future Work.** Our work has a few limitations. First, CADE ranks all the drifting samples in a single list. However, in practice, the drifting samples may contain substructures (*e.g.*, multiple new malware families). A practical strategy could be further grouping drifting samples into clusters. In this way, security analysts only need to inspect and interpret a few representative samples per cluster to further save time. Second, certain hyper-parameters of CADE are determined empirically (*e.g.*, the MAD threshold). We have included an Appendix C to test the sensitivity of CADE to hyper-parameters. Future work can look into more systematic strategies to configure the hyper-parameters. Third, CADE is designed based on the assumption that the training set does not have mislabeled samples (or poisoning samples). We defer to future work to robustify our system against low-quality or malicious labels. Fourth, our experiments are primarily focused on detecting new families. In Section 6, we only briefly experimented with concept drift within existing families (in-class evolution). We defer a more in-depth analysis to future work.

Finally, our evaluation in Section 7 is limited to $N = 15$ training classes (and 155 previously unseen testing classes). We limited to $N = 15$ to make sure each training class has enough samples to train an accurate *original classifier*. To test a larger $N$, we tried to apply CADE to several other malware datasets but did not find a suitable one that could meet our need. For example, the Ember-2018 dataset [6] provides malware samples from a large number of families. However, the family labels are not well curated. For instance, a popular malware family name in Ember-2018 is called "high" (8,417 samples) which turns out to be incorrectly parsed from VirusTotal reports: the original entry name in the reports is "Malicious (High Confidence)," which is not a real malware family name. We have observed other similar parsing errors and inconsistencies in the labels. The Ember-2017 dataset [6] and the UCSB packed malware dataset [3] do not provide malware family information. The dataset from Microsoft Malware Classification Challenge [55] only has 9 malware families, which is smaller than our Blue Hexagon dataset. Given our unsuccessful efforts, we defer the examination of a larger number of training classes to future work.

## 9 Related work

**Machine Learning used in Security.** Machine learning has been used to solve many security problems such as malware detection [6,7,17,42], malware family attribution [4,11], and network intrusion detection [24, 34, 48, 60]. More recently, researchers look into using deep learning methods to perform binary analysis [27,69], software vulnerability identification [72], and severity prediction [30]. Most of these machine learning models need to address the concept drift problem when deployed in practice.

**Out of Distribution (OOD) Detection.** Recently, the machine learning community has made progress in out-of-distribution detection [14, 32, 41, 46, 49]. These works are relevant, but have different assumptions and goals compared to ours. At the high-level, most of these methods try to calibrate the "probability" produced by the original classifier to detect OOD samples. The researchers indeed recognized that the probability could be untrustworthy when it comes to previously unseen distributions [14, 32]. To avoid assigning a high probability to an OOD sample, the proposed methods usually need to introduce an *auxiliary OOD dataset* to the training data. These methods are difficult to realize in security applications for two reasons. First, auxiliary OOD dataset (*i.e.*, previously unseen attacks) is extremely difficult to obtain in the first place. Second, these solutions require re-designing the original classifier (*e.g.*, a functional malware detector), which is inconvenient to do in the production environment. Instead, our method does not rely on auxiliary OOD dataset and is decoupled from the original classifier.

**Classification Trustworthiness.** A related line of work aims to assess the trustworthiness of the classification results [11, 37, 50]. A common goal is to identify untrusted predictions, *e.g.*, predictions on adversarial attacks. Most of these methods are based on the idea of "nearest neighbors". The intuition is, an untrusted prediction is more likely to have a different label from its nearest neighbors. For example, DkNN [50] derives a trust score by comparing a testing sample with its neighboring training samples at each layer of a Deep Neural Network (DNN). Another recent work [37] compute the trust score based on the neighboring "high-density-sets". However, such neighbor-based methods still rely on a good distance function. As acknowledged in [37], their method may suffer in a high dimensional space. Overall, these methods are focused on different problems from ours. Their goal is to identify misclassifications within existing classes (not drifting samples from new classes). Another system EC2 [11] uses a threshold of *prediction probability* to filter out untrustworthy predictions. Related to this direction, active learning methods also use prediction probability to select low-confidence samples to be labeled for model retraining [47, 73]. As discussed before (see [32]), the prediction probability itself can be misleading under concept drift.

**Machine Learning Explanation.** A collection of recent works focus on post-hoc interpretation methods for machine learning classifiers [8, 22, 35, 58, 59] and study the robustness of explanations [15, 71]. Given a testing sample, the goal is to pinpoint important features to explain the classification decisions. Most methods are designed for deep neural networks. For example, perturbation-based methods would subtly manipulate the input and observe the variation of output to identify important features [13, 18, 21, 22]. Gradient-based methods (*e.g.*, saliency maps) back-propagate gradients through the deep neural network to measure the sensitivity of each feature [56, 58, 59, 61]. Other explanation methods treat the target classifier as a blackbox [53, 54]. Systems such as LIME [53], LEMNA [28], and SHAP [44] try to use a simpler model (*e.g.*, linear regression) to approximate the decision boundary near the input sample, and then use the simpler model to pinpoint features to generate the explanations.

Our method falls into the category of perturbation-based method. A key difference is existing methods are designed for *supervised classifiers* and try to explain the decision boundaries. Our method is focusing on explaining distance changes, which are more suitable for outlier detection. Only a few works aim to explain unsupervised models [19, 43]. We used COIN [43] as a baseline in our evaluation, and showed the advantage of distance-based explanation.

## 10 Conclusion

In this paper, we build a novel system CADE to complement supervised classifiers to combat concept drift in security contexts. Using a contrastive autoencoder and a distance-based explanation method, CADE is designed to detect drifting samples that deviate from the original training distribution and provide the corresponding explanations to reason the meaning of the drift. Using various datasets, we show that CADE outperforms existing methods. Working with an industry partner, we demonstrate CADE's ability to detect and explain drifting samples from previously unseen families.

## Acknowledgment

We thank our shepherd David Freeman and anonymous reviewers for their constructive comments and suggestions. This work was supported in part by NSF grants CNS-2030521 and CNS-1717028, and Amazon Research Award.

## References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proc. of USENIX OSDI*, 2016.

[2] Hervé Abdi and Lynne J. Williams. Principal component analysis. *WIREs Computational Statistics*, 2010.

[3] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In *Proc. of NDSS*, 2020.

[4] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proc. of CODASPY*, 2016.

[5] Bruce An. More adware and plankton variants seen in app stores. TrendMicro, 2012.

[6] Hyrum S Anderson and Phil Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[7] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. of NDSS*, 2014.

[8] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 2015.

[9] Manuel Baena-Garcıa, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, R Gavalda, and R Morales-Bueno. Early drift detection method. In *Fourth international workshop on knowledge discovery from data streams*, 2006.

[10] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *Proc. of SDM*, 2007.

[11] Tanmoy Chakraborty, Fabio Pierazzi, and VS Subrahmanian. Ec2: Ensemble clustering and classification for predicting android malware families. *TDSC*, 2017.

[12] Eshwar Chandrasekharan, Mattia Samory, Anirudh Srinivasan, and Eric Gilbert. The bag of communities: Identifying abusive behavior online with preexisting internet data. In *Proc. of CHI*, 2017.

[13] Chun-Hao Chang, Elliot Creager, Anna Goldenberg, and David Duvenaud. Explaining image classifiers by counterfactual generation. In *Proc. of ICLR*, 2019.

[14] Jiefeng Chen, Yixuan Li, Xi Wu, Yingyu Liang, and Somesh Jha. Robust out-of-distribution detection in neural networks. *arXiv preprint arXiv:2003.09711*, 2020.

[15] Jiefeng Chen, Xi Wu, Vaibhav Rastogi, Yingyu Liang, and Somesh Jha. Robust attribution regularization. In *Proc. of NeurIPS*, 2019.

[16] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. *arXiv:2002.05709*, 2020.

[17] Yizheng Chen, Shiqi Wang, Dongdong She, and Suman Jana. On training robust pdf malware classifiers. In *Proc. of USENIX Security*, 2020.

[18] Piotr Dabkowski and Yarin Gal. Real time image saliency for black box classifiers. In *Proc. of NeurIPS*, 2017.

[19] Xuan Hong Dang, Ira Assent, Raymond T Ng, Arthur Zimek, and Erich Schubert. Discriminative features for identifying and interpreting outliers. In *Proc. of ICDE*, 2014.

[20] Denis Moreira dos Reis, Peter Flach, Stan Matwin, and Gustavo Batista. Fast unsupervised online drift detection using incremental kolmogorov-smirnov test. In *Proc. of KDD*, 2016.

[21] Ruth C Fong, Mandela Patrick, and Andrea Vedaldi. Understanding deep networks via extremal perturbations and smooth masks. In *Proc. of ICCV*, 2019.

[22] Ruth C Fong and Andrea Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *Proc. of ICCV*, 2017.

[23] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 2014.

[24] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 2009.

[25] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *Proc. of ICLR*, 2015.

[26] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. 2017.

[27] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. Deepvsa: Facilitating value-set analysis with deep learning for post-mortem program analysis. In *Proc. of USENIX Security*, 2019.

[28] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *Proc. of CCS*, 2018.

[29] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *Proc. of CVPR*, 2006.

[30] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. Learning to predict severity of software vulnerability using only vulnerability description. In *Proc. of ICSME*, 2017.

[31] Maayan Harel, Shie Mannor, Ran El-Yaniv, and Koby Crammer. Concept drift detection through resampling. In *Proc. of ICML*, 2014.

[32] Dan Hendrycks and Kevin Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. In *Proc. of ICLR*, 2017.

[33] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 2006.

[34] Elike Hodo, Xavier Bellekens, Andrew Hamilton, Christos Tachtatzis, and Robert Atkinson. Shallow and deep networks intrusion detection system: A taxonomy and survey. *arXiv preprint arXiv:1701.02145*, 2017.

[35] Sara Hooker, Dumitru Erhan, Pieter-Jan Kindermans, and Been Kim. A benchmark for interpretability methods in deep neural networks. In *Proc. of NeurIPS*, 2019.

[36] Steve TK Jan, Qingying Hao, Tianrui Hu, Jiameng Pu, Sonal Oswal, Gang Wang, and Bimal Viswanath. Throwing darts in the dark? detecting bots with limited data using neural data augmentation. In *Proc. of S&P*, 2020.

[37] Heinrich Jiang, Been Kim, Melody Guan, and Maya Gupta. To trust or not to trust a classifier. In *Proc. of NeurIPS*, 2018.

[38] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *Proc. of USENIX Security*, 2017.

[39] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D. Joseph, and J. D. Tygar. Approaches to adversarial drift. In *Proc. of AISec*, 2013.

[40] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 2013.

[41] Shiyu Liang, Yixuan Li, and Rayadurgam Srikant. Enhancing the reliability of out-of-distribution image detection in neural networks. *Proc. of ICLR*, 2018.

[42] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Prof. of COMPSAC*, 2015.

[43] Ninghao Liu, Donghwa Shin, and Xia Hu. Contextual outlier interpretation. In *Proc. of IJCAI*, 2018.

[44] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Proc. of NeurIPS*, 2017.

[45] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *Proc. of ICLR*, 2017.

[46] Marc Masana, Idoia Ruiz, Joan Serrat, Joost van de Weijer, and Antonio M Lopez. Metric learning for novelty and anomaly detection. In *Proc. of BMVC*, 2018.

[47] Brad Miller, Alex Kantchelian, Sadia Afroz, Rekha Bachwani, Edwin Dauber, Ling Huang, Michael Carl Tschantz, Anthony D. Joseph, and J.D. Tygar. Adversarial active learning. In *Proc. of AISec*, 2014.

[48] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. In *Proc. of NDSS*, 2018.

[49] Aristotelis-Angelos Papadopoulos, Mohammad Reza Rajati, Nazim Shaikh, and Jiamian Wang. Outlier exposure with confidence control for out-of-distribution detection. *arXiv preprint arXiv:1906.03509*, 2019.

[50] Nicolas Papernot and Patrick McDaniel. Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. *arXiv preprint arXiv:1803.04765*, 2018.

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

[52] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. TESSERACT: Eliminating experimental bias in malware classification across space and time. In *Proc. of USENIX Security*, 2019.

[53] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?" explaining the predictions of any classifier. In *Proc. of KDD*, 2016.

[54] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *Proc. of AAAI*, 2018.

[55] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135*, 2018.

[56] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proc. of ICCV*, 2017.

[57] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *Prof. of ICISSP*, 2018.

[58] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *Proc. of ICML*, 2017.

[59] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *Workshop at ICLR*, 2014.

[60] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proc. of S&P*, 2010.

[61] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. In *Proc. of ICLR*, 2015.

[62] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proc. of ICML*, 2017.

[63] Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com/, April 2017.

[64] Robert Tibshirani and Guenther Walther. Cluster validation by prediction strength. *Journal of Computational and Graphical Statistics*, 2005.

[65] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 2019.

[66] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 2008.

[67] Alexander Warnecke, Daniel Arp, Christian Wressnegger, and Konrad Rieck. Don't paint it black: White-box explanations for deep learning in computer security. In *Proc. of Euro S&P*, 2020.

[68] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *Proc. of DIMVA*, 2017.

[69] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proc. of CCS*, 2017.

[70] Rowland Yu. Ginmaster: a case study in android malware. In *Virus bulletin conference*, 2013.

[71] Xinyang Zhang, Ningfei Wang, Shouling Ji, Hua Shen, and Ting Wang. Interpretable deep learning under fire. In *Proc. of USENIX Security*, 2020.

[72] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proc. of NeurIPS*, 2019.

[73] Jingbo Zhu, Huizhen Wang, Eduard Hovy, and Matthew Ma. Confidence-based stopping criteria for active learning for data annotation. *ACM Trans. Speech Lang. Process.*, 2010.

[74] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. A survey on unsupervised outlier detection in high-dimensional numerical data. *Statistical Analysis and Data Mining*, 2012.

## Appendix A: Boundary-based Explanation

To perform the boundary-based explanation, we first need to approximate the detection boundary of the drift detection module with a *parametric function*. We need to run approximation because the true boundary of the drift detector is threshold-based, which is not parametric. Specifically, we used an MLP classifier to perform the approximation in the latent space. Due to the limited number of drifting samples, to approximate the decision boundary, we first synthesized more drifting samples by adding Gaussian noise to the latent representations of the detected drifting samples. Then, we trained an MLP $g(\boldsymbol{z})$ to classify the latent representations of the in-distribution samples from the drifting samples. After obtaining the approximation model, we combined it with the contrastive autoencoder $f$ to construct a supervised approximation of the detection module (*i.e.* $g(f(\boldsymbol{x}))$). We conducted the approximation in the latent space rather than the input space for two reasons. First, training an MLP in a low dimensional space is more efficient than in a high dimensional space. Second, directly utilizing the original contrastive autoencoder enables a higher fidelity of the supervised approximation than approximating the autoencoder with another network. Using the supervised approximation, we then applied the perturbation-based explanation method [22] to explain each drifting sample. Similar to CADE, this method also outputs a mask indicating the feature importance. We ranked the $\boldsymbol{m}_i$ and pinpointed the features with high $\boldsymbol{m}_i$ as the important ones.

## Appendix B: CADE Implementation Details

**CADE.** We implemented CADE based on the Keras [26] package with Tensorflow [1] as the backend. The hyper-parameters of CADE and the baselines are configured as the following. As for CADE, we set the encoder as an MLP with the architecture of 1340-512-128-32-7 for the Drebin dataset (the first dimension could vary when using different families as the unseen family) and 83-64-32-16-3 for the IDS2018 dataset. The activation function for each hidden layer is the ReLU function. We applied the Adam optimizer with the learning rate of 0.0001 and epochs of 250 to train both networks. The batch size for Drebin and IDS2018 are 32 and 256, respectively. As for the hyper-parameters introduced by the contrastive loss in Eqn. (1), we set $\lambda = 0.1$ and $m = 10$. We applied the widely used empirical value for the MAD threshold and coefficient: $T_{\text{MAD}} = 3.5$ and $b = 1.4826$. For the hyper-parameters introduced by the explanation loss in Eqn. (2), we set $\lambda_1 = 1e-3$ and used the Adam optimizer with the learning rate of 0.01 to solve the optimization function. The training epoch is set as 250.

**Drift Detection Baselines.** The vanilla autoencoder baseline was implemented as a variant of our system without using contrastive learning. We also implemented a multi-class version of Transcend based on the source code provided by the authors. The hyper-parameters of the vanilla AE baseline are almost the same with CADE except for the MAD threshold $T_{MAD} = 0$. We tried $T_{MAD} = 3.5$ for this method, which resulted in zero precision and recall. The reason is that the distance in vanilla AE's latent space is not optimized to compare different samples and thus MAD lost its effectiveness.

For Transcend, we used an MLP with the architecture of 1340-100-30-7 for the Drebin dataset and 83-30-3 for the IDS2018 dataset to train a multi-class classifier. Then we used the negative output probability $-p$ as the non-conformity measure of Transcend. We set the threshold of the credibility $p$-value as 1.0. That is, a testing sample is marked as a drifting sample if its $p$-value is lower than 1.0.

**Explanation Baselines.** We implemented the boundary-based explanation method and the random selection as described in the main text. For COIN, we used the source code released by the authors as the implementation.[5] The network architectures of the approximation function $g$ in the boundary-based explanation are 7-15-2 and 3-15-2 for Drebin and IDS2018, respectively. The optimizer, batch size, and number of epochs are the same as those used in our sys-

[5] https://github.com/ninghaohello/Contextual-Outlier-Interpreter

| Parameter | Drebin (Avg±Std) | | IDS2018 (Avg±Std) | |
|---|---|---|---|---|
| | $F_1$ | Norm. Effort | $F_1$ | Norm. Effort |
| $m = 5$ | $0.95 \pm 0.05$ | $0.97 \pm 0.05$ | $0.72 \pm 0.39$ | $0.72 \pm 0.39$ |
| $m = 10$ | $0.96 \pm 0.03$ | $1.00 \pm 0.09$ | $0.96 \pm 0.06$ | $0.95 \pm 0.07$ |
| $m = 15$ | $0.91 \pm 0.06$ | $1.00 \pm 0.14$ | $0.77 \pm 0.33$ | $0.76 \pm 0.34$ |
| $m = 20$ | $0.93 \pm 0.03$ | $1.06 \pm 0.13$ | $0.98 \pm 0.02$ | $1.02 \pm 0.02$ |
| $\lambda = 1$ | $0.95 \pm 0.03$ | $1.05 \pm 0.11$ | $0.94 \pm 0.09$ | $1.00 \pm 0.00$ |
| $\lambda = 0.1$ | $0.96 \pm 0.03$ | $1.00 \pm 0.09$ | $0.96 \pm 0.06$ | $0.95 \pm 0.07$ |
| $\lambda = 0.01$ | $0.94 \pm 0.03$ | $1.05 \pm 0.09$ | $0.67 \pm 0.47$ | $0.71 \pm 0.42$ |
| $\lambda = 0.001$ | $0.89 \pm 0.10$ | $1.19 \pm 0.33$ | $0.95 \pm 0.05$ | $0.93 \pm 0.08$ |
| $T_{MAD} = 2.0$ | $0.96 \pm 0.03$ | $1.00 \pm 0.09$ | $0.94 \pm 0.09$ | $0.99 \pm 0.02$ |
| $T_{MAD} = 2.5$ | $0.96 \pm 0.03$ | $1.00 \pm 0.09$ | $0.95 \pm 0.07$ | $0.97 \pm 0.04$ |
| $T_{MAD} = 3.0$ | $0.96 \pm 0.03$ | $1.00 \pm 0.09$ | $0.95 \pm 0.07$ | $0.96 \pm 0.05$ |
| $T_{MAD} = 3.5$ | $0.96 \pm 0.03$ | $1.00 \pm 0.09$ | $0.96 \pm 0.06$ | $0.95 \pm 0.07$ |

Table 9: Sensitivity test of three hyper-parameters on detecting drifting samples. For each evaluation metric, we report the mean value and the standard deviation across all the settings.

| $\lambda_1$ | Drebin-FakeDoc | | IDS2018-Infiltration | |
|---|---|---|---|---|
| | distance (Avg ± Std) | Ratio | distance (Avg ± Std) | Ratio |
| 0.1 | $0.119 \pm 0.058$ | 91.34% | $2.669 \pm 3.343$ | 1.99% |
| 0.01 | $0.085 \pm 0.039$ | 96.85% | $2.403 \pm 3.266$ | 1.36% |
| 0.001 | $0.065 \pm 0.035$ | 97.64% | $2.349 \pm 3.238$ | 1.41% |
| 0.0001 | $0.064 \pm 0.027$ | 99.21% | $2.322 \pm 3.240$ | 1.69% |

Table 10: Sensitivity test on the hyper-parameter $\lambda_1$ of explaining a drifting sample. "Ratio" means the percentage of perturbed samples that cross the decision boundary.

tem. The hyper-parameters of solving the explanation masks (*i.e.* optimizer and epoch) are also the same as our system. Finally, we used the default choices of the hyper-parameters from the authors' code of COIN.

The original implementation of COIN provided by the authors can be very slow when the dataset has a large number of samples and outliers. For each detected outlier, COIN runs KMeans clustering on its 10% of nearest neighbors to get its contexts. To determine the best number of clusters ($K$), COIN iterates $K$ from 1 to a pre-defined threshold and adopts the measure of *prediction strength* [64] to assess the choice of $K$. *Prediction strength* can be computationally expensive as it requires pair-wise comparison on the labels predicted by KMeans. To make it feasible, on the large IDS2018 dataset, we only choose 1% of nearest neighbors and fix the number of clusters as a value between 1 and 4 for each outlier. Also, the LinearSVM classifier does not converge on about 6% of outliers even we set max iterations as 200,000. We report the best average result on the converged cases obtained from `COIN`. For the Drebin dataset, we keep all the hyper-parameters the same as the original code.

## Appendix C: Hyper-parameter Sensitivity

In Section 3.2, the loss function of contrastive autoencoder has two hyper-parameters: $\lambda$ and $m$. Here, we evaluate the sensitivity of `CADE`'s performance to these hyper-parameters. Our experiment methodology is to fix one parameter and swap the other one. We fix $\lambda$ as 0.1 and set $m$ as 5, 10, 15, 20. As shown in Table 9, `CADE` achieves a high $F_1$ score on the Drebin dataset when $m = 5$ and $m = 10$, but has some minor degradation on $m = 15$ and $m = 20$. The detection performance on the IDS2018 dataset is good when $m$ is set to a higher number *e.g.*, $m = 20$. Recall that $m$ is the threshold to control the upper-bound distance that will be considered. A dissimilar pair can contribute to the loss function only when their distance is within the radius of $m$. As such, $m$ can be set to be higher if the dataset is more dispersed and noisy.

To test the effect of $\lambda$, we fix $m = 10$ as before, and set $\lambda$ as 1, 0.1, 0.01, and 0.001. $\lambda$ controls the weight of the contrastive loss. We can observe from Table 9 that if $\lambda$ is too small, it hurts `CADE`'s performance. The results confirm the importance of the contrastive loss.

In Algorithm 1, we set the threshold of MAD $T_{MAD}$ as 3.5, which is an empirical value [40]. We also tested other commonly used MAD thresholds of 2, 2.5, 3. A smaller MAD threshold will detect more samples as potential drifting samples, but it may not affect the ranking procedure. As shown in Table 9, the average results of the detected drifting samples keep the same as $T_{MAD} = 3.5$ on Drebin and minor fluctuations on the IDS2018 dataset, indicating $T_{MAD}$ has subtle effects on detecting drifting samples.

To assess the sensitivity of the hyper-parameter $\lambda_1$ in the loss function (Eqn.( 2)) of distance-based explanation, we set $\lambda_1$ as 0.1, 0.01, 0.001, and 0.0001. As shown in Table 10, we notice that smaller $\lambda_1$ can have a slightly smaller average distance to the nearest centroid on both Drebin and IDS2018 datasets. Also, a smaller $\lambda_1$ can increase the ratio of perturbed samples that cross the decision boundary from 91.34% to 99.21% on Drebin-FakeDoc. While for IDS-Infiltration, the ratio could vary on different values of $\lambda_1$. But overall, both evaluation metrics do not have significant differences among different values of $\lambda_1$.

| Sampling Rate | 10% | 15% | 20% | 25% | 30% |
|---|---|---|---|---|---|
| $F_1$ score | 0.96 | 0.98 | 0.98 | 0.98 | 0.97 |

Table 11: Sampling rate of IDS2018 dataset vs. $F_1$ score of CADE.

## Appendix D: IDS2018 Additional Results

In our experiment, we only sampled 10% of the network traffic from the IDS2018 dataset. Traffic sampling is a common approach in intrusion detection, which allows us to comprehensively test different experimental setups. We also find that including more traffic only increases the computational overhead and has a negligible influence upon the performance. As shown in Table 11, as the sampling rate increases, `CADE`'s $F_1$ scores remain consistently high.

# CADE: Detecting and Explaining Concept Drift Samples for Security Applications (Supplementary Materials)

## 1 Feature Engineering Details

We describe the detailed steps to perform feature engineering on the selected datasets.

**Drebin.** We followed the original paper [1] and used one-hot encoding to construct the feature vectors. As we mentioned, to be realistic, we only used the features (*e.g.*, strings, permissions, APIs) that appeared in the training data. The original feature dimensionality is between 5,055 and 7,296 (depending on which class is chosen as the unseen family). Since the feature vectors are very sparse, we reduced the dimensionality to about 1,000 based on the variance of each feature. We used the VarianceThreshold function in the scikit-learn package [2] to perform the dimension reduction.

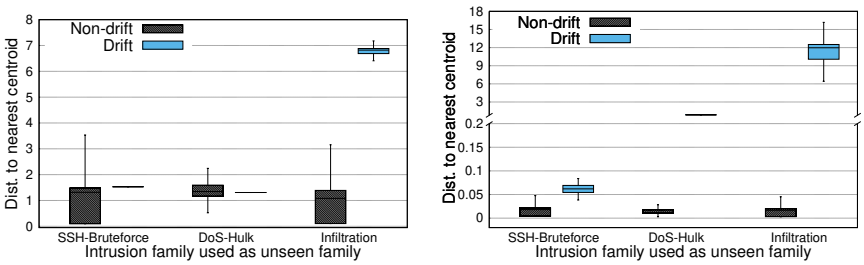
**IDS2018.** Each sample is a network flow, and the features are computed based on both directions: forward flow and backward flow [3]. Each sample has 80 features originally, two of which are categorical features (*i.e.* "Dst Port" and 'Protocol'), and the rest are numerical features. We used one-hot encoding for the categorical features. "Dst Port" is mapped into three categories based on its frequency of appearance (high, medium, and low). We did not encode the port number directly because the vector will be too high-dimensional. Specifically, if the destination port appears more than 10,000 times, it would be mapped to a high-frequency port. Destination port shows between 1,000 and 10,000 times are regarded as medium-frequency port. The rest of the ports are mapped to low-frequency port. "Protocol" is also mapped to a three-dimensional vector based on its value (TCP, UDP, IPv6). For the numerical features, we used MinMaxScaler in the scikit-learn package [2] to normalize them within $[0,1]$.

After the pre-processing, each sample is a vector of 83 dimensions, where each feature is within $[0,1]$. Similar to the Drebin dataset, we also built the encoding dictionary and the MinMaxScaler based on the training set and applied them to the testing samples. We mapped the unseen values of the categorical features to all zeros, indicting these values are not exist in the training set. Note that the feature "Timestamp" is originally represented by a string, we transform the string into milliseconds, which are numerical values. We will release the pre-processed datasets along the final version of the paper.

## 2 Distance-based Explanation using Gradient Method

Our distance-based explanation method in `CADE` is perturbation-based. An alternative way of designing the distance-based explanation method is to use *gradients*. Intuitively, after approximating the detection boundary, we can compute the partial derivative of the approximation model output with respect to the input and identify the features with the highest gradients. The higher gradients indicate that perturbing these features will have a stronger influence upon the latent distance between the drifting sample and the nearest centroid. However, this gradient-based method involves approximating the boundary as well as the direction to maximize the distance changes. To reduce the errors by the approximation, we can directly compute the direction that triggers the largest distance change. Since the latent space learned by the contrastive learning is a Euclidean space, this direction refers to $f(\boldsymbol{x}_t) - \boldsymbol{c}_{y_t}$, the direction from a drifting sample $\boldsymbol{x}_t$ to its nearest centroid $\boldsymbol{c}_{y_t}$. As such, we identify the important features by computing the gradient of $f(\boldsymbol{x}_t) - \boldsymbol{c}_{y_t}$ with respect to $\boldsymbol{x}_t$ and selecting the features with the highest gradients.

We run an evaluation of this idea and choose the same number of important features as `CADE`. Using the evaluation metrics introduced in the main paper (*i.e.* fidelity and the ratio of perturbed samples cross the boundary), we find this idea is not working well. More specifically, the gradient-based method shows a similar performance with the *random baseline*, indicating it fails to identify important features. We suspect the reason is that gradient can only reflect feature sensitivity to a small perturbation. However, we need a relatively large perturbation to impose a significant influence on the distance. As such, perturbation-based explanations are more suitable




(a) Original space

(b) Latent space (CADE)

Figure 1: Boxplot of the distances between testing samples and their nearest centroids in both the *original space* and *latent space* for the IDS2018 dataset. Samples from previously unseen family are regarded as drifting samples.

for our problem than gradient-based explanations.

## 3 IDS2018 Additional Results

Figure 1 shows the distance of the (non-)drifting samples to their closest centroid in the input space and the latent space of CADE. The results are aligned with those on the Drebin dataset, confirming that contrastive learning helps to separate drifting samples from non-drifting ones.

In Table 1 we present a case study on the explanation results of a drifting sample in IDS2018. We use the setting when *Infiltration* is the unseen family and randomly pick one drifting sample for explanation. According to CADE, the closest family is *DoS-Hulk*. After running the explanation, we locate 18 features.

We manually examine the selected features in Table 1, and look up the documentation of IDS2018 [3] to determine if the features have relevant semantic meanings. Specifically, *Infiltration* means the attacker first sends a malicious file via an email to the victim to exploit the victim's host. If successful, attackers will further run portscan (*e.g.*, using Nmap) and exploit more vulnerabilities [3]. *DoS-Hulk* is a particular type of Denial of Service (DoS) attack that aims to over-load the targeted host with superfluous network requests.

Different from Drebin (whose features only have binary values), features in the IDS2018 dataset could be numerical. To interpret the meaning of the features, we also compare the sample's feature values with those of the centroid of *DoS-Hulk*. We use $\uparrow$ (or $\downarrow$) to indicates whether the Infiltration flow has a higher (or lower) feature value.

We find most selected features make intuitive sense. These features are mainly describing the port-scan activities caused by the Infiltration. Some features are about the characteristics of the DoS attacks. First, the FSH flag is frequently set in the Infiltration flow ("FSH Flag Cnt") which indicates port scanning (*e.g.*, XMAS scan). The next four features show Infiltration backward flow's inter-arrival time ("BWD IAT") is shorter. This makes sense because DoS attack will cause large delays to backward flow as the target host is overwhelmed. It also makes sense that Infiltration flow's "Down/Up Ratio" is higher since DoS attack barely has any down-flow traffic. Then the lower "Fwd Pkts/s" indicates that the Infiltration forward flow is sending packets not as fast as DoS. Then the features on the next row are mostly showing the packet sizes of Infiltration are larger than those of DoS-Hulk. Interestingly, Infiltration has sent more packets in total (due to the portscan). Overall, the selected features are useful to explain why the infiltration sample is different from DoS-Hulk attacks. The only mis-selected feature is the "Timestamp" feature, which only indicates the two attacks happened at a different time (not about attack characteristics).

## Appendix-G: The Impact of Concept Drift on Supervised Classifier

We use the two datasets to examine how drifting samples impact the accuracy of the original classifier. We trained an MLP multi-class classifier using the training set. Then we test the trained MLP classifier on the testing sets with and without the previously unseen family. As shown in Table 2, when the testing set does not have any previously-unseen family, the original classifier performs well with an average accuracy of over 99.7% for both datasets. Then if we add the previously unseen families into the testing set (we iteratively test each family as the unseen family), the overall accuracy drops below 62%. As such, it is necessary to monitor incoming samples to detect drifting samples.

## Appendix-H: Drifting Detection with Prediction Probability

We provide additional evidence to show that the softmax outputted prediction probability is not a good indicator for drifting samples. We construct a baseline for drifting detection by ranking this prediction probability. We take the testing samples with a lower softmax prediction probability as drifting samples. Similar to Transcend, we ignore testing samples

| **IDS2018 Case-B:** Drifting Sample Family: **Infiltration**; Closest Family: **DoS-Hulk** |
|---|
| PSH Flag Cnt ↑ , Bwd IAT Tot ↓ , Bwd IAT Mean ↓ , Bwd IAT Min ↓ , Flow IAT Max ↓ , Down/Up Ratio ↑ , Fwd Pkts/s ↓ , Bwd Seg Size Avg ↑ , Bwd Pkt Len Max ↑ , Pkt Len Mean ↑ , Pkt Len Std ↑ , Pkt Size Avg ↑ , Fwd Seg Size Avg ↑ , Fwd Pkt Len Max ↑ , Subflow Fwd Byts ↑ , Tot Fwd Pkts↑ , Flow Byts/s ↑ , Timestamp ↑. |

Table 1: Case study of explaining why a given sample a drifting sample. The highlighted features represent those that match the semantic characteristics that differentiate the drifting sample with the closest family. ↑ means the drifting sample has a higher feature value compared to the closest family (DoS-Hulk); ↓ means the drifting sample has a lower feature value. "Pkt": packets; "Seg": segment; "Fwd": forward; "Bwd": backward; "Tot": total; "IAT": inter-arrival time.

| **Testing Set** | **Drebin** Avg ± Std | **IDS2018** Avg ± Std |
|---|---|---|
| Testing set (w/o drifting) | 0.9976 ± 0.00 | 1.0000 ± 0.00 |
| Testing set (w/ drifting) | 0.6201 ± 0.18 | 0.5602 ± 0.20 |

Table 2: Average accuracy of the original classifier on the testing sets (with and without the drifting samples).

| Metrics | **Drebin (Avg±Std)** | **IDS2018 (Avg±Std)** |
|---|---|---|
| Precision | 0.95 ± 0.08 | 0.50 ± 0.39 |
| Recall | 0.88 ± 0.11 | 0.67 ± 0.47 |
| F1 | 0.91 ± 0.06 | 0.56 ± 0.41 |
| Norm. Effort | 0.94 ± 0.17 | 1.70 ± 0.49 |

Table 3: The average drifting detection results for Drebin and IDS2018 datasets using the softmax outputted probability. For each evaluation metric, we report the mean value and the standard deviation.

with a prediction probability of 1.0 (*i.e.*, they are not considered as drifting). We report the average results on Drebin and IDS2018 datasets in Table 3. We can observe that the softmax outputted probability works OK on the Drebin dataset (91% of F1 score) but only achieves 56% of F1 score on the IDS2018 dataset, validating its inefficiency on detecting drifting samples.

## References


[1] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. of NDSS*, 2014.

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

[3] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *Prof. of ICISSP*, 2018.