ALPHA: Active Learning with PAC-Bayesian Theory for Android Malware Detection

Yaomengxi Han*
Technical University of Munich
Munich, Germany
maxcharm.han@tum.de

Yunru Wang*
Ludwig-Maximilians-Universität München
Munich Center for Machine Learning
Munich, Germany
yunru.wang@ifi.lmu.de

Debarghya Ghoshdastidar
Technical University of Munich
Munich Center for Machine Learning
Munich, Germany
ghoshdas@cit.tum.de

Johannes Kinder
Ludwig-Maximilians-Universität München
Munich Center for Machine Learning
Munich, Germany
johannes.kinder@lmu.de

Abstract

Learning-based malware detection for Android is sensitive to multiple forms of distribution drift. Temporal drift includes (i) the emergence of new families and (ii) variant-level evolution within existing families, while spatial drift manifests as (iii) population-level shifts in the overall app distribution. Although recent work applies active learning to mitigate the resulting performance degradation, it commonly relies on margin-based sampling, which prioritizes samples near the decision boundary and lacks theoretical grounding for improving adaptation to test distribution.

We propose ALPHA, a drift-aware active learning framework guided by PAC-Bayes theory, to mitigate this limitation. The PAC-Bayes bound decomposition expresses test-domain error as the combination of the empirical training error and three discrepancy terms capturing, respectively, out-of-support mass, boundary instability, and distributional density mismatch. These components align closely with the three drift types of Android malware, allowing us to derive active learning strategies that are theoretically motivated by the decomposition. Using this perspective, we first examine two commonly used Android malware benchmarks and show that they exhibit substantially different degrees of distribution drift. Evaluating ALPHA we show that it improves the classification F1-score by $15.4-22.8\%$ over uncertainty-based sampling strategies. Further, ALPHA achieves greater gains on the high-drift benchmark, and we validate this relationship through statistical analysis. Finally, through targeted case studies, we provide empirical evidence that connects the PAC-Bayes decomposition to the three forms of drift observed in the evaluated Android malware datasets.

CCS Concepts

• **Security and privacy → Domain-specific security and privacy architectures**.

Keywords

Android Malware Detection, Distribution Drift, Active Learning, PAC-Bayes Theory, Machine Learning

ACM Reference Format:
Yaomengxi Han, Yunru Wang, Debarghya Ghoshdastidar, and Johannes Kinder. 2026. ALPHA: Active Learning with PAC-Bayesian Theory for Android Malware Detection. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '26), June 1–5, 2026, Bangalore, India.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779208.3807494

1 Introduction

Android malware detection has increasingly relied on machine learning techniques to cope with the scale and diversity of modern mobile threats [6, 27, 46]. By learning patterns from large collections of static [1, 6] or dynamic features [37, 41], detectors have achieved promising accuracy under controlled settings. However, Android malware evolves continuously, and detectors deployed in practice face distributions with substantial drift from training data [31].

Prior studies have shown that this drift arises in several distinct forms [10]. First, unseen-family temporal drift arises when new Android capabilities create fresh attack surfaces and spawn malware families with novel abuse logic. For example, the adoption of mobile payment and cryptocurrency apps has coincided with Android banking trojans like Anubis, leveraging overlay attacks and accessibility abuse [30]. Second, even within seen families, repackaging and obfuscation can induce drift by producing variants that increasingly resemble benign apps or obscure their malicious payloads, pushing them toward the decision boundary and causing unstable classification. For example, AnserverBot samples were observed to repackage different benign applications, such as paid apps `com.camelgames.mxmotor`, while embedding their malicious payloads behind benign-looking package names such as `com.sec.android.provider.drm` [49]. Finally, at the population level, spatial drift arises from highly imbalanced and changing malware family dominance in the Android ecosystem. Large-scale analysis of more than 1.28 million samples shows that some families dominate only during specific periods before declining, such as PIRATES in mid-2013 and large adware families like AIRPUSH and KUGUO peaking in particular years rather than remaining consistently dominant over time [38].

*Both authors contributed equally to this research and are listed in alphabetical order.

This work is licensed under a Creative Commons Attribution 4.0 International License.
ASIA CCS '26, Bangalore, India
© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2356-8/26/06
https://doi.org/10.1145/3779208.3807494

Recent work attempts to reduce the impact of drift through representation learning, training encoders on Drebin-style raw features [6] or graph-based analyses to obtain more robust feature spaces [9, 17, 47, 48]. These representations can reduce the impact of superficial edits, but once the underlying distribution begins to drift, the learned representation itself becomes outdated and thus unable to support reliable malware detection in the long run.

Active learning (AL) provides a more direct mechanism for adapting detectors to the distribution drift, and is orthogonal to representation learning methods [9, 12, 32]. Generally, in practical deployments, detectors periodically request labels for a small number of newly observed apps and update both the encoder and classifier accordingly. This creates a natural opportunity to select the most informative samples for labeling, highlighting the importance of a principled sampling strategy. However, most AL sampling methods used in Android malware rely on heuristics, primarily margin-based scores [7, 9, 12, 22] or out-of-distribution (OOD) detectors [18, 45]. Margin-based scores mainly identify samples that lie near the current decision boundary, while OOD detectors flag apps that appear unusual with respect to their position in the feature space. Although these may succeed in limited settings, they do not distinguish between different underlying causes of drift. This limits their ability to ensure that labels are allocated to the test apps affected by drift in a way that best reduces the test-domain error.

This gap motivates the need for a principled basis for drift-aware sample selection. We draw on the PAC-Bayes family of generalization bounds [13, 14, 26], which provide theoretical guarantees to relate a model's performance on the training distribution to its expected performance on unseen test data. When extended to settings with distribution shift, the resulting bound decomposes the difference between the expected test error, which measures the classifier's misclassification rate on unseen apps, and the training error, which measures the misclassification rate on the labeled training set, into three components [14]. For clarity, Fig. 1 provides a schematic illustration of this decomposition, where Fig. 1a shows the assumed training (red) and test (blue) distributions.

The three components are: (i) an *out-of-support (OOS) term*, measuring the fraction of test apps with feature patterns outside the region of feature space occupied by training data (Fig. 1b). (ii) a *disagreement term*, reflecting how unstable the classifier's predictions are on the unlabeled test set. Samples with high disagreement typically lie near the decision boundary (Fig. 1c). And (iii) a *divergence term*, measuring how differently the train and test data are distributed across the shared feature space (Fig. 1d). These terms provide a structured view of the decomposition of the test-domain error. As the decomposition is defined on the joint feature–label distribution, it implicitly covers various forms of shift studied in machine learning theory (e.g., covariance shift, label shift, and concept drift). This generality makes the framework robust to diverse drift phenomena, rather than tailored to a specific shift pattern.

Interestingly, the decomposition aligns remarkably well with the three drift patterns observed in Android malware. The OOS term captures the effect of unseen families; the disagreement term reflects within-family changes that push variants toward the decision boundary; and the divergence term captures population-level shifts driven by non-stationary and imbalanced malware families. This conceptual alignment provides a compelling theoretical basis for drift-aware adaptation strategies in Android malware.

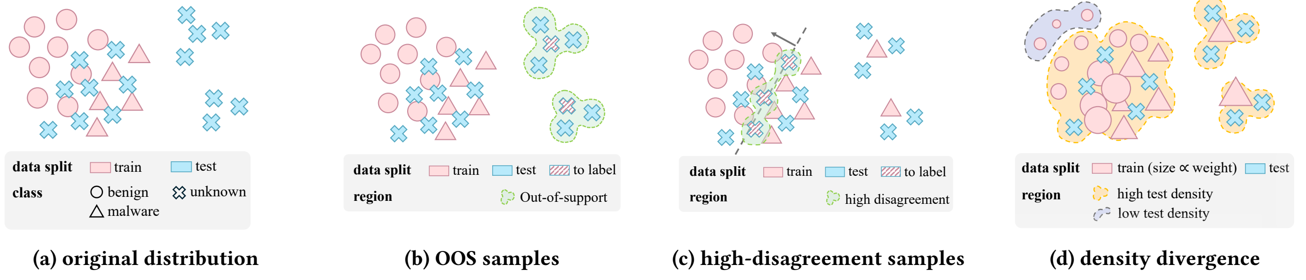
However, applying PAC-Bayes theory to real-world Android malware raises two fundamental challenges. First, in practice, the PAC-Bayes bound is often treated as a training loss and optimized to reduce the upper bound on the test-domain error, yet the three forms of drift in Android occur at different scales. As a result, collapsing them into a single loss mixes incompatible signals and prevents any form of drift from being handled effectively. Second, the decomposed PAC-Bayes terms are defined at the distribution level, whereas AL generally operate at the sample level. This mismatch in granularity requires us to convert distribution-level terms into sample-level criteria only when theoretically sound, and incorporate the remaining distribution-level adjustments into the AL update when no meaningful per-sample interpretation exists, so that the effectiveness of the bound is preserved.

These challenges motivate ALPHA (**A**ctive **L**earning with **P**AC-Bayesian T**H**eory on **A**ndroid), which addresses both issues through a hierarchical, coarse-to-fine structure that mirrors the decomposition. Our framework instantiates the PAC-Bayes terms into three stages aligned with the AL loop. First, we operationalize the OOS term using a theory-derived density threshold to identify an OOS region in the test distribution, rather than ranking individual samples, thus preserving the distribution-level view. Second, we instantiate the disagreement term with a closed-form surrogate that provides a principled disagreement score, without requiring multiple separately trained models as in conformal-based approaches [7, 19]. Third, we instantiate the divergence term via nearest-neighbor reweighting [24] to account for long-term imbalances in malware family prevalence. Arranged in a coarse-to-fine order, these stages translate PAC-Bayes decomposition into an AL-compatible procedure while preserving sample-level actionable signals and distribution-level effectiveness. In particular, we make the following contributions:

- We propose ALPHA, a drift-aware AL framework that operationalizes the PAC-Bayes decomposition for Android malware detection. By aligning each theoretical term with a concrete form of drift, ALPHA provides a principled, interpretable basis for AL, bridging the gap between PAC-Bayes theory and practical Android malware adaptation for the first time.
- Building on the discrepancy terms, we derive two drift indicators to reassess widely used Android malware benchmarks. Our analysis reveals that APIGraph, one of the most widely adopted datasets, exhibits minimal drift, which limits its ability to assess the effectiveness of drift-adaptation methods.
- We conduct extensive experiments comparing ALPHA with classical AL-based Android malware detection. The results show that ALPHA yields consistent improvements in F1 and FNR under substantial drift and limited budgets. Notably, ALPHA reduces FNR by 20% and improves F1 by 21% over uncertainty sampling in Drebin feature space under the smallest budget. It also reduces runtime costs by 25%–50% compared to SOTA strategy HCC [9]. Case studies further validate the correspondence between decomposed terms and Android drift forms.

Our code and dataset are publicly available[1].

[1]Repository: https://github.com/lmu-plai/ALPHA.git



(a) original distribution (b) OOS samples (c) high-disagreement samples (d) density divergence

Figure 1: Drift-aware PAC-Bayes Decomposition.

2 Background: PAC-Bayes Bound Theory

Adapting Android malware detectors in the presence of distribution shift is challenging due to the diverse forms of drift and their interactions. Existing heuristics-based sampling strategies can improve performance locally, but they do not explain why certain samples are chosen for labeling or how they relate to specific types of drift. Without this understanding, labeling effort may be misallocated, leaving the most critical drifted test samples unaddressed.

PAC-Bayes theory [26] provides a principled framework to bridge this gap. It bounds a model's expected error on unseen distributions by explicitly relating it to the empirical training error and a measure of distribution shifts. By decomposing the expected test error into interpretable components, this theory provides a formal basis for guiding active learning (AL) sampling.

2.1 Classical PAC-Bayes Generalization Bound

We first review PAC-Bayes bounds without drift. Let $\mathcal{D}$ be the unknown distribution over feature-label space, where the training $S_{\text{train}} = \{(x_i, y_i)\}_{i=1}^{n}$ and the test set $S_{\text{test}} = \{(x_i, y_i)\}_{i=1}^{m}$ are drawn. PAC-Bayes theory bounds the inaccessible *true error* of classifier f_w with loss function ℓ, defined as $L(w) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(y, f_w(x))]$.

While test error is an unbiased proxy for $L(w)$, it lacks formal guarantees. PAC-Bayes theory bounds $L(w)$ with empirical error $\hat{L}(w) = \frac{1}{n}\sum_{(x,y)\in S_{\text{train}}} \ell(y, f_w(x))$, where w follows the posterior Q. McAllester et al. [26] prove that with probability at least $1-\delta$:

$$\mathbb{E}_{w\sim Q}[L(w)] \leq \mathbb{E}_{w\sim Q}[\hat{L}(w)] + \sqrt{\frac{\mathrm{KL}(Q\|P) + \ln(1/\delta)}{n}}, \quad (1)$$

where P is the *prior* distribution over model parameters before seeing any data, and Q is the *posterior* distribution after training. Eq. 1 shows that the train-test gap is small when the posterior stays close to the prior (measured by the KL divergence) and the size of S_{train} is large. This provides a formal and interpretable measure of generalization that can be extended to settings with distribution shift, motivating our drift-aware malware detection framework.

2.2 PAC-Bayes Bounds Under Distribution Shift

With distinct training and test distributions ($\mathcal{D}_{\text{train}} \neq \mathcal{D}_{\text{test}}$), the adapted PAC-Bayes bound [14] shows with probability at least $1-\delta$:

$$\mathbb{E}_{w\sim Q}[L(w)] \leq 2\eta_{\text{oos}} + d_Q(S_{\text{test}}) + 2\beta(\mathcal{D}_{\text{test}}\|\mathcal{D}_{\text{train}})\sqrt{e_Q(S_{\text{train}})} + R_{\text{PB}}, \quad (2)$$

where the η_{oos}-term captures the percentage of the test distribution outside the feature-label region covered by the training set:

$$\eta_{\text{oos}} = \Pr_{(x,y)\sim\mathcal{D}_{\text{test}}}[(x,y) \notin \mathrm{supp}(\mathcal{D}_{\text{train}})]; \quad (3)$$

and the disagreement term $d_Q(S_{\text{test}})$ measures how often two classifiers sampled from Q predict differently on the same test sample:

$$d_Q(S_{\text{test}}) = \mathbb{E}_{w,w'\sim Q}\left[\frac{1}{m}\sum_{x\in S_{\text{test}}} \ell\left(f_w(x), f_{w'}(x)\right)\right]; \quad (4)$$

the divergence measure $\beta(\mathcal{D}_{\text{test}}\|\mathcal{D}_{\text{train}})$ characterizes the distance between the train and test domains by taking the expected ratio of the test density $p_{\text{test}}(x,y)$ to the train density $p_{\text{train}}(x,y)$; $e_Q(S_{\text{train}})$ measures the joint misclassification rate of two classifiers sampled from Q over training set. They are defined respectively as:

$$\beta(\mathcal{D}_{\text{test}}\|\mathcal{D}_{\text{train}}) = \sqrt{\left(\mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{train}}}\left[\left(\frac{p_{\text{test}}(x,y)}{p_{\text{train}}(x,y)}\right)^2\right]\right)}, \quad (5)$$

$$e_Q(S_{\text{train}}) = \mathbb{E}_{w,w'\sim Q}\left[\frac{1}{n}\sum_{(x,y)\in S_{\text{train}}} \ell(f_w(x), y)\cdot \ell(f_{w'}(x), y)\right], \quad (6)$$

the final term R_{PB} is the PAC-Bayes regularizor similar to the last term in Eq. 1, ensuring the posterior Q stays close to the prior P.

Although not previously used for Android malware detection, these components can quantify how detectors trained on past malware may perform on emerging samples. The three terms in Eq. 3-5 align with the three phenomena arising under drift: test apps with behaviours unseen in historical datasets, instability in model predictions on unfamiliar variants, and shifts in feature distributions caused by imbalanced malware family prevalence over time.

2.3 Challenge and Motivation

Applying the PAC-Bayes bound decomposition to AL for Android malware detection introduces two practical challenges.

C1: In theoretical work, PAC-Bayes bounds are typically used as loss functions optimized during training [13, 14]. However, such losses are only effective when the different components are properly balanced. The divergence $\beta(\mathcal{D}_{\text{test}}\|\mathcal{D}_{\text{train}})$ is inherently a constant because both $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ are fixed. Thus, using Eq. 2 as a loss objective with the divergence as a scaling factor requires that all decomposed terms in the loss be of comparable scale. Although the terms $d_Q(S_{\text{test}})$ and $\sqrt{e_Q(S_{\text{train}})}$ naturally take values in $[0,1]$, $\beta(\mathcal{D}_{\text{test}}\|\mathcal{D}_{\text{train}})$ can in principle be arbitrarily large or small (approaching 0 or ∞). This imbalance can bias the objective toward one term. Therefore, careful hyperparameter tuning is needed to

achieve a reasonable scale in these losses. However, such tuning may compromise the theoretical guarantee of the bound due to its empirical nature. To avoid this, we adopt an alternative approach: instead of treating the PAC-Bayes bound as a global loss, we minimize its components hierarchically, in a multi-stage procedure.

C2: The decomposition is defined at the joint distribution level (see Eq. 3-5), while AL requires sample-level scores. Meanwhile, only the disagreement term $d_Q(S_{\text{test}})$ admits stable sample-level surrogates. The other quantities, η_{oos} and $\beta(\mathcal{D}_{\text{test}} \| \mathcal{D}_{\text{train}})$, do not decompose into meaningful per-sample contributions, but inherently require distribution-level adjustments. Applying the decomposition in AL thus demands a hybrid design: sample-level criteria when feasible and distribution-level updates when not.

These challenges guide the design of our drift-aware AL framework, which is developed in detail in the following sections.

3 ALPHA

3.1 Overview

As shown in Fig. 2, in the standard active learning (AL) workflow, the detector repeatedly encodes newly observed apps, selects a predefined budget to label, and updates the encoder and classifier with labeled samples. Our framework focuses on the sampling stage and can be applied to different feature spaces, although the decomposition details require calibration for different representations.

Guided by the decomposition of drift-based PAC-Bayes theory (Eq. 2-6), we design three signals in ALPHA, each corresponding to a form of drift commonly observed in Android malware. We process these signals in a coarse-to-fine structure so that larger forms of drift are corrected before finer adaptations. First, we identify hard drift related to unseen families (Fig. 2, red). We approximate the training density function by fitting a kernel density estimator (KDE) on training embeddings. A theory-guided threshold then flags test samples in low-density regions, circumventing empirical p-value tuning. We cluster the OOS candidates and request labels only for centroids to update the encoder and classifier. Next, we address within-family drift by labeling samples with unstable predictions (blue). We use a theory-derived closed-form surrogate to estimate test-time disagreement from classifier output probabilities, and query labels for high-disagreement samples to update the encoder. Finally, we address imbalanced malware families through nearest-neighbor reweighting in the representation space, adjusting training sample weights to better match the test distribution (orange). The classifier is retrained using these weights.

Under a fixed labeling budget, the two sampling stages share annotations, and we tune their ratio on a small validation set to balance hard drift against within-family evolution.

3.2 Unseen-Family Sample Selection

New Android malware families often arise when attackers introduce or substantially rework core malicious capabilities, yielding distinct behavior patterns (e.g., C&C and privilege abuse) [10]. Detecting new-family samples early is significant as downstream adaptation cannot reliably correct errors on families the model has never seen. Fig. 1b shows that such samples generally lie outside the region of historical apps, aligning with the η_{oos}-term in the drift-aware PAC-Bayes bound, which reflects the percentage of the test distribution that falls outside the region of the training distribution. To adapt this insight in AL, we introduce a sample-level criterion that estimates whether an individual test app is likely to lie outside the train domain while preserving the distribution-level effectiveness of the η_{oos}-term by a novel thresholding strategy.

Many learning-based approaches for Android malware analysis rely on learned representations [9, 16, 28, 48] whose dimensionality, while moderate for representation learning, is already prohibitively high for reliable density estimation, even with commonly used embedding sizes such as 128 dimensions. This limitation is fundamental rather than algorithmic. Due to the curse of dimensionality, the sample complexity of nonparametric density estimation grows exponentially with the data dimension [34], rendering direct density estimation statistically unstable even at moderate dimensions (e.g., dimension $d > 20$). Moreover, because test samples are unlabeled, we cannot perform density estimation on the joint distribution and must instead operate on the feature marginals. Considering all the factors above, we use principle component analysis (PCA) to project the feature embeddings of both training and test apps into a lower-dimensional space, which still preserves structural patterns while reducing the sparsity of the original representation.

On this reduced space of dimension d, we fit a kernel density estimation (KDE) $\hat{p}^X_{\text{train}}(\cdot)$ with the training set. Then a target app is flagged as out-of-support if its estimated density falls below a threshold. Prior approaches [3] often compute p-values from the sample-level density estimates, rank the test points accordingly, and label those with the smallest p-values as out-of-support. However, this sample-ranking procedure does not preserve the distribution-level notion of η_{oos} and it offers no guarantee that with the same labeling budget, this term will decrease in the most effective fashion.

To address this, we derive a theoretically grounded threshold by upper-bounding the estimation error of the KDE. First, we consider the supremum deviation between the KDE and the true marginal density function of the training distribution p^X_{train} as: $\Delta = \sup_{x \in \text{supp}(\mathcal{D}^X_{\text{test}})} \left|\hat{p}^X_{\text{train}}(x) - p^X_{\text{train}}(x)\right|$. We then derive a probabilistic upper bound on the sup-deviation Δ (see Lemma A.1 in Appendix A): with probability greater than 1-δ, we have that $\Delta \leq \tau$. Hence, we set the threshold as τ and flag any test app x with $\hat{p}^X_{\text{train}}(x) < \tau$ as out-of-support. The strategy of adaptive threshold selection provably bounds the error of the calculated out-of-support ratio, as detailed in Theorem A.2.

The intuition of the choice of threshold is as follows: any test sample x that we did not reject will have an estimated density $\hat{p}^X_{\text{train}}(x) \geq \tau$. As the KDE can overestimate the true density by at most Δ, the true density of this point must satisfy $p^X_{\text{train}}(x) \geq \hat{p}^X_{\text{train}}(x) - \Delta \geq \hat{p}^X_{\text{train}}(x) - \tau \geq 0$.

In this way, any app with $\hat{p}^X_{\text{train}}(x) \geq \tau$ has non-negative density under the training distribution and therefore is not treated as an OOS sample. Test apps outside this region are thus natural candidates for belonging to unseen malware families.

Finally, we cluster all flagged apps with KMeans++ and request labels only for the cluster centroids, shown as the red striped cross markers inside the green region in Fig. 1b. This preserves the goal of reducing the out-of-support mass through labeling while keeping the labeling cost low, and the selected representatives summarize the principal new behaviors introduced by unseen families.

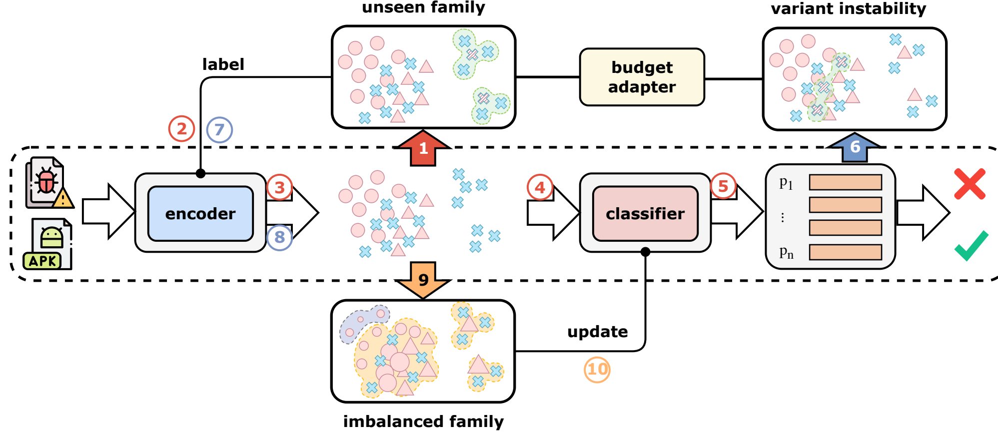


Figure 2: Overview of ALPHA. The dashed box shows the baseline Android malware detection pipeline. The active learning update is organized into three sequential stages. Steps 1–5 (red) operate in the feature space to select samples from previously unseen malware families, which are used to retrain both the encoder and the classifier. Steps 6–8 (blue) focus on variant instability by selecting samples with unstable classifier predictions, and use them to further refine the encoder. Steps 9–10 (orange) address imbalanced malware families via reweighting in the learned feature space, followed by retraining the classifier with a weighted loss. A budget-ratio adapter dynamically allocates the labeling budget between the first two stages.

3.3 Variant-Instability Sample Selection

Prior work shows that malware variants created via repackaging and obfuscation can shift observable features toward benign apps, leading to unstable classifier behavior [35]. This drift is well recognized in Android malware detection, where margin-based criteria (e.g., uncertainty or loss) are widely used in AL or sample rejection [7, 9, 17], and invariant representation learning is adopted to retain stable family-level features while discarding fake changes [48].

This "within-family drift" is also caught by the disagreement term (Eq. 4) in the decomposition. While margin-based criterion usually tracks the confidence of a single classifier, the disagreement term measures how often classifiers drawn from a posterior distribution disagree on a test sample. This ensemble style focuses directly on variants that induce unstable decisions under model perturbations, rather than on raw scores from a single model.

A related line of work [7, 19] uses conformal prediction to obtain calibrated p-values for drift detection. However, these methods require training multiple classifiers or repeatedly perturbing the model to approximate a nonconformity distribution, which is computationally expensive in AL settings. In contrast, we avoid explicit ensembles by leveraging the closed-form surrogate of the PAC-Bayes disagreement term proposed in [14]. For a test app x, let $q(x) = \Pr_{w\sim Q}(\text{sign}(w \cdot x) = 1)$ denote the probability that a classifier drawn from the posterior Q predicts the malware label. The probability that two independent classifiers sampled from Q disagree on x is then $d_Q(x) = 2\,q(x)(1 - q(x))$.

Under the Gaussian posterior assumption, the prediction probability $q(x)$ can be computed directly from the closed-form distribution of $w \cdot x$, without training or storing an explicit ensemble. This assumption is reasonable for Android malware classifiers trained with l_2-regularized objectives, whose PAC-Bayes posterior concentrates around learned weights in approximately Gaussian form.

In practice, we directly use $d_Q(x)$ as a per-sample score. Apps with larger $d_Q(x)$ lie closer to the decision boundary under posterior perturbations and are thus prioritized for labeling, as the samples in the shaded green region in Fig. 1c.

3.4 Imbalanced-Family Reweighting

Spatial drift in Android malware is commonly described as changes in the malware–benign ratio [31], but the malware population itself is highly long-tailed and non-stationary across families [38]. Such shifts in family prevalence change the effective training distribution, even when the malware–benign ratio remains stable. Thus, we focus on this finer-grained, family-level form of spatial drift.

This family-level spatial drift can persist even after the first two stages, where newly labeled samples are added to the training set. The resulting family composition may still differ from the test distribution, biasing the classifier toward dominant families. Theoretically, this residual mismatch reflects local density differences captured by the divergence term in the PAC-Bayes bound.

We provide a sample-level view to understand the divergence term. For each test app $x_i \in S_{\text{test}}$, we identify its k-nearest neighbors among both training and test sets. Let n_i and m_i be the number of its train and test neighbors ($m_i + n_i = k$). The ratio $\frac{n_i}{m_i}$ directly shows whether the area around x_i contains fewer training apps than test apps. A small ratio indicates that this region is now populated predominantly by test samples, signaling a local density increase in the test distribution and hence a shift in feature prevalence.

Although $\frac{n_i}{m_i}$ provides a local view of density mismatch, it does not identify which target samples should be relabeled. Relabeling a few points with low ratios does not correct the mismatch. Addressing this form of drift therefore requires adjusting the train distribution globally rather than selecting specific samples.

Therefore, we leverage the idea of Nearest-Neighbor Reweighting (NNW) [24]. Similarly, NNW assigns each train app a weight proportional to how often it appears in the k-nearest-neighbor sets of test apps. Training apps close to dense test regions therefore receive larger weights, while points lying in outdated regions receive smaller ones, as shown in Fig. 1d where larger sizes mean larger weights. This shifts the effective training distribution towards the test distribution. During the subsequent classifier update in the AL pipeline, these weights are incorporated directly into retraining.

By amplifying the influence of underrepresented families and down-weighting obsolete regions, NNW mitigates spatial drift and stabilizes model updates under evolving malware distributions. This effect is strengthened when unseen families and unstable variants are removed, as these samples would otherwise disrupt the density estimates used for reweighting. Their removal allows reweighting to target residual family-level imbalance more accurately.

4 Evaluation

This section provides a comprehensive evaluation of ALPHA. Section 4.1 outlines our methodology, including datasets, evaluation metrics, baselines, and active learning (AL) setup. Based on this framework, we investigate the following four research questions:
RQ1. Are existing Android malware benchmarks suitable for evaluating drift-aware detection methods? (Section 4.2)
RQ2. How well does ALPHA perform under distribution drift compared to competing strategies? (Section 4.3)
RQ3. What is the computational overhead of ALPHA compared to competing strategies? (Section 4.4)
RQ4. What is the contribution of each component in ALPHA? And how sensitive is ALPHA to hyperparameters? (Section 4.5)

Finally, Section 4.6 presents case studies that verify how each stage of ALPHA reflects its intended drift-removal intuition.

4.1 Methodology

4.1.1 Dataset. We use two publicly available Android malware datasets used in prior work [9], APIGraph [47] and an AndroZoo-based dataset [4]. Both datasets include malware family labels and follow a monthly temporal split. APIGraph spans 2012–2018, while AndroZoo covers 2019–2021.

4.1.2 Evaluation Metrics. In our setting, malware samples are treated as the positive class and benign samples as the negative class. We evaluate detection performance using False Negative Rate (FNR), False Positive Rate (FPR), and F1 score. FNR measures missed malware, while FPR reflects false alarms on benign apps. We additionally report F1 to summarize performance under class imbalance.

4.1.3 Competitors. AL for malware detection typically involves three elements: input features, a binary-classifier, and a query strategy for sample selection and model update. Our method focuses on the query strategy and is compatible with different feature representations and classifiers.
Input Features. We consider two feature settings: (i) Drebin features [6], where apps are represented as high-dimensional sparse binary vectors, which we further reduce to 500 dimensions with truncated SVD; and (ii) HCC embeddings [9], learned via a hierarchical contrastive framework that encourages clustering within malware families and separation between malware and benign apps.
Query Strategy. We compare three strategies: (i) Hierarchical-Contrastive-Loss Sampling (Pseudo Loss), proposed together with HCC [9], uses the hierarchical contrastive loss as a pseudo loss to select samples that are most inconsistent with the embedding structure. (ii) Uncertainty Sampling (UNC), selects samples whose predicted class probability is closest to 0.5, i.e., $unc(x) = 1 - |\Pr(y = 1 \mid x) - 0.5|$, so higher values indicate more uncertain predictions. (iii) ALPHA, our hierarchical coarse-to-fine update strategy, is inspired by PAC-Bayes and applies three stages to address unseen families, within-family drift, and family imbalance.

HCC [9] evaluates multiple baseline strategies (e.g., CADE [45], Transcendent [7]) and identifies uncertainty sampling as the strongest baseline aside from HCC pseudo loss. We thus include uncertainty sampling and omit other alternatives.
Classifier. For ALPHA and UNC, we use a linear SVM classifier, as prior studies have shown that it often outperforms alternatives such as MLP and GBDT in malware detection [9, 17]. For Pseudo Loss, we use its jointly trained classifier.

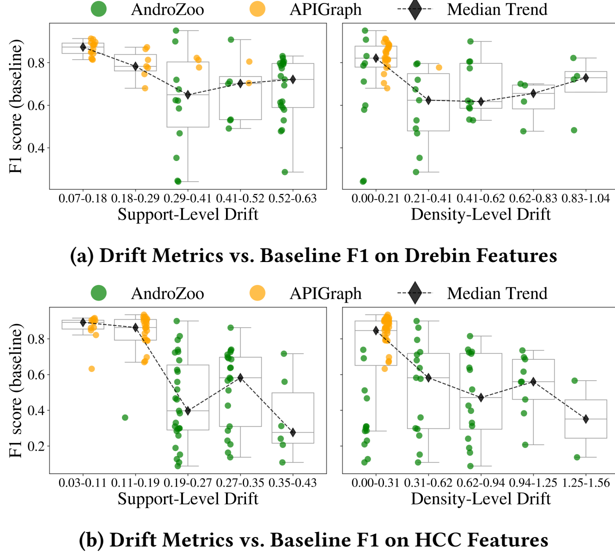
4.1.4 Evaluation Protocol. Following [9], we adopt a temporal split for both datasets in 4.1.1: the first year for training, the first half of the second year for validation, and the remaining data for testing. An initial classifier is trained on the training set. The validation period is used to tune the budget allocation between the first two stages by varying the ratio of the first stage in $[0, 1]$ with a step size of 0.2, while fixing the total budget to 50, 100, 200, or 400. Other hyperparameters (KDE bandwidth, PCA dimension, and neighborhood size) are detailed in Appendix B. During testing, we follow a sequential time-window protocol with a window size of 1 (monthly) or 12 (yearly). For each window, we evaluate the current detector, select the fixed budget of samples for labeling, and retrain both the encoder and classifier. The updated encoder and classifier are then evaluated on the next window. This setup helps to quantify performance degradation under non-stationary drift scenarios.

4.2 RQ1: Evaluation of the Benchmarks

Many studies on drift in Android malware detection [2, 5, 9, 18] rely on the same well-known Android benchmark datasets. These datasets were not designed to capture distribution drift, and some of the steps used to build them can even reduce changes that create real drift. This raises a significant question: **Can we develop a general evaluation procedure to quantify distribution shift in existing malware benchmarks, and further to assess whether they are appropriate for studying drift-aware methods?**

This question motivates a two-part analysis. Section 4.2.1 proposes two metrics to quantify drift magnitude, using the baseline detector's performance degradation as a reference, applicable across datasets. Section 4.2.2 examines how the relative performance of ALPHA and competing methods against baseline varies with drift.

4.2.1 Correlation between drift metrics and baseline performance. To quantify drift within a benchmark, η_{oos} (Eq. 3) and $\beta(\mathcal{D}_{\text{test}} \| \mathcal{D}_{\text{train}})$ (Eq. 5) present themselves as natural candidates. These two terms



(a) Drift Metrics vs. Baseline F1 on Drebin Features

(b) Drift Metrics vs. Baseline F1 on HCC Features

Figure 3: Baseline performance vs. drift metrics across Drebin (top) and HCC (bottom). (Left) Support-level drift correlates with reduced median F1 scores and increased variance. (Right) Density-level drift reveals benchmark separation: low-drift APIGraph (orange) remains stable, while high-drift AndroZoo (green) shows great degradation and variance.

inherently capture different aspects of distribution shift: η_{oos} measures the mismatch in support between the train and test sets, while β quantifies the divergence in the density functions across the two domains. Unlike the signals used for selection, now we focus on distribution-level quantities rather than individual samples. Since this is a post-hoc evaluation, we have access to the labels of both train and test sets in the benchmark, allowing us to go beyond marginal distributions in estimating both terms. Therefore, we propose two complementary metrics to measure the degree of the shift:

- **Support-level drift.** This metric is computed similarly to Section 3.2, but with the KDE approximated over the labeled training set, and the threshold computed with labeled test samples.
- **Density-level drift.** Unlike the relative density term β, where values close to 1 indicate strong alignment, our metric is the Pearson-χ^2 divergence [39] between the joint feature-label distributions of the training and test sets, where larger values correspond to greater drift. This formulation is more interpretable and still captures the gradual shift in the joint density over time.

To evaluate whether these two drift metrics meaningfully reflect the distribution shift, we examine their correlation with the performance of a baseline detector. To obtain sufficient points to analyse, each benchmark is divided into 12 consecutive two-month chunks. For each train-test pair (i, j) where $j > i$, we compute the two metrics using labeled samples. The baseline detector is trained on chunk i and evaluated on chunk j. The resulting F1-score serves as a reference for joint distribution shift, since it reflects how the feature-label distribution shifts compromise the fixed decision boundary.

Fig. 3 plots drift magnitude (x-axis) versus baseline F1 score (y-axis) for all train-test pairs, organized by feature space (top: Drebin, bottom: HCC) and metric (left: support-level, right: density-level). To reveal the overall correlation, we aggregate points into equal drift intervals and visualize the F1 distribution via box plots (showing the middle 50% of scores). A dashed line connects the medians across intervals, tracking the performance as drift increases. We summarize the key observations from Fig. 3 as follows:

Two benchmarks show noticeably different drift patterns. Across feature spaces, APIGraph (orange) consistently clusters within low-drift intervals (typically in $[0, 0.3]$) for both drift metrics. In contrast, AndroZoo (green) spans a much wider range across the x-axis, exhibiting significantly higher drifts at both the support and density levels. Quantitatively, the average drift on AndroZoo is approximately 2.4× that of APIGraph regarding both metrics.

Drift metrics correlate with performance degradation and increased instability. As drift increases, we observe a decline in model reliability,most notable at the separation between two benchmarks. The median F1 score declines steeply as the drift magnitude shifts from the APIGraph to the AndroZoo cluster, accompanied by a sharp expansion in variance. The metric not only tracks performance decay, but identifies the critical transition where the baseline enters a high-drift regime characterized by unstable results.

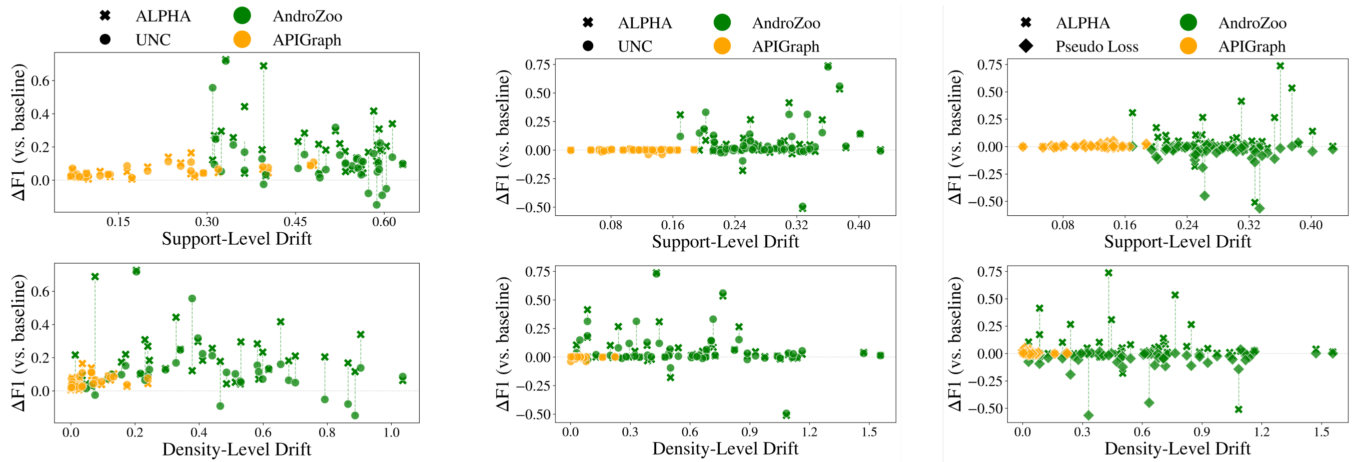
To quantify this relationship, we report the Spearman correlation (ρ) between each drift metric and the baseline detector's performance, which captures monotonic trends and is robust to outliers. We observe consistently strong negative correlations across both feature spaces: for support-level drift, $\rho = -0.52$ (Drebin) and -0.62 (HCC); for density-level drift, $\rho = -0.58$ (Drebin) and -0.53 (HCC). All correlations are statistically significant ($p < 0.001$), indicating that both metrics reliably reflect drift magnitude.

Overall, these metrics can quantify drift and distinguish between stable (e.g., APIGraph) and high-drift datasets (e.g., AndroZoo), the latter of which is more informative to evaluate drift-aware methods.

4.2.2 Impact of distribution shift on the relative improvement of ALPHA. Having established that the proposed drift metrics can reflect the magnitude of distribution shifts, we continue to study whether ALPHA also improves more from the baseline on benchmarks with larger drift, compared to its competing strategies.

Therefore, we extend the previous implementation by incorporating the AL pipeline. We train a baseline classifier on each training chunk i, and perform AL on all the subsequent test chunks $j = i + 1, \ldots, 12$, using three strategies: Pseudo Loss, UNC, and ALPHA. We then measure the performance gain of each method relative to the baseline. Consistent with previous analysis, we compute both drift metrics and the baseline F1 score for every pair.

The relationship between drift magnitude and model improvement is detailed in Fig. 4, where the x-axis shows the drift values and the y-axis shows the ΔF1, defined as the absolute difference between each method and the baseline. In both feature spaces, we observe a clear positive correlation between drift magnitude and performance gain (Fig. 4). This trend is most evident in the high-drift AndroZoo dataset (green), where performance improvements increase with drift, while the low-drift APIGraph dataset (orange) shows limited variance and minimal gains across all methods. In the Drebin feature space (Fig. 4a), both ALPHA and UNC outperform the baseline ($\Delta F1 > 0$) as drift increases; however, ALPHA consistently achieves larger gains, while UNC fluctuates around the



(a) Performance vs. Drift on Drebin Features

(b) Performance vs. Drift on HCC Features

Figure 4: Performance gain (ΔF1) of different strategies vs. drift. (Left Column) ALPHA (×) vs. UNC (•) on Drebin; (Middle Column) ALPHA (×) vs. UNC (•) on HCC; (Right Column) ALPHA (×) vs. Pseudo Loss (♦) on HCC. Rows plot against support-level (top) and density-level (bottom) drift. AndroZoo (green) exhibits significantly higher drift than APIGraph (orange).

Budget	Model		AndroZoo			APIGraph			APIGraph-year		
			Avg. Metrics (%)			Avg. Metrics (%)			Avg. Metrics (%)		
	Feature	Criterion	FNR	FPR	F1	FNR	FPR	F1	FNR	FPR	F1
50	Drebin	UNC	58.98 ± 0.00	0.37 ± 0.00	53.55 ± 0.00	16.65 ± 0.00	0.55 ± 0.00	88.19 ± 0.00	35.66 ± 0.00	0.59 ± 0.00	75.63 ± 0.00
	Drebin	**ALPHA**	46.78 ± 3.25	0.75 ± 0.07	64.58 ± 2.96	17.04 ± 0.31	0.76 ± 0.05	87.06 ± 0.29	31.52 ± 0.68	1.13 ± 0.15	76.11 ± 0.33
	HCC	UNC	41.09 ± 3.01	0.53 ± 0.04	68.39 ± 2.48	16.39 ± 0.77	0.51 ± 0.05	88.51 ± 0.56	40.18 ± 2.05	1.10 ± 0.10	70.03 ± 1.35
	HCC	**ALPHA**	38.51 ± 6.12	0.58 ± 0.09	69.98 ± 5.01	16.64 ± 0.98	0.52 ± 0.11	88.34 ± 0.61	36.42 ± 3.32	1.25 ± 0.34	72.10 ± 2.87
	HCC	Pseudo Loss	40.41 ± 0.11	0.60 ± 0.00	68.95 ± 0.08	16.59 ± 0.85	0.52 ± 0.09	88.39 ± 0.70	35.59 ± 2.63	1.21 ± 0.08	72.86 ± 1.68
100	Drebin	UNC	54.89 ± 0.00	0.52 ± 0.00	56.43 ± 0.00	15.16 ± 0.00	0.61 ± 0.00	88.83 ± 0.00	31.00 ± 0.00	0.71 ± 0.00	78.19 ± 0.00
	Drebin	**ALPHA**	40.05 ± 0.76	0.65 ± 0.04	69.34 ± 0.57	15.05 ± 0.16	0.68 ± 0.03	88.64 ± 0.11	31.12 ± 0.00	0.66 ± 0.00	78.38 ± 0.00
	HCC	UNC	38.84 ± 4.51	0.52 ± 0.03	70.70 ± 3.82	14.20 ± 0.46	0.46 ± 0.04	90.06 ± 0.45	34.87 ± 3.68	1.18 ± 0.20	73.51 ± 3.05
	HCC	**ALPHA**	35.59 ± 7.68	0.50 ± 0.12	72.64 ± 6.16	14.85 ± 1.51	0.44 ± 0.04	89.74 ± 0.89	30.96 ± 2.34	1.16 ± 0.14	76.42 ± 1.36
	HCC	Pseudo Loss	36.60 ± 0.14	0.54 ± 0.00	72.34 ± 0.09	14.37 ± 1.64	0.40 ± 0.03	90.21 ± 0.94	33.13 ± 1.56	1.11 ± 0.19	75.13 ± 1.45
200	Drebin	UNC	53.46 ± 0.00	0.54 ± 0.00	57.64 ± 0.00	15.72 ± 0.00	0.53 ± 0.00	88.84 ± 0.00	26.92 ± 0.00	0.70 ± 0.00	80.96 ± 0.00
	Drebin	**ALPHA**	42.80 ± 2.62	0.49 ± 0.04	67.87 ± 2.54	13.92 ± 0.07	0.63 ± 0.02	89.52 ± 0.04	28.74 ± 1.35	0.83 ± 0.09	79.11 ± 0.74
	HCC	UNC	35.64 ± 2.63	0.53 ± 0.05	72.99 ± 2.18	12.74 ± 0.81	0.41 ± 0.03	91.12 ± 0.46	30.17 ± 3.13	1.17 ± 0.19	76.86 ± 2.71
	HCC	**ALPHA**	34.85 ± 3.72	0.45 ± 0.01	73.58 ± 3.27	13.01 ± 1.46	0.44 ± 0.02	90.85 ± 0.90	26.41 ± 1.20	1.14 ± 0.07	79.51 ± 0.74
	HCC	Pseudo Loss	34.42 ± 0.07	0.50 ± 0.00	74.08 ± 0.04	12.26 ± 0.29	0.40 ± 0.04	91.45 ± 0.12	29.81 ± 1.19	1.15 ± 0.19	77.27 ± 1.38
400	Drebin	UNC	51.29 ± 0.00	0.53 ± 0.00	59.93 ± 0.00	15.25 ± 0.00	0.56 ± 0.00	89.01 ± 0.00	26.72 ± 0.00	0.71 ± 0.00	80.90 ± 0.00
	Drebin	**ALPHA**	39.30 ± 2.18	0.79 ± 0.05	69.20 ± 1.90	14.65 ± 0.24	0.61 ± 0.02	89.20 ± 0.17	26.71 ± 0.00	0.61 ± 0.00	81.34 ± 0.00
	HCC	UNC	32.25 ± 0.51	0.47 ± 0.05	75.72 ± 0.73	11.66 ± 0.77	0.40 ± 0.03	91.81 ± 0.44	27.79 ± 1.43	1.05 ± 0.16	78.99 ± 1.28
	HCC	**ALPHA**	31.99 ± 3.33	0.48 ± 0.05	75.54 ± 2.77	12.84 ± 1.84	0.41 ± 0.01	91.06 ± 1.06	24.14 ± 1.88	1.01 ± 0.12	81.47 ± 0.78
	HCC	Pseudo Loss	32.08 ± 0.01	0.45 ± 0.00	76.02 ± 0.01	10.83 ± 0.14	0.38 ± 0.02	92.35 ± 0.13	27.03 ± 2.08	1.08 ± 0.09	79.36 ± 1.12

Table 1: Stability (mean ± std) comparison of all strategies on all benchmarks. Note that under the Drebin feature, SVM UNC doesn't have any randomness, and ALPHA is also deterministic when hyperparameter tuning chooses a ratio of 0.0. **Blue marks the best mean values under Drebin feature and red marks those under HCC.**

baseline. In contrast, the APIGraph clusters remain stable, with little separation between methods. In the HCC feature space (Fig. 4b), the embedding compresses the observed drift, particularly reducing support-level variation in AndroZoo. Despite this effect, the overall trend persists: performance improvements still scale with drift magnitude, and ALPHA maintains more significant and stable gains compared to UNC and Pseudo Loss.

Taken together, these results address RQ1 from two complementary perspectives. First, **our drift metrics accurately capture the differences in distribution shift across benchmarks**. Various drift magnitudes across benchmarks result in differences in baseline performance, and our drift metrics reflect this relationship. In Apigraph, the baseline detector performs well, consistent with the minimal drift indicated by both metrics. In contrast, Androzoo exhibits substantially higher drift, accompanied by greater baseline degradation. The finding shows that our metrics capture meaningful drifts in AndroZoo and reveals the suitability of this benchmark for evaluating drift-aware methods. Second, the results indicate that **ALPHA exhibits stronger robustness to distribution shift**

compared with other methods. When the drift is small, different approaches perform similarly. In contrast, under substantial drift, ALPHA achieves the greatest performance gains, demonstrating its particular strength in high-shift scenarios.

4.3 RQ2: Effectiveness of ALPHA

Section 4.2 reveals varying degrees of distribution shift across benchmarks, motivating our second question: **how does ALPHA perform under different levels of drift?** To answer this, we compare ALPHA with competitors across three benchmarks (AndroZoo, APIGraph, and APIGraph-year) and two embedding spaces (Drebin and HCC) under four budgets, following [9]. Since APIGraph exhibits relatively low drift and is less suitable for evaluating drift-aware methods, we additionally construct APIGraph-year by increasing the time window to 12 months. Table 1 reports the mean and standard deviation of FNR, FPR, and F1 (in %) for each budget across five runs. The key observations are as follows:

In **AndroZoo** and **APIGraph-year**, with substantial distribution drift, ALPHA achieves the best F1 performance in most cases in both Drebin and HCC features. The advantage is especially pronounced with small budgets (50, 100), where ALPHA improves F1 by 0.13 for Drebin over the second-best method. Additionally, ALPHA consistently lowers FNR across all budgets, with reductions exceeding 0.12 in the Drebin space. In the lower-drift benchmark **APIGraph**, the top-ranked method varies across budgets and metrics, and the F1 differences among the best and worst methods are generally within 0.01. Regarding stability, ALPHA's variance is comparable to other baselines in APIGraph and APIGraph-year. In AndroZoo, ALPHA allocates more budget to the first stage, resulting in slightly higher variance than in the other datasets.

To complement Table 1, Appendix C presents the monthly performance of all strategies for AndroZoo and APIGraph with both feature embeddings with a budget of 200. Across time, ALPHA begins to outperform baselines at different speeds depending on the drift pattern in the dataset: rapidly under strong drift, as in AndroZoo, and more gradually under mild drift, as in APIGraph.

Interestingly, while HCC performs strongly on AndroZoo and APIGraph, it achieves lower F1 than Drebin on APIGraph-year. This may be due to the fact that contrastive representations are better aligned with gradual drift, whereas the year-level aggregation introduces more abrupt shifts. Despite this, ALPHA remains effective on APIGraph-year under both feature spaces, indicating its adaptability to different representations and drift patterns.

Overall, across benchmarks, embedding spaces, and evaluation settings, ALPHA consistently achieves competitive performance compared with existing methods. Its advantage becomes more pronounced under stronger distribution shift or limited labeling budgets, while remaining comparable to the best alternatives in low-drift settings, suggesting its potential applicability in real-world scenarios, where distribution shifts can be larger and more complex.

4.4 RQ3: Computational Overhead

To evaluate **the efficiency and practical overhead of ALPHA**, we analyze its computational cost from two perspectives, the overhead of the selection stage and the end-to-end runtime of the full pipeline. We report three metrics: runtime, maximum CPU memory usage (RSS), and peak GPU memory. Runtime is measured using wall-clock time to capture both CPU and GPU execution. All measurements are performed on the same server equipped with an NVIDIA H100 GPU. To ensure fair comparison across methods, we fix the GPU device for each run and limit the number of CPU threads used by OpenMP and BLAS libraries.

Budget	Criterion	APIGraph-year		
		Time/iter (s)	CPU RSS (GB)	GPU peak (GB)
50	UNC	0.10	2.76	0.82
	ALPHA	98.41	3.31	0.85
	Pseudo Loss	297.02	29.82	2.13
100	UNC	0.10	2.70	0.82
	ALPHA	98.09	3.31	0.85
	Pseudo Loss	284.59	29.83	2.13
200	UNC	0.10	2.76	0.82
	ALPHA	10.65	3.27	0.85
	Pseudo Loss	286.52	29.83	2.13
400	UNC	0.10	2.77	0.82
	ALPHA	10.81	3.28	0.85
	Pseudo Loss	328.24	28.99	2.13

Table 2: Selection-stage runtime on the HCC feature space.

4.4.1 *Selection-Stage Overhead.*

Table 2 reports selection-stage costs on APIGraph-year under different budgets and criteria on the HCC feature space (Drebin in Appendix E). We use APIGraph-year as it has the largest per-iteration sample size, providing a more representative evaluation of selection-stage overhead. The reported cost for the selection stage is averaged over all iterations on the full dataset. ALPHA consists of three components. The disagreement cost is comparable to UNC, while reweighting introduces only a small overhead (around 10s). The remaining cost mainly comes from the OOS stage, which includes PCA, KDE, and clustering. Among these, KDE dominates the runtime as it is fitted on the updated training set and scales with the budget, whereas clustering operates on a small subset and PCA remains fixed. At $B = 200$ and 400, the runtime drops significantly because the entire budget is allocated to disagreement, skipping the OOS stage. To assess scalability, we further evaluate larger budgets. Even with $B = 400$ and 80% allocated to OOS, the selection-stage runtime remains modest (107.18s), comparable to smaller-budget settings.

Overall, UNC is the most efficient strategy, with negligible runtime and low memory usage. ALPHA incurs additional overhead but remains significantly more efficient than Pseudo Loss. On HCC, it is typically 3–30× faster and uses 8–10× less CPU memory. The high cost of Pseudo Loss mainly stems from repeated model evaluations and KD-tree queries, suggesting that ALPHA provides a better efficiency–performance trade-off.

4.4.2 *End-to-end Runtime.*

We evaluate end-to-end runtime on APIGraph-Year (5 iterations) and APIGraph (60 iterations) on the HCC feature space, with ratio = 0.8 for ALPHA. Fig. 5 shows UNC is the most efficient, while ALPHA is consistently faster than Pseudo Loss. The gap between ALPHA and Pseudo Loss increases on APIGraph-year, where selection-stage overhead constitutes a larger fraction of the pipeline, but decreases on APIGraph as costs are amortized over more iterations with smaller per-iteration workload. This also indicates that the difference in selection-stage cost between ALPHA and Pseudo Loss becomes less pronounced in the

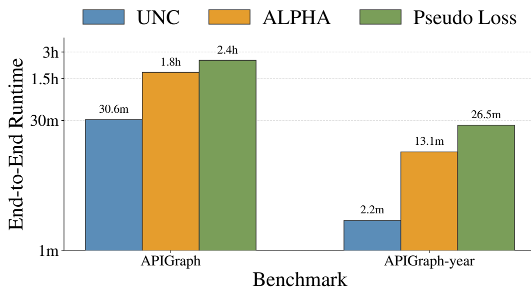


Figure 5: End-to-end Runtime Cost on APIGraph and APIGraph-year (budget 200)

end-to-end pipeline when the iteration size is small. Despite 60 iterations, ALPHA's total runtime remains modest (1.8h), indicating good scalability in practice. Overall, although UNC is more efficient, ALPHA's significant gains justify its manageable overhead. ALPHA also proves more practical than Pseudo Loss, offering comparable F1 with substantially lower runtime.

4.5 RQ4: Ablation Study

To evaluate the contribution of each component in ALPHA, and its sensitivity to hyperparameters, we conduct two ablation studies.

4.5.1 Structural Ablation. We evaluate three variants by removing one stage at a time: **No-OOS**, removing the unseen-family selection stage; **No-DIS**, removing the variant-instability stage; **No-DIV**, removing the imbalanced-family reweighting stage. Table 3 reports all three variants across feature spaces on **AndroZoo** (similar results on APIGraph are documented in Appendix D).

For **Drebin**, ALPHA outperforms all variants in F1 and FNR, with a mild FPR compromise. Removing the first or the last signal (No-OOS, No-DIV) causes clear degradation. With No-DIV, F1 drops by up to 17.05 points. These trends indicate that both unseen-family selection and imbalanced-family reweighting play significant roles in ALPHA's effectiveness, particularly in high-drift settings like AndroZoo, where selection signals tend to have greater impact. For **HCC**, ALPHA performs strongest at small budgets; removing any signal increases FNR by 2%–3% points. At larger budgets (e.g., 400), No-DIV slightly outperforms ALPHA as imbalanced-family reweighting becomes less critical once malware families are extensively labeled. Regardless of these, reweighting remains most effective with small budgets and substantial imbalance.

The structural ablation shows that all three signals contribute significantly to ALPHA, particularly at low budgets, with the full strategy yielding the best overall performance.

4.5.2 Sensitivity Analysis Results. To evaluate how sensitive ALPHA is to key design choices, Table 4 reports the results across varying configurations of PCA dimension (d), KDE bandwidth (h), and neighborhood size (k). The interaction between d and h is central to our OOS-selection signal. For **Drebin**, the theory-guided default configuration (see Appendix B) marginally outperforms alternative combinations of h, d. This result proves the reliability of the theoretical bounds for high-dimensional sparse feature space. Regarding HCC, performance exhibits minimal variance across (h, d) pairs. We adopt the default configuration, even though it is slightly outperformed, as it consistently captures HCC's compact

Budget	Model	Drebin FNR	Drebin FPR	Drebin F1	HCC FNR	HCC FPR	HCC F1
		Avg. Metrics (%)			Avg. Metrics (%)		
50	ALPHA	44.38	0.77	65.74	36.97	0.54	71.30
	No-OOS	57.40	0.40	54.73	39.88	0.65	69.14
	No-DIS	45.25	0.65	64.89	40.37	0.59	68.56
	No-DIV	61.71	0.50	49.60	38.33	0.55	70.37
100	ALPHA	39.97	0.77	69.04	34.87	0.46	73.61
	No-OOS	53.06	0.45	58.73	35.43	0.66	72.46
	No-DIS	41.26	0.74	68.12	39.10	0.44	70.28
	No-DIV	59.72	0.47	51.99	36.37	0.53	72.41
200	ALPHA	40.29	0.43	70.59	31.03	0.45	76.83
	No-OOS	45.90	0.41	65.25	31.82	0.63	75.53
	No-DIS	40.80	0.63	69.03	33.05	0.49	74.60
	No-DIV	56.86	0.49	54.95	29.29	0.53	77.64
400	ALPHA	38.35	0.78	70.23	30.42	0.47	77.06
	No-OOS	44.88	0.53	64.95	30.91	0.57	76.40
	No-DIS	40.61	0.60	69.36	30.19	0.54	76.81
	No-DIV	52.54	0.54	58.63	29.97	0.44	77.59

Table 3: Structural Ablation with both features (AndroZoo).

Variant	Config	Drebin (D) F1 (%)	Drebin (D) FNR (%)	Drebin (D) FPR (%)	HCC (H) F1(%)	HCC (H) FNR(%)	HCC (H) FPR (%)
Baseline	default	68.90	41.72	0.44	74.11	34.30	0.45
bandwidth (h)	0.1	67.41	42.90	0.50	74.89	32.64	0.55
	0.2	68.04	42.70	0.50	74.67	33.03	0.47
	1.0	60.90	50.36	0.41	74.59	33.45	0.59
PCA-dim (d)	10 (D) / 3 (H)	66.68	44.22	0.48	75.79	31.43	0.66
	20 (D) / 7 (H)	61.19	50.44	0.46	75.05	32.26	0.71
#neighbors (k)	2	70.79	40.43	0.44	76.13	31.34	0.50
	10	68.26	41.97	0.59	73.95	33.96	0.47

Table 4: Hyperparameter Sensitivity (AndroZoo, budget 200), Appendix B reports the default configuration.

manifold across all budgets while enabling the fastest KDE computation. Although exhaustive hyperparameter search for different budgets could yield marginal gains, the high computational cost of such tuning is rarely justified given ALPHA's inherent robustness to these hyperparameters. Across both feature spaces, varying the neighborhood size in the reweighting stage has a negligible impact. This robustness simplifies ALPHA's configuration and supports reliable deployment under varying local family densities.

4.6 Case Study

To examine whether each signal in ALPHA captures its targeted drift, we construct a challenging under-sampled setting on APIGraph-year with HCC embeddings by enforcing unseen families to constitute ≥ 20% of each validation and test year. To introduce variant-level drift among the remaining seen families, we train a base detector on Drebin features and under-sample correctly predicted test apps, resulting in around 60% of seen-family malware being misclassified. Though APIGraph originally maintains an unrealistic stable malicious-to-benign ratio (around 1:9), these under-sampling steps introduce noticeable spatial drift across malware families.

The following sections analyze how ALPHA's three components behave on this challenging benchmark, focusing on unseen-family selection (Section 4.6.1), variant-instability selection (Section 4.6.2), and imbalanced-family reweighting (Section 4.6.3).

4.6.1 Support Drift & Unseen Family. To evaluate whether the OOS signal captures unseen families, we compare OOS rates between unseen and seen samples. The yearly OOS rate is defined as the

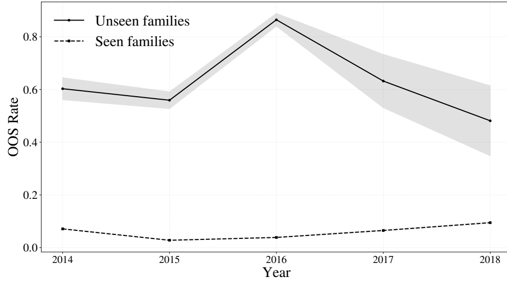


Figure 6: Year-wise OOS rates for unseen and seen sample. Shaded regions denote 95% confidence intervals.

fraction of test apps whose estimated density falls below the threshold given in Section 3.2, computed over both malware and benign apps to reflect the full test distribution. We apply a two-sample proportion test to compare unseen and seen OOS rates, reporting 95% confidence intervals in Fig. 6. Across all years, unseen samples consistently exhibit higher OOS rates than seen, with gaps ranging from 0.32 to 0.48, indicating that the OOS term effectively prioritizes samples from previously unseen families or functionalities.

4.6.2 Disagreement & Variant Instability. To validate that the disagreement signal captures variant-level drift, we analyze the selected samples and their relation to distribution changes in the test set. Most samples selected at this stage were already flagged as OOS but remained unlabeled due to limited budget, whether they belong to unseen families or not. This reveals two key observations:

First, support drift is not exclusive to unseen families. Due to substantial intra-family diversity in Android malware, samples from seen families may also fall outside the learned support [23, 49]. Besides, as the OOS threshold τ is derived as an upper bound: $|\hat{p}^X_{\text{train}}(x) - p^X_{\text{train}}(x)| \le \tau$, the condition $\hat{p}^X_{\text{train}}(x) < \tau$ only makes $p^X_{\text{train}}(x) \le \hat{p}^X_{\text{train}}(x) + \tau < 2\tau$, rather than $p^X_{\text{train}}(x) = 0$. Hence, the OOS set inevitably includes some low-density samples from seen families. Despite this, Section 4.6.1 shows that OOS rates remain significantly higher for unseen families, indicating that the first stage still aligns well with unseen-family drift. Second, although disagreement-selected samples largely overlap with OOS candidates, they follow a different selection principle: OOS emphasizes coverage via cluster centroids, while disagreement ranks samples by prediction inconsistency. As a result, different budget allocations lead to different labeled subsets and downstream performance.

To further illustrate the disagreement signal, we examine samples from the *lockscreen* family (first observed in 2015). Table 5 shows the 2016 samples, with features provided in Appendix F. Although these samples exhibit only minor structural differences, their predictions, densities, and disagreement scores vary significantly. For instance, Sample *1440* differ from sample *1363* by a single additional feature x_{start} = `android.media.mediaplayer.start`, which is strongly associated with benign behavior in the training data ($\Pr(\text{malware} \mid x_{\text{start}} = 1) = 0.0088$). Sample *1363* is selected due to high disagreement and labeled, while *1440* is initially predicted as benign with lower disagreement. After incorporating *1363*, *1440* becomes correctly classified, suggesting that labeling one representative variant helps the model generalize to similar samples. This illustrates that disagreement identifies weak variants whose correction benefits nearby samples in feature space.

Idx	Density	OOS	OOS sel.	Dis.	Dis sel.	Pred.	
						Before	After
115	0.0004	1	0	0.4680	0	1	1
140	0.5697	0	0	0.2655	0	0	0
912	0.0603	1	0	0.2943	0	0	0
1363	0.0006	1	0	0.4751	1	0	1
1440	0.0049	1	0	0.3099	0	0	1

Table 5: Samples of *lockscreen* in 2016. Idx: sample index; Density: KDE density $\hat{p}^X_{\text{train}}(\cdot)$; OOS: flagged as out-of-support or not; OOS sel.: selected in the OOS stage or not; Dis.: disagreement score; Dis sel.: selected in the disagreement stage or not Pred. before/after: predicted label before/after adaptation.

4.6.3 Divergence & Imbalanced Malware Family. To verify the effect of the reweighting stage, we compare the malware-family distribution of the updated training set with that of the test set. For each malware family shared between the training and test sets, we compute its normalized proportion among all malware.

To quantify alignment over time, we track yearly family-level discrepancy using mean absolute deviation (**MAD**) and root-mean-square error (**RMSE**) in Table 6, where Δ_{train}, $\Delta_{reweight}$, and Δ_{ALPHA} denote the discrepancies between the test distribution and the original training set, the reweighting-only set, and the full-ALPHA-updated set, respectively. Table 6 shows that both ALPHA and Reweight reduce mismatch across all years, with larger improvements under stronger drift. In 2018, MAD decreases from 0.205 (Δ_{train}) to 0.158 ($\Delta_{reweight}$) and further to 0.113 (Δ_{ALPHA}), with similar trends in RMSE. A detailed case study on 2018 (Appendix G) further shows that ALPHA produces smaller family-level gaps than reweighting alone, particularly for high-drift families.

These results suggest that reweighting aligns the training distribution with the test, while ALPHA's coarse-to-fine structure further improves alignment by focusing reweighting on family-level imbalance after filtering unseen families and unstable variants.

Year	MAD			RMSE		
	Δ_{train}	$\Delta_{reweight}$	Δ_{ALPHA}	Δ_{train}	$\Delta_{reweight}$	Δ_{ALPHA}
2014	0.035	0.030	0.027	0.066	0.044	0.041
2015	0.046	0.043	0.031	0.080	0.059	0.042
2016	0.042	0.041	0.025	0.077	0.058	0.037
2017	0.088	0.070	0.044	0.155	0.113	0.075
2018	0.205	0.158	0.113	0.308	0.225	0.148

Table 6: Yearly family-ratio discrepancy (MAD & RMSE). Δ_{orig}: discrepancy of original train/test; Δ_{reweight}: discrepancy after reweight-only update; Δ_{ALPHA}: discrepancy after full ALPHA update. Lower values mean closer alignment.

5 Related Work

Drift Adaptation Theory. While our work builds on the PAC-Bayes framework in Section 2, prior work offers alternative theoretical perspectives on domain adaptation. Early foundational work [8] establishes bounds based on the VC-dimension and the $\mathcal{H}\Delta\mathcal{H}$-divergence, proving that adaptation is limited by the divergence between marginal distributions. Mansour et al. [25] introduce the

discrepancy distance, extending generalization guarantees beyond the binary classification setting. Recently, Cortes et al. [11] work on minimizing the discrepancy between train and test distribution by formulating domain adaptation as convex optimization problems to derive tighter generalization bounds. Prior work also addresses samples outside the training support. Netanyahu et al. [29] propose a bilinear transduction framework that bounds generalization under a linear reconstructibility assumption. While suitable for fixed dynamics, this assumption often fails in malware detection due to adversarially driven distribution shifts.

Drift in Malware Detection. Malware detection operates in an evolving ecosystem where performance degrades over time, and a common mitigation is to reject out-of-distribution samples. Transcend [19] and Transcendent [7] use conformal prediction to defer uncertain inputs for manual review, while CADE [45] identifies drifting samples via contrastive learning and distance-based outlier detection. Moving beyond rejection, other works update models using newly labeled data. MADAR [33] proposes distribution-aware replay to preserve heterogeneous malware patterns, while HCC [9] employs hierarchical contrastive learning with loss-based sample selection. DREAM [17] further incorporates concept-level explanations from human experts into adaptation, enabling joint refinement of labels and behavioral concepts. To reduce human labeling efforts, DroidEvolver [44] adopts pseudo-label-based self-training to update models without manual supervision. DroidEvolver++ [20] identifies its susceptibility to error accumulation and proposes refinements to improve stability, but performance degradation under distribution shifts persists. In general, those approaches mainly focus on empirical performance rather than explicitly modeling distribution shifts with theoretical support.

Existing work includes both general-purpose [7, 19, 33, 45] and Android-specific designs [9, 17, 20, 44]. Similar drift patterns appear in both PE and Android malware. However, in PE malware, packing and obfuscation often dominate representation variability [42], causing semantically similar samples to be far apart in feature space. This highlights the Android-specific nature of ALPHA's feature-space stability assumption and suggests that its effectiveness on PE malware depends on the robustness of representations.

6 Discussion and Limitations

While ALPHA seamlessly integrates into active learning (AL) pipelines for Android malware detection, its real-world deployment still raises several concerns:

Scope of Generality. First, while PAC-Bayesian theory provides general guarantees under distribution shift, its decomposition is domain-specific. ALPHA is designed based on observed evolution patterns in Android malware [7, 31], and may transfer to other domains with similar shifts after minor adaptation. Second, although ALPHA can operate on different feature spaces, its effectiveness depends on representation quality. As shown in RQ1 (Section 4.2), stronger representations may reduce the advantage of ALPHA over simpler uncertainty-based methods. Finally, we use an SVM due to its strong performance in Android malware detection [9, 17], but ALPHA can be extended to other classifiers. For classifiers whose posteriors cannot be well approximated by a normal distribution, bootstrapping can be used as an alternative [40, 43]. In our experiments (Appendix H), bootstrap estimates yield performance comparable to or even slightly better than the closed-form surrogate, at the cost of repeated model training.

Ability to Handle Non-stationary Drifts. In real-world deployment, drift may be abrupt and non-stationary, with shifting contributions from support drift (emergence of new families) and disagreement drift (evolution of known families). Currently, ALPHA uses a budget ratio tuned on a small validation set to balance the first two selection stages, which may become biased if the drift characteristics during the validation months do not represent those of the real environment. To address this, future work will explore on-the-fly adaptation of the budget ratio, including periodic recalibration and the use of online algorithms to allocate the budget based on the real-time performance of each signal. By transitioning to a more reactive budget policy, ALPHA can maintain its advantage in more realistic scenarios.

Robustness to Label Noise. Label noise can affect AL methods in general, including uncertainty baselines, by biasing both sample selection and model updates. ALPHA shares this sensitivity, but its multi-stage design may introduce additional considerations: errors from earlier stages, like the clustering-based OOS stage, may propagate and influence downstream decisions. To improve robustness in the first stage, one possibility is to make the OOS clustering more stable under noise, for example by replacing k-means with k-medoids for representative selection or using robust KDE [21] for density estimation. These approaches may help when mislabeled samples appear as outliers, but may be less effective when they lie within high-density regions. To further mitigate label noise in the AL loop, replacing hard labels with robust labels that reflect label uncertainty is an interesting orientation [36]. In practice, this can be achieved by aggregating multiple antivirus reports (e.g., via AVClass), or by querying multiple instances within a cluster rather than relying on a single point.

7 Conclusion

We propose ALPHA, an active learning framework guided by the PAC-Bayes bound to mitigate distribution shift in Android malware detection. We decompose the bound into three distinct Android-specific drift signals and translate distribution-level principles into instance-level sampling criteria. We also introduce two drift metrics and show that benchmarks such as APIGraph exhibit limited drift to reliably evaluate drift-aware methods. Extensive experiments show that ALPHA outperforms baselines across the evaluated metrics and embeddings, particularly in high-drift, low-budget settings. Ablation studies validate that each theoretically grounded signal is significant to robust detection.

Acknowledgments

This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG), reference 378803395 (ConVeY).

References

[1] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *Security and Privacy in Communication Networks, (SecureComm)*, Tanveer A. Zia, Albert Y. Zomaya, Vijay Varadharajan, and Zhuoqing Morley Mao (Eds.), Vol. 127. Springer, 86–103.

[2] Md Tanvirul Alam, Aritran Piplai, and Nidhi Rastogi. 2025. ADAPT: A Pseudo-labeling Approach to Combat Concept Drift in Malware Detection. *CoRR* abs/2507.08597 (2025).
[3] Evgeny Burnaev Aleksandr Safin. 2017. Conformal Kernel Expected Similarity for Anomaly Detection in Time-Series data. *Advances in Systems Science and Applications* 17, 3 (2017), 22–33.
[4] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proc. International Conf. Mining Software Repositories* (Austin, Texas) *(MSR '16)*. 468–471.
[5] Fahad Alotaibi, Euan Goodbrand, and Sergio Maffeis. 2025. Deep Learning from Imperfectly Labeled Malware Data. In *Proc. ACM SIGSAC Conf. Computer and Communications Security, (CCS)*. ACM, 3990–4004.
[6] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Annual Network and Distributed System Security Symposium, (NDSS)*.
[7] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. 2022. Transcending TRANSCEND: Revisiting Malware Classification in the Presence of Concept Drift. In *IEEE Symposium on Security and Privacy, (SP)*. IEEE, 805–823.
[8] Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. 2010. A theory of learning from different domains. *Mach. Learn.* 79, 1-2 (2010), 151–175.
[9] Yizheng Chen, Zhoujie Ding, and David A. Wagner. 2023. Continuous Learning for Android Malware Detection. In *USENIX Security Symposium*. USENIX Association, 1127–1144.
[10] Theo Chow, Zeliang Kan, Lorenz Linhardt, Lorenzo Cavallaro, Daniel Arp, and Fabio Pierazzi. 2023. Drift Forensics of Malware Classifiers. In *Proc. ACM Workshop on Artificial Intelligence and Security, (AISec)*. ACM, 197–207.
[11] Corinna Cortes, Mehryar Mohri, and Andres Muñoz Medina. 2015. Adaptation Algorithm and Theory Based on Generalized Discrepancy. In *Proc. International Conference on Knowledge Discovery and Data Mining*. ACM, 169–178.
[12] Minhong Dong, Liyuan Liu, Mengting Zhang, Sen Chen, Wenying He, Ze Wang, and Yude Bai. 2025. Calmdroid: Core-Set Based Active Learning for Multi-Label Android Malware Detection. In *IEEE/ACM International Conf. Program Comprehension, (ICPC)*. IEEE, 37–48.
[13] Pascal Germain, Amaury Habrard, François Laviolette, and Emilie Morvant. 2013. A PAC-Bayesian Approach for Domain Adaptation with Specialization to Linear Classifiers. In *Proc. International Conf. Machine Learning, (ICML)*. JMLR.org, 738–746.
[14] Pascal Germain, Amaury Habrard, François Laviolette, and Emilie Morvant. 2016. A New PAC-Bayesian Perspective on Domain Adaptation. In *Proc. International Conf. Machine Learning, (ICML)*. JMLR.org, 859–868.
[15] Evarist Giné and Armelle Guillou. 2002. Rates of strong uniform consistency for multivariate kernel density estimators. *Annales de l'I.H.P. Probabilités et statistiques* (2002), 907–921.
[16] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick D. McDaniel. 2017. Adversarial Examples for Malware Detection. In *European Symposium on Research in Computer Security*. Springer, 62–79.
[17] Yiling He, Junchi Lei, Zhan Qin, Kui Ren, and Chun Chen. 2025. Combating Concept Drift with Explanatory Detection and Adaptation for Android Malware Classification. In *Proc. ACM SIGSAC Conf. Computer and Communications Security, (CCS)*. ACM, 978–992.
[18] Alexander Herzog, Aliai Eusebi, and Lorenzo Cavallaro. 2025. Aurora: Are Android Malware Classifiers Reliable under Distribution Shift? *arXiv preprint arXiv:2505.22843* (2025).
[19] Roberto Jordaney, Kumar Sharad, Santanu Kumar Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting Concept Drift in Malware Classification Models. In *USENIX Security Symposium*. USENIX Association, 625–642.
[20] Zeliang Kan, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. 2021. Investigating Labelless Drift Adaptation for Malware Detection. In *Proc. of the 14th ACM Workshop on Artificial Intelligence and Security, (AISec)*. ACM, 123–134.
[21] JooSeuk Kim and Clayton D. Scott. 2008. Robust kernel density estimation. In *Proc. of the IEEE International Conference on Acoustics, Speech, and Signal Processing, (ICASSP)*. IEEE, 3381–3384.
[22] Haodong Li, Xiao Cheng, Yanjie Zhao, Guosheng Xu, Guoai Xu, and Haoyu Wang. 2025. Understanding Model Weaknesses: A Path to Strengthening DNN-Based Android Malware Detection. *Proc. ACM Softw. Eng.* 2, ISSTA (2025), 320–342.
[23] Li Li, Daoyuan Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. 2017. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *IEEE Trans. Inf. Forensics Secur.* 12, 6 (2017), 1269–1284.
[24] Marco Loog. 2012. Nearest neighbor-based importance weighting. In *IEEE International Workshop on Machine Learning for Signal Processing, (MLSP)*. IEEE, 1–6.
[25] Yishay Mansour, Mehryar Mohri, and Afshin Rostamizadeh. 2008. Domain Adaptation with Multiple Sources. In *Proc. Annual Conf. on Neural Information Processing Systems*. Curran Associates, Inc., 1041–1048.
[26] David McAllester and Takintayo Akinbiyi. 2013. PAC-Bayesian Theory. In *Empirical Inference - Festschrift in Honor of Vladimir N. Vapnik*. Springer, 95–103.
[27] Niall McLaughlin, Jesús Martínez del Rincón, BooJoong Kang, Suleiman Y. Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. 2017. Deep Android Malware Detection. In *Proc. ACM Conf. Data and Application Security and Privacy, (CODASPY)*. ACM, 301–308.
[28] Annamalai Narayanan, Charlie Soh, Lihui Chen, Yang Liu, and Lipo Wang. 2018. Apk2vec: Semi-Supervised Multi-view Representation Learning for Profiling Android Applications. In *IEEE International Conf. Data Mining, (ICDM)*. IEEE Computer Society, 357–366.
[29] Aviv Netanyahu, Abhishek Gupta, Max Simchowitz, Kaiqing Zhang, and Pulkit Agrawal. 2023. Learning to Extrapolate: A Transductive Approach. In *International Conf. on Learning Representations, (ICLR)*.
[30] Baodi Ning, Guanqin Zhang, and Zexin Zhong. 2020. An Evolutionary Perspective: A Study of Anubis Android Banking Trojan. In *International Conf. Dependable Systems and Their Applications, (DSA)*. IEEE, 141–150.
[31] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *USENIX Security Symposium*. USENIX Association, 729–746.
[32] Qijing Qiao, Ruitao Feng, Sen Chen, Fei Zhang, and Xiaohong Li. 2022. Multi-label Classification for Android Malware Based on Active Learning. *IEEE Transactions on Dependable and Secure Computing, (TDSC)* (2022), 1–18.
[33] Mohammad Saidur Rahman, Scott E. Coull, Qi Yu, and Matthew Wright. 2025. MADAR: Efficient Continual Learning for Malware Analysis with Diversity-Aware Replay. *CoRR* abs/2502.05760 (2025).
[34] Murray Rosenblatt. 1956. Remarks on Some Nonparametric Estimates of a Density Function. *The Annals of Mathematical Statistics* (1956), 832–837.
[35] Antonio Ruggia, Dario Nisi, Savino Dambra, Alessio Merlo, Davide Balzarotti, and Simone Aonzo. 2024. Unmasking the Veiled: A Comprehensive Analysis of Android Evasive Malware. In *Proc. ACM Asia Conference on Computer and Communications Security, (ASIACCS)*. ACM.
[36] David Stutz, Abhijit Guha Roy, Tatiana Matejovicova, Patricia Strachan, Ali Taylan Cemgil, and Arnaud Doucet. 2023. Conformal prediction under ambiguous ground truth. *Trans. Mach. Learn. Res.* 2023 (2023).
[37] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. 2017. DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware. In *Proc. ACM Conf. on Data and Application Security and Privacy, (CODASPY)*. ACM, 309–320.
[38] Guillermo Suarez-Tangil and Gianluca Stringhini. 2022. Eight Years of Rider Measurement in the Android Malware Ecosystem. *IEEE Trans. Dependable Secur. Comput.* 19, 1 (2022), 107–118.
[39] Masashi Sugiyama, Taiji Suzuki, and Takafumi Kanamori. 2012. *Density Ratio Estimation in Machine Learning*. Cambridge University Press.
[40] Nicholas Syring and Ryan Martin. 2018. Calibrating general posterior credible regions. *Biometrika* 106, 2 (2018), 479–486.
[41] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Annual Network and Distributed System Security Symposium, (NDSS)*.
[42] Antonino Vitale, Simone Aonzo, Savino Dambra, Nanda Rani, Lorenzo Ippolito, Platon Kotzias, Juan Caballero, and Davide Balzarotti. 2025. The Polymorphism Maze: Understanding Diversities and Similarities in Malware Families. In *European Symposium on Research in Computer Security, (ESORICS)*. Springer, 505–525.
[43] Spencer Woody, Novin Ghaffari, and Lauren Hund. 2019. Bayesian Model Calibration for Extrapolative Prediction via Gibbs Posteriors. *arXiv preprint arXiv:1909.05428* (2019).
[44] Ke Xu, Yingjiu Li, Robert H. Deng, Kai Chen, and Jiayun Xu. 2019. DroidEvolver: Self-Evolving Android Malware Detection System. In *IEEE European Symposium on Security and Privacy, (EuroS&P)*. IEEE, 47–62.
[45] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. 2021. CADE: Detecting and Explaining Concept Drift Samples for Security Applications. In *USENIX Security Symposium*. USENIX Association, 2327–2344.
[46] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-Sec: deep learning in android malware detection. In *ACM SIGCOMM*. ACM, 371–372.
[47] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. 2020. Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware. In *Proc. 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 757–770.
[48] Xinran Zheng, Shuo Yang, Edith C. H. Ngai, Suman Jana, and Lorenzo Cavallaro. 2025. Learning Temporal Invariance in Android Malware Detectors. *CoRR* abs/2502.05098 (2025).
[49] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy, (SP)*. IEEE Computer Society, 95–109.

A Derivation of the Adaptive Threshold

In this section, we provide the formal derivation for the adaptive threshold to flag OOS samples. We rely on standard results from non-parametric density estimation to bound the deviation between the Kernel Density Estimator (KDE) and the true marginal distribution.

Lemma A.1. ***(Uniform Convergence of KDE).*** *Let $\hat{p}^X_{\text{train}}$ be the kernel density estimator defined on the training set S_{train} of size n, with bandwidth $h > 0$ and kernel K. Assume the true marginal density p^X_{train} satisfies $\|p^X_{\text{train}}\|_\infty \le c_1$ and $\|\nabla^2 p^X_{\text{train}}\|_\infty \le c_2$. Assume the kernel moments satisfy $\int y^2 K(y)\,\mathrm{d}y \le \mu_K$ and $\int K^2(y)\,\mathrm{d}y \le \sigma_K^2$.*

Then, with probability at least $1-\delta/2$ over the drawing of S_{train}, the supremum deviation is bounded with:

$$\Delta = \sup_x \left|\hat{p}^X_{\text{train}}(x) - p^X_{\text{train}}(x)\right| \le \sigma_K \sqrt{\frac{c_1 \log(2/\delta)}{nh^d}} + \frac{1}{2} c_2 \mu_K h^2.$$

Proof. The estimation error of the KDE at any point x can be decomposed into a variance term and a bias term:

$$\left|\hat{p}^X_{\text{train}}(x) - p^X_{\text{train}}(x)\right| \le \underbrace{\left|\hat{p}^X_{\text{train}}(x) - \mathbb{E}[\hat{p}^X_{\text{train}}(x)]\right|}_{\text{Variance}} + \underbrace{\left|\mathbb{E}[\hat{p}^X_{\text{train}}(x)] - p^X_{\text{train}}(x)\right|}_{\text{Bias}}.$$

Following standard results in non-parametric density estimation [15], the bias can be bounded using the Taylor expansion of p^X_{train} and the Hessian bound c_2, yielding $\frac{1}{2}c_2\mu_K h^2$. The variance term is bounded using the Bernstein concentration inequalities. With probability at least $1-\delta/2$, the following bound holds uniformly for all x:

$$\sup_x \left|\hat{p}^X_{\text{train}}(x) - p^X_{\text{train}}(x)\right| \le \sigma_K \sqrt{\frac{c_1 \log(2/\delta)}{nh^d}} + \frac{1}{2} c_2 \mu_K h^2. \quad (7)$$

□

Theorem A.2. ***(Upper Bound on OOS Risk).*** *Taking the threshold $\tau = \sigma_K \sqrt{\frac{c_1 \log(2/\delta)}{nh^d}} + \frac{1}{2}c_2\mu_K h^2$, given the conditions in Lemma A.1, with probability at least $1-\delta$, the probability of a test sample x falling outside the support of the training distribution is bounded by:*

$$\Pr_{x\sim\mathcal{D}^X_{\text{test}}}(x \notin \operatorname{supp}(\mathcal{D}^X_{\text{train}})) \le \frac{1}{m}\sum_{x\in S_{\text{test}}} \mathbf{1}\{\hat{p}^X_{\text{train}}(x) < \tau\} + \sqrt{\frac{\log(2/\delta)}{2m}}.$$

Proof. For $x \sim \mathcal{D}^X_{\text{test}}$, we decompose $\Pr\left(x \notin \operatorname{supp}\left(\mathcal{D}^X_{\text{train}}\right)\right)$ as:

$$\begin{aligned} &\Pr_{x\sim\mathcal{D}^X_{\text{test}}}(\{x \notin \operatorname{supp}(\mathcal{D}^X_{\text{train}})\}) \\ &= \Pr_{x\sim\mathcal{D}^X_{\text{test}}}(p^X_{\text{train}}(x) = 0) \\ &\le \Pr_{x\sim\mathcal{D}^X_{\text{test}}}(\hat{p}^X_{\text{train}}(x) < \tau) + \mathbb{P}_{x\sim\mathcal{D}^X_{\text{test}}}(|p^X_{\text{train}}(x) - \hat{p}^X_{\text{train}}(x)| \ge \tau). \end{aligned}$$

The second term represents the failure event of the KDE approximation. By Lemma A.1, with probability at least $1-\delta/2$, the maximum deviation is bounded by τ. Under this event, the second term vanishes, as $|p^X_{\text{train}}(x) - \hat{p}^X_{\text{train}}(x)| < \tau$ for all x.

We are thus left with the first term. Since we cannot compute the true probability $\Pr(\hat{p}^X_{\text{train}}(x) < \tau)$, we estimate it using the empirical test set S_{test} of size m. By the Hoeffding inequality, with probability at least $1-\delta/2$ over the drawing of S_{test}:

$$\Pr_{x\sim\mathcal{D}^X_{\text{test}}}(\hat{p}^X_{\text{train}}(x) < \tau) \le \frac{1}{m}\sum_{x\in S_{\text{test}}} \mathbf{1}\{\hat{p}^X_{\text{train}}(x) < \tau\} + \sqrt{\frac{\log(2/\delta)}{2m}}. \quad (8)$$

Applying union bound over the event in Lemma A.1 (w.p. $\delta/2$) and Eq. 8 (w.p. $\delta/2$) gives the result with confidence $1-\delta$. □

B Choice of Hyperparameters

This section reports the default hyperparameter settings used for the evaluation results in Table 1 and Table 4, along with the justification for the specific choices.

Besides the budget allocation ratio between the OOS and disagreement selection stage, which is finetuned on the validation split (see Section 4.1.4), we fixed the following hyperparameters for the evaluation: (i) **regularization strength** C for the SVM classifier used in UNC and ALPHA; (ii) **KDE bandwidth** h and **PCA dimension** d in the OOS-selection stage of ALPHA; (iii) **neighborhood size** k in the reweighting stage of ALPHA.

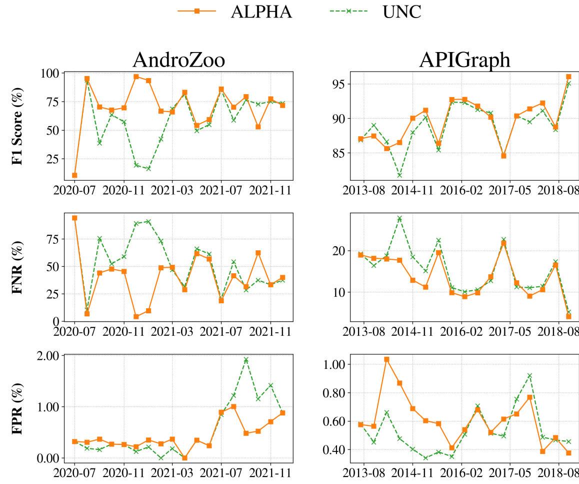
SVM Regularization Strength. For UNC and ALPHA, we adopt a consistent regularization strength across budgets: $C = 0.1$ for AndroZoo and $C = 0.01$ for APIGraph and APIGraph-year, following the recommendation in [9].

Density Estimation h, d. The KDE bandwidth (h) and PCA dimension (d) are specific to the ALPHA framework. We adopt a standard bandwidth choice of $h = 0.5$ and set the PCA dimension to $d = 15$ for the Drebin feature and $d = 1$ for the HCC feature. This selection is guided by Lemma A.1, which suggests that to minimize the gap between the estimated and true densities of the training distribution, h^d should be greater than the inverse of the sample size. This gives us a theoretical upper bound for the PCA dimension, i.e., $d \le 15$. Within this theoretically sound range, a lower dimension is selected for the HCC representation ($d = 1$) than for Drebin ($d = 15$) to account for its more compact manifold structure, enabling effective density estimation with fewer components. As shown in the sensitivity analysis (Table 4), the variances of these two hyperparameters have small impact on ALPHA's performance, demonstrating that ALPHA remains robust as long as h^d stays within the derived safe range.

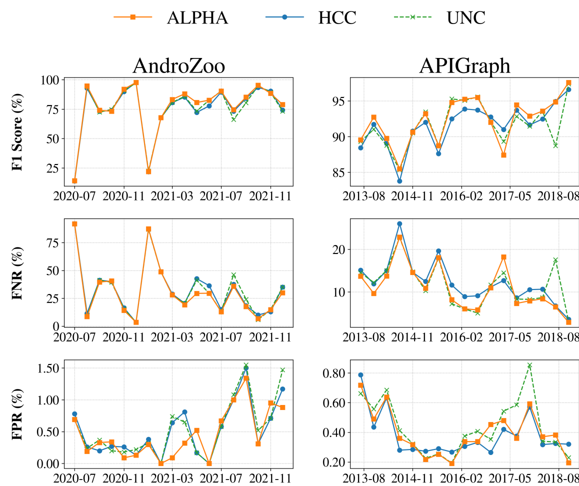
Reweighting k. The neighborhood size k is also unique to ALPHA. We use the advised default setting $k = 5$ from the Adapt library. Similar to h and d, varying k also has a negligible impact on the performance of ALPHA. Therefore, extensive fine-tuning of these parameters is unnecessary for maintaining ALPHA's effectiveness across diverse feature spaces and datasets.

Competitor Settings. For HCC [9], we do not perform additional fine-tuning. To ensure a fair comparison, we strictly adopt the hyperparameter configurations recommended in the original work.

C Monthly Results on Two Benchmarks



(a) Monthly Performance on Drebin Feature



(b) Monthly Performance on HCC Feature

Figure 7: The performance of different selection and update strategies across two benchmarks and two embedding feature spaces. With Drebin features, ALPHA consistently outperforms the uncertainty-based sampling method regarding all metrics. Notably, in terms of FPR, its advantage begins to accumulate as time progresses. With HCC features, all models perform comparably, though ALPHA gradually gains an advantage against the others over time.

D Structural Ablation Analysis on APIGraph

Budget	Model	Drebin Avg. Metrics (%)			HCC Avg. Metrics (%)		
		FNR	FPR	F1	FNR	FPR	F1
50	ALPHA	16.35	0.74	87.56	15.80	0.45	89.18
	No-OOS	16.35	0.74	87.56	14.38	0.52	89.75
	No-DIS	19.69	0.93	84.74	15.73	0.52	88.88
	No-DIV	19.46	0.57	86.38	16.30	0.47	88.77
100	ALPHA	14.63	0.70	88.77	14.49	0.39	90.23
	No-OOS	16.04	0.59	88.35	13.10	0.50	90.57
	No-DIS	17.68	0.77	86.58	14.22	0.48	89.98
	No-DIV	16.52	0.53	88.34	14.47	0.40	90.19
200	ALPHA	14.00	0.61	89.52	11.17	0.40	92.10
	No-OOS	15.30	0.55	88.97	13.34	0.42	90.77
	No-DIS	18.10	0.66	86.81	10.62	0.43	92.26
	No-DIV	15.95	0.51	88.76	13.20	0.41	90.90
400	ALPHA	13.66	0.60	89.74	12.22	0.42	91.41
	No-OOS	13.66	0.60	89.74	10.05	0.50	92.29
	No-DIS	17.80	0.65	87.05	10.86	0.46	92.00
	No-DIV	15.06	0.55	89.13	11.01	0.43	92.06

Table 7: Structural Ablation Study on APIGraph. Compared to improvements on AndroZoo, improvements are still positive, but marginal. In particular, with the Drebin feature, at budgets 50 and 400, ALPHA matches No-OOS, as the budget-ratio adapter selects a ratio of 0 via validation, turning off the unseen-family signal. This weaker dependence on individual components is consistent with APIGraph's minimal drift.

E Drebin Selection-stage Runtime

Budget	Criterion	APIGraph-year Time/iter (s)	CPU RSS (GB)	GPU peak (GB)
50	UNC	48.60	1.83	-
	ALPHA	475.97	2.10	-
100	UNC	52.38	1.72	-
	ALPHA	64.44	1.85	-
200	UNC	54.48	1.84	-
	ALPHA	503.33	2.10	-
400	UNC	61.32	1.75	-
	ALPHA	76.47	1.79	-

Table 8: Selection-stage runtime on the Drebin feature space (CPU-only training and inference).

The results show that UNC maintains stable and low runtime across all budgets, with only a slight increase as the budget grows while ALPHA exhibits notably higher runtime, aligning with the expectations. In terms of memory usage, both methods have similar CPU memory usage, with ALPHA incurring only a modest increase. No GPU memory is used under the CPU-only setting.

F Feature-Level Comparison of *lockscreen* Variants

Listing 1 Common malicious feature subset shared by all *lockscreen* variants listed in Table 5.

```
# shared features
android.permission.receive_boot_completed
landroid/content/context.getsystemservice
ljava/lang/runtime;->exec
landroid/content/pm/packagemanager.getpackageinfo
android.intent.action.main
android.intent.action.boot_completed
```

Listing 2 Remaining features of Sample *1363* in 2016

```
android.permission.system_alert_window
android.permission.mount_unmount_filesystems
android.permission.internet
android.permission.access_network_state
android.permission.write_external_storage
landroid/app/application.getsystemservice
android.app.action.device_admin_enabled
android.permission.send_sms
```

Listing 3 Remaining features of Sample *1440* in 2016

```
android.permission.system_alert_window
android.permission.mount_unmount_filesystems
android.permission.internet
android.permission.access_network_state
android.permission.write_external_storage
landroid/app/application.getsystemservice
android.app.action.device_admin_enabled
android.permission.send_sms
# The only feature that differs from Sample 1363.
android.media.mediaplayer.start
```

Listing 4 Remaining features of Sample *140* in 2016

```
android.permission.get_tasks
android.permission.kill_background_processes
android.permission.access_wifi_state
android.permission.get_tasks
android.permission.change_wifi_state
android.net.wifi.state_change
android.app.activitymanager.killbackgroundprocesses
android.net.conn.connectivity_change
android.permission.wake_lock
mainactivity
android.net.wifi.wifi_state_changed
```

G Imbalanced Family in 2018

Table 9 reports family-level ratios in 2018, where spatial drift is largest relative to the 2012 training distribution. We compare four settings: original training (**Train**), reweighting only (**Reweight**), the full ALPHA update (**ALPHA**), and the test set (**Test**). Δ measures the additional reduction in train–test mismatch achieved by ALPHA over reweighting alone. Both ALPHA and Reweight move the training distribution toward the test distribution, but ALPHA yields smaller family-level gaps, particularly for high-drift families (e.g., adware), indicating that reweighting is more effective after filtering unseen families and unstable variants.

Family	Train	Reweight	ALPHA	Test	Δ
adware	0.6742	0.5093	0.2523	0.0194	0.2570
leadbolt	0.0395	0.1221	0.0230	0.0097	0.0991
execdownload	0.0180	0.0016	0.0737	0.0388	0.0024
dnotua	0.0023	0.0441	0.0465	0.0388	-0.0024
smsreg	0.0643	0.2051	0.2026	0.3981	-0.0025
opfake	0.1669	0.1179	0.2021	0.0097	-0.0842

Table 9: Comparison of family-level ratios for 2018.

H Bootstrap

Table 10: Bootstrap Disagreement vs. Closed-form (Androzoo, Budget=200).

Metric	Drebin		HCC	
	Closed-form	Bootstrap	Closed-form	Bootstrap
F1 (%)	70.59	70.59	74.77	75.85
FNR (%)	40.29	36.96	33.44	32.06
FPR (%)	0.43	0.98	0.45	0.52

Bootstrap disagreement is computed by training multiple classifiers (n=20) and measuring the disagreement for each sample as the sum of pairwise disagreements among classifier outputs. As shown in Table 10, across feature spaces, the closed-form solution achieves performance close to the bootstrap estimate, while the latter incurs substantially higher computational cost due to repeated model training.