R+R: Revisiting Static Feature-Based Android Malware Detection using Machine Learning

1st Md Tanvirul Alam
Rochester Institute of Technology
Rochester, NY, USA
ma8235@rit.edu

2nd Dipkamal Bhusal
Rochester Institute of Technology
Rochester, NY, USA
db1702@rit.edu

3rd Nidhi Rastogi
Rochester Institute of Technology
Rochester, NY, USA
nxrvse@rit.edu

***Abstract*—Static feature-based Android malware detection using machine learning (ML) remains critical due to its scalability and efficiency. However, existing approaches often overlook security-critical reproducibility concerns, such as dataset duplication, inadequate hyperparameter tuning, and variance from random initialization. This can significantly compromise the practical effectiveness of these systems. In this paper, we systematically investigate these challenges by proposing a more rigorous methodology for model selection and evaluation. Using two widely used datasets, Drebin and APIGraph, we evaluate six ML models of varying complexity under both offline and continuous active learning settings. Our analysis demonstrates that, contrary to popular belief, well-tuned, simpler models, particularly tree-based methods like XGBoost, consistently outperform more complex neural networks, especially when duplicates are removed. To promote transparency and reproducibility, we open-source our codebase, which is extensible for integrating new models and datasets, facilitating reproducible security research.**

1. Introduction

The proliferation of Android malware continues to pose significant security threats, making timely detection and mitigation critical. Machine learning (ML), particularly leveraging static analysis features, remains widely adopted in security practice due to its scalability, speed, and suitability for rapidly screening large numbers of applications [1]–[4]. ML models learn patterns from data, adapt to new threats, and scale far beyond the capabilities of traditional signature-based systems. However, recent research underscores critical limitations in their reliability, reproducibility, and security assurance [5], leading to claims that cannot be reliably corroborated. Key issues, as observed in prior works [5]–[7], include sampling bias, incorrect labels, data snooping, inappropriate threat models, and unsuitable baselines. This lack of rigor in the design and evaluation process undermines the validity of many reported findings [8]. Such discrepancies seriously question the reliability of ML-based malware detection when deployed in operational security settings.

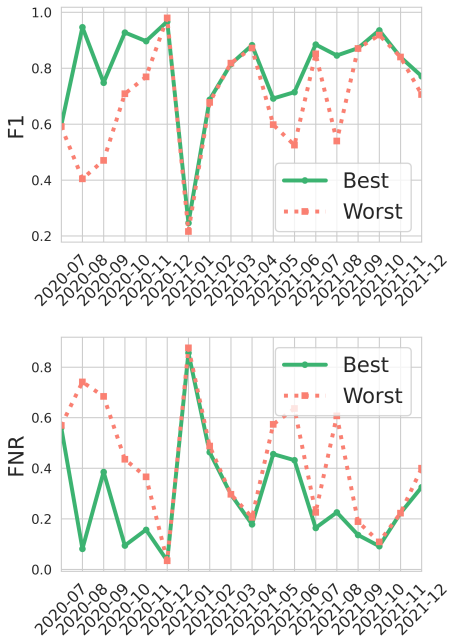


Figure 1: Performance comparison of the best and worst performing neural network models for Android malware detection using the state-of-the-art continuous active learning method from [9]. Top: F1-score. Bottom: False Negative Rate (FNR). Models were initialized with 5 different random seeds. The average F1-score over months differs by 10.6%, and the False Negative Rate (FNR) differs by 13.9% between the two models, despite using the same hyperparameters. This highlights reproducibility challenges in machine learning research for Android malware detection.

Motivation. We showcase an example that highlights how reliably reproducing prior results in Android malware detection can be fraught with challenges, which, if not adequately addressed, can hinder progress in this research field. See Figure 1, where we examine the reproducibility of results reported by [9] for Android malware detection using continuous active learning. Reproduced experiments include artifacts provided by the authors, including code and

datasets, and the author-provided best set of hyperparameters for the Drebin-based dataset [1]. We used five different random seeds and then compared the best and worst-performing models on the test set. Best model achieved an average F1 score of 79.26% over the test months, while the worst model scored 68.65%, indicating a performance difference of 10.6% solely due to variations in random seed initialization.

In this paper, we study these security-critical evaluation issues by systematically studying Android malware detection methodologies. We examine three prevalent reproducibility pitfalls, including (1) data duplication, which artificially inflates model performance; (2) inadequate hyperparameter tuning, causing unfair baseline comparisons; and (3) performance instability due to variance from random model initialization. We systematically discuss how these factors cause irreproducibility and propose ways to mitigate them. We use two widely-recognized Android malware datasets, Drebin [1] and APIGraph [10], and six different models, including Random Forest [11], Support Vector Machine [1], eXtreme Gradient Boosting [12], Multilayer Perceptron [13], Supervised Contrastive Classifier [14], and Hierarchical Contrastive Classifier [9], previously used in Android malware analysis. We conduct comprehensive experiments under realistic offline learning, where the model is trained once, and continuous active learning, where the model is periodically retrained with a subset of annotated samples. We perform detailed model selection and hyperparameter tuning for all models in both settings. While our work builds on recent advances in continuous active learning [9], we broaden the evaluation scope to include diverse settings and emphasize reproducibility in Android malware detection. Contrary to the assumption that complex neural architectures outperform simpler ones, our experiments demonstrate that well-tuned tree-based models, particularly XGBoost, often exhibit greater robustness after duplicate samples are removed. These results underscore the importance of rigorous and reproducible evaluation over architectural sophistication.

Key Contributions:

1) We systematically identify critical reproducibility issues in datasets and evaluation methods commonly used in Android malware detection, and propose rigorous methodological solutions to improve result reliability and enable fairer comparisons across models.
2) Through comprehensive experiments on two standard datasets in both offline and continuous active learning scenarios, we demonstrate that simpler, well-tuned models (e.g., XGBoost) consistently outperform complex neural networks, challenging common assumptions in security literature.
3) We release our extensively validated, extensible evaluation framework as open-source[1], facilitating transparent, reproducible, and security-focused malware detection research.

1. https://github.com/xashru/maldetect

2. Background & Related Work

2.1. Reproducibility & Replicability

We adhere to ACM's definitions of reproducibility and replicability, standardized in 2020 [15]. Computational reproducibility refers to the ability of an independent team to achieve a study's results using the original study's artifacts [16]. This ensures that the reported findings are valid under specified conditions. [8] conducted a longitudinal study on the reproducibility of security papers published in Tier 1 venues (2013-2022), identifying challenges in reproducing results with author-provided artifacts and proposing mitigation strategies.

Replicability, in contrast, implies that an independent group can obtain the same results using artifacts they develop independently. Replicability studies generally involve different datasets and aim to confirm previous research results, considering the underlying system's inherent uncertainty.

Our study examines both aspects of machine learning research for Android malware detection. For instance, the analysis shown in Figure 1 underscores reproducibility issues in prior research, even when author-provided artifacts are used, due to unaccounted variability in published results. Additionally, we discuss pitfalls that must be addressed for a study's claims to be replicable with independently developed artifacts. In this paper, we use the term reproducibility to broadly refer to both computational reproducibility and replicability.

2.2. Android Malware Detection

Android malware detection using machine learning classifies applications as benign or malicious [17]. Three main types of features are used: static, dynamic, and hybrid. Static features are obtained by analyzing the app's source code or related information [1], [3], [10]. Specifically, the primary focus for Android applications is the APK file, the installation package, which includes components like AndroidManifest.xml and smali files obtained through decompilation. Dynamic features are acquired by observing the app's behavior in real or emulated environments, such as sandboxes [18]–[20]. Hybrid features combine both static and dynamic characteristics [21], [22].

Extracted features are input into machine learning models like Random Forests [11], Support Vector Machines [1], and Multilayer Perceptrons [13] to classify apps as malware or benign. The choice of features and models depends on detection objectives and resources. Static analysis is fast and suitable for large-scale detection, whereas dynamic analysis can be more accurate but resource-intensive [23]. This work focuses on static-feature-based machine learning methods due to their prevalence in the literature [9], [10], [24]–[26].

Concept Drift: Malware detection systems encounter challenges due to the evolving nature of malware, leading to concept drift. This drift can arise from new malware families, behavioral changes, evasion attempts, or updates

TABLE 1: Summary statistics of the datasets used in the study

Dataset	Split	Duration	Benign Apps	Malicious Apps	Total	Malware Families	Feature Dimension
Drebin	Train	2019-01 to 2019-12	40,947	4,542	45,489	121	16,978
	Validation	2020-01 to 2020-06	18,109	2,028	20,137	67	
	Test	2020-07 to 2021-12	30,797	3,631	34,428	71	
APIGraph	Train	2012-01 to 2012-12	27,472	3,061	30,533	104	1,159
	Validation	2013-01 to 2013-06	21,310	2,366	23,676	115	
	Test	2013-07 to 2018-12	240,729	25,377	266,106	418	

in API semantics. Traditional supervised learning models trained on static datasets may become outdated as new malware types emerge, reducing detection accuracy [14], [25], [27], [28]. These models often fail to recognize emerging variants, resulting in higher false negatives. To address this, continuous learning methods can enable models to adapt to new data continuously and stay current with malware trends [9], [24].

Reproducibility of Research in Android Malware Detection Using ML: Previous studies have assessed issues in reported results for Android malware detection [29], [30], attempted to replicate prior research [31], or addressed limitations in practical environments [32]. However, these studies do not consider the continuous learning setup, often rely on single global metrics like AUC or F1-score, lacking nuanced analysis of monthly performance variations, factors influencing reproducibility, and inherent variances in machine learning models due to their stochastic nature. Our work aims to address these gaps by analyzing conditions that can lead to poor reproducibility in different experimental settings and how to mitigate them.

3. Study settings

3.1. Machine Learning Scenarios

We evaluate Android malware detection models under 2 distinct ML scenarios relevant to practical security deployment:

1) **Offline Learning:** Models are trained once on the historical dataset and then evaluated on unseen future data, reflecting static deployment scenarios. This involves using annotated data from a specific period (e.g., one year) to train a model.
2) **Continuous Active Learning:** To address the evolving nature of malware (concept drift), we periodically retrain the model [9], [10], [24] using only the samples where the model is least confident, saving annotation resources. We adopt monthly retraining intervals, consistent with established literature.

We treat these settings separately because each requires different model selection methods, such as hyperparameter tuning, as noted in prior work [9].

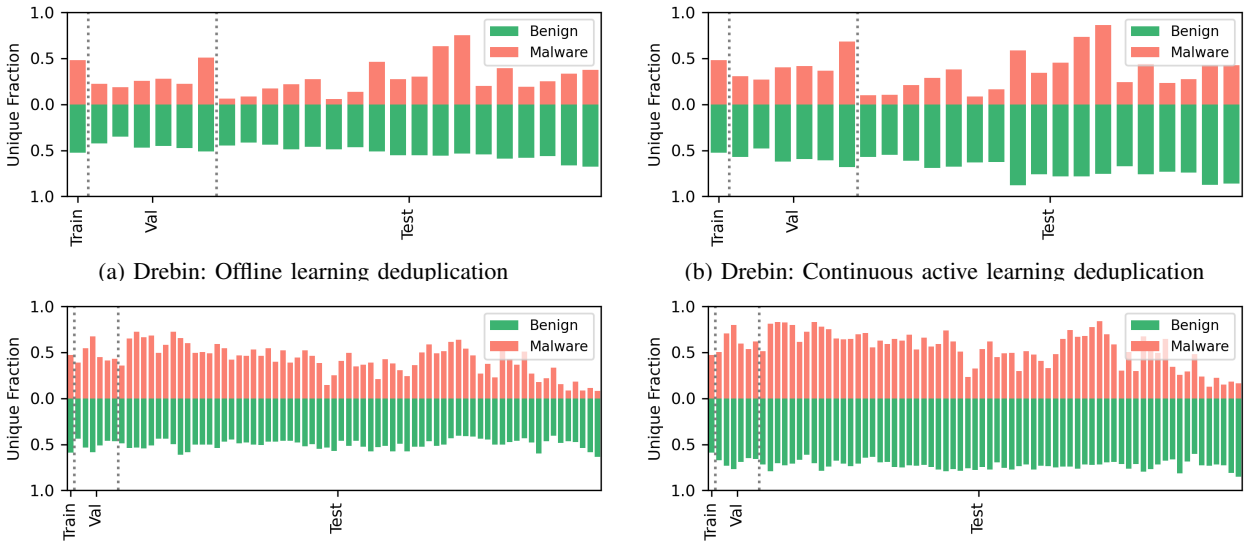
3.2. Datasets

We utilize the publicly available dataset from [9], which includes two datasets based on static feature types: Drebin [1] and APIGraph [10]. For brevity, we refer to them as the Drebin and APIGraph datasets, although the Drebin variant used here was collected between 2019 and 2021, rather than the original 2010–2012 corpus [1]. This ensures comparability with the evaluation setup of [9]. Both datasets have been extensively applied across various contexts, including explainable ML [33], concept drift analysis [26], [32], obfuscation studies [34], and semi-supervised learning [35], [36], reflecting their central role in the literature. However, despite their broad use, prior studies have not adequately examined the inherent reproducibility challenges in these datasets, which complicates efforts to replicate results in new environments. Addressing these issues provides the primary motivation for our work.

Drebin uses eight different sets of features, including access to hardware components, requested permissions, app components, intents, usage of restricted API calls, used permissions, suspicious API calls, and network addresses. APIGraph employs API semantics to cluster similar APIs, significantly reducing the feature space. The resulting feature set is binary for both datasets, indicating whether a feature is present or absent for a specific sample.

The Drebin dataset contains applications collected from 2019 to 2021, while the APIGraph dataset includes applications from 2012 to 2018. Both datasets address temporal and spatial biases present in malware datasets [5], [25]. They consist of approximately 90% benign and 10% malware applications collected from each month, reflecting their real-world ratios. Samples are ordered and approximately evenly distributed over their respective periods. Table 1 provides a summary of the datasets.

Following [9], we use data from the first year as the training set, the next six months for validation, and the remaining months for testing. For offline learning, only the training set data is used to train the model. In the continuous active learning setup, this data is used to train the initial model, which is then periodically retrained using data from subsequent months from the validation and test set. The validation set is used to tune the model hyperparameters in both experimental settings.



(a) Drebin: Offline learning deduplication

(b) Drebin: Continuous active learning deduplication

(c) APIGraph: Offline learning deduplication

(d) APIGraph: Continuous active learning deduplication

Figure 2: Fraction of unique samples retained after the deduplication process

4. Pitfalls in Reproducibility and Evaluation

We critically examine key methodological and dataset-related pitfalls prevalent in existing Android malware detection research. These pitfalls can severely undermine reproducibility and practical security effectiveness, and we propose systematic strategies to mitigate their impact.

4.1. Dataset Duplication

Issues of duplicates in Android malware datasets have been reported in prior studies [29], [30]. The study by [30] analyzed duplicates in the original Drebin dataset [1] using opcode sequences and found that 50.65% of malware samples were unique, with the rest being duplicates. They demonstrated that removing duplicates can alter the performance rankings of different machine learning models. [29] examined duplicates in four malware datasets using APK, DEX, opcode sequences, and API calls to identify them. They also evaluated the impact of duplicates on multiple machine-learning models.

Our study differs from prior works in key ways. First, we are the first to address the reproducibility of reported results in this context. Second, we provide a time-aware analysis of duplicates' effects on malware detection models, as global metrics (e.g., F1-score) may miss monthly variations [25]. Lastly, unlike prior studies focused only on offline learning, we assess duplicates' impact in offline and continuous learning settings, which require different deduplication approaches, as discussed later.

4.1.1. Defining Duplicates. We define duplicates based strictly on feature-vector equality, meaning two samples are duplicates if their extracted feature vectors are identical. Specifically, two samples X and Z are considered duplicates if $x_1 = z_1, x_2 = z_2, \ldots, x_n = z_n$ for an n-dimensional feature vector:

$$X = (x_1, x_2, \ldots, x_n), \quad Z = (z_1, z_2, \ldots, z_n)$$

This definition directly reflects the model's perspective, differing from prior approaches relying on raw or opcode-based duplication checks [29], [30]. Notably, despite using this feature-space definition, we observe a similar percentage of duplicates in the datasets as reported in prior studies. Also, detecting accurate duplicates is particularly challenging in malware analysis, which we further discuss in Section 10.

4.1.2. Deduplication. The process of *deduplication* ensures that the dataset contains unique samples. We establish rigorous deduplication protocols based on different experimental settings in our study:

Offline Learning Deduplication: Here, the model is trained once on a training set and evaluated on validation and test sets. Deduplication involves selecting unique samples within and across data splits. Since the model is evaluated monthly, two levels of deduplication are required. *First*, we remove samples that appear in earlier data splits: duplicates in the validation set are removed if present in the training set, while the test set excludes duplicates from both the training and validation sets. *Second*, within each split, only the earliest occurrence of a sample is retained based on its appearance time.

Continuous Active Learning Deduplication: Here, the definition of duplicates evolves due to monthly model retraining and changes in behavior, such as catastrophic forgetting [37]. Duplicate samples are first removed from the training split, following the same approach as offline learning. For the validation and test sets, duplicates are removed only within the same month, not across different months. This allows duplicates to appear in multiple months

of the test set while ensuring that each month contains unique instances. This approach is crucial for continuous learning, as periodic retraining requires the model to recognize recurring samples accurately.

4.1.3. Duplicate Statistics in the Datasets. Figure 2 illustrates the proportion of unique samples in the datasets after the deduplication process in both experimental settings. Ideally, this fraction would be 1 for both malware and benign apps, indicating no duplicates. However, our analysis shows that many duplicates are present, with varying degrees across different splits. Notably, around 50% of the training set for the Drebin and APIGraph datasets consists of duplicate samples in benign and malware categories.

We observe two key patterns regarding duplicates in the validation and test sets. First, the fraction of duplicate samples varies monthly. Second, malware samples tend to contain more duplicates than benign samples, with certain months particularly affected. For instance, in the first test month of the Drebin dataset (July 2020), only 6.30% of malware samples are unique in the offline-learning setting, and 9.96% are unique in the continuous-learning setting.

4.2. Inadequate Model Selection and Tuning

A common approach for evaluating machine learning models involves splitting the dataset into training, validation, and test sets. The validation set is essential for model selection and hyperparameter tuning to prevent biased parameter choices [5]. However, previous research on Android malware detection often omits a separate validation set, leading to incomplete evaluations [5]. Our study adopts the same train, validation, and test split as in [9] for offline and continuous learning settings.

Another common issue is the inadequate tuning of baseline methods [5]. This often results in insufficient hyperparameter tuning for simpler models compared to more complex ones. For example, prior research has shown that appropriately tuned tree-based methods frequently outperform deep neural networks on tabular datasets [38], [39], even though neural networks excel in tasks like computer vision and natural language processing [**?**], [40], [41].

In Android malware detection, various models have been used in literature, such as support vector machines (SVM), random forests, gradient-boosted decision trees, and deep neural networks [1], [3], [9], [10], [25]. However, these models might not have been adequately calibrated due to the absence of a validation set or insufficient hyperparameter searches. Our analysis emphasizes extensive hyperparameter searches across different baseline methods, ensuring the same hyperparameter budget is allocated to each model. This approach enables a fair comparison between models and aligns with the established norms in the domain generalization literature in machine learning [42].

4.3. Accounting for Variance from Random Initialization

Some machine learning models, particularly neural networks, exhibit additional stochasticity independent of their hyperparameters. Their sensitivity to initial random weight initialization can lead to variations in results that depend on the training system rather than the model parameters [43], [44]. It is common practice in the deep learning community to evaluate performance across multiple random seeds on the test set and report the average [45] to address this. However, prior studies on Android malware detection often neglect this variation, leading to irreproducible outcomes when experiments are conducted with different random seeds. We explore this issue in detail in Section 6.2.

4.4. Delayed Evaluation of Models

Splitting the dataset into non-overlapping temporal train, validation, and test sets enables consistent model selection but introduces a delay between training and test data. While acceptable in standard supervised learning, this temporal gap poses challenges for malware detection, where data distributions evolve. As observed in our study and in [9], a six-month delay between training and test periods can lead to unrealistic evaluations, since deployed models would typically be retrained with more recent data [5].

To mitigate this, we merge the training and validation datasets after hyperparameter tuning and evaluate on the test set. This **Merged Training** setup offers a fairer estimate of offline performance compared to the **Holdout Training** setup, where only the training data are used. Such merging is standard in temporal domain generalization [46], and the impact of both setups is analyzed in Sections 6 and 7.

5. Experiments

5.1. Machine Learning Models

We utilize six different models commonly used in prior works on Android malware detection:

1) **Random Forest (RF):** Random Forest [11] is an ensemble learning method for classification that constructs multiple decision trees during training and outputs the mode of their classes. It has been used in Android malware detection [3], [47].
2) **Support Vector Machine (SVM):** SVM [48] is a classification algorithm that finds the hyperplane that best separates classes in the feature space. It has been widely used for Android malware detection, particularly with Drebin features [1], [24].
3) **eXtreme Gradient Boosting (XGBoost):** XGBoost [12] is a scalable and efficient gradient boosting method that excels in performance on various tabular datasets [38], although it has not shown consistent success in Android malware detection [9].

4) **Multilayer Perceptron (MLP):** An MLP is a type of neural network composed of multiple layers of neurons, with each layer fully connected to the next [13]. MLPs are used for a variety of tasks, including classification and regression, and have been employed in several prior works for Android malware detection [25], [49], [50].
5) **Supervised Contrastive Classifier (SCC):** Contrastive learning has proven effective for detecting and adapting to concept drift in Android malware detection [9], [14]. It learns representations by bringing similar instances closer and pushing dissimilar ones apart [51], [52]. Our implementation uses a triplet loss over the two main classes, benign and malware, where positives share the same label as the anchor and negatives differ. Unlike prior works that model each malware family as a separate class, our binary setup is more annotation efficient and better suited for real world detection.
The learning process combines a supervised binary cross-entropy loss and a triplet margin loss. The binary cross-entropy loss is defined as:

$$\mathcal{L}_{\text{bce}} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$

where y_i is the true label and $\hat{y}_i$ is the predicted probability.
The triplet margin loss is given by:

$$\mathcal{L}_{\text{triplet}} = \frac{1}{N}\sum_{i=1}^{N}\max\Big(0,\ \|\text{enc}(x_i^a) - \text{enc}(x_i^p)\|_2^2 - \|\text{enc}(x_i^a) - \text{enc}(x_i^n)\|_2^2 + m\Big)$$

where x_i^a, x_i^p, x_i^n are the anchor, positive, and negative samples respectively, $\text{enc}(\cdot)$ is the encoder function, and m is the margin parameter.
The combined loss function is:

$$\mathcal{L} = \lambda \cdot \mathcal{L}_{\text{bce}} + \mathcal{L}_{\text{triplet}}$$

where λ is a weight parameter balancing the two loss components.
6) **Hierarchical Contrastive Classifier (HCC):** Following [9], HCC extends supervised contrastive learning with a hierarchical loss that enforces stronger similarity among samples from the same malware family than those from different families. Using an encoder enc, embeddings of benign or cross-family malware pairs satisfy $\|\text{enc}(x_1) - \text{enc}(x_2)\|_2 \leq m$, while benign–malicious pairs are pushed farther apart with $\|\text{enc}(x_1) - \text{enc}(x_2)\|_2 \geq 2m$.
Let d_{ij} denote the Euclidean distance between two samples i and j in the embedding space, defined as $d_{ij} = \|\text{enc}(x_i) - \text{enc}(x_j)\|_2$. The hierarchical contrastive loss is:

$$\begin{aligned}\mathcal{L}_{hc}(i) =& \frac{1}{|P(i,y_i,y_i')|}\sum_{j\in P(i,y_i,y_i')}\max(0, d_{ij}-m)\\ &+\frac{1}{|P_z(i,y_i,y_i')|}\sum_{j\in P_z(i,y_i,y_i')} d_{ij}\\ &+\frac{1}{|N(i,y_i)|}\sum_{j\in N(i,y_i)}\max(0, 2m-d_{ij})\end{aligned}$$

The first term ensures positive pairs, such as (benign, benign) or (malicious, malicious), are close but not overly constrained, penalizing distances only if they exceed m. The second term enforces similarity within the same malware family. The last term aims to separate benign and malicious samples by at least $2m$.
Like the SCC, this loss is combined with binary cross-entropy loss to train the model. Note that this is the only model that requires access to malware family labels to train the model.

5.2. Active Learning Sample Selection Strategy

Active learning seeks to improve model performance with minimal labeled data by selectively querying the most informative samples from a large pool of unlabeled data [53]. For all models except HCC, we employ the standard uncertainty sampling strategy [54], which selects samples where the model is least confident, computed as $1 -$ probability (e.g., the softmax output for binary classifiers). For HCC, we follow the pseudo-label selection method of [9], where pseudo-labels act as ground truth for calculating a contrastive pseudo-loss between test and nearby training embeddings. This loss is combined with binary cross-entropy to identify samples with the highest uncertainty.

5.3. Hyperparameter Tuning

We perform hyperparameter tuning independently for two experimental setups: offline learning and continuous active learning. In the offline setup, each model is trained once on the training dataset, using the validation set to select optimal hyperparameters. We apply random search [55] with a budget of 200 trials per model, ensuring fair comparison across models and accounting for differences in search spaces, and repeat tuning separately for duplicated and deduplicated datasets. In the continuous active learning setup, following [9], the validation data is divided into six-month intervals; the model is retrained with active learning at each interval and evaluated on the subsequent month. Most hyperparameters remain fixed across retraining phases, though parameters such as training epochs may vary. Random search with 100 trials per model and an annotation budget of 50 samples per month is used for tuning. Full hyperparameter details are provided in Appendix A.2.

TABLE 2: Performance comparison of different models on the Drebin dataset for offline learning. The results represent the mean and standard deviation over five runs with different random seeds. The best-performing model is underlined.

Model	Merged Training						Holdout Training					
	Duplicated			Deduplicated			Duplicated			Deduplicated		
	F1	FPR	FNR	F1	FPR	FNR	F1	FPR	FNR	F1	FPR	FNR
RF	46.7±0.99	0.14±0.00	66.8±0.74	41.5±0.30	0.08±0.00	72.5±0.25	39.4±1.31	0.16±0.03	72.4±1.03	33.2±1.01	0.14±0.04	78.7±0.71
SVM	63.6±0.00	0.80±0.00	47.0±0.00	46.6±0.00	0.60±0.00	65.6±0.00	47.2±0.00	0.92±0.00	63.2±0.00	33.8±0.00	0.70±0.00	76.7±0.00
XGBoost	63.7±2.45	0.18±0.01	48.9±2.80	59.5±0.46	0.39±0.02	52.1±0.59	45.2±1.92	0.24±0.04	67.3±2.01	47.3±0.78	0.73±0.02	63.4±0.66
MLP	66.2±3.69	0.41±0.08	45.8±4.10	53.6±2.08	0.39±0.08	59.5±2.12	44.3±2.07	0.87±0.06	66.7±1.72	40.9±1.05	0.57±0.04	69.8±0.82
SCC	74.7±0.78	0.53±0.09	35.0±1.51	57.4±0.44	0.63±0.25	53.7±2.11	45.6±1.18	0.70±0.11	65.7±1.35	37.3±0.26	0.63±0.10	73.0±0.61
HCC	68.4±1.73	0.82±0.21	41.3±2.52	56.7±0.79	1.04±0.12	50.5±0.76	44.9±1.60	0.77±0.11	66.2±1.60	35.8±0.65	0.84±0.15	73.5±1.14

TABLE 3: Performance comparison of different models on the APIGraph Dataset for offline learning

Model	Merged Training						Holdout Training					
	Duplicated			Deduplicated			Duplicated			Deduplicated		
	F1	FPR	FNR	F1	FPR	FNR	F1	FPR	FNR	F1	FPR	FNR
RF	64.6±1.02	0.23±0.00	49.6±1.08	61.3±0.92	0.50±0.03	51.7±0.79	45.3±5.22	0.12±0.02	68.5±4.50	46.8±3.89	0.34±0.08	67.0±3.76
SVM	76.0±0.00	1.20±0.00	30.7±0.00	69.7±0.00	1.41±0.00	34.9±0.00	74.5±0.00	0.86±0.00	34.7±0.00	66.3±0.00	1.42±0.00	39.5±0.00
XGBoost	74.6±0.47	1.21±0.09	32.5±0.49	69.0±0.45	1.61±0.03	34.3±0.57	69.5±1.40	1.03±0.06	40.2±1.65	65.2±0.62	1.33±0.05	41.6±0.88
MLP	68.6±2.28	0.85±0.26	41.9±3.77	63.0±2.73	0.96±0.37	46.32±4.9	55.6±4.26	0.90±0.20	56.2±4.85	57.0±3.52	1.67±0.64	49.9±6.78
SCC	66.1±2.79	1.06±0.26	43.8±3.94	66.2±2.09	2.11±0.57	35.2±1.98	70.36±2.9	1.79±0.16	35.3±3.33	57.0±3.52	1.67±0.64	49.9±6.78
HCC	73.3±2.37	1.35±0.15	33.3±3.90	66.0±1.75	0.92±0.17	42.8±2.66	64.2±5.17	1.60±0.20	43.8±6.61	62.6±2.74	1.59±0.26	43.8±4.13

5.4. Evaluation Metric

After hyperparameter tuning, we evaluate the models on the test set. The continuous active learning setup includes retraining the model each test month and evaluating it the following month. We use the same performance metrics as in [9]: average F1-score, False Positive Rate (FPR), and False Negative Rate (FNR). We evaluate all models on the test set using optimal hyperparameters and five different random seeds, reporting the mean and standard deviation of the metrics. It is worth noting that SVM produces deterministic results across seeds, as it uses the same training dataset without any stochastic elements.

5.5. Experimental Settings

We summarize the different experimental settings below:

1) **Learning Settings:**
   - *Offline learning*: The model is trained once.
   - *Active learning*: The model is retrained monthly with a fixed annotation budget. For RF, SVM, and XGBoost, we train from scratch, while for neural networks, we continue fine-tuning the previous model, following recommendations from [9].
2) **Duplicates:**
   - *Duplicated*: We use the original datasets containing duplicated samples in the train, validation, and test splits.
   - *Deduplicated*: We deduplicate the datasets following the procedures in Section 4.1.2 before conducting the experiments.
3) **Delayed Evaluation:** Based on whether validation data is merged with training data after hyperparameter tuning(4.4):
   - *Merged Training*: Merge validation data with training data before evaluating the model on test data.
   - *Holdout Training*: Train the model using only training data.
4) **Active Learning Budget:** We evaluate two monthly annotation budgets for the active learning setup: 50 and 100 samples. Both settings use the model optimized with a 50-sample budget for hyperparameter tuning.

6. Offline Learning Results & Analysis

6.1. Malware Detection Performance

We present the results of six models in Table 2 for the Drebin dataset and Table 3 for the APIGraph dataset. These results encompass the various experimental settings discussed previously. Key findings are summarized as follows:

1) **Impact of duplicates on model performance:** Deduplicated datasets generally perform worse than those with duplicates, indicating that duplicates may inflate performance metrics. For instance, in the Drebin dataset's merged training setting, the F1-score differences for the SVM and SCC models are 17.0 and 17.3, respectively, between duplicate and deduplicated datasets.

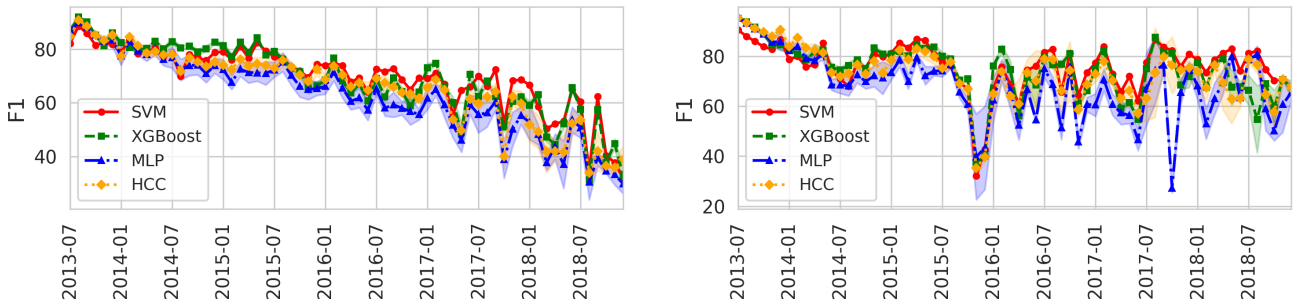
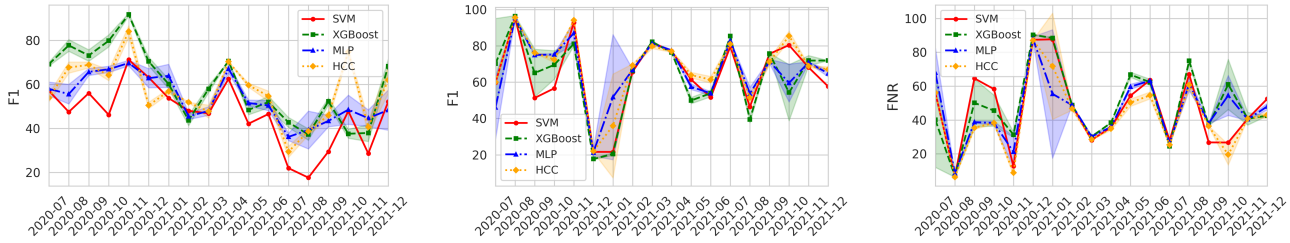


Figure 3: F1-score over the test months on the APIGraph for (left) deduplicated (right) duplicated datasets



(a) F1-score on deduplicated data (b) F1-score on duplicated dataset (c) FNR on duplicated dataset

Figure 4: Performance in test months on the Drebin datasets for Merged Training setting

2) **Benefits of merged training:** Merging validation data with training data before final evaluation consistently enhances performance. In the Drebin deduplicated dataset, the average F1-score improvement ranges from 8.3 (RF) to 20.9 (HCC), demonstrating the benefits of additional training data and its alignment with practical applications.
3) **Duplicates influence model preference:** Duplicates can skew model comparisons. For example, in the Drebin dataset merge-training setting, SCC outperforms XGBoost by 11 in average F1-score. However, after deduplication, XGBoost outperforms SCC by 2.1 in average F1-score.
4) **XGBoost as a strong baseline on deduplicated datasets:** With proper tuning, XGBoost achieves the highest average F1-score on the deduplicated Drebin dataset and remains competitive on APIGraph, highlighting its effectiveness as a baseline model. In contrast, while SVM performs well on APIGraph, its performance drops significantly on the Drebin dataset after deduplication.

6.2. Duplicates and Variance in Performance

Figures 3 and 4 show the month-by-month performance of four models (SVM, XGBoost, MLP, HCC) on the APIGraph and Drebin datasets under the merged training setting for both duplicated and deduplicated datasets. Key conclusions from the results are:

1) **Impact of duplicates on performance trends:** On both datasets, models generally exhibit a downward performance trend over time due to concept drift. However, this trend is less evident in the original datasets with duplicates, where sudden fluctuations occur. For example, all models experienced a performance drop in November 2015 on the APIGraph dataset, explained by duplicate inputs, as shown in Figure 5 (left). Of the 488 malware samples that month, 190 have identical input features. If a model fails to detect one correctly, it fails for all, resulting in a significant performance drop.
2) **High variance due to duplicates:** As seen in Figure 4(b), duplicates can cause high variance in model performance. Figure 4(c) shows that this effect is primarily due to variance introduced by false negative rates (FNRs). Figure 5(right) illustrates the frequency of unique, and top-10 duplicated malware samples for January 2021 on the Drebin dataset. Specifically, 348 out of 385 samples are duplicates of the same input feature, resulting in significant variance for models relying on random initialization, such as neural networks. Two of five MLPs trained with different seeds achieved high F1 scores (93.7 and 94.0), while the others scored much lower (21.4, 24.6, 24.8), indicating high fluctuations due to duplicates. A theoretical classifier correctly identifying all malware samples this month has an FNR of 0. If it fails to detect this single duplicated sample, the FNR rises to 90.4%. Conversely, a classifier detecting no samples has an FNR of 100%, but identifying only the duplicated sample reduces the FNR to 9.6%. This highlights

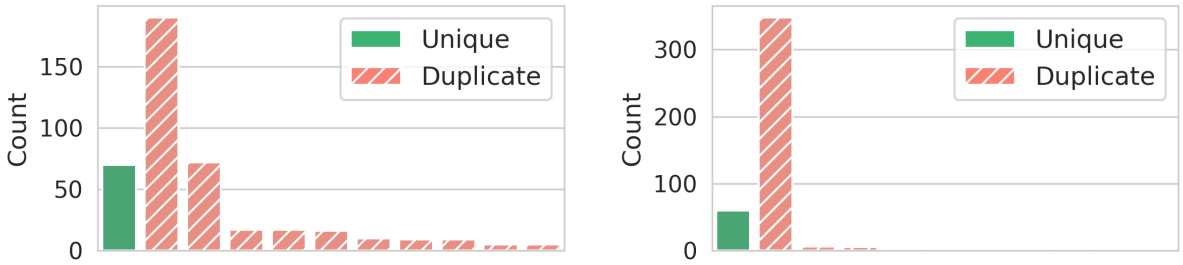


Figure 5: Frequency of unique and duplicate malware samples. Left: APIGraph in November 2015 with 190 of 488 samples sharing identical features. Right: Drebin in January 2021, with 348 of 385 samples duplicates of the same input.

the need to deduplicate datasets for models relying on random initialization to prevent performance disparities caused by different initializations.

3) **Importance of reporting performance with multiple seeds:** Although deduplication reduces extreme variance in model performance for certain months, neural networks can still show performance variation due to random initialization, as seen in Figure 4(a). For instance, in four test months (2021-01, 2021-08, 2021-10, 2021-12), the MLP had a standard deviation of average F1-scores greater than 5, even on the deduplicated Drebin dataset. Therefore, accounting for this randomness when reporting neural network performance is essential.

6.3. Effective Model Selection for Robust Baselines

Our offline learning experiments show that XGBoost consistently outperforms other models across settings, differing from the results in [9] primarily due to an expanded hyperparameter search space. We included additional parameters and wider ranges (see Appendix A.2), with the number of boosting rounds being particularly important. While [9] tested only 10–100 rounds, we found that up to 400 rounds significantly improved performance, especially on the APIGraph dataset. This highlights the importance of thorough hyperparameter tuning for establishing strong baselines [5]. Figure 6 shows the impact of boosting rounds on validation performance across datasets.

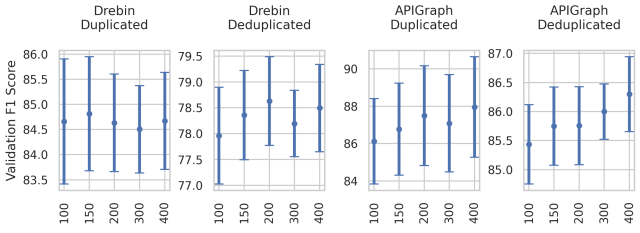


Figure 6: Effect of number of boosting rounds during learning for XGBoost model on different datasets

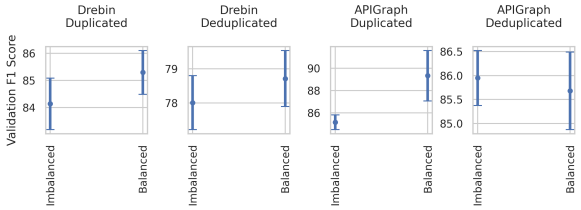


Figure 7: Effect of balancing the weight of the classes during learning for XGBoost model on different datasets

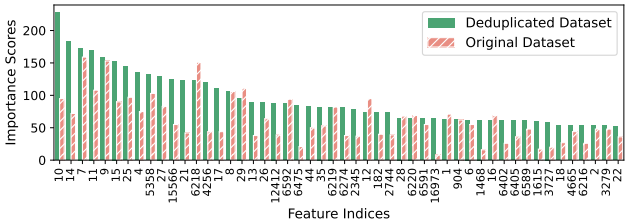


Figure 8: Comparison of feature weights for the top 50 features in XGBoost models trained on the deduplicated Drebin dataset and their corresponding weights in the model trained on the original duplicated dataset.

6.4. Duplicates & Model Selection

The presence of duplicates can affect hyperparameter tuning and model selection. As shown in Figure 7, adjusting the `scale_pos_weight` parameter to balance classes improves validation performance for duplicated datasets but can reduce it on the deduplicated APIGraph dataset, indicating that duplicates influence optimal hyperparameter behavior.

Figure 8 further shows that duplicates alter feature weighting in XGBoost. Comparing models trained on deduplicated and original Drebin datasets, the top 50 features from the deduplicated model display markedly different importance distributions when evaluated on the duplicated data (feature names in Appendix D).

TABLE 4: Performance of different models on the Drebin Dataset for continuous active learning with 50 & 100 annotation budget per month

Model	Budget=50						Budget=100					
	Merged Training			Holdout Training			Merged Training			Holdout Training		
	F1	FPR	FNR	F1	FPR	FNR	F1	FPR	FNR	F1	FPR	FNR
RF	61.9±0.73	0.13±0.02	53.0±0.88	62.7±1.11	0.11±0.00	52.4±1.11	62.8±1.24	0.13±0.01	52.1±1.30	63.6±0.47	0.12±0.03	51.3±0.40
SVM	64.4±0.00	0.22±0.00	48.7±0.00	59.6±0.00	0.23±0.00	54.1±0.00	68.1±0.00	0.24±0.00	44.6±0.00	63.9±0.00	0.26±0.00	49.1±0.00
XGBoost	82.2±0.60	0.15±0.00	27.3±0.76	81.1±0.15	0.14±0.01	29.2±0.21	82.5±0.29	0.15±0.01	26.9±0.32	82.5±0.45	0.14±0.01	27.0±0.57
MLP	67.9±0.65	0.49±0.03	43.3±0.70	61.8±0.54	0.35±0.02	51.0±0.49	70.2±0.42	0.38±0.03	40.6±0.46	65.7±3.37	0.36±0.06	46.2±4.17
SCC	71.0±0.64	0.59±0.04	37.4±0.29	68.2±1.83	0.50±0.07	41.6±2.21	73.7±1.20	0.57±0.10	33.7±1.35	72.1±0.88	0.48±0.07	37.0±1.44
HCC	73.5±0.27	0.66±0.04	32.8±0.85	68.5±3.38	0.57±0.05	40.3±4.37	75.6±0.96	0.68±0.06	29.4±1.07	72.6±2.10	0.57±0.02	35.2±2.64

TABLE 5: Performance of different models on the APIGraph Dataset for continuous active learning with 50 & 100 annotation budget per month

Model	Budget=50						Budget=100					
	Merged Training			Holdout Training			Merged Training			Holdout Training		
	F1	FPR	FNR	F1	FPR	FNR	F1	FPR	FNR	F1	FPR	FNR
RF	87.1±0.15	0.27±0.00	19.4±0.28	87.4±0.20	0.30±0.01	18.6±0.34	89.0±0.11	0.26±0.01	16.4±0.13	89.4±0.11	0.28±0.01	15.6±0.11
SVM	83.7±0.00	0.60±0.00	21.4±0.00	84.4±0.00	0.67±0.00	19.2±0.00	84.8±0.00	0.63±0.00	19.0±0.00	85.2±0.00	0.75±0.00	16.9±0.00
XGBoost	88.7±0.21	0.55±0.01	13.8±0.31	88.9±0.11	0.53±0.03	13.7±0.11	90.4±0.13	0.48±0.01	11.6±0.21	90.6±0.18	0.45±0.01	11.5±0.16
MLP	85.7±0.67	0.47±0.03	19.6±1.12	85.7±0.40	0.47±0.05	19.5±1.10	87.4±0.29	0.47±0.02	16.6±0.49	87.2±0.09	0.49±0.04	16.6±0.60
SCC	85.6±0.37	0.85±0.07	15.1±0.27	86.0±0.71	0.81±0.08	14.8±0.31	86.1±0.38	0.95±0.07	12.8±0.61	85.7±2.80	2.03±2.51	12.9±0.48
HCC	87.1±0.36	0.64±0.05	15.6±0.95	86.7±0.15	0.56±0.03	17.2±0.36	87.5±0.78	0.56±0.03	15.5±1.11	87.4±0.27	0.59±0.02	15.6±0.14

7. Continuous Active Learning Results & Analysis

In the continuous active learning setting, duplicates pose additional challenges beyond those in offline learning. Since active learning relies on human experts to select samples for annotation, increasing the diversity of selected samples is crucial [56]. Duplicates can lead to redundant selections, wasting the annotation budget. A simple heuristic can identify previously annotated samples and replace them with their existing annotations. Thus, using duplicated datasets in an active learning setting is impractical, and our analysis primarily focuses on deduplicated datasets.

We present the results of the six models in Tables 4 and 5 for the Drebin and APIGraph datasets, discussing merged and holdout training settings. Key findings are summarized below:

1) **XGBoost as a strong baseline in continuous active learning:** XGBoost achieves the highest average F1 score across all experimental settings in continuous active learning. This is especially evident in the challenging Drebin dataset, where it surpasses the second-best model by 8.7% in the merged training setting. This highlights the importance of selecting a strong baseline model and underscores XGBoost's effectiveness in Android malware detection.
2) **Benefits of Contrastive Learning for Neural Networks:** The neural networks trained with contrastive learning, SCC, and HCC outperform baseline neural networks in all settings, indicating the effectiveness of contrastive learning in enhancing robustness to concept drift. HCC further outperforms SCC, demonstrating the advantages of hierarchical contrastive learning and the pseudo-loss sample selector introduced in [9]. However, more improvements are needed to match XGBoost's performance.
3) **Random Forest as a Competitive Baseline on APIGraph:** Even simpler tree-based methods like Random Forest achieve competitive performance on the APIGraph dataset, often surpassing more complex neural networks. With consistently low false positives, RF is a strong candidate model for this dataset, illustrating that simpler models can compete effectively with complex ones in a compact feature space.
4) **Performance Gains from Merged Training:** Merging the validation set with the training set can enhance performance on test data, particularly for neural networks. However, the performance difference is less pronounced than offline learning since models are retrained during validation months with a subset of samples.

8. Post-hoc explainability

To examine how learned features change over time, we analyzed XGBoost, MLP, and HCC classifiers on the APIGraph dataset using SHAP-based explanations [57]. For each model, we constructed a reference set by taking the union of the top-10 most important features identified in the first and last months of the initial one-year period (July 2013–June 2014), using TreeSHAP for XGBoost and

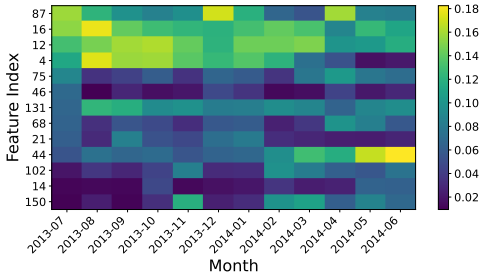


Figure 9: Evolution of normalized XGBoost feature importance from July 2013 to June 2014.

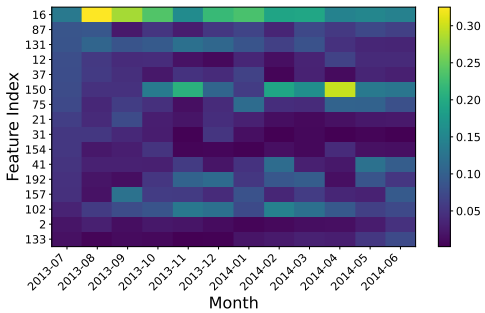


Figure 10: Evolution of feature importance for MLP models from July 2013 to June 2014.

KernelSHAP for MLP and HCC. We then fixed this set as a basis for comparison and, for each subsequent month, recomputed feature importances using correctly classified malware samples from that month's test data. All importances were normalized to ensure comparability. The resulting heatmaps (Figures 9, 10, and 11) reveal the temporal dynamics of feature relevance across models.

Figure 9 shows that in XGBoost models several features (e.g., Feature 12 [`GetDeviceID`], Feature 16 [`GetSubscriberID`], and Feature 87 [`SendSMS`]) remain consistently influential across months. In contrast, Feature 4 [`ReadPhoneStateUsed`] is prominent in the early months but declines in importance toward the end of the year, where Feature 44 [`ReadPhoneStateRequested`] gains greater weight.

For neural network models, the importance distribution is more static, with less variation across months. Despite their different training strategies (HCC being trained with hierarchical contrastive features), MLP and HCC show substantial overlap in their top features (e.g., the top four features are identical in the first month). While MLP consistently assigns high importance to Feature 16 [`GetSubscriberID`], HCC emphasizes both Feature 16 [`GetSubscriberID`] and Feature 87 [`SendSMS`] as particularly influential.

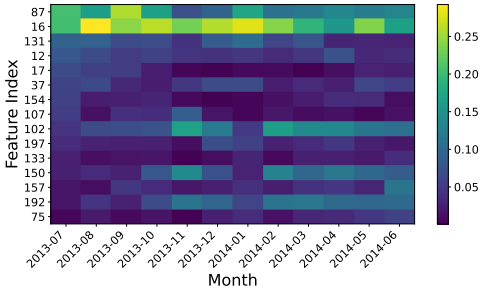


Figure 11: Evolution of normalized feature importance for HCC models from July 2013 to June 2014.

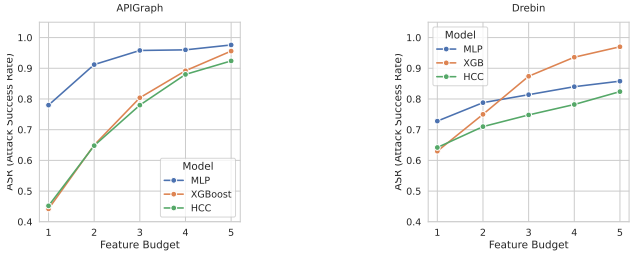


Figure 12: Comparison of attack success rate

9. Adversarial Robustness

We evaluate a pure black-box, score-based, feature-space evasion [58] on APIGraph and Drebin datasets. The Constrained Adaptive Attack (CAA) is a state-of-the-art method for adversarial evasion on tabular data that combines a fast gradient-based phase (CAPGD) with a stronger search phase. In our setting, we disable CAPGD and use only the search phase: a best-first beam search over single-bit add-only flips that batch-queries model scores, retains the lowest-score successors in a fixed-width beam (width 16; node cap 64), and halts upon reaching the benign threshold, exhausting the flip budget, or hitting the node cap. The attacker may flip up to B binary features per sample under an add-only constraint; all features are treated as mutable. We attack 500 randomly sampled malware test points per model, while benign samples remain unchanged.

Results are shown in Figure 12 for budgets $B \in \{1, 2, 3, 4, 5\}$. Even with small budgets, the attack achieves high success across all models. On APIGraph, XGBoost and HCC are more robust than MLP, whereas on Drebin, XGBoost shows higher ASR than MLP for larger budgets ($B = 3, 4, 5$), indicating that higher clean accuracy does not necessarily imply stronger adversarial robustness.

10. Limitations & Discussions

Computation Time. Figure 13 illustrates a comparison of the aggregate inference times for various machine learning models evaluated on the Drebin and APIGraph test datasets. The results show that neural models consistently incur higher inference latency compared to traditional

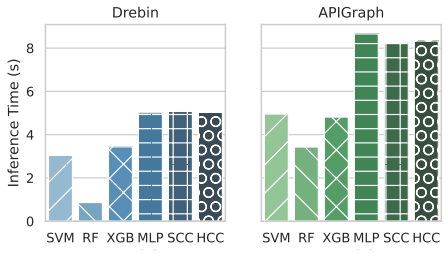


Figure 13: Inference time (in seconds) of different machine learning models on the Drebin (left) and APIGraph (right).

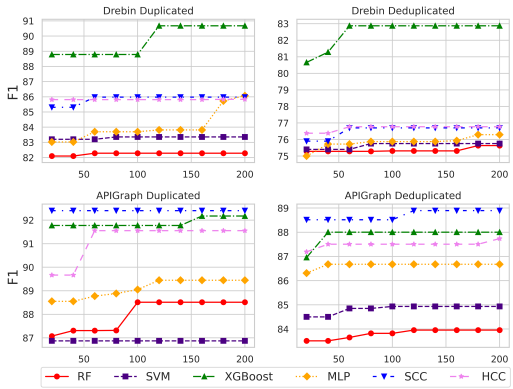


Figure 14: Performance on validation set as number of hyperparameter trials is increased for offline learning

models such as Random Forest (RF) and Support Vector Machine (SVM). The experiments were conducted using Python 3.11.6 on Red Hat Enterprise Linux 9.4, running on an Intel Xeon Gold CPU at 2.20 GHz.

Number of Trials and Performance Convergence. We use random search for hyperparameter tuning, following standard practice in domain generalization where training and test sets are not i.i.d. [42], [59]. This approach ensures fair comparison without biasing toward models with larger search spaces. To maintain rigor, we allocate sufficient trials for stable validation results. As shown in Figures 14 and 15, most models converge within 100 trials for offline learning and even fewer in continuous learning, indicating limited sensitivity to initial hyperparameters. While validation performance generally reflects test performance, exceptions exist, such as the RF model on the Drebin dataset.

Annotation Interval. We use a fixed interval for continuous active learning experiments, as widely adopted in prior work on Android malware detection [9], [24]–[26], [28], [35]. In practical deployments, concept drift detection could decide when to retrain the model [24], but this complicates comparisons across active learning methods since different models may trigger retraining at different times. Moreover, altering the retraining interval while keeping the annotation budget fixed can impact results; for example, increasing the interval to two months may reduce performance because the model is evaluated on a longer horizon of unseen data before retraining, compared to the current one-month setup.

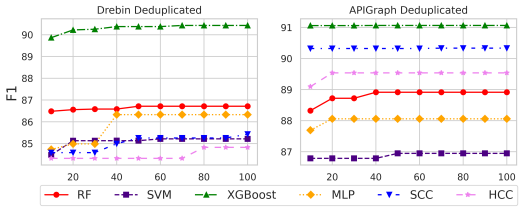


Figure 15: Validation performance versus number of hyperparameter trials in continuous active learning.

Annotation Budget in the Presence of Duplicates. In our active learning setup, we assume a fixed annotation budget. For practical deployment, however, duplicates introduce an additional challenge: the annotation process should prioritize uncertain samples while ensuring that no sample is annotated more than once, as repeated annotations waste valuable resources. While this constraint is relatively easy to enforce for the static-analysis features considered in our study, it can be far more challenging when dealing with dynamic features [17], [60].

Duplicate detection. We treat duplicates as samples that are indistinguishable in the model's feature space. This does not guarantee duplication in the raw input, since obfuscation, packing, and feature drift can map distinct binaries to identical features [61]–[63]. Accurate malware deduplication is difficult and outside our scope; our aim is to assess reproducibility under fixed feature sets and models. Because duplicate handling is induced by the feature extractor rather than the learner, fair model comparison requires deduplication on a common feature set or, at a minimum, explicit reporting of the resulting performance variance.

Label Noise and Obfuscation. Datasets with labels aggregated from sources such as VirusTotal may suffer from noise and inconsistencies in ground truth annotations, as noted in prior studies [23]. Similarly, malware developers often employ obfuscation techniques to evade detection. Both issues can affect model performance and bias comparisons across methods. Addressing these challenges is beyond the scope of our work; however, since our focus is on the reproducibility of machine learning models, we believe our findings remain applicable in such settings.

11. Conclusion

In this paper, we revisited key reproducibility and replicability challenges in static feature-based Android malware detection. We identified pitfalls in dataset curation, model selection, and evaluation, showing that issues such as data duplication, limited hyperparameter tuning, and biased baselines can cause major performance disparities. Through extensive experiments across models in offline and continuous active learning, we found that well-tuned simple models often match or surpass complex architectures. To promote transparency and fairness, we release an open-source framework for standardized, reproducible benchmarking of new models.

References

[1] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *NDSS*, vol. 14, 2014, pp. 23–26.

[2] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé *et al.*, "Deep android malware detection," in *Proceedings of the seventh ACM on conference on data and application security and privacy*, 2017, pp. 301–308.

[3] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. J. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[4] M. Gopinath and S. C. Sethuraman, "A comprehensive survey on deep learning based malware detection techniques," *Computer Science Review*, vol. 47, p. 100529, 2023.

[5] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3971–3988.

[6] R. Flood, G. Engelen, D. Aspinall, and L. Desmet, "Bad design smells in benchmark nids datasets," in *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2024, pp. 658–675.

[7] R. M. Verma, V. Zeng, and H. Faridi, "Data quality for security challenges: Case studies of phishing, malware and intrusion detection datasets," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2605–2607.

[8] D. Olszewski, A. Lu, C. Stillman, K. Warren, C. Kitroser, A. Pascual, D. Ukirde, K. Butler, and P. Traynor, "" get in researchers; we're measuring reproducibility": A reproducibility study of machine learning papers in tier 1 security conferences," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3433–3459.

[9] Y. Chen, Z. Ding, and D. Wagner, "Continuous learning for android malware detection," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1127–1144.

[10] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 757–770.

[11] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[12] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[14] L. Yang, W. Guo, Q. Hao, A. Ciptadi, A. Ahmadzadeh, X. Xing, and G. Wang, "{CADE}: Detecting and explaining concept drift samples for security applications," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2327–2344.

[15] "Artifact review and badging — current," https://www.acm.org/publications/policies/artifact-review-and-badging-current, Aug. 2020, accessed: 2025-09-20.

[16] O. E. Gundersen and S. Kjensmo, "State of the art: Reproducibility in artificial intelligence," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[17] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A review of android malware detection approaches based on machine learning," *IEEE access*, vol. 8, pp. 124 579–124 607, 2020.

[18] U. S. Jannat, S. M. Hasnayeen, M. K. B. Shuhan, and M. S. Ferdous, "Analysis and detection of malware in android applications using machine learning," in *2019 International Conference on Electrical, Computer and Communication Engineering (ECCE)*. IEEE, 2019, pp. 1–7.

[19] V. G. Shankar, G. Somani, M. S. Gaur, V. Laxmi, and M. Conti, "Androtaint: An efficient android malware detection framework using dynamic taint analysis," *2017 ISEA Asia security and privacy (ISEASP)*, pp. 1–13, 2017.

[20] J. Zhang, Z. Qin, K. Zhang, H. Yin, and J. Zou, "Dalvik opcode graph based android malware variants detection using global topology features," *IEEE Access*, vol. 6, pp. 51 964–51 974, 2018.

[21] M. Choudhary and B. Kishore, "Haamd: Hybrid analysis for android malware detection," in *2018 International Conference on Computer Communication and Informatics (ICCCI)*. IEEE, 2018, pp. 1–4.

[22] F. Martinelli, F. Mercaldo, and A. Saracino, "Bridemaid: An hybrid tool for accurate detection of android malware," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 899–901.

[23] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When malware is packin'heat; limits of machine learning classifiers based on static analysis features," in *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[24] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," in *26th USENIX security symposium (USENIX security 17)*, 2017, pp. 625–642.

[25] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: Eliminating experimental bias in malware classification across space and time," in *28th USENIX security symposium (USENIX Security 19)*, 2019, pp. 729–746.

[26] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro, "Transcending transcend: Revisiting malware classification in the presence of concept drift," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 805–823.

[27] T. Chow, Z. Kan, L. Linhardt, L. Cavallaro, D. Arp, and F. Pierazzi, "Drift forensics of malware classifiers," in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 2023, pp. 197–207.

[28] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: Self-evolving android malware detection system," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 47–62.

[29] Y. Zhao, L. Li, H. Wang, H. Cai, T. F. Bissyandé, J. Klein, and J. Grundy, "On the impact of sample duplication in machine-learning-based android malware detection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–38, 2021.

[30] P. Irolla and A. Dey, "The duplication issue within the drebin dataset," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 3, pp. 245–249, 2018.

[31] N. Daoudi, K. Allix, T. F. Bissyandé, and J. Klein, "Lessons learnt on reproducibility in machine learning based android malware detection," *Empirical Software Engineering*, vol. 26, no. 4, p. 74, 2021.

[32] C. Gao, G. Huang, H. Li, B. Wu, Y. Wu, and W. Yuan, "A comprehensive study of learning-based android malware detectors under challenging environments," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[33] Y. Liu, C. Tantithamthavorn, L. Li, and Y. Liu, "Explainable ai for android malware detection: Towards understanding why the models perform so well?" in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 169–180.

[34] C. Gao, M. Cai, S. Yin, G. Huang, H. Li, W. Yuan, and X. Luo, "Obfuscation-resilient android malware analysis based on complementary features," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 5056–5068, 2023.

[35] Z. Kan, F. Pendlebury, F. Pierazzi, and L. Cavallaro, "Investigating labelless drift adaptation for malware detection," in *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, 2021, pp. 123–134.

[36] M. T. Alam, R. Fieblinger, A. Mahara, and N. Rastogi, "Morph: Towards automated concept drift adaptation for malware detection," *arXiv preprint arXiv:2401.12790*, 2024.

[37] M. S. Rahman, S. Coull, and M. Wright, "On the limitations of continual learning for malware classification," in *Conference on Lifelong Learning Agents*. PMLR, 2022, pp. 564–582.

[38] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?" *Advances in neural information processing systems*, vol. 35, pp. 507–520, 2022.

[39] D. McElfresh, S. Khandagale, J. Valverde, V. Prasad C, G. Ramakrishnan, M. Goldblum, and C. White, "When do neural nets outperform boosted trees on tabular data?" *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[41] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[42] I. Gulrajani and D. Lopez-Paz, "In search of lost domain generalization," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=lQdXeXDoWtI

[43] X. Bouthillier, P. Delaunay, M. Bronzi, A. Trofimov, B. Nichyporuk, J. Szeto, N. Mohammadi Sepahvand, E. Raff, K. Madan, V. Voleti *et al.*, "Accounting for variance in machine learning benchmarks," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 747–769, 2021.

[44] D. Picard, "Torch. manual_seed (3407) is all you need: On the influence of random seeds in deep learning architectures for computer vision," *arXiv preprint arXiv:2109.08203*, 2021.

[45] X. Bouthillier and G. Varoquaux, "Survey of machine-learning experimental methods at neurips2019 and iclr2020," Ph.D. dissertation, Inria Saclay Ile de France, 2020.

[46] H. Yao, C. Choi, B. Cao, Y. Lee, P. W. W. Koh, and C. Finn, "Wild-time: A benchmark of in-the-wild distribution shift over time," *Advances in Neural Information Processing Systems*, vol. 35, pp. 10 309–10 324, 2022.

[47] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez, "Puma: Permission usage to detect malware in android," in *International joint conference CISIS'12-ICEUTE 12-SOCO 12 special sessions*. Springer, 2013, pp. 289–298.

[48] C. Cortes, "Support-vector networks," *Machine Learning*, 1995.

[49] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[50] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

[51] Z. Wu, Y. Xiong, S. X. Yu, and D. Lin, "Unsupervised feature learning via non-parametric instance discrimination," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 3733–3742.

[52] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, "Supervised contrastive learning," *Advances in neural information processing systems*, vol. 33, pp. 18 661–18 673, 2020.

[53] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, and X. Wang, "A survey of deep active learning," *ACM computing surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.

[54] D. D. Lewis and W. A. Gale, "A sequential algorithm for training text classifiers," in *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*, W. B. Croft and C. J. van Rijsbergen, Eds. ACM/Springer, 1994, pp. 3–12. [Online]. Available: https://doi.org/10.1007/978-1-4471-2099-5_1

[55] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.

[56] B. Settles, "Active learning literature survey," 2009.

[57] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in neural information processing systems*, vol. 30, 2017.

[58] T. Simonetto, S. Ghamizi, and M. Cordy, "Constrained adaptive attack: Effective adversarial attack against deep neural networks for tabular data," *Advances in Neural Information Processing Systems*, vol. 37, pp. 27 817–27 849, 2024.

[59] H. Yu, X. Zhang, R. Xu, J. Liu, Y. He, and P. Cui, "Rethinking the evaluation protocol of domain generalization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 21 897–21 908.

[60] A. Kapratwar, F. Di Troia, and M. Stamp, "Static and dynamic analysis of android malware," in *International Workshop on FORmal Methods for Security Engineering*, vol. 2. SciTePress, 2017, pp. 653–662.

[61] P. O'Kane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," *IEEE Security & Privacy*, vol. 9, no. 5, pp. 41–47, 2011.

[62] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "Appspear: Bytecode decrypting and dex reassembling for packed android malware," in *International Symposium on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 359–381.

[63] Z. Chen, Z. Zhang, Z. Kan, L. Yang, J. Cortellazzi, F. Pendlebury, F. Pierazzi, L. Cavallaro, and G. Wang, "Is it overkill? analyzing feature-space concept drift in malware detectors," in *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2023, pp. 21–28.

[64] R. Girshick, "Yacs – yet another configuration system," https://github.com/rbgirshick/yacs, 2020.

[65] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural networks: Tricks of the trade: Second edition*. Springer, 2012, pp. 437–478.

Appendix

1. Implementation Details

1.1. Environment. We conducted all experiments using Python 3.11.6 on Red Hat Enterprise Linux (Version 9.4). Neural network training was performed using a single NVIDIA A100 GPU. Table 6 lists the Python pip packages used in the implementation:

Pip Package	Version	Utility
scikit-learn	1.5.0	Used to train RF, SVM
torch	2.1.0	Used to train neural networks (MLP, SCC, HCC)
xgboost	2.1.0	Used to train XGBoost
numpy	1.26.4	Used to process datasets
pytorch-metric-learning	2.6.0	Used for creating data sampler for HCC
yacs	0.1.8	Configuration management for experiments

TABLE 6: Python pip packages used in the implementation

1.2. Code Structure. The code is organized into multiple key modules to facilitate easy extension:

1) **Dataset:** We preprocess data into a single `.npz` file containing features, labels, timestamps, and duplication indicators. The `intra_split_dupes` field marks duplicate within the current timestamp, referring to earlier indices or set to `-1` if unique. The `cross_split_dupes` field tracks duplicates across splits, indicating the relevant split and index or `none` if unique. This format supports active and offline learning deduplication. The `MalwareDataset` class handles dataset loading and includes the HalfSampler method used in HCC [9], as well as a class for working with Triplet Datasets.
2) **Models:** This module implements six models, all inheriting from a `BaseModel` class, which takes `params` for hyperparameters and `cfg` for experiment-specific settings (e.g., output classes, GPU usage). Each model includes methods for fitting (`fit`), prediction (`predict`, `predict_proba`), and active learning sample selection (`sample_active_learning`).
3) **Tasks:** We separate offline and active learning into distinct tasks within the `tasks` module, each with specific training and evaluation procedures. Tasks use a `yacs` configuration [64], provided as a YAML file, specifying the task type, model, and relevant settings. This separation allows easy extension for other malware analysis tasks, such as family classification or concept drift detection, with minimal code changes.

2. Hyperparameter Search-Space

The detailed hyperparameter search space is shown in Table 7. Most hyperparameter names follow standard terminologies in the scikit-learn, XGBoost, or PyTorch APIs. Below, we describe those with different or new terminology:

- **XGBoost/MLP/SCC - class-weight:** This parameter, "scale_pos_weight" in XGBoost, defaults to 1 if set to False. When True, it adjusts for class imbalance by weighting the positive class (malware) according to the benign-to-malware ratio in the training data. For MLP and SCC, samples are either balanced in each batch with equal benign and malware samples (balance=True) or drawn randomly (balance=False).
- **MLP/SCC/HCC - cont_learning_epochs:** In continuous active learning with neural networks, initial and retraining epochs differ, similar to [9]. For HCC, it specifies the number of epochs; for MLP and SCC, it represents the fraction of the original training epochs.
- **HCC - cont_learning_lr:** For HCC, the initial learning rate is varies the learning rate for each update, while MLP and SCC continue with the current rate using the Adam optimizer.
- **SCC/HCC - xent_lambda:** This controls the binary cross-entropy loss weight in contrastive learning. We use a value of 100, as suggested in [9].
- **SCC/HCC - margin:** This sets the margin for computing contrastive loss. We use a value of 10, following [9].

We adopt strategies from prior works on Android malware detection [9], machine learning on tabular data [39], and deep neural architectures [65] to design the hyperparameter search space.

While we mostly follow the hyperparameters for HCC from [9], there are a few minor differences. These choices ensure consistency with other neural network architectures and ease of implementation. We explore two architectures for the encoder and MLP layers, each instead of the single one used in their work. We include dropout as a hyperparameter instead of setting it to a fixed value of 0.2 in their study. Only the Adam optimizer is used during the continuous learning retraining phase. We employ a step learning rate scheduler and omit the cosine annealing scheduler. Regardless, we verify that our search space includes the best hyperparameter set reported in [9]; for instance, cosine annealing did not outperform step-based learning rates in any dataset in their results.

3. Variance in Results for Continuous Learning

Figure 16 shows the monthly F1 scores on the Drebin dataset in continuous active learning settings, comparing three neural networks on both duplicated and deduplicated datasets. Separate hyperparameter tuning was performed for MLP and SCC, while for HCC, we used the artifacts

TABLE 7: Hyperparameter Search Spaces for Different Models. U indicates sampling from a uniform random distribution over values in the range. * indicates hyperparameters that are only used during continuous learning retraining phase.

Model	Hyperparameter	Candidate Values
Random Forest	n_estimators	2^x, where $x \in \mathrm{U}[5, 10]$
	max_depth	2^y, where $y \in \mathrm{U}[5, 10]$
	criterion	{gini, entropy, log_loss}
	class_weight	{None, "balanced"}
SVM	C	10^z, where $z \in \mathrm{U}[-4, 3]$
	class_weight	{None, "balanced"}
XGBoost	max_depth	2^w, where $w \in \mathrm{U}[3, 7]$
	alpha	10^a, where $a \in \mathrm{U}[-8, 0]$
	lambda	10^b, where $b \in \mathrm{U}[-8, 0]$
	eta	3.0×10^c, where $c \in \mathrm{U}[-2, -1]$
	balance	{True, False}
	num_boost_round	{100, 150, 200, 300, 400}
	subsample	x, where $x \in \mathrm{U}[0.8, 1.0]$
	colsample_bytree	x, where $x \in \mathrm{U}[0.8, 1.0]$
MLP	mlp_layers	{[100, 100], [512, 256, 128], [512, 384, 256, 128], [512, 384, 256, 128, 64]}
	learning_rate	10^d, where $d \in \mathrm{U}[-5, -3]$
	dropout	x, where $x \in \mathrm{U}[0.0, 0.5]$
	batch_size	2^e, where $e \in$ {5, 6, 7, 8, 9, 10}
	epochs	{25, 30, 35, 40, 50, 60, 80, 100, 150}
	optimizer	{Adam}
	balance	{True, False}
	cont_learning_epochs*	{0.1, 0.2, 0.3, 0.4, 0.5}
SCC	encoder_layers	{[512, 256, 128], [512, 384, 256, 128]}
	mlp_layers	{[100], [100, 100]}
	learning_rate	10^f, where $f \in \mathrm{U}[-5, -3]$
	dropout	y, where $y \in \mathrm{U}[0.0, 0.25]$
	batch_size	2^g, where $g \in$ {9, 10, 11}
	epochs	{25, 30, 35, 40, 50, 60, 80, 100}
	xent_lambda	{100}
	margin	{10}
	optimizer	{Adam}
	balance	{True, False}
	cont_learning_epochs*	{0.1, 0.2, 0.3, 0.4, 0.5}
HCC	encoder_layers	{[512, 256, 128], [512, 384, 256, 128]}
	mlp_layers	{[100], [100, 100]}
	learning_rate	{0.001, 0.003, 0.005, 0.007}
	dropout	z, where $z \in \mathrm{U}[0.0, 0.25]$
	batch_size	2^{10}
	epochs	{100, 150, 200, 250}
	xent_lambda	{100}
	margin	{10}
	optimizer	{Adam, SGD}
	scheduler_step	{10}
	scheduler_gamma	{0.5, 0.95}
	cont_learning_lr*	{0.01, 0.05}
	cont_learning_epochs*	{50, 100}

provided in [9] for the duplicated setting to maintain consistency with their study. Notably, we observe extreme variance in model performance in certain months (e.g., December 2020) in the duplicated datasets.

4. Feature Index Mapping

Tables 8 and 9 show the feature index mappings for the important features used by different models on the APIGraph and Drebin datasets, respectively.

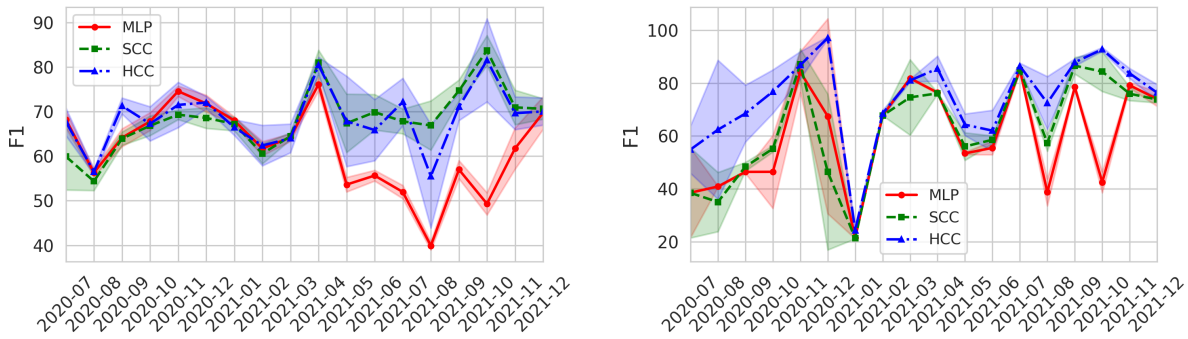


Figure 16: F1-score over the test months for active learning on the (left) deduplicated (right) duplicated Drebin datasets

TABLE 8: APIGraph API feature mapping

Feature ID	Feature Name
2	usedpermissionslist_android.permission.get_tasks
4	usedpermissionslist_android.permission.read_phone_state
12	suspiciousapilist_landroid/telephony/telephonymanager.getdeviceid
14	suspiciousapilist_system/bin/su
16	suspiciousapilist_landroid/telephony/telephonymanager.getsubscriberid
17	suspiciousapilist_landroid/content/pm/packagemanager.getpackageinfo
21	urldomainlist_10.0.0.172
31	activitylist_com.waps.offerswebview
37	requestedpermissionlist_android.permission.receive_boot_completed
41	requestedpermissionlist_android.permission.get_tasks
44	requestedpermissionlist_android.permission.read_phone_state
46	suspiciousapilist_landroid/telephony/gsm/smsmanager.sendtextmessage
68	usedpermissionslist_android.permission.send_sms
75	suspiciousapilist_landroid/telephony/smsmanager.sendtextmessage
87	requestedpermissionlist_android.permission.send_sms
102	intentfilterlist_android.intent.action.package_added
107	intentfilterlist_android.intent.action.package_removed
131	requestedpermissionlist_com.android.launcher.permission.install_shortcut
133	requestedpermissionlist_com.android.browser.permission.read_history_bookmarks
150	intentfilterlist_android.intent.action.user_present
154	requestedpermissionlist_android.permission.read_contacts
157	usedpermissionslist_android.permission.get_accounts
192	requestedpermissionlist_android.permission.system_alert_window
197	requestedpermissionlist_android.permission.call_phone

TABLE 9: Drebin API feature mapping

Feature ID	Feature Name
1	suspiciousapilist_android.content.res.assetmanager.open
2	suspiciousapilist_android.content.context.getassets
4	suspiciousapilist_android.app.application.oncreate
6	intentfilterlist_com.android.vending.install_referrer
7	intentfilterlist_android.intent.action.boot_completed
8	requestedpermissionlist_android.permission.read_external_storage
9	requestedpermissionlist_android.permission.receive_boot_completed
10	requestedpermissionlist_android.permission.system_alert_window
11	requestedpermissionlist_android.permission.get_tasks
12	requestedpermissionlist_android.permission.write_settings
13	requestedpermissionlist_android.permission.access_wifi_state
14	requestedpermissionlist_android.permission.read_phone_state
15	requestedpermissionlist_android.permission.change_wifi_state
16	requestedpermissionlist_android.permission.write_external_storage
17	suspiciousapilist_android.app.alertdialog\$builder.setcancelable
18	suspiciousapilist_android.app.alertdialog\$builder.setmessage
21	suspiciousapilist_android.view.window.settype
22	suspiciousapilist_android.content.res.resources.getassets
25	intentfilterlist_android.intent.action.view
26	requestedpermissionlist_android.permission.access_coarse_location
27	requestedpermissionlist_android.permission.camera
28	requestedpermissionlist_android.permission.access_fine_location
29	requestedpermissionlist_android.permission.vibrate
35	usedpermissionslist_android.permission.access_network_state
44	suspiciousapilist_android.app.application.getpackagename
182	suspiciousapilist_android.content.context.getsharedpreferences
904	suspiciousapilist_android.app.application.registeractivitylifecyclecallbacks
1468	suspiciousapilist_android.content.intent.setflags
1615	suspiciousapilist_android.widget.toast.show
2345	suspiciousapilist_android.content.pm.packagemanager\$namenotfoundexception.printstacktrace
2744	suspiciousapilist_android.telephony.telephonymanager.getdeviceid
3279	suspiciousapilist_ljava/io/ioexception;->printstacktrace
3727	suspiciousapilist_android.net.wifi.wifiinfo.getmacaddress
4256	suspiciousapilist_android.content.context.getdir
4665	suspiciousapilist_android.view.layoutinflater.from
5358	suspiciousapilist_ljava/lang/runtime;->exec
6216	intentfilterlist_com.google.android.c2dm.intent.receive
6218	contentproviderlist_android.support.v4.content.fileprovider
6219	hardwarecomponentslist_android.hardware.camera.autofocus
6220	hardwarecomponentslist_android.hardware.camera
6274	suspiciousapilist_android.view.surfaceview.<init>
6402	suspiciousapilist_android.app.activitymanager.getrunningservices
6405	suspiciousapilist_android.content.pm.packagemanager.getinstalledpackages
6475	suspiciousapilist_android.content.pm.applicationinfo.loadlabel
6589	activitylist_com.google.android.gms.ads.adactivity
6591	activitylist_com.google.android.gms.common.api.googleapiactivity
6592	requestedpermissionlist_android.permission.get_accounts
12412	requestedpermissionlist_com.android.vending.billing
15566	suspiciousapilist_android.content.pm.packagemanager.setcomponentenabledsetting
16973	contentproviderlist_mono.monoruntimeprovider