TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time (Extended Version)

ZELIANG KAN, King's College London, UK and University College London, UK
SHAE MCFADDEN, King's College London, UK and The Alan Turing Institute, UK
DANIEL ARP, TU Berlin, Germany
FEARGUS PENDLEBURY, University College London, UK
ROBERTO JORDANEY, Independent Researcher, UK
JOHANNES KINDER, Ludwig-Maximilians-Universität München, Germany
FABIO PIERAZZI, King's College London, UK
LORENZO CAVALLARO, University College London, UK

Machine learning (ML) plays a pivotal role in detecting malicious software. Despite the high F_1-scores reported in numerous studies reaching upwards of 0.99, the issue is not completely solved. Malware detectors often experience performance decay due to constantly evolving operating systems and attack methods, which can render previously learned knowledge insufficient for accurate decision-making on new inputs. This paper argues that commonly reported results are inflated due to two pervasive sources of experimental bias in the detection task: *spatial bias* caused by data distributions that are not representative of a real-world deployment; and *temporal bias* caused by incorrect time splits of data, leading to unrealistic configurations. To address these biases, we introduce a set of constraints for fair experiment design, and propose a new metric, AUT, for classifier robustness in real-world settings. We additionally propose an algorithm designed to tune training data to enhance classifier performance. Finally, we present TESSERACT, an open-source framework for realistic classifier comparison. Our evaluation encompasses both traditional ML and deep learning methods, examining published works on an extensive Android dataset with 259,230 samples over a five-year span. Additionally, we conduct case studies in the Windows PE and PDF domains. Our findings identify the existence of biases in previous studies and reveal that significant performance enhancements are possible through appropriate, periodic tuning. We explore how mitigation strategies may support in achieving a more stable and better performance over time by employing multiple strategies to delay performance decay.

CCS Concepts: • **Security and privacy → Software and application security**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: Concept Drift, Experimental Bias, Malware Detection, Performance Decay

111

1 INTRODUCTION

Machine learning (ML) has become a fundamental tool in malware research within the academic security community, with its applications spanning a wide range of domains. These include Windows malware [4, 18, 47, 67], PDF malware [38, 45, 62], malicious URLs [39, 64], malicious JavaScript [17, 56], and Android malware [6, 33, 46, 75]. While the high performance figures associated with machine learning solutions might lead one to believe that the issue of malware classification is practically solved, this is far from true.

Malware classifiers are challenged by the dynamic nature of operational environments and attacking techniques. The constant emergence of new malware variants and families often leads to a degradation in the predictive accuracy of these classifiers over time [46]. Ensuring temporal consistency is therefore a critical factor in evaluating the effectiveness of these classifiers. A skewed outcome can result if the experimental setup inadvertently provides the classifier with foreknowledge, leading to biased results [2, 48].

These biases are not isolated instances but are widespread in different security domains. In this paper, we primarily focus on the domain of Android malware, arguing that many detection approaches, including those cited [6, 20, 29, 33, 46, 65, 73, 74] and even encompassing our previous research, are often evaluated under conditions that do not accurately reflect real-world scenarios. Then in order to broaden our perspective, we also extend our investigation into the Windows and PDF malware domains by case study [4, 62].

We identify two types of experimental bias: *spatial bias* and *temporal bias*. *Spatial bias* occurs when there are unrealistic assumptions about the ratio of goodware to malware in the dataset, which must be consistently enforced during the testing phase to mimic a realistic scenario. The ratio of goodware to malware varies by domain. For example, most Android apps in the wild are goodware [3, 32, 42], whereas most URLs in software download events are malicious [44, 55]. *Temporal bias* occurs when there is a temporal inconsistency between the training and testing datasets, which can lead to the integration of future knowledge into the training phase or create unrealistic settings, especially for concept drift detection [2, 8, 48]. This issue is particularly concerning in the case of families of closely related malware, where including even one variant in the training set may enable the algorithm to identify many variants in the testing.

We believe that the pervasiveness of these issues is due to two main reasons: first, possible sources of evaluation bias are not common knowledge; second, accounting for time complicates the evaluation and does not allow a comparison to other approaches using headline evaluation metrics such as the F_1-Score or AUROC. We address these issues in this paper by systematizing evaluation bias for Android malware classification and providing new constraints for sound experiment design along with new metrics and framework support.

Prior work has investigated challenges and experimental bias in security evaluations [2, 7, 48, 57, 61, 70]. The *base-rate fallacy* [7] describes how evaluation metrics such as True Positive Rate (*TPR*) and False Positive Rate (*FPR*) are misleading in intrusion detection, due to significant class imbalance (e.g., most traffic is benign); in contrast, we identify and address experimental settings that give misleading results *regardless* of the adopted metrics—even when correct metrics are reported. Previous works by Sommer and Paxson [61], Rossow *et al.* [57], and Kouwe *et al.* [70] propose guidelines for sound security evaluations, but none of them identify temporal and spatial

bias or quantify the impact of errors on classifier performance. While Allix *et al.* [2] and Miller *et al.* [48] identify initial temporal constraints in Android malware classification, we demonstrate that recent work following their guidelines (e.g., [4, 6]) still suffer from temporal and spatial bias. Our paper is the first to identify and address these sources of bias with novel, actionable constraints, metrics, and tool support.

This article makes the following contributions:

- We identify two sources of bias that affect the evaluation of ML-based malware classifiers: *temporal* bias associated with incorrect train-test splits and *spatial* bias related to unrealistic assumptions in dataset distribution. As an extension of our previous conference paper, we experimentally verify our hypothesis on an extended Android dataset of 259,230 apps (with 10% malware) spanning over five years that, due to bias, performance can decrease up to 50% in practice in the well-known Android malware classifiers. Furthermore, we explore the impact of these biases on popular classifiers in other security domains, specifically targeting Windows PE [4] and PDF malware [62].
- We propose novel building blocks for more robust evaluations of malware classifiers: a set of spatio-temporal constraints to be enforced in experimental settings; a new metric, AUT, that captures a classifier's robustness to time decay with optional evaluation parameters, and allows for fair comparison of different algorithms; and an updated tuning algorithm that empirically optimizes the classification performance when malware represents the minority class. In comparing performances across Android, Windows PE, and PDF malware domains, we demonstrate how eliminating biases can provide counter-intuitive results on real performance. Additionally, our findings underscore the necessity of regular retraining and re-tuning to maintain stable classifier performance.
- We implement and publicly release the Python framework for our methodology, TESSERACT. Accompanied by a comprehensive user guide. It demonstrates how our findings evaluate the performance-cost trade-offs of solutions designed to mitigate time decay, such as active learning [60] and conformal prediction (CP) [59]. Additionally, we implement the state-of-the-art CP framework-TRANSCENDENT [8]-to be compatible with deep neural networks (DNN). This adaptation involves the implementation of a non-conformity measurement (NCM) based on the output from the SoftMax layer of a DNN model.

This article extends our work published at USENIX Security Symposium 2019 [54]. In this journal version, we focus on deepening and expanding the most promising aspects observed in our initial study on a larger dataset. A key enhancement is our exploration of the deep learning-based malware detection approach, previously referred to as 'DL' in [54] and detailed in Appendix A.1. This version presents an in-depth analysis of the deep learning approach, particularly examining how it performs in comparison to the traditional ML method when applied to identical spatial and temporal bias. Furthermore, we extend our exploration to additional security domains, specifically Windows PE and PDF malware. Our findings reveal that, even in these domains where concept drift is less pronounced, applying our proposed tuning algorithm to the training data can still enhance detection capabilities.

Our TESSERACT framework offers the research community a tool for producing consistent results, uncovering unexpected performance insights, and assessing classifiers in real-world industrial scenarios. While we make strides in understanding spatio-temporal biases in experimental setups, there remains a vast expanse of unexplored territory. The potential to further investigate diverse security domains, encompassing a range of malware families, operating systems, and digital environments, is substantial. We advocate for the broader security community to engage in this exploration, building upon the foundational principles we have laid out.

The structure of this article is as follows: In § 2, we present the background knowledge pertinent to our Android malware classification evaluation. Subsequently, in § 3, we identify and validate potential sources of experimental bias. Our methodology for conducting a spatio-temporal bias-free evaluation is delineated in § 4. In § 5, we assess the influence of these biases on the Windows PE and PDF domains by case study. § 6 demonstrates how TESSERACT can be utilized to compare and assess the trade-offs associated with various budget-constrained strategies to mitigate time decay. Then, we discuss related research in § 7. Lastly, we discuss guidelines, assumptions, limitations, and availability in § 8 and § 9, and make conclusions in § 10.

> **Note: Use of the term "bias"** In this article, the term *(experimental) bias* is employed to denote the discrepancies between an experimental setting and the conditions observed in genuine real-world deployments. These discrepancies might give rise to evaluations that are potentially misleading. It is pivotal to differentiate that we are not referencing the trade-off between classifier bias and variance as delineated in the conventional machine learning literature [11].

2 ANDROID MALWARE CLASSIFICATION

We first delve into the Android domain. In this section, we outline the reference of examined Android classification approaches in § 2.1. We also discuss the specific prevalence of malware within the Android domain in § 2.2, detail the composition of the Android dataset under validation in § 2.3, and elaborate on our chosen features and feature selection methodology § 2.4.

2.1 Reference Algorithms in Android Classification

In this study, we primarily focus on two popular malware detection approaches: DREBIN and DEEPDREBIN. We opt for DREBIN's approach and its feature because it is a highly regarded technique in state-of-the-art machine learning security research [8, 14, 19]. It utilizes a linear support vector machine (SVM) with high-dimensional binary feature vectors derived from efficient static analysis. Additionally, we incorporate DEEPDREBIN [33], a deep learning approach that uses the same input features as DREBIN. Deep learning models are known for their ability to discover latent feature spaces, thus potentially offering enhanced resilience against time decay [30]. Our previous work, published at USENIX Security 2019 [54], indicated that deep learning frameworks could outperform traditional machine learning methods in terms of robustness, it becomes interesting to investigate whether these initial findings persist under varied evaluation scenarios.

To ensure a robust experimental framework, we have thoroughly re-implemented both DREBIN and DEEPDREBIN based on the detailed methodologies outlined in their foundational papers [6, 33]. Successfully replicating baseline results from these studies highlights their scientific rigor and forms the foundation upon which we build our research. These methods are selected for their accessibility and the stable baselines they provide, allowing us to concentrate on the implications of our findings within the wider malware detection research field. The hyper-parameters of the re-implemented algorithms are detailed in Appendix A.1. In contrast, we have excluded MAMADROID [46] from this study due to its limited efficacy, as observed in our previous analysis spanning 2014 to 2016. Our findings indicated that MAMADROID's performance had plateaued, making further exploration of its Markov-chain feature inefficient.

2.2 Estimating in-the-wild Malware Ratio

The proportion of malware in a dataset can have a significant impact on the performance of a classifier (see § 3). In order to conduct unbiased experiments, it is crucial to use a dataset with a

realistic percentage of malware to goodware. This can be achieved by down-sampling the majority class, for instance. However, it is important to note that each malware domain has its own unique ratio of malware to goodware that is typically encountered in the wild. It is therefore essential to determine whether malware is a minority, majority, or equal-sized class compared to goodware. For example, malware is the minority class in network traffic [7] and Android [42], but it is the majority class in binary download events [55]. Estimating the percentage of malware in the wild for a given domain can be challenging, but it can be informed by measurement papers, industry telemetry, and publicly available reports.

In the Android ecosystem, there is variation in the proportion of malware among all the apps. For instance, industry telemetry by a key player suggests that around 6% of Android apps are malicious. On the other hand, AndRadar measurement study [42] indicates that the ratio of Android malware in the wild is approximately 8%. The 2017 Google's Android security report [32] estimates the malware percentage to be between 6% and 10%, while in 2023 the AndroZoo [3] dataset analysis reveals an incidence of 16.3% of malicious apps among the whole dataset. This data suggests that, on average, malware is the minority class in the Android domain. To stabilize the percentage of malware in our dataset, we choose to set it to 10%, which is an approximate average across different estimates, with monthly values ranging from 8% to 12%. This decision allows us to collect a dataset with statistically significant per-month malware samples. Aggressive under-sampling of the majority class would have decreased the statistical significance of the dataset, while oversampling of the goodware class would have been too resource-intensive. (see § 2.3)

It is important to emphasize that the decision to set the malware percentage at 10% in our dataset should not be interpreted as a universal benchmark for Android malware detection. The landscape of malware and software development is dynamic, with constantly evolving threats and defensive measures. Therefore, researchers and practitioners should exercise discretion in selecting an appropriate malware-to-goodware ratio for their specific context. This selection should be informed by the latest trends, industry reports, and the specific characteristics of the domain being studied. Regular reassessment of this ratio is recommended to ensure the continued relevance and accuracy of the malware detection methodologies being employed.

2.3 Android Dataset

In our research, we utilize AndroZoo [3] as our data source, which is publicly available and contains more than 22.6 million Android apps when we write the article. Each sample from AndroZoo is labeled with a timestamp and many of them also include VirusTotal metadata results. This dataset is constantly updated through crawling from various markets, including Google Play Store and third-party markets such as Anzhi and AppChina. We selected AndroZoo dataset for our experiments because of its large size and extensive time-span, which allows us to perform realistic experiments that are aware of both space and time factors.

Defining goodware and malware. Ideally, goodware in software security refers to legitimate Android apps that do not contain any malicious behavior. Malware, on the other hand, refers to those that have been classified as containing malicious behavior. However, different anti-virus engines may have different criteria for what is malicious. Since AndroZoo's metadata reports the number p of positive anti-virus reports on VirusTotal [31] for samples in the AndroZoo dataset, we chose $p = 0$ for goodware and $p \geq 4$ for malware, following Miller *et al.*'s [48] advice for a reliable ground-truth. About 9% of AndroZoo apps can be called grayware as they have $0 < p < 4$ when we write the article. We exclude grayware when constructing our dataset, as including gray samples could disadvantage classifiers whose features were designed with a different labeling threshold.

Time-span selection. In this extended article, one of our goals is to validate our previous findings by employing a larger, more comprehensive Android dataset. The vastness of the complete
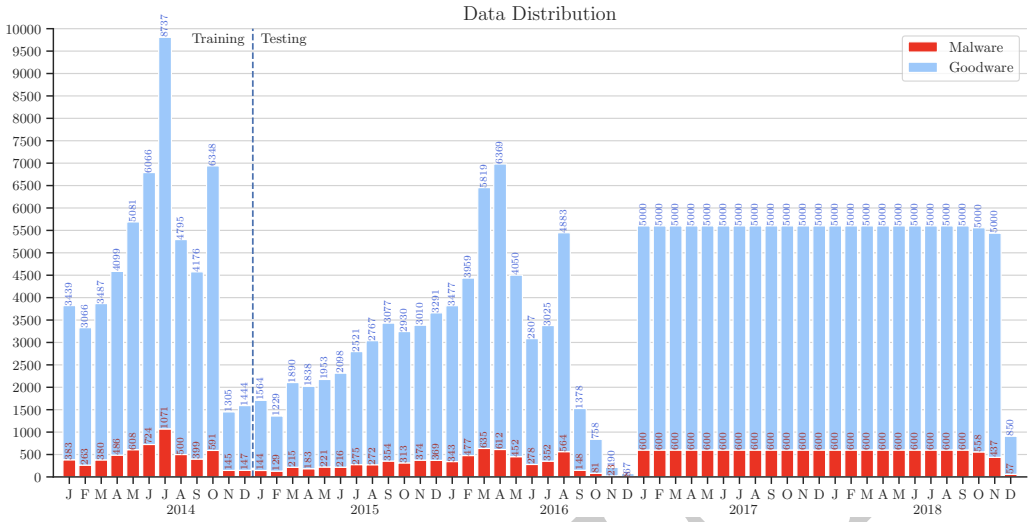
Fig. 1. Data distribution of the Android dataset using for this study. The figure shows a stacked histogram illustrating the monthly distribution of Android APKs we sourced from AndroZoo. It comprises 259, 230 Android applications, with approximately 10% being malware each month, covering a period from Jan. 2014 to Dec. 2018. The vertical dashed line indicates the split for all time-aware experiments in this study, with training data from 2014 and testing data from 2015 to 2018 if there is no special illustration.

AndroZoo dataset, which total size of DEX files stands at over 127TB, coupled with the constraints of our research infrastructure, poses significant challenges in extracting features from the entire dataset. For context, utilizing our research infrastructure — which consists of three high-end Dell PowerEdge R730 nodes (boasting 2 x 14 cores in hyperthreading, leading to a total of 168 vCPU threads, complemented by 1.2TB of RAM and an accompanying 100TB NAS)—to extract DREBIN's features for the entire dataset is estimated to take close to three years.

Practical considerations, including the costs of feature extraction and storage space requirements, drive our decision to find a balance between the temporal coverage and representativeness of our dataset. We select the period from 2014 to 2018 for analysis, a timeframe frequently cited in the assessment of contemporary methodologies, as highlighted in [8, 14, 15]. This choice, we believe, provides substantial statistical significance, crucial for identifying and thoroughly evaluating the influence of experimental biases on model performance. In our examination of time decay within the Android domain, we adopt a monthly granularity and uniformly sample 259K AndroZoo apps from January 2014 to December 2018. Furthermore, we maintain an overall malware average of 10%, in line with the malware ratio around 2017 as detailed in § 2.2, while allowing a monthly variation between 8% and 12% to ensure some degree of fluctuation. This five-year span ensures a robust dataset, offering over 1,000 apps per month, with the exception of the final quarter of 2016, which experienced a lower application crawl rate.

Dataset summary. The final dataset used in our study consists of 259,230 Android applications, with 232,843 labeled as goodware and 26,387 labeled as malware. To visualize the distribution of goodware and malware over time, we provide a stack histogram in Figure 1. The histogram displays the per-month distribution, and each bar represents a specific time period. For clarity, the figure also indicates the number of malware and goodware samples in each bar. If not explicitly stated, all subsequent time-aware experiments discussed in this paper involve training on data from 2014
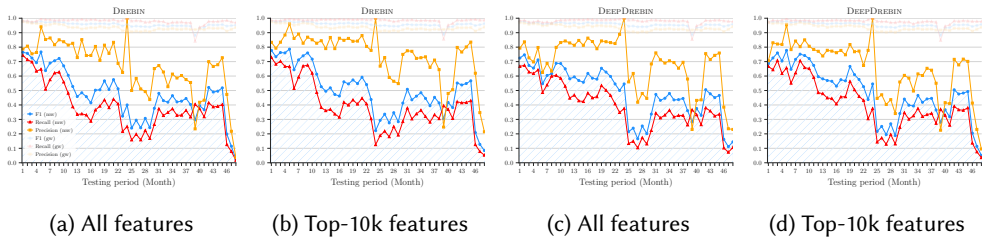
(a) All features	(b) Top-10k features	(c) All features	(d) Top-10k features

Fig. 2. Performance before and after feature selection. *'All features'* stands for the approach's performance on the full extracted DREBIN feature space, and *'Top-10k features'* represents the performance with the most important 10,000 features which selected based on the weight vector of classifier.

and testing on data from 2015 to 2018. This time range is indicated by the vertical dashed line in Figure 1, illustrating the division between the training and testing periods.

2.4 Feature Selection

We also confront the challenge of a large DREBIN [6] feature space present in the dataset mentioned earlier, which encompasses over two million features across eight different aspects of Android apps, including permissions, intents, API calls, among others [6]. This large feature space significantly increases computational demands and extends experiment duration, particularly problematic when analyzing DEEPDREBIN in detail. Owing to DEEPDREBIN's architecture, such a vast number of two million input neurons leads to an exponential increase in the number of parameters, resulting in prolonged training/testing times.

To address this, we employ feature selection techniques that aim to reduce the size of the feature space while maintaining its effectiveness in discriminating between different samples. Previous research studies [14, 22, 58] have shown that utilizing only the top 10,000 features does not significantly affect DREBIN classification performance. This is due to only a small number of features displaying significant discrimination and being assigned non-zero weights by the SVM learning algorithm used in DREBIN classification.

We begin by initializing the DREBIN approach on the full feature space with data from 2014. Since the chosen classification algorithm is a linear SVM, the decision function after training will be $f(x) = \mathbf{w}^T \mathbf{x} = \sum_{i=1}^{n} w_i x_i$. Here, n is the dimension of original feature space, $\mathbf{w}$ is the model's weight vector, and $\mathbf{x}$ represents the input sample. According to the algorithm, the absolute value of each w_i in its weight vector is positively correlated with the importance of the ith feature in making a classification decision.

We reduce the DREBIN feature space by selecting the top 10,000 features based on the largest absolute values in $\mathbf{w}$ [22]. Subsequently, only the indices of these chosen features are retained for ongoing experiments. We validate the performance of both the DREBIN and DEEPDREBIN classifiers post this feature selection. Testing results spanning from 2015 to 2018 are depicted in Figure 2. The left two plots, Figure 2a and Figure 2b, illustrate the performance between DREBIN's full feature space and its reduced 10,000 feature space, while the Figure 2c and Figure 2d on the right are dedicated to DEEPDREBIN. It is important to note that after the initial model training with 2014 data, no further retraining occurs during the testing phase. Key metrics like the F_1-score, Precision, and Recall for both malware and goodware classes are evaluated. The results show that in both the DREBIN and DEEPDREBIN set, the performance in the reduced feature space remains largely consistent, aligning with findings from previous studies [14, 22, 58].

In summary, feature selection has been effective in addressing the computational and time-consuming challenges associated with large feature spaces, especially for deep learning approaches

Table 1. F_1-Score results that show impact of spatial (in columns) and temporal (in rows) experimental bias. Values with red backgrounds are experimental results of *unrealistic* settings similar to those considered in papers of DREBIN [6] and DEEPDREBIN [33]; cells with green background are results in the *realistic* settings we identify. The dataset consists of five years (Figure 1), and each square in the sample dates column of the table represents a twelve month time-frame data: if training (resp. testing) objects are sampled from that time frame, we use a black square (■); if not, we use a gray square (▪).

			% mw in testing set Ts							
			10% (*realistic*)				90% (*unrealistic*)			
			% mw in training set Tr				% mw in training set Tr			
	Sample dates		10%	90%	10%	90%	10%	90%	10%	90%
Experimental setting	Training	Testing	DREBIN [6]		DEEPDREBIN [33]		DREBIN [6]		DEEPDREBIN [33]	
10-fold CV	gw: ■■■■■ mw: ■■■■■	gw: ■■■■■ mw: ■■■■■	**0.85**	0.55	**0.84**	0.51	0.91	0.98	0.88	0.98
Temporally inconsistent	gw: ▪▪■▪▪ mw: ▪▪■▪▪	gw: ■■▪■■ mw: ■■▪■■	0.64	0.48	0.62	0.38	0.71	0.49	0.66	0.92
Temporally inconsistent gw/mw windows	gw: ■▪■▪▪ mw: ■▪▪▪▪	gw: ▪▪▪▪■ mw: ▪▪■▪▪	0.30	0.42	0.37	0.42	0.31	0.77	0.39	0.78
Temporally consistent (*realistic*)	gw: ■▪▪▪▪ mw: ■▪▪▪▪	gw: ▪■■■■ mw: ▪■■■■	**0.47**	**0.39**	**0.36**	**0.39**	0.53	0.80	0.39	0.84

[30]. Our experiments demonstrate that reducing the DREBIN feature space does not negate the occurrence of concept drift; in fact, it can enhance performance by eliminating irrelevant information. Therefore, we believe that utilizing feature selection is a practical way to improve the efficiency and effectiveness of malware classifiers. Thus, we decide to use the reduced DREBIN feature space, with the most important 10,000 features, in the subsequent experiments.

3 SOURCES OF EXPERIMENTAL BIAS

In this section, we provide the rationale for our exploration of experimental biases via conducting experiments with DREBIN and DEEPDREBIN (§ 3.1). Subsequently, we delve into the origins of temporal bias (§ 3.2) and spatial bias (§ 3.3) that impact ML-based classification of Android malware.

3.1 Motivational Example

We present a motivating example where we manipulate experimental biases to better illustrate the problem. The results are summarized in Table 1, which showcases the F1-score for DREBIN and DEEPDREBIN under different experimental configurations (as shown in cells under the *Experimental settings* column). The rows represent various sources of temporal experimental bias, while the columns represent different sources of spatial experimental bias.

In the *Sample dates* column of Table 1, squares (■ / ▪) indicate the time frames from which the training and testing samples are obtained. Each square corresponds to a twelve-month period within the range from January 2014 to December 2018. Black squares (■) indicate that samples are taken from that specific time frame, while gray squares (▪) represent periods that are not utilized. On the right side of the table, the columns denote different percentages of malware in the training set (*Tr*) and the testing set (*Ts*).

The data presented in Table 1 reveal that both DREBIN and DEEPDREBIN exhibit significantly lower performance in realistic scenarios (highlighted in bold with a green background in the last row, corresponding to columns with 10% malware in testing) compared to settings similar to those described in prior works such as [6, 33] (highlighted in bold with a red background). This discrepancy can be attributed to unintentional experimental biases, which we elaborate on in the following sections.

> **Note.** We would like to clarify the reference to the similar settings in [6, 33] mentioned in the cells with a red background in Table 1. The original paper of DREBIN [6] utilizes a hold-out approach by performing 10 random splits, with 66% of the data used for training and 33% for testing. On the other hand, the paper on DEEPDREBIN [33] briefly mentions using the same dataset and experimental setting as [6]. Since hold-out is nearly equivalent to k-fold cross-validation (CV) and is susceptible to the same spatio-temporal biases, for the sake of simplicity in this section, we refer to a k-fold cross validation setting for both DREBIN and DEEPDREBIN.

3.2 Temporal Experimental Bias

Concept drift is a problem that occurs in machine learning when a model becomes obsolete as the distribution of incoming data at test-time differs from that of training data, i.e., when the assumption does not hold that data is independent and identically distributed (i.i.d.) [8, 37]. In the ML community, this problem is also known as *dataset shift* [66]. *Time decay* is the decrease in model performance over time caused by dataset shift.

Concept drift in malware combined with similarities among malware within the same family causes *k-fold cross validation* (CV) to be *positively biased*, artificially inflating the performance of malware classifiers [2, 48, 49]. K-fold CV is likely to include in the training set at least one sample of each malware family in the dataset, whereas new families will be unknown at training time in a real-world deployment. The all-black squares in Table 1 for 10-fold CV refer to each training/testing fold of the 10 iterations containing at least one sample from each time frame. The use of k-fold CV is widespread in malware classification research [17, 18, 38, 44, 47, 49, 55, 65, 67, 74]; while a useful mechanism to prevent overfitting [11] or estimate the performance of a classifier in the *absence* of concept drift when the i.i.d. assumption holds (see considerations in § 4.4), it has been unclear how it affects the real-world performance of machine learning techniques with non-stationary data that are affected by time decay. Hence, in the first row of Table 1, we quantify the performance impact with 10-fold CV in the Android domain.

The second row of Table 1 reports an experiment in which a classifier's ability to detect both past and future objects is evaluated [6, 33]. Although this characteristic is important, high performance should be expected from a classifier in such a scenario: if the classifier contains at least one variant of a past malware family, it will likely identify similar variants. We thus believe that experiments on the performance achieved on the detection of past malware can be misleading; therefore future research should focus on building malware classifiers that are robust against time decay.

In the third row of Table 1, we identify a novel form of temporal bias that arises when goodware and malware correspond to different time periods, often due to originating from distinct data sources (e.g., as observed in [46]). The black and gray squares in Table 1 indicate that, even though the testing objects for malware come after the training objects, the time windows for goodware and malware do not overlap. This situation can lead to the classifier learning to distinguish between applications from different time periods rather than differentiating between goodware and malware, thereby resulting in artificially inaccurate performance. For instance, spurious features such as new API methods may exhibit strong discriminatory power simply because malicious applications predate the introduction of those APIs.

Temporal inconsistency in data can result in *Spurious Correlations*. As described by [5], this refers to a scenario where a ML model forms shortcut patterns to differentiate classes based on artifacts that are irrelevant to the actual security issue. For instance, consider a malware detector that is trained on a dataset heavily dominated by certain malware families prevalent at the time. The model
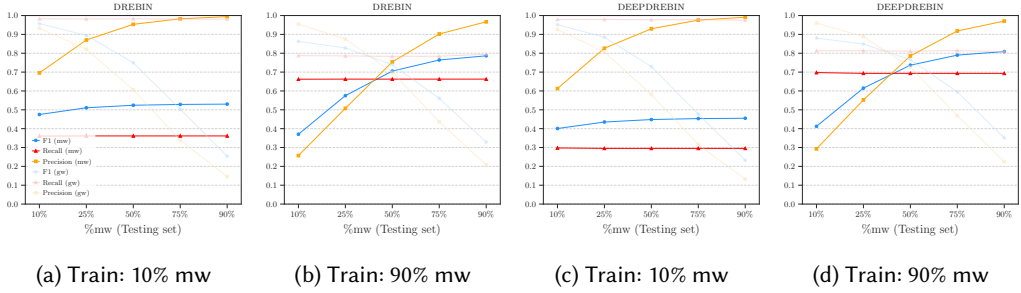
(a) Train: 10% mw   (b) Train: 90% mw   (c) Train: 10% mw   (d) Train: 90% mw

Fig. 3. *Spatial experimental bias in testing*. The models are trained on data from 2014 and tested on data from the remaining four years. In this unrealistic setting, where the percentage of malware in the testing is artificially increased, Precision for malware increases while Recall remains similar. Consequently, the overall F_1-Score also increases with the rising percentage of malware in the testing. However, it is important to note that this setting with more malware than goodware in testing does not reflect the true in-the-wild distribution of 10% malware (§ 2.2), rendering it unrealistic and leading to biased results.

might inadvertently learn to identify features that are specific to these families rather than the broader characteristics of malware. Such features, although effective in the short-term, may become redundant as new malware families emerge, which were not present during the training phase. This leads to a temporal inconsistency in the model's performance, as the model is tailored to the specific context of the training data and may not generalize well to future threats [5]. Consequently, it is essential for designers to be mindful of and avoid potential temporal inconsistencies during the data collection and sampling process.

The last row of Table 1 shows that the realistic setting, where training is temporally precedent to testing, causes the worst classifier performance in the majority of cases. We present decay plots and a more detailed discussion in § 4.

3.3 Spatial Experimental Bias

We identify two main types of spatial experimental bias based on assumptions on percentages of malware in testing and training sets. For all experiments in this section, we assume temporal consistency, meaning that the model is trained on data from 2014 and tested on data from 2015 onward (as the setting indicated in the last row of Table 1). This temporal setup allows us to examine spatial bias independently, without the confounding influence of temporal bias.

Spatial experimental bias in testing. The percentage of malware in the testing distribution needs to be estimated (§ 2.2) and *cannot* be changed, if one wants results to be representative of in-the-wild deployment of the malware classifier. To understand why this leads to biased results, we artificially vary the testing distribution to illustrate our point. Figure 3 reports performance (F_1-Score, Precision, Recall) for increasing the percentage of malware during testing on the x-axis. We change the percentage of malware in the testing set by randomly down-sampling goodware, so the number of malware remains fixed throughout the experiments. For completeness, we report the two training settings from Table 1 with 10% and 90% malware, respectively.

Note that we choose to down-sample goodware (gw) to achieve up to 90% of malware (mw) for testing because of the computational and storage resources required to achieve such a ratio by oversampling. However, this does not alter the conclusions of our analysis. Consider a scenario where gw quantity is constant, but mw is increased via oversampling. The precision ($P_{mw} = TP/(TP + FP)$) would increase because TPs would increase for any mw detection, and FPs would not change—because the number of gw remains the same; if training (resp. testing) observations
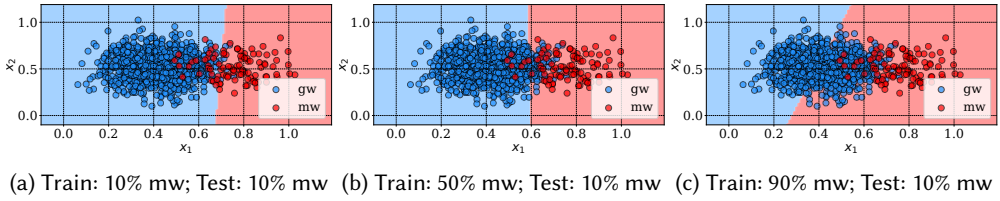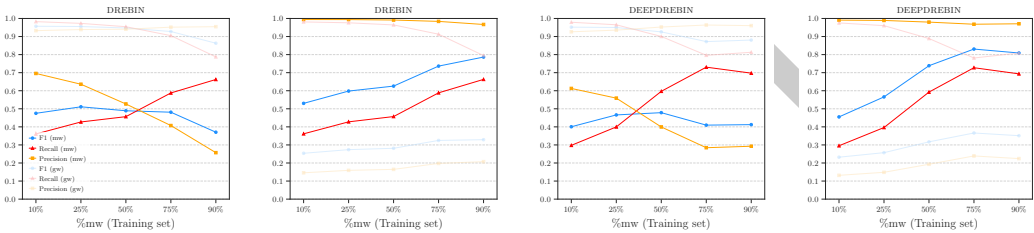
(a) Train: 10% mw; Test: 10% mw   (b) Train: 50% mw; Test: 10% mw   (c) Train: 90% mw; Test: 10% mw

Fig. 4. Motivating example for the intuition of *spatial experimental bias in training* with Linear-SVM and two features, x_1 and x_2. The training changes, but the testing points are fixed: 90% gw and 10% mw. When the percentage of malware in the training increases, the decision boundary moves towards the goodware class, improving Recall for malware but decreasing Precision.

(a) Test: 10% mw   (b) Test: 90% mw   (c) Test: 10% mw   (d) Test: 90% mw

Fig. 5. *Spatial experimental bias in training.* The models were trained on data from 2014 and tested on the remaining four years data. As the percentage of malware in the training set increases, Precision decreases while Recall increases, aligning with the motivations illustrated in the example of Figure 4. In § 4.3, we present an algorithm to determine the optimal training configuration for optimizing Precision, Recall, or F_1-Score based on user requirements.

are sampled from a distribution similar to the mw in the original dataset (e.g., new training mw is from 2014 and new testing mw comes from 2015 and 2016), then Recall ($R_{mw} = TP/(TP + FN)$) would be stable—it would have the same proportions of TPs and FNs because the classifier will have a similar predictive capability for finding mw. Hence, if the number of mw in the dataset increases, the F_1-Score would increase as well, because Precision increases while Recall remains stable.

Let us first focus on the malware performance (highlighted lines). All plots in Figure 3 exhibit consistent Recall, and increasing Precision when increasing percentage of malware in the testing. Precision for the malware (mw) class - the positive class - is defined as $P_{mw} = TP/(TP + FP)$ and Recall as $R_{mw} = TP/(TP + FN)$. In this scenario, we can observe that TPs (i.e., malware objects correctly classified as malware) and FNs (i.e., malware objects incorrectly classified as goodware) do not change, because the number of malware does not increase; hence, Recall remains stable. The increase in number of FPs (i.e., goodware objects misclassified as malware) decreases as we reduce the number of goodware in the dataset; hence, Precision improves. Since the F_1-Score is the harmonic mean of Precision and Recall, it goes up with Precision. We also observe that, inversely, the Precision for the goodware (gw) class - the negative class - $P_{gw} = TN/(TN + FN)$ decreases (see yellow dimmed lines in Figure 3), because we are reducing the TNs while the FNs do not change. This example shows how considering an unrealistic testing distribution with more malware than goodware in this context (§ 2.2) positively inflates Precision and hence the F_1-Score of classifiers.

Spatial experimental bias in training. The impact of altering the malware-to-goodware ratios in training can be understood through a motivating example with a linear SVM in a 2D feature space, using features x_1 and x_2. In this example, Figure 4 presents three scenarios, each with the same 10% malware in testing but varying percentages of malware in training: 10%, 50%, and 90%.

It is observed that as the percentage of malware in training increases, the hyperplane of the SVM shifts towards the goodware region. Specifically, it improves the Recall of malware while reducing its Precision. Conversely, the opposite trend is observed for goodware. In order to minimize the overall error rate $Err = (FP+FN)/(TP+TN+FP+FN)$ (i.e., maximize Accuracy), it is advisable to train the dataset with the same distribution that is expected in the testing. However, in certain cases, there may be a greater interest in identifying objects of the minority class (e.g., "more malicious samples") by improving Recall while ensuring a constraint on the maximum FPR.

Figure 5 shows the performance for DREBIN and DEEPDREBIN, with increasing percentages of malware in training on the x-axis. For the sake of completeness (as it is not possible to artificially change the test distribution for realistic evaluations), results are reported for both 10% malware in testing and 90% malware in testing. However, it should be noted that in the Android setting, the estimated proportion of malware in the wild is 10% (see § 2.2). These plots confirm the trends observed in our motivating example (Figure 4), where R_{mw} increases while P_{mw} decreases. For the plots with 10% malware in testing, it can be observed that there is a point at which the F_1-Score for malware is maximized while keeping the error for the goodware (gw) class within 5%.

In § 4.3, we propose a novel algorithm to enhance the performance of the malware class based on the user's objective (e.g., high Precision, Recall, or F_1-Score), while ensuring a maximum tolerated error. Furthermore, in § 4, we introduce constraints and metrics to ensure bias-free evaluations and reveal counter-intuitive results.

4 SPACE-TIME AWARE EVALUATION

We now outline the process for conducting an evaluation of an Android malware classifier that is free from spatio-temporal bias. Firstly, we establish a novel set of constraints that must be adhered to in order to ensure realistic evaluations (§ 4.1). Secondly, we introduce a new time-aware metric called AUT, which provides a comprehensive measure of the impact of time decay on a classifier (§ 4.2). Thirdly, we propose a novel tuning algorithm that empirically optimizes the performance of a classifier, while considering a maximum acceptable error (§ 4.3). Finally, we introduce TESSERACT, a framework that enables unbiased evaluations and presents counter-intuitive results (§ 4.4). For ease of reference, we provide a table of major symbols used throughout the remainder of this paper in Appendix A.2 to enhance readability.

4.1 Evaluation Constraints

Let us consider D as a labeled dataset with two classes: malware (positive class) and goodware (negative class). Let us define $s_i \in D$ as an *object* (e.g., testing sample) with timestamp $time(s_i)$. To evaluate the classifier, the dataset D must be split into a training dataset Tr with a time window of size W, and a testing dataset Ts with a time window of size S. Here, we consider $S > W$ in order to estimate long-term performance and robustness to decay of the classifier. A user may consider different time splits depending on their objectives, provided each split has a significant number of samples. We emphasize that, although we have the labels of objects in $Ts \subseteq D$, all the evaluations and tuning algorithms *must* assume that labels y_i of objects $s_i \in Ts$ are unknown.

To evaluate performance over time, the test set Ts must be split into time-slots of size Δ. For example, for a testing set time window of size $S = 2$ years, we may have $\Delta = 1$ month, for time window of size $S = 6$ months, we may choose $\Delta = 1$ week, etc. This granularity parameter is chosen by the user, but it is important that the chosen Δ allows for a statistically significant number of objects in each test window $[t_i, t_i + \Delta)$.

We now formalize three constraints that must be enforced when dividing D into Tr and Ts for a realistic setting that avoids spatio-temporal experimental bias (§ 3). While C1 was proposed in past work [2, 48], we are the first to propose C2 and C3—which we show to be fundamental in § 4.4.

(C1) Temporal training consistency. All the objects in the training must be *strictly* temporally precedent to the testing ones:

$$time(s_i) < time(s_j), \forall s_i \in Tr, \forall s_j \in Ts \tag{1}$$

where s_i (resp. s_j) is an object in the training set Tr (resp. testing set Ts). Eq. 1 must hold; its violation inflates the results by including future knowledge in the classifier (§ 3.2).

(C2) Temporal gw/mw time-window consistency. In every testing slot of size Δ, all test objects s_i must be from the same time window:

$$t_i^{min} \le time(s_i) \le t_i^{max}, \quad \forall s_i \text{ in time slot } [t_i, t_i + \Delta) \tag{2}$$

where $t_i^{min} = \min_k time(s_i)$ and $t_i^{max} = \max_k time(s_i)$. The same should hold for the training: although violating Equation 2 in the training data does not bias the evaluation, it may affect the sensitivity of the classifier to unrelated artifacts. Equation 2 has been violated in the past when goodware and malware have been collected from different time windows (e.g., [46])—if violated, the results are biased because the classifier may learn and test on artifactual behaviors that, for example, distinguish goodware from malware just by their different API versions.

(C3) Realistic malware-to-goodware ratio in testing. Let us define φ as the average percentage of malware in training data, and δ as the average percentage of malware in the testing data. Let $\hat{\sigma}$ be the estimated percentage of malware in the wild. To have a realistic evaluation, the average percentage of malware in the testing (δ) must be as close as possible to the estimated percentage of malware in the wild ($\hat{\sigma}$), so that:

$$\delta \simeq \hat{\sigma} \tag{3}$$

For example, we have estimated that in the Android scenario, especially during our evaluated period (2014 - 2018), goodware is predominant over malware, with $\hat{\sigma} \approx 10\%$ (§ 2.2). If C3 is violated by overestimating the percentage of malware, the results are positively inflated (§ 3.3). We highlight that, although the testing distribution δ cannot be changed (in order to get realistic results), the percentage of malware in the training φ may be tuned (§ 4.3).

4.2 Time-aware Performance Metrics

We introduce a time-aware performance metric that allows for the comparison of different classifiers while considering time decay. Let Θ be a classifier trained on Tr; we capture the performance of Θ for each time frame $[t_i, t_i + \Delta]$ of the testing set Ts (e.g., each month). We identify two options to represent per-month performance:

- **Point estimates** (*pnt*): The value plotted on the Y-axis for $X_k = k\Delta$ (where k is the test slot number) computes the performance metric (e.g., F_1-Score) only based on predictions $\hat{y}_i$ of Θ and true labels y_i in the interval $[W + (k-1)\Delta, W + k\Delta)$.
- **Cumulative estimates** (*cml*): The value plotted on the Y-axis for $X_k = k\Delta$ (where k is the test slot number) computes the performance metric (e.g., F_1-Score) only based on predictions $\hat{y}_i$ of Θ and true labels y_i in the cumulative interval $[W, W + k\Delta)$.

Point estimates are always to be preferred to represent the real performance of an algorithm. The cumulative estimates can be used to highlight a smoothed trend and to show overall performance up to a certain point, but can be misleading if reported on their own if objects are too sparsely distributed in some test slots Δ. Hence, we report only point estimates in the remainder of the paper, while an example of cumulative estimate plots is reported in Appendix A.3.

To facilitate the comparison of different time decay plots, we define a new metric, *Area Under Time* (**AUT**), the area under the performance curve over time. Formally, based on the trapezoidal rule (as in AUROC [11]), AUT is defined as follows:

$$AUT(\mathbb{P}, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{[\mathbb{P}(X_{k+1}) + \mathbb{P}(X_k)]}{2} \tag{4}$$

where: $\mathbb{P}(X_k)$ is the value of the point estimate of the performance metric $\mathbb{P}$ (e.g., F_1) evaluated at point $X_k := (W + k\Delta)$; N is the number of test slots, and $1/(N-1)$ is a normalization factor so that AUT $\in [0, 1]$. The perfect classifier with robustness to time decay in the time window S has AUT = 1. By default, AUT is computed as the area under point estimates, as they capture the trend of the classifier over time more closely; if the AUT is computed on cumulative estimates, it should be explicitly marked as AUT$_{cml}$. As an example, AUT(F_1, 12m) is the point estimate of F_1-Score considering time decay for a period of 12 months, with a 1-month interval.

We highlight that the simplicity of computing the AUT should be seen as a benefit rather than a drawback; it is a simple yet effective metric that captures the performance of a classifier with respect to time decay, de-facto promoting a fair comparison across different approaches.

AUT($\mathbb{P}$,N) is a metric that allows us to evaluate performance $\mathbb{P}$ of a malware classifier against time decay over N time units in realistic experimental settings—obtained by enforcing C1, C2, and C3 (§ 4.1). The next sections leverage AUT for tuning classifiers and comparing different solutions (§ 4.4).

4.3 Tuning Training Ratio

We propose a novel algorithm that allows for the adjustment of the training ratio φ when the dataset is imbalanced, in order to optimize a user-specified performance metric (F_1-Score, Precision, or Recall) on the minority class, subject to a maximum tolerated error, while aiming to reduce time decay. The high-level intuition of the impact of changing φ is described in § 3.3. We also observe that ML literature has shown ROC curves to be misleading on highly imbalanced datasets [21, 36]. Choosing different thresholds on ROC curves *shifts* the decision boundary, but (as seen in the motivating example of Figure 4) re-training with different ratios φ (as in our algorithm) also changes the shape of the decision boundary, better representing the minority class.

Our tuning algorithm is inspired by one proposed by Weiss and Provost [71]; they propose a progressive sampling of training objects to collect a dataset that improves AUROC performance of the minority class in an imbalanced dataset. However, they did not take temporal constraints into account (§ 3.2), and heuristically optimize only AUROC. Conversely, we enforce C1, C2, C3 (§ 4.1), and rely on AUT to achieve three possible targets for the malware class: higher F_1-Score, higher Precision, or higher Recall. Also, we assume that the user already has a training dataset Tr and wants to use as many objects from it as possible, while still achieving a good performance trade-off; for this purpose, we perform a *progressive sub-sampling* of the goodware class.

Algorithm 1 formally presents our methodology for tuning the parameter φ to find the value $\varphi_{\mathbb{P}}^*$ that optimizes $\mathbb{P}$ subject to a maximum error rate E_{max}. The algorithm aims to solve the following optimization problem:

$$\text{maximise}_\varphi \{\mathbb{P}\} \quad \text{subject to: } E \leq E_{max} \tag{5}$$

where $\mathbb{P}$ is the target performance: the F_1-Score (F_1), Precision (Pr) or Recall (Rec) of the targeted class (malware in this article); E_{max} is the maximum tolerated error; depending on the target $\mathbb{P}$, the error rate E has a different formulation:

- if $\mathbb{P} = F_1 \rightarrow E = 1 - \text{Acc} = \frac{(FP+FN)}{(TP+TN+FP+FN)}$

- if $\mathbb{P} = Rec \rightarrow E = FPR = \frac{FP}{(TN+FP)}$

- if $\mathbb{P} = Pr \rightarrow E = FNR = \frac{FN}{(TP+FN)}$

Each of these definitions of E is targeted to limit the error induced by the specific performance. The choice of a particular error rate for each target performance metric, $\mathbb{P}$, is grounded in the inherent characteristics and priorities of these metrics. When the target is the F_1-Score ($\mathbb{P} = F_1$), the error rate is defined as $1 - \text{Accuracy}$ (Acc) because the F_1-Score seeks a detecting balance between both class (in multi-class detection, it will be the targeted class and other classes), thereby necessitating a comprehensive view of both classification errors (FP and FN). Accuracy encapsulates this balance by considering the correct classifications (TP and TN).

Algorithm 1: Tuning φ.

Input: Training dataset Tr

Parameters: Learning rate μ, target performance $\mathbb{P} \in \{F_1, Pr, Rec\}$, max error rate E_{max}

Output: $\varphi_{\mathbb{P}}^*$, optimal percentage of mw to use in training to achieve the best target performance $\mathbb{P}$
subject to $E < E_{max}$.

1 Split the training set Tr into two subsets: actual training ($ProperTr$) and validation set (Val), while enforcing C1, C2, C3 (§ 4.1), also implying $\delta = \hat{\sigma}$

2 Divide Val into N non-overlapped subsets, each corresponding to a time-slot Δ, so that $Val_{array} = [V_0, V_1, ..., V_N]$

3 Train a classifier Θ on $ProperTr$

4 $P^* \leftarrow \text{AUT}(\mathbb{P}, N)$ on Val_{array} with Θ

5 $\varphi_{\mathbb{P}}^* = \hat{\sigma}$

6 **for** $(\varphi = \mu; \varphi < 1.0; \varphi = \varphi + \mu)$ **do**

7 $\quad$ Downsample gw in $ProperTr$ so that percentage of mw is φ

8 $\quad$ Train the classifier Θ_φ on $ProperTr$ with φ mw

9 $\quad$ performance $P_\varphi \leftarrow \text{AUT}(\mathbb{P}, N)$ on Val_{array} with Θ_φ

10 $\quad$ error $E_\varphi \leftarrow$ Error rate on Val_{array} with Θ_φ

11 $\quad$ **if** $(P_\varphi > P^*)$ **and** $(E_\varphi \leq E_{max})$ **then**

12 $\quad\quad$ $P^* \leftarrow P_\varphi$

13 $\quad\quad$ $\varphi_{\mathbb{P}}^* \leftarrow \varphi$

14 **return** $\varphi_{\mathbb{P}}^*$;

In contrast, when targeting Recall ($\mathbb{P} = Rec$), the error rate is focused on the FPR. This is because when we improve a classifier's ability to catch positive class, the side effect, which we consider as errors, will be the increase of false positives (FP). FPR directly measures the extent to which a model incorrectly classifies negatives as positives. Similarly, when Precision ($\mathbb{P} = Pr$) is the goal, the error rate is defined by the False Negative Rate (FNR) because in this scenario, the errors will be positive samples which are wrongly detected as negative.

As a result, if we want to maximize F_1-Score for the malware class, we need to limit both FPs and FNs. if $\mathbb{P} = Pr$, we minimize FNs, so we constrain FNR. And if $\mathbb{P} = Rec$, we constrain FPR.

Algorithm 1 consists of two phases: *initialization* (lines 1–5) and *grid search* of $\varphi_{\mathbb{P}}^*$ (lines 6–14). In the initialization phase, the training set Tr is split into a proper training set $ProperTr$ and a validation set Val; this split is according to the space-time evaluation constraints in § 4.1, so that all the objects in $ProperTr$ are temporally anterior to Val, and the malware percentage δ in Val is equal
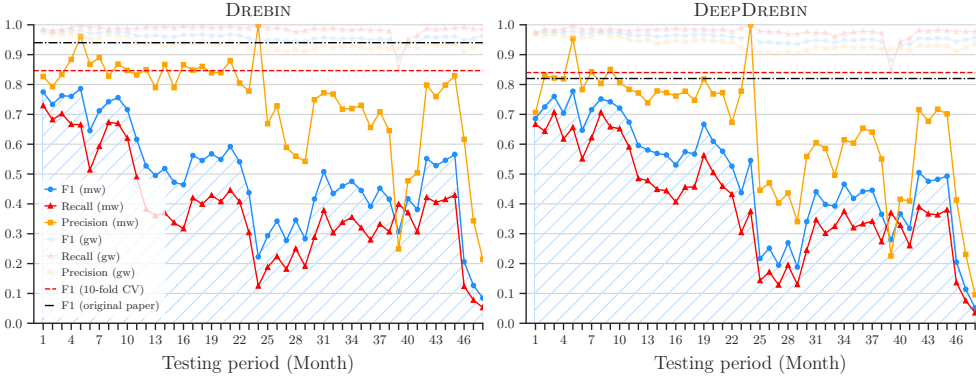
Fig. 6. Performance decay over time of DREBIN [6] and DEEPDREBIN [33]—with AUT(F_1, 48m) of 0.50 and 0.49, respectively. Training on data from 2014 and testing on data from 2015 to 2018. Both distribution have roughly 10% malware.

to $\hat{\sigma}$, the in-the-wild malware percentage. The maximum performance observed P^* and the optimal training ratio $\varphi_{\mathbb{P}}^*$ are initialized by assuming the estimated in-the-wild malware ratio $\hat{\sigma}$ for training; in Android, $\hat{\sigma} \approx 10\%$ (see § 2.2).

The grid-search phase iterates over different values of φ, with a learning rate μ (e.g., $\mu = 0.05$), and keeps as $\varphi_{\mathbb{P}}^*$ the value leading to the best performance, subject to the error constraint. To reduce the chance of discarding high-quality points while down-sampling goodware, we prioritize the most uncertain points (e.g., points close to the decision boundary of the SVM) [60]. This approach differs from the one in the conference version of TESSERACT [54], where we relax the constraint on line 6 from ($\hat{\sigma} \leq \varphi \leq 0.5$) to a broader range of ($\mu \leq \varphi \leq 1.0$). We find that imposing a stringent boundary is unnecessary when an error rate control is already in place (see algorithm 1, line 11). Finally, the grid-search explores multiple values of φ and stores the best ones. To capture time-aware performance, we rely on AUT (§ 4.2), and the error rate is computed according to the target $\mathbb{P}$ (see above). Tuning examples are in § 4.4.

4.4 TESSERACT: Revealing Hidden Performance

Here, we show how our methodology can reveal hidden performance of DREBIN [6] and DEEP-DREBIN [33], along with their robustness to time decay.

We introduce TESSERACT, an open-source performance evaluation framework that rigorously applies constraints C1, C2, and C3 (§ 4.1), computes AUT (§ 4.2), and possesses the capability to refine a classifier utilizing our proposed tuning algorithm (§ 4.3). TESSERACT functions comparably to traditional Python ML libraries, yet it distinctively requires not only the features matrix X and labels y, but also an array of timestamps t, which sorts the dates corresponding to each data entry. Details on TESSERACT's architecture and its versatility are thoroughly discussed in Appendix A.4.

Figure 6 reports several performance metrics of the two selected classification approaches as point estimates over time. The X-axis reports the testing slots in month granularity, whereas the Y-axis reports different scores between 0.0 and 1.0. The areas highlighted in blue correspond to the AUT(F_1, 48m) over the 48 testing months. The black dash-dotted horizontal lines represent the best F_1-Score from the original works Arp $et~al.$[6] and Papernot $et~al.$[33], corresponding to results obtained with 10 hold-out random splits for DREBIN and DEEPDREBIN, respectively; both these settings are analogous to k-fold from a temporal bias perspective, and violate both C1 and C2. The red dashed horizontal lines correspond to 10-fold F_1 obtained on our dataset, which satisfies C3.

Table 2. AUT comparison across various observation windows (τ) for the DREBIN and DEEPDREBIN approaches. Rows under the 'AUT Observation Window τ' show the size of selected observation windows. Each column under the 'Testing Month' cell indicates the endpoint of the testing period, starting from the end of the preceding cell. For instance, when τ is 3-month, the cell under column '12' denotes the AUT result for the window spanning months 10 to 12. Similarly, for the row of τ is 6-month, the cell under column '12' corresponds to the AUT result from months 7 to 12. Time windows where one model significantly outperforms the other, determined by a minimum AUT difference of 0.02, are highlighted - orange for DREBIN and blue for DEEPDREBIN. Note that this table's $\mathbb{P}$ is F_1-Scores, and the AUT granularity is a month.

Time-Aware AUT Comparison																		
AUT Observation Window τ		Testing Month																
		3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48
3-month	DREBIN	0.75	0.74	0.74	0.62	0.50	0.53	0.57	0.41	0.31	0.33	0.46	0.44	0.40	0.43	0.55	0.14
	DEEPDREBIN	0.72	0.70	0.69	0.67	0.59	0.56	0.60	0.41	0.23	0.26	0.44	0.44	0.37	0.41	0.52	0.13
6-month	DREBIN	0.75		0.69		0.51		0.50		0.32		0.46		0.40		0.35	
	DEEPDREBIN	0.71		0.69		0.57		0.52		0.24		0.44		0.38		0.33	
12-month	DREBIN	0.72				0.51				0.39				0.39			
	DEEPDREBIN	0.69				0.55				0.35				0.37			
24-month	DREBIN	0.61								0.39							
	DEEPDREBIN	0.62								0.36							
48-month	DREBIN	0.50															
	DEEPDREBIN	0.49															

The figure illustrates that DEEPDREBIN's performance aligns closely with that reported in the original study by Papernot *et al.*[33], as indicated by the similarity between the red-dashed and black dash-dotted lines. In contrast, DREBIN exhibits a slightly reduced performance compared to its original evaluation in Arp *et al.*[6]. We believe the difference of performance is due to different data distribution. Both approaches show a similar trend of performance in the testing data. The 10-fold CV results are 0.85 and 0.84, with an AUT(F_1, 48m) of 0.50 and 0.49 for DREBIN and DEEPDREBIN, respectively. Focusing on the malware side (highlighted lines), we observe that both F_1-Scores and Recall rates initially decrease, followed by a modest recovery after Month 28. Precision remains stable in the first 24 months for both methods, then experiences a notable decline after Month 24 and generally remains lower thereafter. There is no obvious performance decay, apart from Month 39, from goodware's side (dimmed lines).

Notably, between these two detection approaches, DEEPDREBIN demonstrates greater robustness, especially in the second year of the testing period, achieving higher Recall and F_1-Score. We present a more detailed AUT analysis in Table 2.

Observation window in AUT evaluation. The nature of the AUT metric we propose shows the overall performance of a classifier in the time-aware evaluation. As a result, it sometimes ignores performance details for a specific period. For instance, even if the AUT(F_1, 48m) for DREBIN and DEEPDREBIN are 0.50 and 0.49, respectively, we can still observe that at some periods (e.g., first five months and last seven months), the performance can vary.

When evaluating performance with the AUT metric, a new feature in TESSERACT framework is that users can define the observation time window (τ, by default, it equals the full length of the testing period) for more detailed performance change. Different from the granularity parameter, which defines the size of test time slots Δ (Equation 2), τ can be considered as the length of the period over which we assess the classifier's performance. This observation window is important as it shapes the scope of performance evaluation. For example, a smaller τ might involve assessments every three months, while a larger one could span several years. Adjusting τ of AUT is essential for understanding how a classifier's effectiveness in malware detection evolves over time. Different τ can lead to distinct insights, enabling researchers to pinpoint when and how a classifier's performance starts to decay, thus guiding the development of robust systems.

As shown in Table 2, the AUT comparison between DREBIN and DEEPDREBIN highlights the impact of varying the AUT observation window τ. The table presents AUT values across different τ, ranging from 3-month to 48-month, illustrating how performance fluctuates over time. Cells highlighted in the table indicate where one model notably outperforms the other, signaling areas for further investigation. Both approaches are initialized with training data from 2014, with the AUT metric evaluated over the four years of testing data. When we set the time window as default, where $\tau = 48$-month, DREBIN and DEEPDREBIN show similar performances of 0.50 and 0.49, respectively. However, by reducing τ reveals variations in performance. When $\tau = 24$-month, we can find that both approaches perform similarly in the two evaluation periods, with a better AUT in the first 24 months. Notably, when τ gets even smaller, DEEPDREBIN exhibits more consistent and robust performance in the second year (Month 13 to 24) and maintains this trend across all smaller time windows (e.g., $\tau = 3$-month, 6-month, and 12-month).

This variability in smaller windows suggests that τ can provide deeper insights into a classifier's immediate adaptability and response to emerging malware types. Larger windows might offer a broader view of long-term stability and resilience. Shorter windows are beneficial in environments where malware threats evolve rapidly, and early detection is crucial. These findings highlight the significance of selecting the appropriate AUT observation window τ.

The table also highlights another important aspect: the need to investigate the causes behind the performance discrepancies at different granularities. Such an investigation could involve Explainable AI (XAI) techniques, which can provide insights into the decision-making processes of classifiers [16]. Understanding why a classifier's performance varies with time can inform improvements in classifier design and contribute to developing more robust malware detection systems.

In practice, the selection of τ should be driven by the specific needs of the malware detection context and the desired balance between immediate responsiveness and enduring robustness. The capability to compute AUT over different windows demonstrates its versatility and enables a more strategic and customized approach to classifier evaluation and comparison.

Violating the consistency of temporal training (C1) and gw/mw time-window (C2). The removal of temporal bias unveils the real performance of each algorithm amid concept drift. The AUT(F_1, 48m) captures such performance over all 48 testing months: 0.50 for DREBIN, and 0.49 for DEEPDREBIN, respectively. We also evaluate the two approaches on the first 24 months of testing data: the AUT(F_1, 24m) stands at 0.61 for DREBIN and 0.64 for DEEPDREBIN, respectively. Across all quartets of scenarios, the AUT consistently falls behind the 10-fold F_1 score, as the latter violates constraint C1 and may violate C2 if the dataset classes are not evenly distributed across the timeline.

Robustness of deep learning approach. In our initial analysis documented in the original paper (refer to the 'DL' plot of Figure 5 in [54]), the deep learning method appeared to outshine the other two traditional ML approaches (Alg1(Drebin) and Alg2(MaMaDroid)) in terms of robustness after enforcing C1, C2 and C3. However, upon closer inspection and a more extended evaluation, this claim does not hold consistently. It is true that during the first two year of evaluation, the deep learning approach demonstrated a better and relatively stable performance. Nonetheless, when considering the broader scope of the analysis, it becomes evident that the overall performance of the deep learning method aligns closely with that of traditional machine learning techniques, if retraining is not possible. This revelation underscores that while deep learning may offer advantages in certain temporal slices, its efficacy is comparable to traditional models when viewed across a comprehensive timeline. This unresolved inquiry is delegated to future investigative efforts to discern the conditions under which DEEPDREBIN, or deep learning in general, exhibits superior performance within certain testing intervals.

Tuning algorithm. We now evaluate whether our tuning algorithm (Algorithm 1 in § 4.3) improves robustness to time decay of a malware classifier for a given target performance. We first

Table 3. Testing AUTs performance when training with $\hat{\sigma}$, $\varphi_{F_1}^*$, φ_{Pr}^*, and φ_{Rec}^*. Results are shown for both the entire 48-month period and the first 24-month period, offering a comprehensive view of performance over different time frames. Best performances are highlighted in gray color.

Algorithm	φ	FP (48m)	FN (48m)	AUT($\mathbb{P}$, 48m)			FP (24m)	FN (24m)	AUT($\mathbb{P}$, 24m)		
				F_1	Pr	Rec			F_1	Pr	Rec
DREBIN [6]	$\hat{\sigma}$ ($\sim$10%)	3,201	13,251	0.50	0.75	0.39	596	3,716	0.61	0.85	0.49
	$\varphi_{F_1}^*$ (55%)	12,053	10,409	0.51	0.49	0.54	2,970	2,113	0.64	0.61	0.69
	φ_{Pr}^* (5%)	2,616	15,637	0.39	0.72	0.28	421	4,025	0.58	0.87	0.45
	φ_{Rec}^* (95%)	48,891	5,878	0.37	0.25	0.75	13,978	466	0.46	0.31	0.91
DEEPDREBIN [33]	$\hat{\sigma}$ ($\sim$10%)	4,711	13,180	0.49	0.65	0.40	989	3,292	0.64	0.80	0.54
	$\varphi_{F_1}^*$ (20%)	6,381	11,509	0.53	0.62	0.48	1,494	2,605	0.68	0.74	0.63
	φ_{Pr}^* (5%)	2,069	16,961	0.32	0.69	0.22	302	4,792	0.47	0.89	0.33
	φ_{Rec}^* (95%)	86,487	1,844	0.30	0.18	0.91	28,364	165	0.32	0.19	0.95

(a) Tr: 2014, φ = 0.55 Ts: 2015-2018   (b) Tr: 2015, φ = 0.35 Ts: 2016-2017   (c) Tr: 2016, φ = 0.35 Ts: 2017-2018

(d) Tr: 2014, φ = 0.2 Ts: 2015-2018   (e) Tr: 2015, φ = 0.35 Ts: 2016-2017   (f) Tr: 2016, φ = 0.35 Ts: 2017-2018
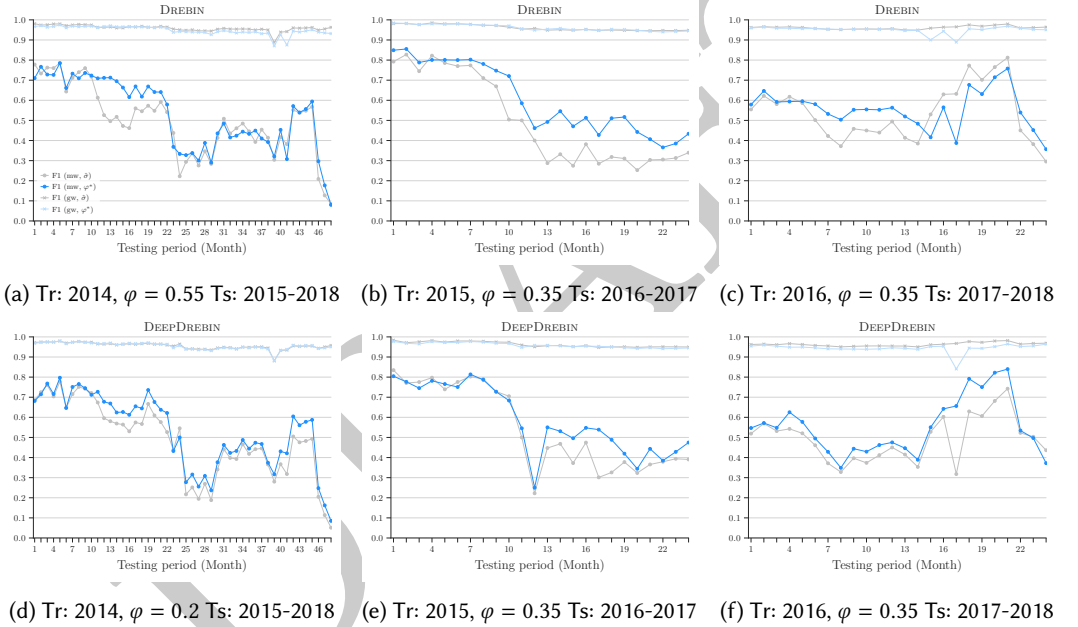
Fig. 7. Performance comparison of DREBIN and DEEPDREBIN before and after applying Tuning Algorithm (algorithm 1). Each graph displays results from both malware (mw) and goodware (gw) perspectives. The gray lines indicate the original F_1-Score performance, while the blue lines show the F_1-Score after tuning. The tuned outcomes in each scenario demonstrate improved performance over the original ones, achieving either better performance or increased stability. The sub-captions for each graph detail the period of training (Tr) and testing (Ts) data used. It is important to note that the final four months of data in the training set are utilized as the validation set for the tuning algorithm. Additionally, each sub-caption specifies the optimal malware percentage $\varphi = \varphi_{F_1}^*$ identified during the training phase.

aim to maximize $\mathbb{P} = F_1$-Score of malware class, subject to E_{max} = 10%. After running Algorithm 1 on DREBIN [6] and DEEPDREBIN [33], we set $\varphi_{F_1}^*$ = 0.55 for DREBIN, and $\varphi_{F_1}^*$ = 0.20 for DEEPDREBIN. Figures 7a and 7d report the respective improvements in test performance of the two approaches on four years of testing data Ts after applying $\varphi_{F_1}^*$ to the full training set Tr.

We remark that the choice of $\varphi_{F_1}^*$ uses only training information (see Algorithm 1) and no test information is used—the optimal value is chosen from a 4-month validation set extracted from the one year of training data; this is to simulate a realistic deployment setting in which we have no a priori information about testing. Figure 7a (DREBIN) and Figure 7d (DEEPDREBIN) show that our approach for finding the best $\varphi_{F_1}^*$ improves the F_1-Score on malware at most of the four-year testing time, at the cost of slightly reduced goodware performance. Table 3 shows details of how total FPs, total FNs, and AUT changed by training DREBIN and DEEPDREBIN with $\varphi_{F_1}^*$, φ_{Prec}^*, and φ_{Rec}^* instead of $\hat{\sigma}$. These training ratios have been computed subject to $E_{max} = 5\%$ for φ_{Rec}^*, $E_{max} = 10\%$ for $\varphi_{F_1}^*$, and $E_{max} = 15\%$ for φ_{Prec}^*; the difference in the maximum tolerated errors is motivated by the class imbalance in the dataset—which causes lower FPR and higher FNR values (see definitions in § 4.3), as there are many more goodware than malware.

Table 3 details the performance over the 48-month period when employing various training strategies for the DREBIN and DEEPDREBIN algorithms. When trained with $\varphi_{F_1}^*$, a decrease in Precision is noted due to an increase in False Positives (FPs), while an increase in Recall is observed as a result of a decrease in False Negatives (FNs). This trade-off leads to a slight improvement of AUT for both classifiers. For each specific performance metric $\mathbb{P}$, the tailored performance AUT($\mathbb{P}$, 48m) with $\varphi_{\mathbb{P}}^*$ generally surpasses the performance achieved using the original training ratio $\hat{\sigma}$ for both DREBIN and DEEPDREBIN. The notable exception is when focusing on Precision; in this case, the tuning slightly underperforms compared to the original training ratio. This discrepancy might be attributed to the concept shift over the long-term evaluation.

To provide a more detailed perspective, AUT($\mathbb{P}$, 24m) metrics are also presented, which consistently show that the results of tuning are superior to those obtained when training with $\hat{\sigma}$. It is also noteworthy that the AUT for Precision may vary slightly even with a similar number of total FPs; this is due to AUT(Pr)'s sensitivity to the timing of the occurrence of FPs. A corresponding observation holds true for the total FNs and AUT for Recall. Post-tuning, DEEPDREBIN's performance in terms of the F1-Score, as measured by AUT, remains comparatively higher, with a AUT(F_1, 24m) of 0.68 compares to DREBIN's 0.64. This demonstrates the effectiveness of the tuning approach, especially over the first 24-month period of evaluation, where the improvements over training with $\hat{\sigma}$ are more pronounced.

Furthermore, a three-year evaluation window was also employed to assess the tuning algorithm, as evidenced in Figures 7b, 7c for DREBIN, and Figures 7e, 7f for DEEPDREBIN. This shorter period of evaluation suggests the necessity for regular updates to the tuning process to keep pace with the dynamic nature of malware threats. Aligning with the full dataset protocol, the first year of data from this three-year period was used to ascertain the optimal $\varphi_{F_1}^*$, and subsequent tests were conducted over the following two years (as listed in each sub-captions). The optimal $\varphi_{F_1}^*$ was found to be 0.35 on both years 2015 and 2016 for both DREBIN and DEEPDREBIN. Compared to the baseline performance (indicated by gray lines in the graphs), the tuned outcomes demonstrate enhanced robustness in the latter two years, comparing to in the long-run evaluation.

These results highlight the effectiveness of employing a short-term evaluation interval for both tuning and testing our algorithm. The graphical analyses, as demonstrated in Figures 7b, 7c, 7e, and 7f, reveal that our tuning algorithm not only enhances the overall robustness of the malware classifiers (DREBIN and DEEPDREBIN) but also contributes to more consistent performance over time. This is particularly evident in the post-tuning improvements in the F_1-Score, Precision, and Recall metrics across different time frames. The tuning process, guided by Algorithm 1, effectively addresses the challenge of time decay, as shown by the improved AUT metrics over various periods.

Moreover, the findings highlight the importance of regular re-calibration of the tuning parameters to adapt to the evolving nature of malware threats. The optimal values of $\varphi_{F_1}^*$ obtained from the

three-year data suggest that periodic adjustments in the tuning process can lead to better detection rates and more reliable classification. Such an approach is crucial in maintaining the efficacy of malware detection systems in a rapidly changing digital landscape.

Conclusion. Our investigation strongly recommends the adoption of a dynamic tuning and testing regimen. Researchers should consider the specific characteristics of their datasets and the prevailing trends in malware evolution when determining the frequency of tuning intervals. Regular updates and adjustments to the tuning parameters, as dictated by the nature of the threat environment and the performance goals, are key to sustaining the reliability and accuracy of malware classifiers over time.

5 WINDOWS AND PDF MALWARE

We now discuss the impact of the spatial and temporal bias on ML-driven detection performance in other domains: Windows Portable Executables (PEs) and PDFs. Before diving into the results, we provide reflection and intuitions on the relationship between Android and these other domains.
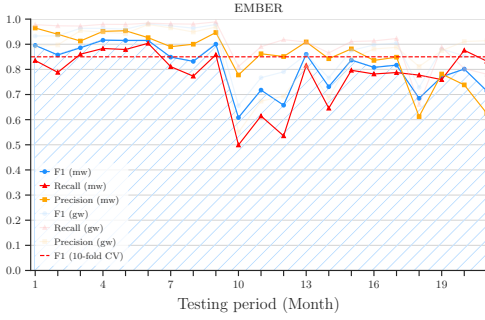
Comparing Android and Windows PE / PDF domains. In the realm of cybersecurity, the distinction between the Android domain and the Windows PE and PDF domains is marked and significant, especially when considering spatial and temporal aspects. The Android ecosystem, predominantly comprising mobile applications, presents a unique challenge due to its vast, diverse app market and the rapid evolution of mobile technologies. This results in a highly dynamic temporal landscape where malware characteristics and trends can shift swiftly [32, 63, 77]. The diversity in the source and nature of applications, ranging from official app stores to third-party platforms, adds another layer of complexity [27]. In contrast, the Windows PE and PDF domains, typically associated with desktop environments, display distinctive traits. Windows PE files, crucial for executables and libraries, are integrally linked with Windows' architecture, leading to a steadier malware evolution, in sync with the relatively longer operation system (OS) development cycle [23, 43]. PDF files, serving mainly for document exchange, exhibit a stable structure in format and usage [25, 34, 62]. Malicious activities via PDFs often exploit software and OS vulnerabilities. Intuitively, we could expect a slower onset of concept drift—along with a more gradual decline in model performance—in these domains.

Malware ratio in the wild. The Android ecosystem predominantly features goodware [32]. Conversely, the ratio of malware to goodware in Windows PE and PDF domains is not easily quantified due to a lack of comprehensive evidence. Hence, we refer to the ratios in two popular, existing datasets: one on the EMBER dataset for Windows PE files [4] (released by the EndGame company), and the other on the Hidost dataset for PDF files [62].
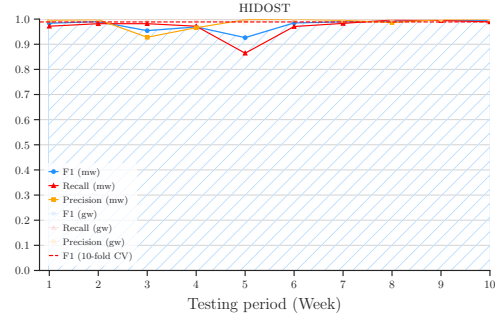
5.1 Case Study: EMBER

The EMBER dataset consists of a comprehensive suite of features derived from PE files to facilitate cybersecurity research [4], such as headers, content-based histograms, and textual data like URL frequencies. Our analysis spans both the EMBER 2017 v2 and EMBER 2018 datasets, encompassing the years 2017 and 2018, resulting in a composite dataset of 255,499 benign and 148,231 malicious executables (labeled as having 40+ VirusTotal AV detections), constrained to this two-year period. We employ Gradient Boosted Decision Trees (GBDT) [28] as our classifier, in line with the EMBER study's methods. We initialize the classifier on data from the first three months and test it on subsequent data.

The original EMBER evaluation adheres to the constraints C1 and C2 delineated in § 4.1. Defining a precise malware-to-goodware ratio for the Windows PE domain is challenging; however, the dataset's size and heterogeneity of sources lend credence to its adherence to constraint C3. The performance degradation illustrated in Figure 8a corroborates the presence of concept drift,

(a) Tr: Jan.-Mar.2017, Ts: Apr.2017 - Dec.2018.

(b) Tr: Week 1-4, Ts: Week 5-14.

Fig. 8. Time-aware evaluation on Windows PE dataset (EMBER [4]), and PDF dataset (HIDOST [62]). The Tr and Ts periods for each experiment are shown in the sub-caption.
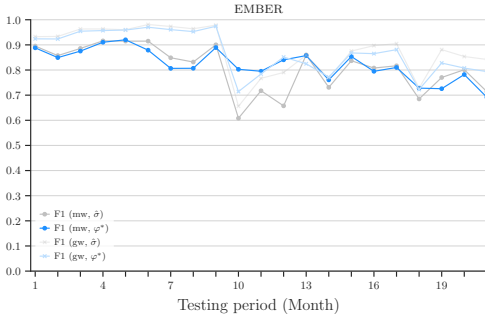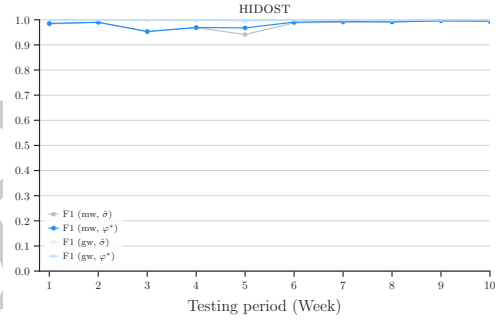

(a) Train on Jan 2017 and tune on the remaining training dataset, $\varphi = 0.25$, $\varphi_{F_1}^* = 0.5$.

(b) Train on the first two weeks of the training set and tune on the remaining two. $\varphi = 0.04$, $\varphi_{F_1}^* = 0.1$.

Fig. 9. Performance comparison of EMBER and HIDOST before and after applying Tuning Algorithm

particularly notable in the F1-score decline post-January 2018, which stabilizes at a lower benchmark. A marked reduction in the Recall metric at the onset of 2018 further underscores this point. These observations are consistent with those in the original EMBER paper [4], which deliberately incorporated more elusive malware into the 2018 dataset.

Applying our Tuning Algorithm on the training set improves the model's performance on the testing set, with the overall AUT(F_1, 21m) increasing from 0.80 to 0.83. We choose the first month data from training set to initialize the tuning model, and validate the better $\varphi_{F_1}^*$ on the remaining two months data. We find the $\varphi_{F_1}^* = 0.5$, compares to the original $\varphi = 0.25$ on the training data. Post-tuning, as shown in Figure 9a, the model displays a consistent and robust performance, maintaining a relatively stable F1-score throughout 2018. The stability was achieved by prioritizing an increase in Recall at the expense of Precision. This approach underscores the significance of establishing a realistic malware-to-goodware ratio in the training set, which is crucial for enhancing the model's practical applicability in real-world scenarios (C3).

5.2   Case Study: Hidost

Hidost, an open-source tool for extracting document structures from PDF and SWF files, forms the basis of our analysis [62]. We utilize the Hidost evaluation dataset, which includes data from a 14-week period between July 16 and October 21, 2012, comprising 1,465,737 benign and 102,234 malicious files (identified with 5+ VirusTotal AV detections). The dataset's features are derived from static analysis of the PDF files, extracting structural paths in their hierarchy, resulting in both Boolean and numeric features. Following the original Hidost study, we employ a Random Forest (RF) algorithm [12], training it on data from the initial four weeks and testing on the subsequent ten weeks.

The Hidost evaluation adheres to constraints C1 and C2. Defining a realistic malware-to-goodware ratio in the PDF domain, akin to the Windows PE domain, remains challenging. Therefore, we rely on the class balance in the original dataset. Figure Figure 8b displays stable performance across ten testing weeks, with minor dips in weeks 3 and 5. Two primary factors could explain the stable performance observed. Firstly, the robustness of the feature space is a significant contributor, especially considering that many malicious attacks in PDFs are executed via embedded JavaScript code, making them potentially more identifiable. Secondly, the limited duration of the evaluation period might not be sufficient for a concept drift to manifest significantly. This hypothesis necessitates further in-depth research to understand the true extent and nature of any potential concept drift in this context.

For tuning on the Hidost dataset, we use algorithm 1 by initializing the model with the first two weeks of the training set, and determining the optimal $\varphi_{F_1}^* = 0.1$ on the subsequent two weeks (compared to the original $\varphi = 0.04$). Figure Figure 9b illustrates the post-tuning changes. While the overall improvement is modest due to the already high baseline performance, the tuning eradicates the performance drop in week 5 and slightly improves the overall AUT(F_1, 10w) from 0.97 to 0.98, underscoring the benefits of adjusting the malware-to-goodware ratio.

5.3   Summary of Findings

In exploring the Windows PE and PDF malware domains, we encounter the challenge of accurately defining an in-the-wild malware-to-goodware ratio, critical for training effective detection models. Our case studies on both EMBER and Hidost reveal significantly less concept drift compared to the Android dataset, likely due to the more stable and predictable nature of malware evolution in these environments, as discussed in [15]. This stability is likely linked to the slower OS development cycle in Windows PE and the consistent format and usage in PDFs. Additionally, the criteria used for labeling an instance as malware vary across different approaches and domains. For instance, in the EMBER dataset, an instance is labeled as malware if it triggers 40 or more VirusTotal alerts. Notably, we find that applying our algorithm 1 to tune the malware-to-goodware ratio during the training process enhances detection performance in testing within these domains. This also indicates the importance of further research to define this ratio accurately and to understand the mechanisms behind the benefits of tuning.

6   DELAYING TIME DECAY

In this section, we explore delay time decay strategies using the Android dataset (§ 2.3). We exclude the Windows PE and PDF malware datasets (§ 5) from this analysis because they suffer less from concept drift. This focus on the Android dataset, with its more pronounced data changes, provides a clearer perspective on the effectiveness of these strategies in dynamic environments.
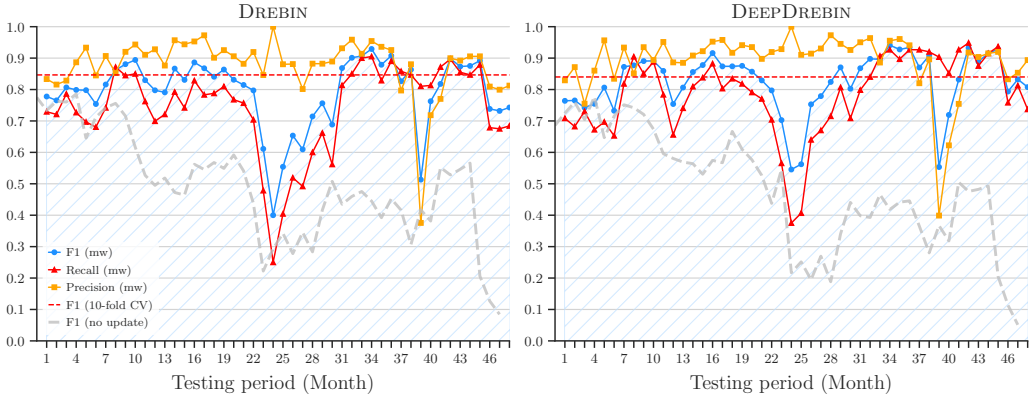
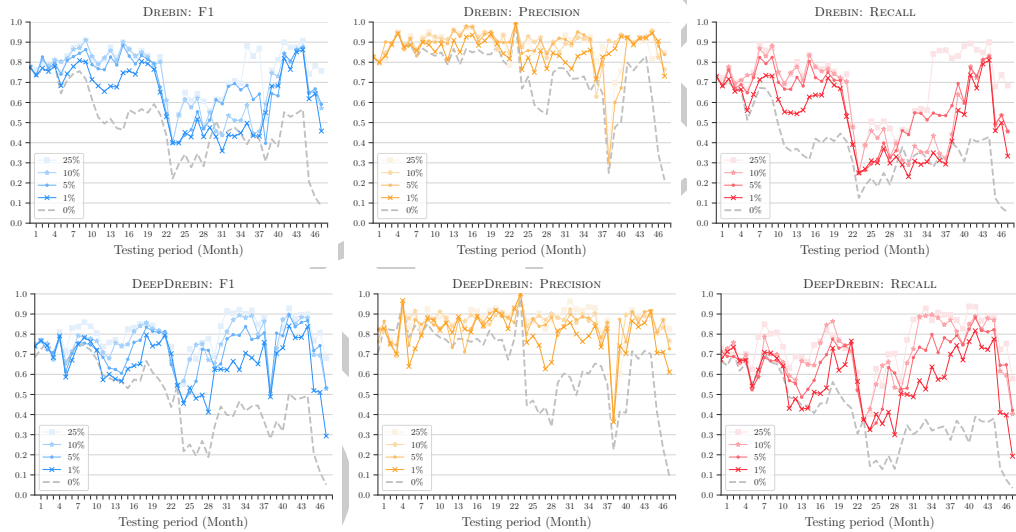Fig. 10. Delaying time decay: incremental retraining.

Fig. 11. Delay time decay: performance with active learning based on uncertainty sampling.

6.1 Delay Strategies

While our work does not propose novel delay strategies, it leverages TESSERACT to facilitate the comparative analysis of several popular approaches aimed at mitigating time decay. This provides a framework for researchers to employ TESSERACT in assessing various strategies fairly, especially when proposing new solutions to delay time decay within certain budget constraints. Below, we summarize the delay strategies under consideration and present results from our dataset. Additional information on these strategies is available for interested readers.

Incremental Retraining. We begin by examining an approach that sets an ideal performance benchmark: including *all* points in a monthly retraining regime. Figure 10 illustrates the performance of DREBIN and DEEPDREBIN with this monthly incremental retraining strategy. The results indicate a comparable upper performance limit for both methods, exhibiting an AUT(F_1, 48m) of
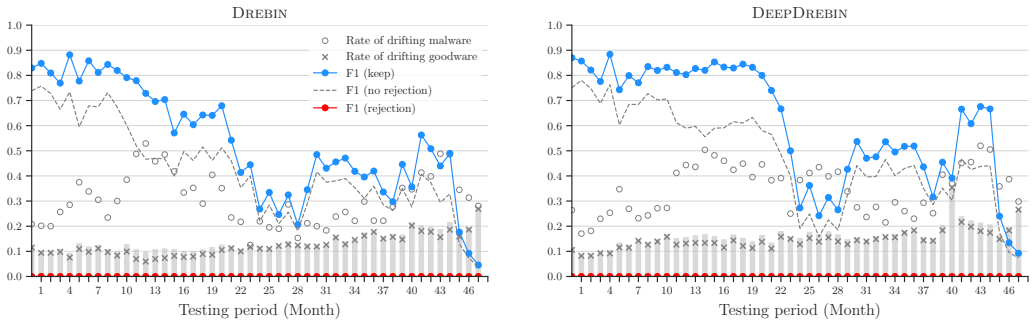
Fig. 12. Delay time decay: classification with TRANSCENDENT [8] rejection.

0.79 for DREBIN and 0.82 for DEEPDREBIN. Though representing an upper bound in performance, this strategy is likely impractical due to the continuous need for labeling a vast amount of objects. Despite the potential use of VirusTotal, the increased API usage incurs additional costs. Moreover, this approach might not be suitable in various security contexts.

Active Learning. Active Learning (AL) strategies focus on selecting a subset of test objects, whose labels are unknown, for manual labeling and inclusion in the training set. This is based on the premise that these objects are most beneficial for updating the classification model [60]. In our study, we employ the widely-used AL query strategy known as *uncertainty sampling* [40]. Here, objects with the most uncertain predictions are chosen for retraining, under the assumption that they are key to refining decision boundaries. Figure 11 displays the active learning outcomes via uncertainty sampling, detailing the F_1 score (blue), Precision (yellow), and Recall (red) metrics. Notably, the data indicates discernible performance gains with the retraining of as few as 1% of samples monthly. It is observed that both DREBIN and DEEPDREBIN achieve optimal performance when the monthly retraining rate is at its maximum of 25%, reinforcing the notion that a larger subset of retrained data is closely associated with enhanced model efficacy. It should be noted that at a monthly retraining rate of 100%, the AL approach effectively equates to Incremental Retraining.

Classification with Rejection. As a mitigation approach, classifiers can designate decisions as "low confidence" and defer them to a later date, effectively placing uncertain predictions into a quarantine zone pending manual review [9]. Figure 12 demonstrates how both DREBIN and DEEP-DREBIN perform when implementing a rejection mechanism for dubious classifications, utilizing the TRANSCENDENT framework [8, 37]. This method employs conformal evaluation theory to set thresholds that filter out the least reliable classifier decisions. Specifically, the inductive conformal evaluator (ICE) is used to pinpoint and exclude examples that deviate from expected patterns. While the original research focused on the nonconformity measure (NCM) for traditional ML algorithms, an adaptation of NCM has been developed for DNN (DEEPDREBIN) in this work, drawing upon the probabilistic outputs from its final SoftMax layer. Background gray histograms in the figures quantify the monthly count of quarantined objects, with cross and circle markers indicating the proportion of rejected malware and goodware, respectively.

6.2 Analysis of Delay Methods

To quantify performance-cost trade-offs of methods to delay time decay, we characterize the following three elements:

Performance (P): the performance measured in terms of AUT to capture robustness against time decay (§ 4.2);

Table 4. Performance-cost comparison of delay methods.

Delay method	Costs				Performance							
	L_c(48m)		Q_c(48m)		P : AUT(F_1, 48m)				P : AUT(F_1, 24m)			
					$\varphi = \hat{\sigma}$		$\varphi = \varphi_{F_1}^*$		$\varphi = \hat{\sigma}$		$\varphi = \varphi_{F_1}^*$	
	Drebin	DeepDrebin	Drebin	DeepDrebin	Drebin	DeepDrebin	Drebin	DeepDrebin	Drebin	DeepDrebin	Drebin	DeepDrebin
No update	0	0	0	0	0.496	0.489	0.505	0.534	0.609	0.636	0.645	0.667
AL: 1%	1,993	1,993	0	0	0.632	0.651	0.655	0.657	0.723	0.660	0.740	0.686
AL: 5%	10,052	10,052	0	0	0.698	0.713	0.722	0.735	0.778	0.709	0.800	0.735
AL: 10%	20,127	20,127	0	0	0.705	0.753	0.738	0.764	0.808	0.757	0.815	0.761
AL: 25%	50,355	50,355	0	0	0.763	0.793	0.789	0.801	0.809	0.775	0.814	0.791
Inc. retrain	201,490	201,490	0	0	0.794	0.819	0.802	0.838	0.808	0.812	0.810	0.834
Rejection ($\hat{\sigma}$)	0	0	29,391	35,127	0.537	0.613	–	–	0.658	0.779	–	–
Rejection ($\varphi_{F_1}^*$)	0	0	10,891	19,695	–	–	0.558	0.635	–	–	0.717	0.804

Labeling Cost (L_c): the number of testing objects (if any) that must be labeled—the labeling must occur periodically (e.g., every month), and is particularly costly in the malware domain as manual inspection requires many resources (infrastructure, time, expertise, etc)—for example, Miller *et al.* [48] estimated that an average company could only manually label 80 objects per day;

Quarantine Cost (Q_c): the number of objects (if any) rejected by the classifier—these must be manually verified, so there is a cost for leaving them in quarantine.

Table 4 delivers an in-depth performance-cost analysis of various delay methods applied to the Drebin and DeepDrebin malware classifiers. Using the AUT for the F_1-Score across two intervals - 24 and 48 months - the table outlines the implications of each method in terms of over all L_c, Q_c and performance gains.

The performance metrics are delineated based on two training scenarios: one with an estimated base ratio of malware in the training set ($\hat{\sigma}$) set at approximately 10% for both classifiers, and another with the optimized ratio ($\varphi_{F_1}^*$) determined to be 55% for Drebin and 20% for DeepDrebin, as detailed in Table 3. The best AUT outcomes for Drebin and DeepDrebin, accentuated in the table with distinct color highlights (orange and blue, respectively) , facilitate direct comparison and underscore the impact of the tuning process.

The initial setup with no updates provides a baseline for performance, just as shown in Figure 6 in § 4.4. The training ratios with $\varphi_{F_1}^*$, however, exhibit better results for both classifiers without incurring additional costs, underscoring the importance of precise initial training/tuning settings for sustained performance.

Upon examining delay methods aimed at mitigating performance degradation over time, the tuned training ratio $\varphi_{F_1}^*$ consistently yields performance enhancements for both classifiers. Furthermore, over the 48-month period, DeepDrebin demonstrates greater resilience in maintaining performance, suggesting its robustness to temporal decay—a significant attribute for systems intended for prolonged use without continuous modification.

A closer inspection of the costs involved in Table 4 reveals that performance improvements in the F_1-Score are accompanied by financial considerations. The more data we label, the better retrained performance we get. For example, employing Active Learning (AL) with a 25% relabeling rate results in a labeling cost of 50,355 samples over 48 months for both Drebin and DeepDrebin. This data prompts a careful evaluation of the balance between performance gains and the associated financial outlay. Additionally, we have to remember the costs that come from dealing with uncertain cases, which may not always be harmful but are difficult for the system to handle.

In the rejection scenarios using Transcendent, the advantages of the tuning algorithm are particularly noticeable, especially for DeepDrebin. A comparison between the two rejection rows clearly shows that the application of the optimal training ratio $\varphi_{F_1}^*$ allows the classifier to more effectively quarantine dubious decisions. For Drebin, Transcendent leads to a significant decrease in rejected samples—two-thirds less—while still improving performance: AUT(F_1, 24m) rises from

0.658 to 0.717, and AUT(F_1, 48m) from 0.537 to 0.558 respectively. Notably, Transcendent even enables DeepDrebin to outperform the AUT(F_1, 24m) of the Active Learning scenario with a 25% budget, while quarantining fewer samples.

In conclusion, Table 4 offers not just an academic exercise but a concrete tool for the industry. It underlines the critical need for an in-depth understanding of classifier performance within actual operational contexts. This involves striking a delicate balance between analytical assessment of algorithms and the practical aspects of resource distribution for labeling and quarantining actions. The methodology outlined in this table provides a pathway to reduce costs and improve classifier performance, making it a crucial asset for the field. The challenge of creating cost-effective and sturdy classification systems is significant, yet sharing the code related to this research is expected to encourage ongoing innovation and real-world usage.

7   RELATED WORK

Experimental Bias. The *base rate fallacy* [7] is a well-known experimental bias in security, which highlights the limitations of TPR and FPR as performance metrics in highly imbalanced datasets. For instance, in network intrusion detection, where most traffic is benign, even a low FPR of 1% may correspond to millions of false positives and only thousands of true positives. In contrast, our work uncovers experimental settings that are misleading regardless of the metrics used, and that remain incorrect even if the correct metrics are employed (§ 4.4). While previous works such as Sommer and Paxson [61], Rossow *et al.* [57], and Kouwe *et al.* [70] provide valuable insights and guidelines for conducting security experiments, they do not address the specific issue of temporal and spatial bias in the Android platform. Furthermore, their guidelines would not eliminate all sources of such biases as identified in our work. For example, Rossow *et al.* [57] evaluate the percentage of objects in previously adopted datasets that are incorrectly labeled, without assessing the impact of such errors on classifier performance. Zhou *et al.* [76] show that Hardware Performance Counters (HPCs) are not very effective for malware classification, but their focus is narrow, and they rely on 10-fold cross-validation in the evaluation. Finally, a recent paper published by Arp *et al.* [5] identifies ten pitfalls for machine learning in computer security, which relates to the spatio-temporal biases we discuss. The pitfalls of sampling bias and base rate fallacy overlap with spatial bias. Along with data snooping and spurious correlations relating to temporal bias. It is important to note that these connections are in part due to the paper building off our original release of Tesseract [54]. However, Arp *et al.* [5] provide the prevalence, implications, and recommendations for these pitfalls opposed to the concrete solutions provided in this work for spatio-temporal biases.

Time-aware Detection. Allix *et al.* [2] made an important contribution by evaluating malware classifiers with respect to time and demonstrating how future knowledge can inflate performance. However, they did not propose a solution for comparable evaluations and only identified constraint C1. Additionally, Allix *et al.* [1] investigated the differences between in-the-lab and in-the-wild scenarios and found that a greater presence of goodware leads to lower performance. In contrast, our work systematically analyzes and explains these issues, and addresses them by formalizing a set of constraints that jointly consider the impact of temporal and spatial bias. We introduce AUT as a unified performance metric for fair time-aware comparisons of different solutions, and offer a tuning algorithm to leverage the effects of training data distribution. While Miller *et al.* [48] identified *temporal sample consistency* (equivalent to our constraint C1), they did not identify C2 or C3, which are fundamental (§ 4.4). Moreover, they considered the test period to be a uniform time slot, whereas we take time decay into account. Roy *et al.* [58] questioned the use of recent or older malware as training objects and the performance degradation in testing real-world object ratios. However, their experiments were designed without considering time, which reduces the reliability of their conclusions. Past work has highlighted some sources of experimental bias [1, 2, 48, 58],

but it has given little consideration to classifiers' goals. Different scenarios may have different objectives, which may not necessarily involve maximizing F_1. In our work, we demonstrate the effects of different training settings on performance goals and propose an algorithm to properly tune a classifier accordingly (§ 4.3). Following the publication of [53, 54], numerous studies have adopted time-aware evaluations to demonstrate their performance across various temporal stages. Notable examples include [8, 14, 72, 75].

Unbiased Methods from Other Domains. The pursuit of unbiased methods is a crucial aspect in various other fields [13, 36, 71]. However, since these studies do not originate from the security domain, they only focus on some aspects of spatial bias and do not consider temporal bias. Concept drift is especially problematic in Android malware compared to other applications, such as image and text classification [8, 37]. Fawcett [26] highlights challenges in spam detection, where one of the challenges is similar to spatial bias, but without providing any solution. In contrast, we propose C3 to address spatial bias and demonstrate how its violation inflates performance (§ 4.4). Torralba and Efros [68] discuss dataset bias in computer vision, which is different from our security setting where there are fewer benchmarks, and the negative class (e.g., "not cat") in images can grow arbitrarily, which is less likely in the malware context. Moreno-Torres *et al.* [50] systematize different types of drift and mention sample-selection bias, which resembles spatial bias, but they do not propose any solution or experiments to evaluate its impact on ML performance. Other related work emphasizes the importance of choosing appropriate performance metrics to avoid an incorrect interpretation of the results (e.g., ROC curves are misleading in an imbalanced dataset [21, 35]). In this paper, we consider imbalance, propose actionable constraints and metrics, and provide tool support to evaluate the performance decay of classifiers over time.

Summary. Our research is motivated by several studies on bias, but none of them comprehensively address the problem in the context of evolving data where the i.i.d. assumption does not hold. While Miller *et al.* [48] introduced Constraint C1, it alone is insufficient to eliminate bias. It is demonstrated by other evaluations such as [46], which only enforces C1. In § 4.4 of our paper, we clarify why our novel constraints C2 and C3 are fundamental and show how our AUT metric can effectively reveal the true performance of algorithms. Our approach provides counter-intuitive results that highlight the importance of considering both spatial and temporal bias when evaluating classifiers on evolving data.

8 DISCUSSION

We now discuss guidelines, our assumptions, and how we address limitations of our work.

Further Considerations on Temporal Consistency (Constraint C1). One might be tempted to test how their classifier performs with respect to older data. This essentially requires to modify C1 to account for two distinct testing time frames that do not overlap with the training data one. For instance, antivirus companies might intentionally use newer threats to make sure their models can still detect older yet relevant security risks. In this case C1 can be modified to:

$$time(s_j) < time(s_i) < time(s_k), \forall s_i \in Tr, \forall s_j, s_k \in Ts \qquad (6)$$

This raises important philosophical questions about classifier evaluation methods, representing an open research question that we suggest to address as future work.

Generalization to other security domains. Our TESSERACT methodology, while demonstrated in three security domains, is broadly applicable across various ML-driven security areas for bias-free evaluations. It adapts to different domains through certain specific parameters like time granularity and test duration. This need for domain-specific settings is an inherent aspect, not a limitation, of our approach. Generally, spatio-temporal bias is likely to impact other security domains experiencing concept drift, but further research is needed for definitive conclusions.

The effectiveness of TESSERACT in understanding this bias hinges on having access to extensive timestamped datasets, realistic class ratio knowledge, and either the code or detailed information to reproduce existing baselines.

Domain-specific in-the-wild malware percentage $\hat{\sigma}$. Correctly estimating the malware percentage in dataset is a challenging task and we encourage further measurement studies [42, 69] and data sharing to obtain realistic experimental settings in different security domains.

Correct observation labels. We assume labels in datasets we use are correct. Miller *et al.* [48] found that AVs sometimes change their outcome over time: some goodware may eventually be tagged as malware. However, they also found that VirusTotal detections stabilize after one year; since we are using observations up to Dec 2018, we consider labels we used as reliable. In the future, we may integrate approaches for *noisy oracles* [24], which select only trustful observations.

Timestamps in the dataset. It is important to consider that some timestamps in a public dataset could be incorrect or invalid. In this paper, we rely on the public AndroZoo dataset maintained at the University of Luxembourg, and we rely on the dex_date attribute as the approximation of an observation timestamp, as recommended by the dataset creators [3]. We further verified the reliability of the dex_date attribute by re-downloading VirusTotal [31] reports for 25K apps[1] and verifying that the first_seen attribute always matched the dex_date within our time span. In general, we recommend performing some sanitization of a timestamped dataset before performing any analysis on it: if multiple timestamps are available for each object, consider the most reliable timestamp you have access to (e.g., the timestamp recommended by the dataset creators, or the VirusTotal's first_seen attribute) and discard objects with "impossible" timestamps (e.g., with dates which are either too old or in the future), which may be caused by incorrect parsing or invalid values of some timestamps. To improve trustworthiness of the timestamps, one could verify whether a given object contains time inconsistencies or features not yet available when the app was released [41]. We encourage the community to promptly notify dataset maintainers of any date inconsistencies. In the TESSERACT's project website (section 9), we will maintain an updated list of timestamped datasets publicly available for the security community.

Resilience of malware classifiers. In our study, we analyze both traditional ML approach DREBIN and DNN approach DEEPDREBIN. One could argue that other classifiers may show consistently high performance even with space-time bias eliminated. And this should indeed be the goal of research on malware classification. TESSERACT provides a mechanism for an unbiased evaluation that we hope will support this kind of work.

Adversarial ML. Adversarial ML focuses on perturbing training or testing observations to compel a classifier to make incorrect predictions [10]. Both relate to concepts of *robustness* and one can characterize adversarial ML as an artificially induced worst-case concept drift scenario. While the adversarial setting remains an open problem, the experimental bias we describe in this work—endemic in Android malware classification—must be addressed prior to realistic evaluations of adversarial mitigations.

9 AVAILABILITY

We make TESSERACT's code and data available to the research community to promote the adoption of a sound and unbiased evaluation of classifiers. The TESSERACT project website with instructions to request access is at https://github.com/s2labres/tesseract-ml/, and all experiments related code can be found at https://github.com/s2labres/tesseract-journal-experiments/. We will also keep maintaining an updated list of publicly available security-related datasets with timestamped objects.

---

[1]We downloaded only 25K VirusTotal reports (corresponding to about 10% of our dataset) due to restrictions on our VirusTotal API usage quota.

10 CONCLUSIONS

We have identified novel temporal and spatial biases in ML-driven malware detection, and proposed novel constraints, metrics and tuning algorithm to address such issues. We have built and released TESSERACT as an open-source tool that integrates our methods. We have shown the two experimental biases commonly exist in detection tasks, and how TESSERACT can reveal the real performance of malware classifiers that remain hidden in wrong experimental settings in a non-stationary context. TESSERACT is fundamental for the correct evaluation and comparison of different solutions, in particular when considering mitigation strategies for time decay.

TESSERACT offers valuable insights for both researchers and practitioners. It enables evaluation of a classifier's inherent robustness to performance decay using a baseline AUT performance metric, highlighting true performance and sometimes surprising results. This is particularly useful in deployment scenarios where frequent retraining is not feasible due to financial or computational constraints. Our experiments reveal that even with retraining, classifiers may not remain consistent over time, and TESSERACT assists in eliminating spatio-temporal bias in evaluations, allowing researchers to focus on developing more robust algorithms. Our experiments also indicate the necessity of regular retraining and tuning for sustained performance. Additionally, TESSERACT supports various time granularities for analysis and allows different observation time windows for evaluation, enabling researchers to assess and compare the robustness of algorithms over specific time frames, such as the upcoming three months, ensuring bias-free comparisons and consideration of time decay.

We envision that future work on malware classification will use TESSERACT to produce realistic, comparable and unbiased results. We encourage the community to adopt TESSERACT to deeply evaluate the impact of temporal and spatial bias in any security domain where concept drift appears.

REFERENCES

[1] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. 2016. Empirical Assessment of Machine Learning-Based Malware Detectors for Android. *Empirical Software Engineering* (2016).

[2] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Are Your Training Datasets Yet Relevant?. In *ESSoS*. Springer.

[3] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting Millions of Android Apps for the Research Community. In *Mining Software Repositories*. ACM.

[4] H. S. Anderson and P. Roth. 2018. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints* (April 2018). arXiv:1804.04637 [cs.CR]

[5] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*. 3971–3988.

[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 *NDSS*.

[7] Stefan Axelsson. 2000. The Base-Rate Fallacy and the Difficulty of Intrusion Detection. *ACM TISSEC* (2000).

[8] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. 2022. Transcending transcend: Revisiting malware classification in the presence of concept drift. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 805–823.

[9] Peter L Bartlett and Marten H Wegkamp. 2008. Classification with a reject option using a hinge loss. *JMLR* (2008).

[10] Battista Biggio and Fabio Roli. 2018. Wild Patterns: Ten Years After The Rise of Adversarial Machine Learning. *Pattern Recognition* (2018).

[11] Christopher M Bishop. 2006. *Pattern Recognition and Machine Learning*.

[12] Leo Breiman. 2001. Random forests. *Machine learning* 45 (2001), 5–32.

[13] Nitesh V Chawla, Nathalie Japkowicz, and Aleksander Kotcz. 2004. Special Issue on Learning From Imbalanced Data Sets. *ACM SIGKDD Explorations Newsletter* (2004).

[14] Yizheng Chen, Zhoujie Ding, and David Wagner. 2023. Continuous Learning for Android Malware Detection. In *USENIX Security Symposium*.

[15] Zhi Chen, Zhenning Zhang, Zeliang Kan, Limin Yang, Jacopo Cortellazzi, Feargus Pendlebury, Fabio Pierazzi, Lorenzo Cavallaro, and Gang Wang. 2023. Is It Overkill? Analyzing Feature-Space Concept Drift in Malware Detectors. In *2023 IEEE Deep Learning Security and Privacy Workshop (DLSP)*. IEEE.

[16] Theo Chow, Zeliang Kan, Lorenz Linhardt, Daniel Arp, Lorenzo Cavallaro, and Fabio Pierazzi. 2023. Drift Forensics of Malware Classifiers. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*. ACM.

[17] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *USENIX Security*.

[18] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. 2013. Large-Scale Malware Classification Using Random Projections and Neural Networks. In *Int. Conf. Acoustics, Speech and Signal Processing (ICASSP)*. IEEE.

[19] Nadia Daoudi, Kevin Allix, Tegawendé François Bissyandé, and Jacques Klein. 2022. A deep dive inside drebin: An explorative analysis beyond android malware detection scores. *ACM Transactions on Privacy and Security* 25, 2 (2022), 1–28.

[20] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. 2016. Droidscribe: Classifying Android Malware Based on Runtime Behavior. In *MoST-SPW*. IEEE.

[21] Jesse Davis and Mark Goadrich. 2006. The Relationship Between Precision-Recall and ROC Curves. In *Proceedings of the 23rd international conference on Machine learning*. ACM, 233–240.

[22] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. 2017. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE transactions on dependable and secure computing* 16, 4 (2017), 711–724.

[23] K Divya and Venkata Krishna Kumar. 2016. Comparative analysis of smart phone operating systems Android, Apple IOS and Windows. *International Journal of Scientific Engineering and Applied Science (IJSEAS)* 2, 2 (2016), 432–439.

[24] Jun Du and Charles X Ling. 2010. Active Learning with Human-Like Noisy Oracle. In *ICDM*. IEEE.

[25] Guillaume Endignoux, Olivier Levillain, and Jean-Yves Migeon. 2016. Caradoc: A pragmatic approach to pdf parsing and validation. In *2016 IEEE Security and Privacy Workshops (SPW)*. Ieee, 126–139.

[26] Tom Fawcett. 2003. In vivo spam filtering: a challenge problem for KDD. *ACM SIGKDD Explorations Newsletter* (2003).

[27] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. 2011. A survey of mobile malware in the wild. *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (2011). https://doi.org/10.1145/2046614.2046618

[28] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[29] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural Detection of Android Malware using Embedded Call Graphs. In *AISec*. ACM.

[30] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. MIT press Cambridge.

[31] Google. 2004. *VirusTotal*. https://www.virustotal.com

[32] Google. March 2018. Android Security 2017 Year In Review. https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf.

[33] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial examples for malware detection. In *ESORICS*. Springer.

[34] Jiaxiang Gu, Rui Kong, He Sun, Honglin Zhuang, Fan Pan, and Zhechao Lin. 2022. A novel detection technique based on benign samples and one-class algorithm for malicious PDF documents containing JavaScript. In *International Conference on Computer Application and Information Security (ICCAIS 2021)*, Vol. 12260. SPIE, 599–607.

[35] David J Hand. 2009. Measuring Classifier Performance: a Coherent Alternative to the Area Under the ROC Curve. *Machine Learning* (2009).

[36] Haibo He and Edwardo A Garcia. 2009. Learning From Imbalanced Data. *IEEE TKDE* (2009).

[37] 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*.

[38] Pavel Laskov and Nedim Šrndić. 2011. Static Detection of Malicious JavaScript-Bearing PDF Documents. In *ACSAC*. ACM.

[39] Sangho Lee and Jong Kim. 2012. WarningBird: Detecting Suspicious URLs in Twitter Stream. In *NDSS*.

[40] David D Lewis and Jason Catlett. 1994. Heterogeneous uncertainty sampling for supervised learning. In *Machine learning proceedings 1994*. Elsevier, 148–156.

[41] Li Li, Tegawendé Bissyandé, and Jacques Klein. 2018. MoonlightBox: Mining Android API Histories for Uncovering Release-time Inconsistencies. In *Symp. on Software Reliability Engineering*. IEEE.

[42] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. 2014. AndRadar: Fast Discovery of Android Applications in Alternative Markets. In *DIMVA*. Springer.

[43] Xiang Ling, Lingfei Wu, Jiangyu Zhang, Zhenqing Qu, Wei Deng, Xiang Chen, Yaguan Qian, Chunming Wu, Shouling Ji, Tianyue Luo, et al. 2023. Adversarial attacks against Windows PE malware detection: A survey of the state-of-the-art. *Computers & Security* (2023), 103134.

[44] Federico Maggi, Alessandro Frossi, Stefano Zanero, Gianluca Stringhini, Brett Stone-Gross, Christopher Kruegel, and Giovanni Vigna. 2013. Two Years of Short URLs Internet Measurement: Security Threats and Countermeasures. In *WWW*. ACM.

[45] Davide Maiorca, Giorgio Giacinto, and Igino Corona. 2012. A Pattern Recognition System for Malicious PDF Files Detection. In *Intl. Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer.

[46] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *NDSS*.

[47] Zane Markel and Michael Bilzor. 2014. Building a Machine Learning Classifier for Malware Detection. In *Anti-malware Testing Research Workshop*. IEEE.

[48] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. 2016. Reviewer Integration and Performance Measurement for Malware Detection. In *DIMVA*. Springer.

[49] Bradley Austin Miller. 2015. *Scalable Platform for Malicious Content Detection Integrating Machine Learning and Manual Review*. University of California, Berkeley.

[50] Jose G Moreno-Torres, Troy Raeder, RocíO Alaiz-RodríGuez, Nitesh V Chawla, and Francisco Herrera. 2012. A unifying view on dataset shift in classification. *Pattern Recognition* (2012).

[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. https://pytorch.org/.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-Learn: Machine Learning in Python. *JMLR* (2011).

[53] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2018. POSTER: Enabling Fair ML Evaluations for Security. In *CCS*. ACM.

[54] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium (USENIX Security 19)*. 729–746.

[55] Babak Rahbarinia, Marco Balduzzi, and Roberto Perdisci. 2017. Exploring the Long Tail of (Malicious) Software Downloads. In *DSN*. IEEE.

[56] Konrad Rieck, Tammo Krueger, and Andreas Dewald. 2010. Cujo: Efficient Detection and Prevention of Drive-By-Download Attacks. In *ACSAC*. ACM.

[57] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. 2012. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *Symp. S&P*. IEEE.

[58] Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, Xinming Ou, Venkatesh Prasad Ranganath, Hongmin Li, and Nicolais Guevara. 2015. Experimental Study with Real-World Data for Android App Security Analysis Using Machine Learning. In *ACSAC*. ACM.

[59] Craig Saunders, Alexander Gammerman, and Volodya Vovk. 1998. Ridge regression learning algorithm in dual variables. (1998).

[60] Burr Settles. 2012. Active Learning Literature Survey. *Synthesis Lectures on Artificial Intelligence and Machine Learning* (2012).

[61] Robin Sommer and Vern Paxson. 2010. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *Symp. S&P*. IEEE.

[62] Nedim Šrndić and Pavel Laskov. 2016. Hidost: a static machine-learning-based detector of malicious files. *EURASIP Journal on Information Security* 2016 (2016), 1–20.

[63] Caio Steglich, Sabrina Marczak, Luiz Pedro Guerra, Luiz Henrique Mosmann, Marcelo Perin, Fernando Figueira Filho, and Cleidson de Souza. 2019. Revisiting the mobile software ecosystems literature. In *2019 IEEE/ACM 7th International Workshop on Software Engineering for Systems-of-Systems (SESoS) and 13th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (WDES)*. IEEE, 50–57.

[64] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. 2013. Shady Paths: Leveraging Surfing Crowds to Detect Malicious Web Pages. In *CCS*. ACM.

[65] 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 *CODASPY*. ACM.

[66] Masashi Sugiyama, Neil D Lawrence, Anton Schwaighofer, et al. 2009. *Dataset Shift in Machine Learning*. The MIT Press.

[67] Gil Tahan, Lior Rokach, and Yuval Shahar. 2012. Mal-Id: Automatic Malware Detection Using Common Segment Analysis and Meta-features. *JMLR* (2012).

[68] Antonio Torralba and Alexei A Efros. 2011. Unbiased look at dataset bias. In *CVPR*. IEEE.

[69] Phani Vadrevu, Babak Rahbarinia, Roberto Perdisci, Kang Li, and Manos Antonakakis. 2013. Measuring and Detecting Malware Downloads in Live Network Traffic. In *ESORICS*. Springer.

[70] Erik van der Kouwe, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Gernot Heiser. 2018. Benchmarking Crimes: An Emerging Threat in Systems Security. *arXiv preprint* (2018).

[71] Gary M Weiss and Foster Provost. 2003. Learning when Training Data Are Costly: The Effect of Class Distribution on Tree Induction. *Journal of Artificial Intelligence Research* (2003).

[72] Ke Xu, Yingjiu Li, Robert Deng, Kai Chen, and Jiayun Xu. 2019. Droidevolver: Self-evolving android malware detection system. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 47–62.

[73] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-sec: Deep Learning in Android Malware Detection. In *SIGCOMM Computer Communication Review*. ACM.

[74] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual Api Dependency Graphs. In *CCS*. ACM.

[75] 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 *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 757–770.

[76] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. 2018. Hardware Performance Counters Can Detect Malware: Myth or Fact?. In *ASIACCS*. ACM.

[77] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*. IEEE, 95–109.

A   APPENDIX

A.1   Algorithm Hyperparameters

In the study of malware detection algorithms, hyperparameters play a critical role in defining the behavior and performance of the classification models. This section details the specific hyperparameters utilized for the two algorithms tested: DREBIN and DEEPDREBIN.

The DREBIN algorithm is instantiated using a linear Support Vector Machine (SVM) from the SCIKIT-LEARNlibrary, specifically employing *sklearn.svm.LinearSVC*. In alignment with the original DREBIN study, the SVM was fine-tuned with a regularization parameter C set to 1. This choice reflects a balance between the correct classification of training examples and the maximization of the decision function's margin. The dual parameter's automatic setting ensures the algorithm optimally chooses between the primal and dual problem formulations, contingent on the dataset's dimensionality. The solver's iterations were limited to 5000 to establish a convergence threshold, reflecting a practical tolerance for the training data's optimization process.

This hyperparameter configuration is a deliberate effort to replicate the original DREBIN paper's findings. Despite slight modifications to accommodate the current study's dataset, this setup achieved a commendable 10-fold cross-validated F_1 score of around 0.91, a figure modestly trailing the original performance, thereby validating the reproducibility and robustness of the approach.

The DEEPDREBIN model represents a deep learning approach, architected with a multi-layer sequential design. It comprises an input layer, succeeded by two hidden layers with 200 neurons each. Post each hidden layer, a ReLU (Rectified Linear Unit) function is applied, injecting non-linearity, coupled with a dropout strategy at a rate of 0.5 to mitigate overfitting. The terminal layer of the network is binary, outputting two values for the classification task.

Training hyperparameters for DEEPDREBIN were meticulously set: the model underwent 10 epochs, ensuring thorough exposure to the training dataset. The batch size was chosen as 64, balancing computational load with frequency of parameter updates. A learning rate of 0.05 was

selected to navigate the parameter space effectively. Stochastic Gradient Descent (SGD) serves as the optimizer, steering the loss function towards an optimal minimum.

In adherence to the protocols in [33], DeepDrebin was re-implemented using PyTorch. Initial inputs were aligned with those utilized by Drebin. We mirrored the neural network architecture that demonstrated optimal results as per [33]. Yet, we encountered a lack of comprehensive detail on hyperparameter optimization in the source material. Our refined implementation, however, culminated in a marginally elevated F_1 score within a 10-fold cross-validation framework (§ 4.4). We postulate that this increment stems from the original research's optimization for the Accuracy metric [11], which can be deceptive in imbalanced datasets [7]—a common characteristic in the Android domain, dominated by goodware.

A.2 Symbol Table

Table 5 is a legend of the main symbols used throughout this paper to improve readability.

Table 5. Symbol table.

Symbol	Description
gw	Short version of goodware.
mw	Short version of malware.
ML	Short version of Machine Learning.
D	Labeled dataset with malware (mw) and goodware (gw).
Tr	Training dataset.
W	Size of the time window of the training set (e.g., 1 year).
Ts	Testing dataset.
S	Size of the time window of the testing set (e.g., 2 years).
TP, FP, TN, FN	Stand for True Positive, False Positive, True Negative, and False Negative, respectively
Δ	Size of the test time-slots for time-aware evaluations (e.g., months).
$\mathbb{P}$	Performance target of the tuning algorithm in § 4.3; it can be F_1-Score (F_1), Precision (Pr) or Recall (Rec).
$AUT(\mathbb{P},N)$	Area Under Time, a new metric we define to measure performance over time decay and compare different solutions (§ 4.2). It is always computed with respect to a performance function $\mathbb{P}$ (e.g., F_1-Score) and N is the number of time units considered (e.g., 24 months)
τ	Size of AUT observation time window.
$\hat{\sigma}$	Estimated percentage of malware (mw) in the wild.
φ	Percentage of malware (mw) in the training set.
δ	Percentage of malware (mw) in the testing set.
$\varphi_{\mathbb{P}}^*$	Percentage of malware (mw) in the training set, to improve performance $\mathbb{P}$ on the malware (mw) class (§ 4.3).
E	Error rate (§ 4.3).
E_{max}	Maximum error rate when searching $\varphi_{\mathbb{P}}^*$ (§ 4.3).
Θ	Model learned after training a classifier.
L_c	Labeling cost.
Q_c	Quarantine cost.
P	Actual performance; depending on the context, it can refer to AUT with F_1 or Pr or Rec.

A.3 Cumulative Plots for Time Decay

Figure 13 shows the cumulative performance plot defined in § 4.2. This is the cumulative version of Figure 6 on the Android dataset.
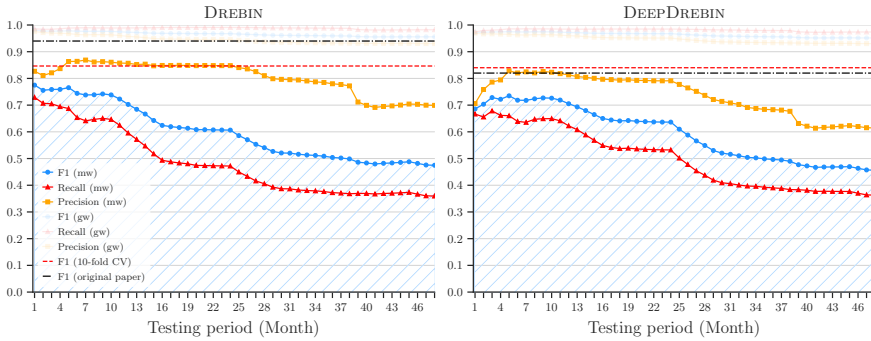
Fig. 13. Performance time decay with cumulative estimate for DREBIN and DEEPDREBIN. Testing distribution has $\delta = 10\%$ malware, and training distribution has $\varphi = 10\%$ malware.

A.4 TESSERACT Implementation

Our Python library, TESSERACT, incorporates our constraints, metrics, and algorithms, and is designed for seamless integration with standard workflows. Its API, closely aligned with and fully compatible with SCIKIT-LEARN [52] and PYTORCH [51], ensures familiarity for users of these libraries. A detailed overview of TESSERACT's core modules is available in [53].

loader.py is designed to transform the input dataset into a structure compatible with TESSERACT. It outputs a vectorized matrix X, a label array y, and an associated datetime array t. This transformation can be executed by calling the load_features() function. Additionally, the file contains the feature selection function (feature_reduce()), utilized in § 2.4, for efficient feature processing.

temporal.py enhances traditional machine learning frameworks that utilize input features X and target variables y by incorporating datetime objects t. This addition enables time-sensitive data operations, such as dataset partitioning which adheres to temporal constraints C1 and C2, with functions like time_aware_partition() and time_aware_train_test_split(). A granularity parameter is also provided, allowing the specification of the evaluation time window.

spatial.py This module allows the user to alter the proportion of the positive class in a given dataset. downsample_set() can be used to simulate the natural class distribution $\hat{\sigma}$ expected during deployment or to tune the performance of the model by over-representing a class during training. To this end we provide an implementation of Algorithm 1 (§ 4.3) for finding the optimal training proportion φ^* (search_optimal_train_ratio()). This module can also assert that constraint C3 (§ 4.1) has not been violated.

metrics.py As TESSERACT aims to encourage comparable and reproducible evaluations, we include functions for visualizing classifier assessments and deriving metrics such as the accuracy or total errors from slices of a time-aware evaluation. Importantly we also include aut(), with optional parameter granularity (δ) and observation time window (τ), for computing the AUT (§ 4.2) for a given metric (F_1, Precision, Recall, etc.) over a given time period.

evaluation.py Here we include the predict() and fit_predict_update() functions that accept a classifier, dataset and set of parameters (as defined in § 4.1) and return the results of a time-aware evaluation (§ 4.4) performed across the chosen periods.

selection.py *and* **rejection.py** For extending the evaluation to testing model update strategies, these modules provide hooks for novel query and reject strategies to be easily plugged into the evaluation cycle. We already implement many of the methods discussed in § 6 and include them with our release. We hope this modular approach lowers the bar for future researches.