This is the peer reviewd version of the followng article:

Tesseract: Eliminating experimental bias in malware classification across space and time / Pendlebury, F.; Pierazzi, F.; Jordaney, R.; Kinder, J.; Cavallaro, L.. - (2019), pp. 729-746. (Intervento presentato al convegno 28th USENIX Security Symposium tenutosi a usa nel 2019).

USENIX Association

Terms of use:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

17/04/2024 19:48

(Article begins on next page)

TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time

Feargus Pendlebury*†‡, Fabio Pierazzi*†‡, Roberto Jordaney†‡, Johannes Kinder§, Lorenzo Cavallaro†

† *King's College London*

‡ *Royal Holloway, University of London*

§ *Bundeswehr University Munich*

Abstract

Is Android malware classification a solved problem? Published F_1 scores of up to 0.99 appear to leave very little room for improvement. In this paper, we argue that results are commonly inflated due to two pervasive sources of experimental bias: *spatial bias* caused by distributions of training and testing data that are not representative of a real-world deployment; and *temporal bias* caused by incorrect time splits of training and testing sets, leading to impossible configurations. We propose a set of space and time constraints for experiment design that eliminates both sources of bias. We introduce a new metric that summarizes the expected robustness of a classifier in a real-world setting, and we present an algorithm to tune its performance. Finally, we demonstrate how this allows us to evaluate mitigation strategies for time decay such as active learning. We have implemented our solutions in TESSERACT, an open source evaluation framework for comparing malware classifiers in a realistic setting. We used TESSERACT to evaluate three Android malware classifiers from the literature on a dataset of 129K applications spanning over three years. Our evaluation confirms that earlier published results are biased, while also revealing counter-intuitive performance and showing that appropriate tuning can lead to significant improvements.

1 Introduction

Machine learning has become a standard tool for malware research in the academic security community: it has been used in a wide range of domains including Windows malware [12, 34, 51], PDF malware [27, 32], malicious URLs [28, 48], malicious JavaScript [11, 43], and Android malware [4, 22, 33]. With tantalizingly high performance figures, it seems malware should be a problem of the past.

Malware classifiers operate in dynamic contexts. As malware evolves and new variants and families appear over time, prediction quality decays [26]. Therefore, temporal consistency matters for evaluating the effectiveness of a classifier. When the experimental setup allows a classifier to train on what is effectively future knowledge, the reported results become biased [2, 36].

This issue is widespread in the security community and affects multiple security domains. In this paper, we focus on Android malware and claim that there is an endemic issue in that Android malware classifiers [4, 13, 18, 22, 33, 49, 56, 57] (including our own work) are not evaluated in settings representative of real-world deployments. We choose Android because of the availability of (a) a public, large-scale, and timestamped dataset (AndroZoo [3]) and (b) algorithms that are feasible to reproduce (where all [33] or part [4] of the code has been released).

We identify experimental bias in two dimensions, *space* and *time*. *Spatial bias* refers to unrealistic assumptions about the ratio of goodware to malware in the data. The ratio of goodware to malware is domain-specific, but it must be enforced consistently during the testing phase to mimic a realistic scenario. For example, measurement studies on Android suggest that most apps in the wild are goodware [21, 30], whereas for (desktop) software download events most URLs are malicious [31, 41]. *Temporal bias* refers to temporally inconsistent evaluations which integrate future knowledge about the testing objects into the training phase [2, 36] or create unrealistic settings. This problem is exacerbated by families of closely related malware, where including even one variant in the training set may allow 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 tool support.

Prior work has investigated challenges and experimental bias in security evaluations [2, 5, 36, 44, 47, 54]. The *base-rate*

*Equal contribution.

fallacy [5] describes how evaluation metrics such as *TPR* and *FPR* are misleading in intrusion detection, due to significant class imbalance (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. Sommer and Paxson [47], Rossow et al. [44], and van der Kouwe et al. [54] discuss possible guidelines for sound security evaluations; but none of these works identify temporal and spatial bias, nor do they *quantify* the impact of errors on classifier performance. Allix et al. [2] and Miller et al. [36] identify an initial temporal constraint in Android malware classification, but we show that even results of recent work following their guidelines (e.g., [33]) suffer from other temporal and spatial bias (§4.4). To the best of our knowledge, we are the first to identify and address these sources of bias with novel, actionable constraints, metrics, and tool support (§4).

This paper makes the following contributions:

- We identify *temporal* bias associated with incorrect train-test splits (§3.2) and *spatial* bias related to unrealistic assumptions in dataset distribution (§3.3). We experimentally verify on a dataset of 129K apps (with 10% malware) that, due to bias, performance can decrease up to 50% in practice (§3.1) in two well-known Android malware classifiers, DREBIN [4] and MAMADROID [33], which we refer to as ALG1 and ALG2, respectively.
- We propose novel building blocks for more robust evaluations of malware classifiers: a set of spatio-temporal constraints to be enforced in experimental settings (§4.1); a new metric, AUT, that captures a classifier's robustness to time decay in a single number and allows for the fair comparison of different algorithms (§4.2); and a novel tuning algorithm that empirically optimizes the classification performance, when malware represents the minority class (§4.3). We compare the performance of ALG1 [4], ALG2 [33] and DL [22] (a deep learning-based approach), and show how removing bias can provide counter-intuitive results on real performance (§4.4).
- We implement and publicly release the code of our methodology (§4), TESSERACT, and we further demonstrate how our findings can be used to evaluate performance-cost trade-offs of solutions to mitigate time decay such as active learning (§5).

TESSERACT can assist the research community in producing comparable results, revealing counter-intuitive performance, and assessing a classifier's prediction qualities in an industrial deployment (§6).

We believe that our methodology also creates an opportunity to evaluate the extent to which spatio-temporal experimental bias affects security domains other than Android malware, and we encourage the security community to embrace its underpinning philosophy.

Use of the term "bias": We use *(experimental) bias* to refer to the details of an experimental setting that depart from the conditions in a real-world deployment and can have a misleading impact (*bias*) on evaluations. We do not intend it to relate to the classifier bias/variance trade-off [8] from traditional machine learning terminology.

2 Android Malware Classification

We focus on Android malware classification. In §2.1 we introduce the reference approaches evaluated, in §2.2 we discuss the domain-specific prevalence of malware, and in §2.3 we introduce the dataset used throughout the paper.

2.1 Reference Algorithms

To assess experimental bias (§3), we consider two high-profile machine learning-driven techniques for Android malware classification, both published recently in top-tier security conferences. The first approach is **ALG1** [4], a linear support vector machine (SVM) on high-dimensional binary feature vectors engineered with a lightweight static analysis. The second approach is **ALG2** [33], a Random Forest (RF) applied to features engineered by modeling caller-callee relationships over Android API methods as Markov chains. We choose ALG1 and ALG2 as they build on different types of static analysis to generate feature spaces capturing Android application characteristics at different levels of abstraction; furthermore, they use different machine learning algorithms to learn decision boundaries between benign and malicious Android apps in the given feature space. Thus, they represent a broad design space and support the generality of our methodology for characterizing experimental bias. For a sound experimental baseline, we reimplemented ALG1 following the detailed description in the paper; for ALG2, we relied on the implementation provided by its authors. We replicated the baseline results for both approaches. After identifying and quantifying the impact of experimental bias (§3), we propose specific constraints and metrics to allow fair and unbiased comparisons (§4). Since ALG1 and ALG2 adopt traditional ML algorithms, in §4 we also consider **DL** [22], a deep learning-based approach that takes as input the same features as ALG1 [4]. We include DL because the latent feature space of deep learning approaches can capture different representations of the input data [19], which may affect their robustness to time decay. We replicate the baseline results for DL reported in [22] by re-implementing its neural network architecture and by using the same input features as for ALG1.

It speaks to the scientific standards of these papers that we were able to replicate the experiments; indeed, we would like to emphasize that we do not criticize them specifically. We use these approaches for our evaluation because they are available and offer stable baselines.

We report details on the hyperparameters of the reimplemented algorithms in §A.1.

2.2 Estimating in-the-wild Malware Ratio

The proportion of malware in the dataset can greatly affect the performance of the classifier (§3). Hence, unbiased experi-

ments require a dataset with a realistic percentage of malware over goodware; on an already existing dataset, one may enforce such a ratio by, for instance, downsampling the majority class (§3.3). Each malware domain has its own, often unique, ratio of malware to goodware typically encountered in the wild. First, it is important to know if malware is a minority, majority, or an equal-size class as goodware. For example, malware is the minority class in network traffic [5] and Android [30], but it is the majority class in binary download events [41]. On the one hand, the estimation of the percentage of malware in the wild for a given domain is a non-trivial task. On the other hand, measurement papers, industry telemetry, and publicly-available reports may all be leveraged to obtain realistic estimates.

In the Android landscape, malware represents 6%–18.8% of all the apps, according to different sources: a key industrial player[1] reported the ratio as approximately 6%, whereas the AndRadar measurement study [30] reports around 8% of Android malware in the wild. The 2017 Google's Android security report [21] suggests 6–10% malware, whereas an analysis of the metadata of the AndroZoo dataset [3] totaling almost 8M Android apps updated regularly, reveals an incidence of 18.8% of malicious apps. The data suggests that, in the Android domain, malware is the minority class. In this work, we decide to stabilize its percentage to 10% (a de-facto average across the various estimates), with per-month values between 8% and 12%. Settling on an average overall ratio of 10% Android malware also allows us to collect a dataset with a statistically sound number of per-month malware. An aggressive undersampling would have decreased the statistical significance of the dataset, whereas oversampling goodware would have been too resource intensive (§2.3).

2.3 Dataset

We consider samples from the public AndroZoo [3] dataset, consisting of more than 8.5 million Android apps between 2010 and early 2019: each app is associated with a timestamp, and most apps include VirusTotal metadata results. The dataset is constantly updated by crawling from different markets (e.g., more than 4 million apps from Google Play Store, and the remaining from markets such as Anzhi and AppChina). We choose to refer to this dataset due to its size and timespan, which allow us to perform realistic space- and time-aware experiments.

Goodware and malware. AndroZoo's metadata reports the number p of positive anti-virus reports on VirusTotal [20] for applications in the AndroZoo dataset. We chose $p = 0$ for goodware and $p \geq 4$ for malware, following Miller et al.'s [36] advice for a reliable ground-truth. About 13% of AndroZoo apps can be called grayware as they have $0 < p < 4$. We exclude grayware from the sampling as including it as either goodware or malware could disadvantage classifiers whose features were designed with a different labeling threshold.

[1] Information obtained through confidential emails with the authors.

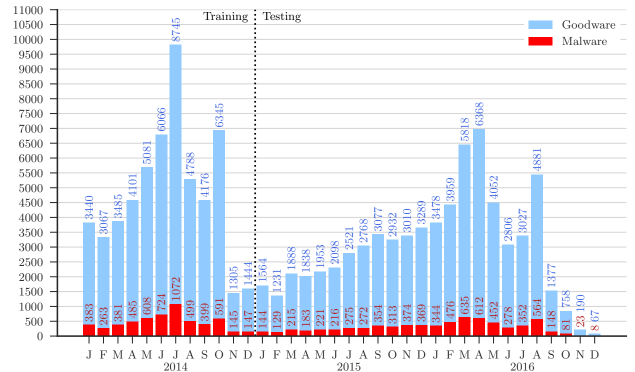


Figure 1: *Details of the dataset considered throughout this paper*. The figure reports a stack histogram with the monthly distribution of apps we collect from AndroZoo: $129,728$ Android applications (with average 10% malware), spanning from Jan 2014 to Dec 2016. The vertical dotted line denotes the split we use in all time-aware experiments in this paper (see §4 and §5): training on 2014, testing on 2015 and 2016.

Choosing apps. The number of objects we consider in our study is affected by the feature extraction cost, and partly by storage space requirements (as the full AndroZoo dataset, at the time of writing, is more than 50TB of apps to which one must add the space required for extracting features). Extracting features for the whole AndroZoo dataset may take up to three years on our research infrastructure (three high-spec Dell PowerEdge R730 nodes, each with 2 x 14 cores in hyperthreading—in total, 168 vCPU threads, 1.2TB of RAM, and a 100TB NAS), thus we decided to extract features from 129K apps (§2.2). We believe this represents a large dataset with enough statistical significance. To evaluate time decay, we decide on a granularity of one month, and we uniformly sample 129K AndroZoo apps in the period from Jan 2014 to Dec 2016, but also enforce an overall average of 10% malware (see §2.2)–with an allowed percentage of malware per month between 8% and 12%, to ensure some variability. Spanning over three years ensures 1,000+ apps per month (except for the last three months, where AndroZoo had crawled less applications). We consider apps up to Dec 2016 because the VirusTotal results for 2017 and 2018 apps were mostly unavailable from AndroZoo at the time of writing; moreover, Miller et al. [36] empirically evaluated that antivirus detections become stable after approximately one year—choosing Dec 2016 as the finishing time ensures good ground-truth confidence in objects labeled as malware.

Dataset summary. The final dataset consists of 129,728 Android applications (116,993 goodware and 12,735 malware). Figure 1 reports a stack histogram showing the per-month distribution of goodware/malware in the dataset. For the sake of clarity, the figure also reports the number of malware and goodware in each bin. The training and testing splits

used in §3 are reported in Table 1; all the time-aware experiments in the remainder of this paper are performed by training on 2014 and testing on 2015 and 2016 (see the vertical dotted line in Figure 1).

3 Sources of Experimental Bias

In this section, we motivate our discussion of bias through experimentation with ALG1 [4] and ALG2 [33] (§3.1). We then detail the sources of temporal (§3.2) and spatial bias (§3.3) that affect ML-based Android malware classification.

3.1 Motivational Example

We consider a motivational example in which we vary the sources of experimental bias to better illustrate the problem. Table 1 reports the F_1-score for ALG1 and ALG2 under various experimental configurations; rows correspond to different sources of temporal experimental bias, and columns correspond to different sources of spatial experimental bias. On the left-part of Table 1, we use squares (■/■) to show from which time frame training and testing objects are taken; each square represents six months (in the window from Jan 2014 to Dec 2016). Black squares (■) denote that samples are taken from that six-month time frame, whereas periods with gray squares (■) are not used. The columns on the right part of the table correspond to different percentages of malware in the training set Tr and the testing set Ts.

Table 1 shows that both ALG1 and ALG2 perform far worse in realistic settings (bold values with green background in the last row, for columns corresponding to 10% malware in testing) than in settings similar to those presented in [4, 33] (bold values with red background). This is due to inadvertent experimental bias as outlined in the following.

Note. We clarify to which similar settings of [4, 33] we refer to in the cells with red background in Table 1. The paper of ALG2 [33] reports in the abstract performance "up to 99% F_1", which (out of the many settings they evaluate) corresponds to a scenario with 86% malware in both training and testing, evaluated with 10-fold CV; here, we rounded off to 90% malware for a cleaner presentation (we have experimentally verified that results with 86% and 90% malware-to-benign class ratio are similar). ALG1's original paper [4] relies on hold-out by performing 10 random splits (66% training and 33% testing). Since hold-out is almost equivalent to k-fold CV and suffers from the same spatio-temporal biases, for the sake of simplicity in this section we refer to a k-fold CV setting for both ALG1 and ALG2.

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.) [26]. In the ML community, this problem is also known as *dataset shift* [50]. *Time decay* is the decrease in model performance over time caused by concept drift.

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 [2,36,37]. 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 [11,12,27,31,34,37,41,49,51,57]; while a useful mechanism to prevent overfitting [8] 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. Here, in the first row of Table 1, we quantify the performance impact in the Android domain.

The second row of Table 1 reports an experiment in which a classifier's ability to detect past objects is evaluated [2, 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; the community should focus on building malware classifiers that are robust against time decay.

In the third row, we identify a novel temporal bias that occurs when goodware and malware correspond to different time periods, often due to having originated from different data sources (e.g., in [33]). The black and gray squares in Table 1 show that, although malware testing objects are posterior to malware training objects, the goodware/malware time windows do not overlap; in this case, the classifier may learn to distinguish applications from different time periods, rather than goodware from malware—again leading to artificially high performance. For instance, spurious features such as new API methods may be able to strongly distinguish objects simply because malicious applications predate that API.

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. All experiments in this section assume temporal consistency. The model is trained on 2014 and tested on 2015 and 2016 (last row of Table 1) to allow the analysis of spatial

	Sample dates		% mw in testing set Ts							
			10% (*realistic*)				90% (*unrealistic*)			
			% mw in training set Tr				% mw in training set Tr			
			10%	90%	10%	90%	10%	90%	10%	90%
Experimental setting	Training	Testing	ALG1 [4]		ALG2 [33]		ALG1 [4]		ALG2 [33]	
10-fold CV	gw: ■■■■■■ mw: ■■■■■■	gw: ■■■■■■ mw: ■■■■■■	**0.91**	0.56	0.83	0.32	0.94	0.98	0.85	**0.97**
Temporally inconsistent	gw: □□□□■■ mw: □□□□■■	gw: ■■■■□□ mw: ■■■■□□	0.76	0.42	0.49	0.21	0.86	0.93	0.54	**0.95**
Temporally inconsistent gw/mw windows	gw: □□□□■□ mw: ■■□□□□	gw: □□□□□■ mw: □□■■□□	0.77	0.70	0.65	0.56	0.79	0.94	0.65	**0.93**
Temporally consistent (*realistic*)	gw: ■■□□□□ mw: ■■□□□□	gw: □□■■■■ mw: □□■■■■	**0.58**	**0.45**	**0.32**	**0.30**	0.62	0.94	0.33	0.96

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 ALG1 [4] and ALG2 [33]; values with green background (last row) are results in the realistic settings we identify. The dataset consists of three years (§2.3), and each square on the left part of the table represents a six month time-frame: if training (resp. testing) objects are sampled from that time frame, we use a black square (■); if not, we use a gray square (□).

bias without the interference 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 2 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 downsampling goodware, so the number of malware remains fixed throughout the experiments.[2] For completeness, we report the two training settings from Table 1 with 10% and 90% malware, respectively.

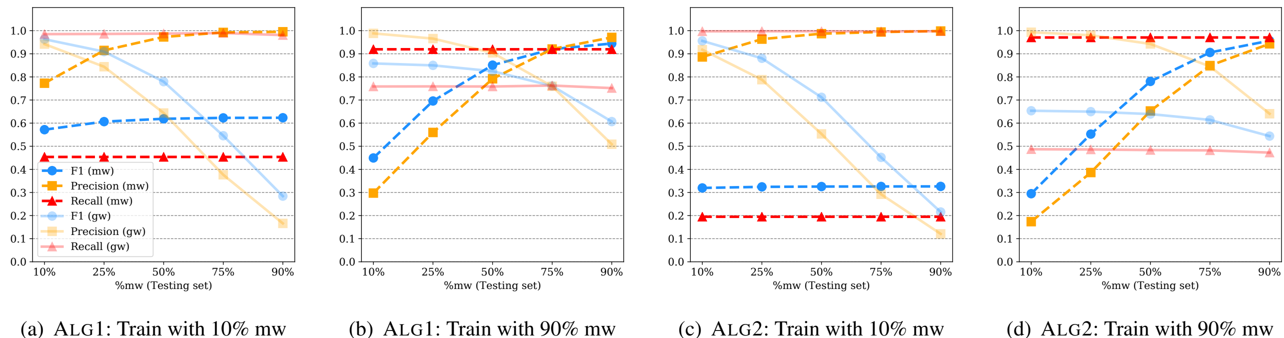
Let us first focus on the malware performance (dashed lines). All plots in Figure 2 exhibit constant Recall, and increasing Precision for increasing percentage of malware in the testing. Precision for the malware (mw) class—the positive class—is defined as $P_{mw} = TP/(TP+FP)$ and Recall as $R_{mw} = TP/(TP+FN)$. 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 solid lines in Figure 2), 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 the malware classifier.

Spatial experimental bias in training. To understand the impact of altering malware-to-goodware ratios in training, we now consider a motivating example with a linear SVM in a 2D feature space, with features x_1 and x_2. Figure 3 reports three scenarios, all with the same 10% malware in testing, but with 10%, 50%, and 90% malware in training.

We can observe that with an increasing percentage of malware in training, the hyperplane moves towards goodware. More formally, it improves Recall of malware while reducing its Precision. The opposite is true for goodware. To minimize the overall error rate $Err = (FP+FN)/(TP+TN+FP+FN)$ (i.e., maximize Accuracy), one should train the dataset with the same distribution that is expected in the testing. However, in this scenario one may have more interest in finding objects of the minority class (e.g., "more malware") by improving Recall subject to a constraint on maximum FPR.

Figure 4 shows the performance for ALG1 and ALG2, for increasing percentages of malware in training on the X-axis; just for completeness (since one cannot artificially change the test distribution to achieve realistic evaluations), we report results both for 10% mw in testing and for 90% malware in testing, but we remark that in the Android setting we have estimated 10% mw in the wild (§2.2). These plots confirm the trend in our motivating example (Figure 3), that is, R_{mw} increases but P_{mw} decreases. For the plots with 10% mw in

[2] We choose to downsample goodware 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. This does not alter the conclusions of our analysis. Let us assume a scenario in which we keep the same number of goodware (gw), and increase the percentage of mw in the dataset by oversampling mw. 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 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.



(a) ALG1: Train with 10% mw (b) ALG1: Train with 90% mw (c) ALG2: Train with 10% mw (d) ALG2: Train with 90% mw

Figure 2: *Spatial experimental bias in testing*. Training on 2014 and testing on 2015 and 2016. For increasing % of malware in the testing (unrealistic setting), Precision for malware increases and Recall remains the same; overall, F_1-Score increases for increasing percentage of malware in the testing. However, having more malware than goodware in testing does not reflect the in-the-wild distribution of 10% malware (§2.2), so the setting with more malware is unrealistic and reports biased results.

testing, we observe there is a point in which F_1-Score$_{mw}$ is maximum while the error for the gw class is within 5%.

In §4.3, we propose a novel algorithm to improve the performance of the malware class according to the objective of the user (high Precision, Recall or F_1-Score), subject to a maximum tolerated error. Moreover, in §4 we introduce constraints and metrics to guarantee bias-free evaluations, while revealing counter-intuitive results.

4 Space-Time Aware Evaluation

We now formalize how to perform an evaluation of an Android malware classifier free from spatio-temporal bias. We define a novel set of constraints that must be followed for realistic evaluations (§4.1); we introduce a novel time-aware metric, AUT, that captures in one number the impact of time decay on a classifier (§4.2); we propose a novel tuning algorithm that empirically optimizes a classifier performance, subject to a maximum tolerated error (§4.3); finally, we introduce TESSERACT and provide counter-intuitive results through unbiased evaluations (§4.4). To improve readability, we report in Appendix A.2 a table with all the major symbols used in the remainder of this paper.

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., Android app) 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 his 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. This parameter is chosen by the user, but it is important that the chosen granularity 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, 36], 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:

$$time(s_i) < time(s_j), \forall s_i \in Tr, \forall s_j \in Ts \qquad (1)$$

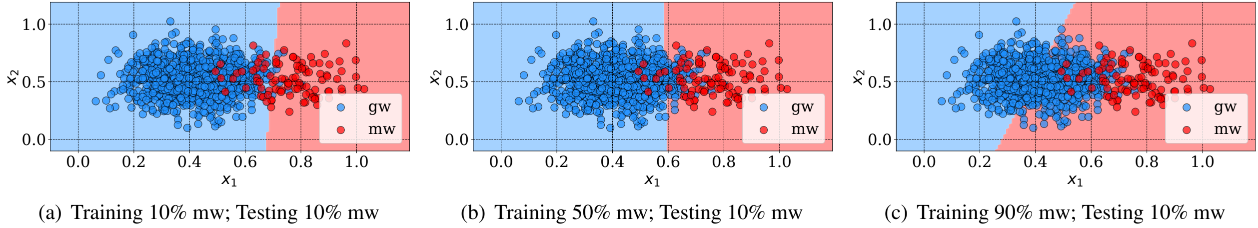
where s_i (resp. s_j) is an object in the training set Tr (resp. testing set Ts). Eq. 1 must hold; its violation inflates the results by including future knowledge in the classifier (§3.2).

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

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

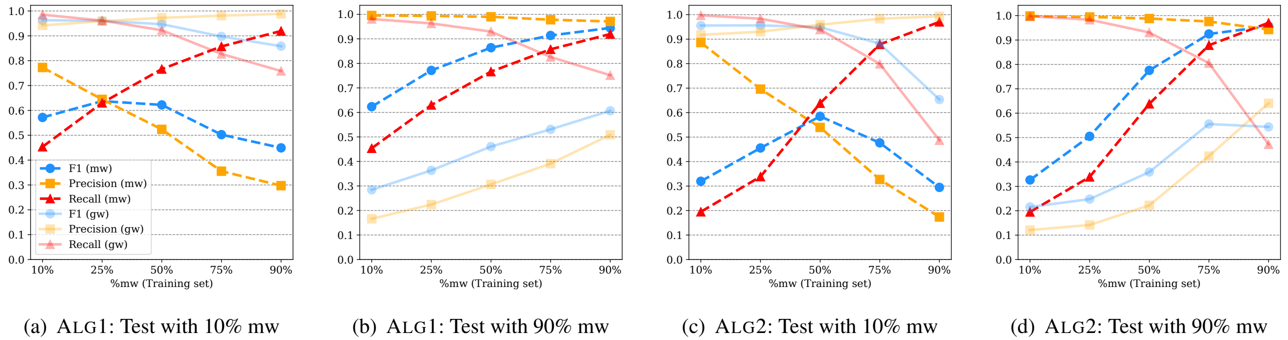
where $t_i^{min} = \min_k time(s_k)$ and $t_i^{max} = \max_k time(s_k)$. The same should hold for the training: although violating Eq. 2 in the training data does not bias the evaluation, it may affect the sensitivity of the classifier to unrelated artifacts. Eq. 2 has been violated in the past when goodware and malware have been collected from different time windows (e.g., ALG2 [33], re-evaluated in §4.4)—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



(a) Training 10% mw; Testing 10% mw (b) Training 50% mw; Testing 10% mw (c) Training 90% mw; Testing 10% mw

Figure 3: 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 % of malware in the training increases, the decision boundary moves towards the goodware class, improving Recall for malware but decreasing Precision.



(a) ALG1: Test with 10% mw (b) ALG1: Test with 90% mw (c) ALG2: Test with 10% mw (d) ALG2: Test with 90% mw

Figure 4: *Spatial experimental bias in training.* Training on 2014 and testing on 2015 and 2016. For increasing % of malware in the training, Precision decreases and Recall increases, for the motivations illustrated in the example of Figure 3. In §4.3, we devise an algorithm to find the best training configuration for optimizing Precision, Recall, or F_1 (depending on user needs).

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 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 goodware is predominant over malware, with $\hat{\sigma} \approx 0.10$ (§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 (e.g., in §4.4), 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 [8]), AUT is defined as follows:

$$AUT(f, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{[f(x_{k+1}) + f(x_k)]}{2} \tag{4}$$

where: $f(x_k)$ is the value of the point estimate of the performance metric f (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, $\text{AUT}(F_1, 12m)$ 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**(f,N) is a metric that allows us to evaluate performance f 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, 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 [14, 25]. Choosing different thresholds on ROC curves *shifts* the decision boundary, but (as seen in the motivating example of Figure 3) 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 [55]; 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 subsampling* 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:

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

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

- if $\mathbb{P} = F_1 \rightarrow E = 1 - Acc = {}^{(FP+FN)}/_{(TP+TN+FP+FN)}$
- if $\mathbb{P} = Rec \rightarrow E = FPR = {}^{FP}/_{(TN+FP)}$
- if $\mathbb{P} = Pr \rightarrow E = FNR = {}^{FN}/_{(TP+FN)}$

Each of these definitions of E is targeted to limit the error induced by the specific performance—if we want to maximize F_1 for the malware class, we need to limit both FPs and FNs; if $\mathbb{P} = Pr$, we increase FNs, so we constrain FNR.

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 is split 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 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 downsampling goodware, we prioritize the most uncertain points (e.g., points close to the decision boundary in an SVM) [46]. The constraint on line 6 ($\hat{\sigma} \leq \varphi \leq 0.5$) is to ensure that one does not under-represent the minority class (if $\varphi < \hat{\sigma}$) and that one does not let it become the majority class (if $\varphi > 0.5$); also, from §3.3 it is clear that if $\varphi > 0.5$, then the error rate becomes too high for the goodware class. 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 ALG1 [4], ALG2 [33], and DL [22] (§2.1), and their robustness to time decay.

We develop TESSERACT as an open-source Python framework that enforces constraints C1, C2, and C3 (§4.1), computes AUT (§4.2), and can train a classifier with our tuning algorithm (§4.3). TESSERACT operates as a traditional Python ML library but, in addition to features matrix X and labels y, it also takes as input the timestamp array t containing dates for each object. Details about TESSERACT's implementation and generality are in §A.5.

Figure 5 reports several performance metrics of the three algorithms as point estimates over time. The X-axis reports the testing slots in months, whereas the Y-axis reports different scores between 0 and 1. The areas highlighted in blue correspond to the $\text{AUT}(F_1, 24m)$. The black dash-dotted horizontal lines represent the best F_1 from the original papers [4, 22, 33],

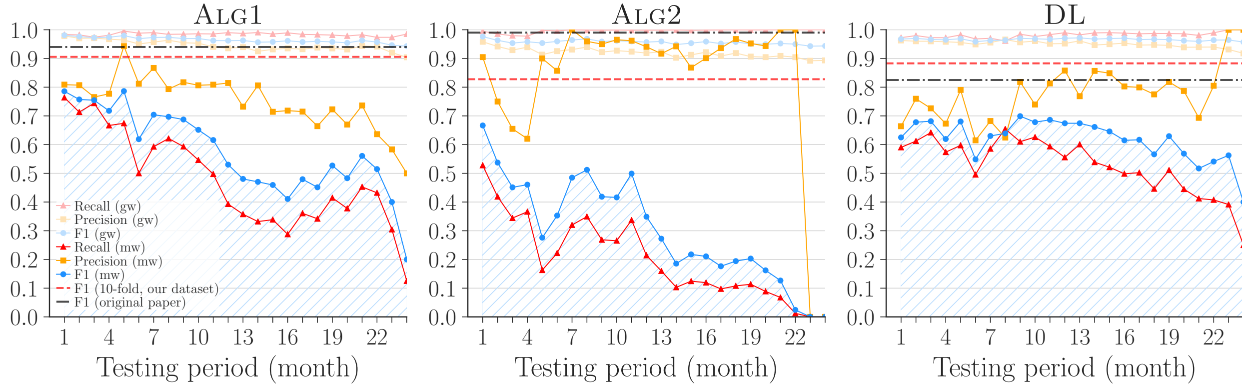


Figure 5: Time decay of ALG1 [4], ALG2 [33] and DL [22]—with AUT(F_1, 24m) of 0.58, 0.32 and 0.64, respectively. Training and test distribution both have 10% malware. The drop in the last 3 months is also related to lower samples in the dataset.

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. $P^* \leftarrow$ AUT($\mathbb{P}$,N) on Val_{array} with Θ
5. $\varphi^*_{\mathbb{P}} = \hat{\sigma}$
6. **for** ($\varphi = \hat{\sigma}$; $\varphi \leq 0.5$; $\varphi = \varphi + \mu$) **do**
7. Downsample gw in $ProperTr$ so that percentage of mw is φ
8. Train the classifier Θ_φ on $ProperTr$ with φ mw
9. performance $P_\varphi \leftarrow$ AUT($\mathbb{P}$, N) on Val_{array} with Θ_φ
10. error $E_\varphi \leftarrow$ Error rate on Val_{array} with Θ_φ
11. **if** ($P_\varphi > P^*$) **and** ($E_\varphi \leq E_{max}$) **then**
12. $P^* \leftarrow P_\varphi$
13. $\varphi^*_{\mathbb{P}} \leftarrow \varphi$
14. **return** $\varphi^*_{\mathbb{P}}$;

corresponding to results obtained with 10 hold-out random splits for ALG1, 10-fold CV for ALG2, and random split for DL; all 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.

Differences in 10-fold F_1. We discuss and motivate the differences between the horizontal lines representing original papers' best F_1 and replicated 10-fold F_1. The 10-fold F_1 of ALG1 is close to the original paper [4]; the difference is likely related to the use of a different, more recent dataset. The 10-fold F_1 of ALG2 is much lower than the one in the paper. We verified that this is mostly caused by **violating C3**: the best F_1 reported in [33] is on a setting with 86% malware—hence, spatial bias increases even 10-fold F_1 of ALG2. Also **violating C2** tends to inflate the 10-fold performance as the classifier may learn artifacts. The 10-fold F_1 in DL is instead slightly higher than in the original paper [22]; this is likely related to a hyperparameter tuning in the original paper that optimized Accuracy (instead of F_1), which is known to be misleading in imbalanced datasets. Details on hyperparameters chosen are in §A.1. From these results, we can observe that even if an analyst wants to estimate what the performance of the classifier would be in the *absence* of concept drift (i.e., where objects coming from the same distribution of the training dataset are received by the classifier), she still needs to enforce C2 and C3 while computing 10-fold CV to obtain valid results.

Violating C1 and C2. Removing the temporal bias reveals the real performance of each algorithm in the presence of concept drift. The AUT(F_1, 24m) quantifies such performance: 0.58 for ALG1, 0.32 for ALG2 and 0.64 for DL. In all three scenarios, the AUT(F_1, 24m) is lower than 10-fold F_1 as the latter violates constraint C1 and may violate C2 if the dataset classes are not evenly distributed across the timeline (§4).

Best performing algorithm. TESSERACT shows a counter-intuitive result: the algorithm that is most robust to time decay and has the highest performance over the 2 years testing is the DL algorithm (after removing space-time bias), although for the first few months ALG1 outperforms DL. Given this outcome, one may prefer to use ALG1 for the first few months and then DL, if retraining is not possible (§5). We observe that this strongly contradicts the performance obtained in the presence of temporal and spatial bias. In particular, if we only looked at the best F_1 reported in the original papers, ALG2 would have been the best algorithm (because spatial bias was present). After enforcing C3, the k-fold on our dataset would have suggested that DL and ALG1 have similar performance (because of temporal bias). After enforcing C1, C2 and C3, the AUT reveals that DL is actually the algorithm most robust to time decay.

Different robustness to time decay. Given a training dataset, the robustness of different ML models against perfor-

mance decay over time depends on several factors. Although more in-depth evaluations would be required to understand the theoretical motivations behind the different robustness to time decay of the three algorithms in our setting, we hereby provide insights on possible reasons. The performance of ALG2 is the fastest to decay likely because its feature engineering [33] may be capturing relations in the training data that quickly become obsolete at test time to separate goodware from malware. Although ALG1 and DL take as input the same feature space, the higher robustness to time decay of DL is likely related to feature representation in the *latent feature space* automatically identified by deep learning [19], which appears to be more robust to time decay in this specific setting. Recent results have also shown that linear SVM tends to overemphasize a few important features [35]—which are the few most effective on the training data, but may become obsolete over time. We remark that we are *not* claiming that deep learning is always more robust to time decay than traditional ML algorithms. Instead, we demonstrate how, in this specific setting, TESSERACT allowed us to highlight higher robustness of DL [22] against time decay; however, the prices to pay to use DL are lower explainability [23, 42] and higher training time [19].

Tuning algorithm. We now evaluate whether our tuning (Algorithm 1 in §4.3) improves robustness to time decay of a malware classifier for a given target performance. We first aim to maximize $\mathbb{P} = F_1$-Score of malware class, subject to E_{max}= 10%. After running Algorithm 1 on ALG1 [4], ALG2 [33] and DL, we find that $\varphi^*_{F_1} = 0.25$ for ALG1 and DL, and $\varphi^*_{F_1} = 0.5$ for ALG2. Figure 6 reports the improvement on the test performance of applying $\varphi^*_{F_1}$ to the full training set Tr of 1 year. 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 1 year of training data; this is to simulate a realistic deployment setting in which we have no a priori information about testing. Figure 6 shows that our approach for finding the best $\varphi^*_{F_1}$ improves the F_1-Score on malware at test time, at the cost of slightly reduced goodware performance. Table 2 shows details of how total FPs, total FNs, and AUT changed by training ALG1, ALG2, and DL 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. As expected (§3.3), Table 2 shows that when training with $\varphi^*_{F_1}$ Precision decreases (FPs increase) but Recall increases (because FNs decrease), and the overall AUT increases slightly as a trade-off. A similar reasoning follows for the other performance targets. We observe that the AUT for Precision may slightly differ even with a similar number of total FPs—this is because AUT$(Pr, 24m)$ is sensitive to the

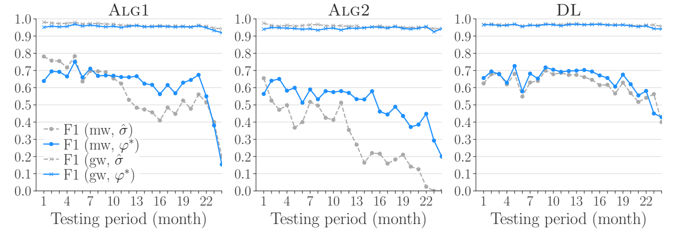


Figure 6: Tuning improvement obtained by applying $\varphi^*_{F_1} = 25\%$ to ALG1 and DL, and $\varphi^*_{F_1} = 50\%$ to ALG2. The values of $\varphi^*_{F_1}$ are obtained with Algorithm 1 and one year of training data (trained on 8 months and validated on 4 months).

Algorithm	φ	FP	FN	AUT($\mathbb{P}$,24m)		
				F_1	Pr	Rec
ALG1 [4]	10% ($\hat{\sigma}$)	965	3,851	0.58	0.75	0.48
	25% ($\varphi^*_{F_1}$)	2,156	2,815	0.62	0.65	0.61
	10% (φ^*_{Pr})	965	3,851	0.58	0.75	0.48
	50% (φ^*_{Rec})	3,728	1,793	0.64	0.58	0.74
ALG2 [33]	10% ($\hat{\sigma}$)	274	5,689	0.32	0.77	0.20
	50% ($\varphi^*_{F_1}$)	4,160	2,689	0.53	0.50	0.60
	10% (φ^*_{Pr})	274	5,689	0.32	0.77	0.20
	50% (φ^*_{Rec})	4,160	2,689	0.53	0.50	0.60
DL [22]	10% ($\hat{\sigma}$)	968	3,291	0.64	0.78	0.53
	25% ($\varphi^*_{F_1}$)	2,284	2,346	0.65	0.66	0.65
	10% (φ^*_{Pr})	968	3,291	0.64	0.78	0.53
	25% (φ^*_{Rec})	2,284	2,346	0.65	0.66	0.65

Table 2: Testing AUTs performance over 24 months when training with $\hat{\sigma}$, $\varphi^*_{F_1}$, φ^*_{Pr} and φ^*_{Rec}.

time at which FPs occur; the same observation is valid for total FNs and AUT Recall. After tuning, the F_1 performance of ALG1 and DL become similar, although DL remains higher in terms of AUT. The tuning improves the AUT$(F_1, 24m)$ of DL only marginally, as DL is already robust to time decay even before tuning (Figure 5).

The next section focuses on the two classifiers less robust to time decay, ALG1 and ALG2, to evaluate with TESSERACT the performance-cost trade-offs of budget-constrained strategies for delaying time decay.

5 Delaying Time Decay

We have shown how enforcing constraints and computing AUT with TESSERACT can reveal the real performance of Android malware classifiers (§4.4). This *baseline AUT performance* (without retraining) allows users to evaluate the general robustness of an algorithm to time decay. A classifier may be retrained to update its model. However, *manual labeling* is costly (especially in the Android malware setting), and the ML community [6, 46] has worked extensively on mitigation strategies—e.g., to identify a limited number of *best* objects to label (active learning). While effective at postponing time decay, strategies like these can further complicate the fair evaluation and comparison of classifiers.

In this section, we show how TESSERACT can be used

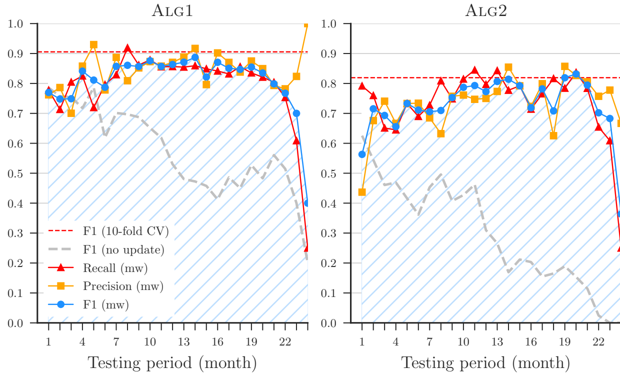


Figure 7: Delaying time decay: incremental retraining.

to compare and evaluate the trade-offs of different budget-constrained strategies to delay time decay. Since DL has shown to be more robust to time decay (§4.4) than ALG1 and ALG2, in this section we focus our attention these to show performance-cost trade-offs of different mitigations.

5.1 Delay Strategies

We do not propose novel delay strategies, but instead focus on how TESSERACT allows for the comparison of some popular approaches to mitigating time decay. This shows researchers how to adopt TESSERACT for the fair comparison of different approaches when proposing novel solutions to delaying time decay under budget constraints. We now summarize the delay strategies we consider and show results on our dataset. For interested readers, we include additional background knowledge on these strategies in §A.4.

Incremental retraining. We first consider an approach that represents an ideal upper bound on performance, where *all* points are included in retraining every month. This is likely unrealistic as it requires continuously labeling all the objects. Even assuming a reliance on VirusTotal, there is still an API usage cost associated with higher query rates and the approach may be ill-suited in other security domains. Figure 7 shows the performance of ALG1 and ALG2 with monthly incremental retraining.

Active learning. Active Learning (AL) strategies investigate how to select a subset of test objects (with unknown labels) that, if manually labeled and included in the training set, should be the most valuable for updating the classification model [46]. Here, we consider the most popular AL query strategy, *uncertainty sampling*, in which the points with the most uncertain predictions are selected for retraining, under the intuition that they are the most relevant to adjust decision boundaries. Figure 8 reports the active learning results obtained with uncertainty sampling, for different percentages of objects labeled per month. We observe that even with 1% AL, the performance already improves significantly.

Delay method	Costs				Performance			
	L		Q		$P:\mathrm{AUT}(F_1, 24m)$			
					$\varphi = \hat{\sigma}$		$\varphi = \varphi^*_{F_1}$	
	ALG1	ALG2	ALG1	ALG2	ALG1	ALG2	ALG1	ALG2
No update	0	0	0	0	0.577	0.317	0.622	0.527
Rejection ($\hat{\sigma}$)	0	0	10,283	3,595	0.717	0.280	–	–
Rejection ($\varphi^*_{F_1}$)	0	0	10,576	24,390	–	–	0.704	0.683
AL: 1%	709	709	0	0	0.708	0.456	0.703	0.589
AL: 2.5%	1,788	1,788	0	0	0.738	0.509	0.758	0.667
AL: 5%	3,589	3,589	0	0	0.782	0.615	0.784	0.680
AL: 7.5%	5,387	5,387	0	0	0.793	0.641	0.801	0.714
AL: 10%	7,189	7,189	0	0	0.796	0.656	0.802	0.732
AL: 25%	17,989	17,989	0	0	0.821	0.674	0.823	0.732
AL: 50%	35,988	35,988	0	0	0.817	0.679	0.828	0.741
Inc. retrain	71,988	71,988	0	0	0.818	0.679	0.830	0.736

Table 3: Performance-cost comparison of delay methods.

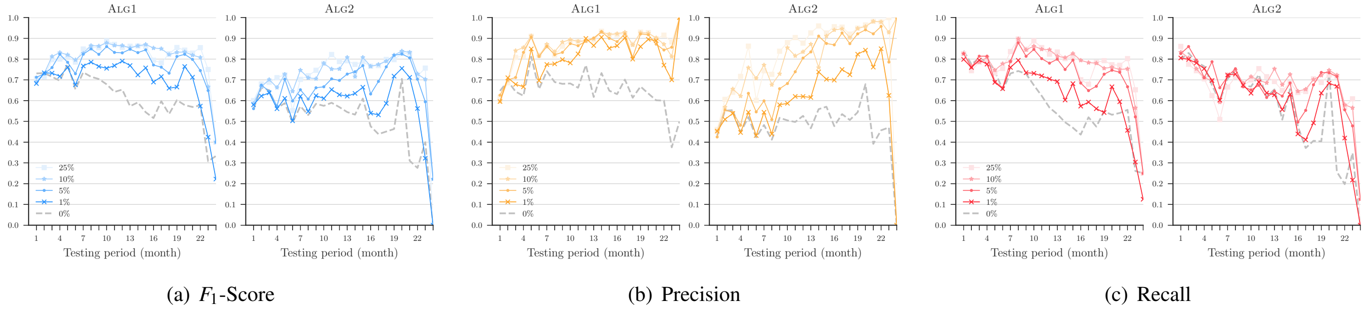
Classification with rejection. Another mitigation strategy involves rejecting a classifier's decision as "low confidence" and delaying the decision to a future date [6]. This isolates the rejected objects to a *quarantine* area which will later require manual inspection. Figure 9 reports the performance of ALG1 and ALG2 after applying a reject option based on [26]. In particular, we use the third quartile of probabilities of incorrect predictions as the rejection threshold [26]. The gray histograms in the background report the number of rejected objects per month. The second year of testing has more rejected objects for both ALG1 and ALG2, although ALG2 overall rejects more objects.

5.2 Analysis of Delay Methods

To quantify performance-cost trade-offs of methods to delay time decay without changing the algorithm, we characterize the following three elements: **Performance** (P), the performance measured in terms of AUT to capture robustness against time decay (§4.2); **Labeling Cost** (L), 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. [36] estimated that an average company could manually label 80 objects per day; **Quarantine Cost** (Q), 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 3, utilizing AUT(F_1,24m) while enforcing our constraints, reports a summary of labeling cost L, quarantine cost Q, and two performance columns P, corresponding to training with $\hat{\sigma}$ and $\varphi^*_{F_1}$ (§4.3), respectively. In each row, we highlight in purple cells (resp. orange) the column with the highest AUT for ALG2 (resp. ALG1). Table 3 allows us to: (i) examine the effectiveness of the training ratios $\varphi^*_{F_1}$ and $\hat{\sigma}$; (ii) analyze the AUT performance improvement and the corresponding costs for delaying time decay; (iii) compare the performance of ALG1 and ALG2 in different settings.

First, let us compare $\varphi^*_{F_1}$ with $\hat{\sigma}$. The first row of Table 3 represents the scenario in which the model is trained only once at the beginning—the scenario for which we originally



(a) F_1-Score (b) Precision (c) Recall

Figure 8: Delay time decay: performance with active learning based on uncertainty sampling.

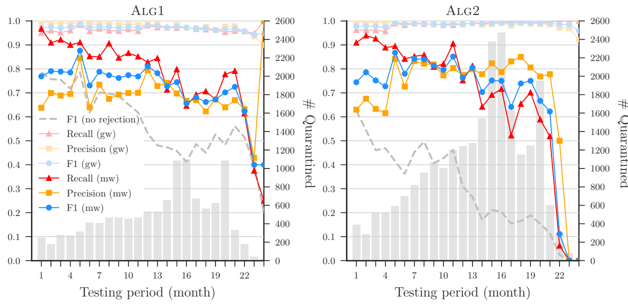


Figure 9: Delay time decay: classification with rejection.

designed Algorithm 1 (§4.3 and Figure 6). Without methods to delay time decay, $\varphi^*_{F_1}$ achieves better performance than $\hat{\sigma}$ for both ALG1 and ALG2 at no cost. In all other configurations, we observe that training $\varphi = \varphi^*_{F_1}$ always improves performance for ALG2, whereas for ALG1 it is slightly advantageous in most cases except for rejection and AL 1%—in general, the performance of ALG1 trained with $\varphi^*_{F_1}$ and $\hat{\sigma}$ is consistently close. The intuition for this outcome is that $\varphi^*_{F_1}$ and $\hat{\sigma}$ are also close for ALG1: when applying the AL strategy, we re-apply Algorithm 1 at each step and find that the average $\varphi^*_{F_1} \approx 15\%$ for ALG1, which is close to 10% (i.e., $\hat{\sigma}$). On the other hand, for ALG2 the average $\varphi^*_{F_1} \approx 50\%$, which is far from $\hat{\sigma}$ and improves all results significantly. We can conclude that our tuning algorithm is most effective when it finds a $\varphi^*_{\mathbb{P}}$ that differs from the estimated $\hat{\sigma}$.

Then, we analyze the performance improvement and related cost of using delay methods. The improvement in F_1-Score granted by our algorithm comes at no labeling or quarantine cost. We can observe that one can improve the in-the-wild performance of the algorithms at some cost L or Q. It is important to observe that objects discarded or to be labeled are not necessarily malware; they are just the objects most uncertain according to the algorithm, which the classifier may have likely misclassified. The labeling costs L for ALG1 and ALG2 are identical (same dataset); in AL, the percentage of retrained objects is user-specified and fixed.

Finally, Table 3 shows that ALG1 consistently outperforms ALG2 on F_1 for all performance-cost trade-offs. This confirms the trend seen in the realistic settings of Table 1.

This section shows that TESSERACT is helpful to both researchers and industrial practitioners. Practitioners need to estimate the performance of a classifier in the wild, compare different algorithms, and determine resources required for L and Q. For researchers, it is useful to understand how to reduce costs L and Q while improving classifiers performance P through comparable, unbiased evaluations. The problem is challenging, but we hope that releasing TESSERACT's code fosters further research and widespread adoption.

6 Discussion

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

Actionable points on TESSERACT. It is relevant to discuss how both researchers and practitioners can benefit from TESSERACT and our findings. A *baseline AUT performance* (without classifier retraining) allows users to evaluate the general robustness of an algorithm to performance decay (§4.2). We demonstrate how TESSERACT can reveal true performance and provide counter-intuitive results (§4.4). Robustness over extended time periods is practically relevant for deployment scenarios without the financial or computational resources to label and retrain often. Even with retraining strategies (§5), classifiers may not perform consistently over time. Manual labeling is costly, and the ML community has worked on mitigation strategies to identify a limited number of *best* objects to label (e.g., active learning [46]). TESSERACT takes care of removing spatio-temporal bias from evaluations, so that researchers can focus on the proposal of more robust algorithms (§5). In this context, TESSERACT allows for the creation of comparable baselines for algorithms in a time-aware setting. Moreover, TESSERACT can be used with different time granularity, provided each period has a significant number of samples. For example, if researchers are interested in increasing robustness to decay for the upcoming 3 months, they can use TESSERACT to produce bias-free comparisons of their approach with prior research, while considering time decay.

Generalization to other security domains. Although we used TESSERACT in the Android domain, our methodology

generalizes and can be immediately applied to any machine learning-driven security domain to achieve an evaluation without spatio-temporal bias. Our methodology, although general, requires some domain-specific parameters that reflect realistic conditions (e.g., time granularity Δ and test time length). This is not a weakness of our work, but rather an expected requirement. In general, it is reasonable to expect that spatio-temporal bias may afflict other security domains when affected by concept drift and i.i.d. does not hold—however, further experiments in other domains (e.g., Windows malware, code vulnerabilities) are required to make any scientific conclusion. TESSERACT can be used to understand the extent to which spatio-temporal bias affects such security domains; however, the ability to generalize requires access to large timestamped datasets, knowledge of realistic class ratios, and code or sufficient details to reproduce baselines.

Domain-specific in-the-wild malware percentage $\hat{\sigma}$. In the Android landscape, we assume that $\hat{\sigma}$ is around 10% (§2.2). Correctly estimating the malware percentage in the testing dataset is a challenging task and we encourage further representative measurement studies [30, 53] and data sharing to obtain realistic experimental settings.

Correct observation labels. We assume goodware and malware labels in the dataset are correct (§2.3). Miller et al. [36] 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 2016, we consider VirusTotal's labels as reliable. In the future, we may integrate approaches for *noisy oracles* [15], which assume some observations are mislabeled.

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 [20] reports for 25K apps[3] 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 [29]. We encourage the community to promptly notify dataset maintainers of any date inconsistencies. In the TESSERACT's project website (§8), we will maintain an updated list of timestamped datasets publicly available for the security community.

Choosing time granularity (Δ). Choosing the length of the time slots (i.e., time granularity) largely depends on the sparseness of the available dataset: in general, the granularity should be chosen to be as small as possible, while containing a statistically significant number of samples—as a rule of thumb, we keep the buckets large enough to have at least 1000 objects, which in our case leads to a monthly granularity. If there are restrictions on the number of time slots that can be considered (perhaps due to limited processing power), a coarser granularity can be used; however if the granularity becomes too large then the true trend might not be captured.

Resilience of malware classifiers. In our study, we analyze three recent high-profile classifiers. 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 [7]. 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.

7 Related Work

A common experimental bias in security is the *base rate fallacy* [5], which states that in highly-imbalanced datasets (e.g., network intrusion detection, where most traffic is benign), TPR and FPR are misleading performance metrics, because even FPR = 1% may correspond to *millions* of FPs and only *thousands* of TPs. In contrast, our work identifies experimental settings that are misleading *regardless* of the adopted metrics, and that remain incorrect even if the right metrics are used (§4.4). Sommer and Paxson [47] discuss challenges and guidelines in ML-based intrusion detection; Rossow et al. [44] discuss best practices for conducting malware experiments; van der Kouwe et al. [54] identify 22 common errors in system security evaluations. While helpful, these works [44, 47, 54] do not identify temporal and spatial bias, do not focus on Android, and do not *quantify* the impact of errors on classifiers performance, and their guidelines would not prevent all sources of temporal and spatial bias we identify. To be precise, Rossow et al. [44] evaluate the percentage of objects—in previously adopted datasets—that are "incorrect" (e.g., goodware labeled as malware, malfunctioning malware),

[3] We downloaded only 25K VT reports (corresponding to about 20% of our dataset) due to restrictions on our VirusTotal API usage quota.

but without evaluating impact on classifier performance. Zhou et al. [58] have recently shown that Hardware Performance Counters (HPCs) are not really effective for malware classification; while interesting and in line with the spirit of our work, their focus is very narrow, and they rely on 10-fold CV in the evaluation.

Allix et al. [2] broke new ground by evaluating malware classifiers in relation to time and showing how future knowledge can inflate performance, but do not propose any solution for comparable evaluations and only identify C1. As a separate issue, Allix et al. [1] investigated the difference between in-the-lab and in-the-wild scenarios and found that the greater presence of goodware leads to lower performance. We systematically analyze and explain these issues and help address them by formalizing a set of constraints (jointly considering the impact of temporal and spatial bias), introducing AUT as a unified performance metric for fair time-aware comparisons of different solutions, and offering a tuning algorithm to leverage the effects of training data distribution. Miller et al. [36] identified *temporal sample consistency* (equivalent to our constraint C1), but not 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. [45] questioned the use of recent or older malware as training objects and the performance degradation in testing real-world object ratios; however, most experiments were designed without considering time, reducing the reliability of their conclusions. While past work highlighted some sources of experimental bias [1, 2, 36, 45], it also gave little consideration to classifiers' aims: different scenarios may have different goals (not necessarily maximizing F_1), hence in our work we show the effects of different training settings on performance goals and propose an algorithm to properly tune a classifier accordingly (§4.3).

Other works from the ML literature investigate imbalanced datasets and highlighted how training and testing ratios can influence the results of an algorithm [9, 25, 55]. However, not coming from the security domain, these studies [9, 25, 55] focus only on some aspects of spatial bias and do *not* consider temporal bias. Indeed, concept drift is less problematic in some applications (e.g., image and text classification) than in Android malware [26]. Fawcett [16] focuses on challenges in spam detection, one of which resembles spatial bias; no solution is provided, whereas we introduce C3 to this end and demonstrate how its violation inflates performance (§4.4). Torralba and Efros [52] discuss the problem of *dataset bias* in computer vision, distinct from our security setting where there are fewer benchmarks; moreover in images the negative class (e.g., "not cat") can grow arbitrarily, which is less likely in the malware context. Moreno-Torres et al. [38] systematize different *drifts*, and mention *sample-selection bias*; while this resembles spatial bias, they do not propose any solution/experiments for its impact on ML performance. Other related work underlines 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 [14, 24]). In this paper, we take imbalance into account, and we propose actionable constraints and metrics with tool support to evaluate performance decay of classifiers over time.

Summary. Several studies of bias exist and have motivated our research. However, none of them address the entire problem in the context of evolving data (where the i.i.d. assumption does not hold anymore). Constraint C1, introduced by Miller et al. [36], is by itself insufficient to eliminate bias. This is evident from the original evaluation in MAMADROID [33], which enforces only C1. The evaluation in §4.4 clarifies why our novel constraints C2 and C3 are fundamental, and shows how our AUT metric can effectively reveal the true performance of algorithms, providing counter-intuitive results.

8 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://s2lab.kcl.ac.uk/projects/tesseract/. We will also maintain an updated list of publicly available security-related datasets with timestamped objects.

9 Conclusions

We have identified novel temporal and spatial bias in the Android domain and proposed novel constraints, metrics and tuning to address such issues. We have built and released TESSERACT as an open-source tool that integrates our methods. We have shown 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. We are currently working on integrating a time-varying percentage of malware in our framework to model still more realistic scenarios, and on how to use the slope of the performance decay curve to better differentiate algorithms with similar AUT.

We envision that future work on Android malware classification will use TESSERACT to produce realistic, comparable and unbiased results. Moreover, we also encourage the security community to adopt TESSERACT to evaluate the impact of temporal and spatial bias in other security domains where concept drift still needs to be quantified.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Roya Ensafi, for their constructive feedback, which has improved the overall quality of this work. This research has been partially sponsored by the UK EP/L022710/1 and EP/P009301/1 EPSRC research grants.

References

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

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

[4] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

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

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

[7] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 2018.

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

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

[10] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

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

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

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

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

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

[16] Tom Fawcett. In vivo spam filtering: a challenge problem for kdd. *ACM SIGKDD Explorations Newsletter*, 2003.

[17] Giorgio Fumera, Ignazio Pillai, and Fabio Roli. Classification with reject option in text categorisation systems. In *Int. Conf. Image Analysis and Processing*. IEEE, 2003.

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

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

[20] Google. VirusTotal, 2004.

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

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

[23] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. LEMNA: Explaining Deep Learning based Security Applications. In *CCS*. ACM, 2018.

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

[25] Haibo He and Edwardo A Garcia. Learning From Imbalanced Data. *IEEE TKDE*, 2009.

[26] Roberto Jordaney, Kumar Sharad, Santanu Kumar Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting Concept Drift in Malware Classification Models. In *USENIX Security*, 2017.

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

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

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

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

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

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

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

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

[35] Marco Melis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. Explaining Black-box Android Malware Detection. *EUSIPCO*, 2018.

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

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

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

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

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

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

[42] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why Should I Trust You?: Explaining the Predictions of Any Classifier. In *KDD*. ACM, 2016.

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

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

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

[46] Burr Settles. Active Learning Literature Survey. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2012.

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

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

[49] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware. In *CODASPY*. ACM, 2017.

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

[51] Gil Tahan, Lior Rokach, and Yuval Shahar. Mal-id: Automatic malware detection using common segment analysis and meta-features. *JMLR*, 2012.

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

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

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

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

[56] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: Deep learning in android malware detection. In *SIGCOMM Computer Communication Review*. ACM, 2014.

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

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

A Appendix

A.1 Algorithm Hyperparameters

We hereby report the details of the hyperparameters used to replicate ALG1, ALG2 and DL.

We replicate the settings and experiments of ALG1 [4] (linear SVM with C=1) and ALG2 [33] (package mode and RF with 101 trees and max depth of 64) as described in the respective papers [4, 33], successfully reproducing the published results. Since on our dataset the ALG1 performance is slightly lower (around 0.91 10-fold F_1), we also reproduce the experiment on their same dataset [4], achieving their original performance of about 0.94 10-fold F_1. We have used SCIKIT-LEARN, with *sklearn.svm.LinearSVC* for ALG1 and *sklearn.ensemble.RandomForestClassifier* for ALG2.

We then follow the guidelines in [22] to re-implement DL with KERAS. The features given as initial input to the neural network are the same as ALG1. We replicated the best-performing neural network architecture of [22], by training with 10 epochs and batch size equal to 1,000. To perform the training optimization, we used the stochastic gradient descent class *keras.optimizers.SGD* with the following parameters: *lr*=0.1, *momentum*=0.0, *decay*=0.0, *nesterov*=False. Some low-level details of the hyperparameter optimization were missing from the original paper [22]; we managed to obtain slightly higher F_1 performance in 10-fold setting (§4.4) likely because they have performed hyperparameter optimization on the Accuracy metric [8]—which is misleading in imbalanced datasets [5] where one class is prevalent (goodware, in Android).

A.2 Symbol table

Table 4 is a legend of the main symbols used throughout this paper to improve readability.

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).
Δ	Size of the test time-slots for time-aware evaluations (e.g., months).
AUT(f,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 f (e.g., F_1-Score) and N is the number of time units considered (e.g., 24 months)
$\hat{\sigma}$	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.
$\mathbb{P}$	Performance target of the tuning algorithm in §4.3; it can be F_1-Score, Precision (Pr) or Recall (Rec).
$\varphi^*_{\mathbb{P}}$	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	Labeling cost.
Q	Quarantine cost.
P	Performance; depending on the context, it will refer to AUT with F_1 or Pr or Rec.

Table 4: Symbol table.

A.3 Cumulative Plots for Time Decay

Figure 10 shows the cumulative performance plot defined in §4.2. This is the cumulative version of Figure 5.

A.4 Delay Strategies

We discuss more background details on the mitigation strategies adopted in Section 5.

Incremental retraining. Incremental retraining is an approach that tends towards an "ideal" performance P^*: *all* test objects are periodically labeled manually, and the new knowledge introduced to the classifier via retraining. More formally, the performance of month m_i is determined from the predictions of a model Θ trained on: $Tr \cup \{m_0, m_1, ..., m_{i-1}\}$, where $\{m_0, m_1, ..., m_{i-1}\}$ are testing objects, which are manually labeled. The dashed gray line represents the F_1-Score *without* incremental retraining (i.e., stationary training). Although incremental retraining generally achieves optimal performance throughout the whole test period, it also incurs the highest labeling cost L.

Active learning. Active learning is a field of machine learning that studies *query strategies* to select a small number of testing points close to the decision boundaries, that, if included in the training set, are the most relevant for updating the classifier. For example, in a linear SVM the slope of the decision boundary greatly depends on the points that are closest to it, the *support vectors* [8]; all the points further from the SVM decision boundary are classified with higher confidence, hence have limited effect on the slope of the hyperplane.

We evaluate the impact of one of the most popular active learning strategies: *uncertainty sampling* [36,46]. This query strategy selects the most points the classifier is least certain about, and uses them for retraining; we apply it in a time-aware scenario, and choose a percentage of objects to retrain per month. The intuition is that the most uncertain elements are the ones that may be indicative of concept drift, and new, correct knowledge about them may better inform the decision boundaries. The number of objects to label depends on the user's available resources for labeling.

More formally, in binary classification uncertainty sampling gives a score x^*_{LC} (where LC stands for *Least Confident*) to each sample [46]; this score is defined as follows[4]:

$$x^*_{LC} := \text{argmax}_x \{1 - P_\Theta(\hat{y}|x)\} \quad (6)$$

where $\hat{y} := \text{argmax}_y P_\Theta(y|x)$ is the class label with the highest posterior probability according to classifier Θ. In a binary classification task, the maximum uncertainty for an object is achieved when its prediction probability equal to 0.5 for both classes (i.e., equal probability of being goodware or malware). The test objects are sorted by descending order of uncertainty x^*_{LC}, and the top-n most uncertain are selected to be labeled for retraining the classifier.

Depending on the percentage of manually labeled points, each scenario corresponds to a different labeling cost L. The labeling cost L is known a priori since it is user specified.

Classification with rejection. Malware evolves rapidly over time, so if the classifier is not up to date, the decision region may no longer be representative of new objects. Another approach, orthogonal to active learning, is to include a *reject option* as a possible classifier outcome [17,26]. This discards the most uncertain predictions to a *quarantine* area for manual inspection at a future date. At the cost of rejecting some objects, the overall performance of the classifier (on the remaining objects) increases. The intuition is that in this way only high confidence decisions are taken into account. Again, although performance P improves, there is a quarantine cost Q associated with it; in this case, unlike active learning, the cost is not known a priori because, in traditional classification with rejection, a threshold on the classifier confidence is applied [17,26].

[4] In multi-class classification, there is a query strategy based on the entropy of the prediction scores array; in binary classification, the entropy-based query strategy is proven to be equivalent to the "least confident" [46].

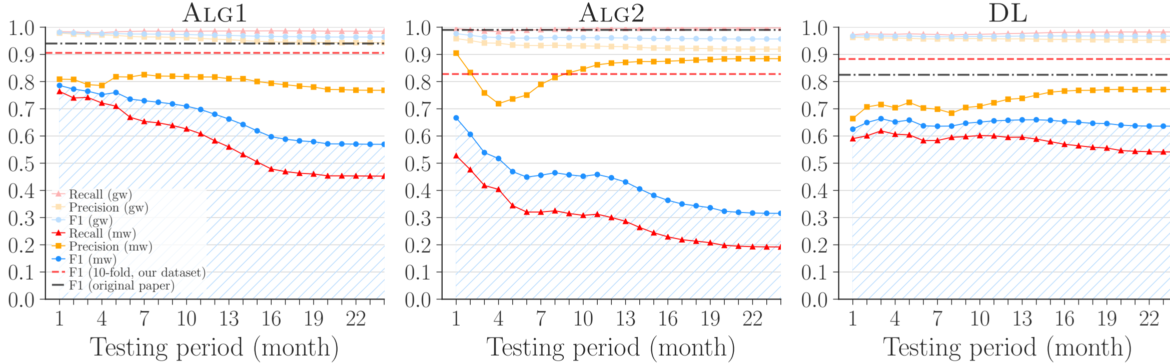


Figure 10: Performance time decay with cumulative estimate for ALG1, ALG2 and DL. Testing distribution has $\delta = 10\%$ malware, and training distribution has $\varphi = 10\%$ malware.

A.5 TESSERACT Implementation

We have implemented our constraints, metrics, and algorithms as a Python library named TESSERACT, designed to integrate easily with common workflows. In particular, the API design of TESSERACT is heavily inspired by and fully compatible with SCIKIT-LEARN [39] and KERAS [10]; as a result, many of the conventions and workflows in TESSERACT will be familiar to users of these libraries. Here we present an overview of the library's core modules while further details of the design can be found in [40].

temporal.py While machine learning libraries commonly involve working with a set of predictors `X` and a set of output variables `y`, TESSERACT extends this concept to include an array of `datetime` objects `t`. This allows for operations such as the time-aware partitioning of datasets (e.g., `time_aware_partition()` and `time_aware_train_test_split()`) while respecting temporal constraints C1 and C2.

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 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()` for computing the AUT for a given metric (F_1, Precision, AUC, 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 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 §5 and include them with our release. We hope this modular approach lowers the bar for follow-up research in these areas.

A.6 Summary of Datasets Evaluated by Prior Work

As a reference, Table 5 reports the composition of the dataset used in our paper (1st row) and of the datasets used for experimentally biased evaluations in prior work ALG1 [4], ALG2 [33], and DL [22] (2nd and 3rd row). In this paper, we always evaluate ALG1, ALG2 and DL with the dataset in the first row (more details in §2.3 and Figure 1), because we have built it to allow experiments without spatio-temporal bias by enforcing constraints C1, C2 and C3. Details on experimental settings that caused spatio-temporal bias in prior work are described in §3 and §4. *We never use the datasets of prior work [4, 22, 33] in our experiments.*

Work	Apps	Date Range	# Objects	Total	Violations
TESSERACT (this work)	Benign	Jan 2014 - Dec 2016	116,993	116,993	-
	Malicious		12,735	12,735	
[4], [22]	Benign	Aug 2010 - Oct 2012	123,453	123,453	C1
	Malicious		5,560	5,560	
[33]	Benign	Apr 2013 - Nov 2013	5,879	8,447	(C1) C2 C3
		Mar 2016	2,568		
	Malicious	Oct 2010 - Aug 2012	5,560	35,493	
		Jan 2013 - Jun 2013	6,228		
		Jun 2013 - Mar 2014	15,417		
		Jan 2015 - Jun 2015	5,314		
		Jan 2016 - May 2016	2,974		

Table 5: Summary of datasets composition used by our paper (1st row) and by prior work [4, 22, 33] (2nd and 3rd row).