Data Drift in Android Malware Detection

LUCA MINNEI[1], HICHAM EDDOUBI[1], ANGELO SOTGIU[1], MAURA PINTOR[1], AMBRA DEMONTIS[1], BATTISTA BIGGIO[1]

[1]University of Cagliari, Italy

Abstract:

Android malware detectors are now widely implemented with machine learning algorithms, trained on large datasets of goodware and malware applications gathered at a fixed moment in time. However, as recent work showed, this domain is not stationary, causing detectors to show degrading performance over time. While recent work pinpoints the presence of such drift, little has been done to isolate its causes and understand the underlying reasons. In this work, we show which features cause the data drift, i.e., new features to appear and old ones that become unreliable. Our experimental evaluation highlights that particular feature groups cause the data drift. However, we also show that removing these highly variable features from the feature set is insufficient to achieve good classification performance.

1 Introduction

As mobile technologies evolve quickly, Android has emerged as one of the most popular Operating Systems (OS) for mobile devices. However, this popularity has also made it a prime target for cyber attackers who exploit OS vulnerabilities to develop malicious applications, known as *malware*, which can compromise user data and security. To address the challenge of detecting malware in Android OS, numerous Machine Learning (ML) techniques have been proposed [1], showing impressive performances in detecting malware while allowing the use of benign applications (*goodware*). Nevertheless, real-world data are affected by concept drift, i.e., changes in the data over time. For this reason, ML models using features extracted from these data patterns might become unreliable in classifying new data. Namely, previously captured malware samples might evolve so that they become undetected by the ML models. Android applications are especially susceptible to this phenomenon, as attackers constantly update malware to evade detection. Additionally, the Android API framework frequently updates, introducing new functionalities - such as Application Programming Interfaces (APIs), permissions, etc. - and deprecating old ones. If ML models rely on information from these features, changes in the framework can make detecting the same patterns in older data difficult, further complicating malware detection. This ongoing evolution in the mobile applications landscape makes maintaining high malware detection accuracy over time a significant challenge. Recent work has highlighted how concept drift can worryingly affect the performance of state-of-the-art Android malware detectors over time [2]. However, no other work differentiates between the types of features used by ML models for malware detection, highlighting how some features are defined by the Android framework, such as API calls, which are relatively stable and standardized, and others are user-defined, i.e., they can be created dynamically in code, and can vary widely across different applications. User-defined features are inherently more volatile and subject to change, leading to weaker and less reliable patterns for ML models. However, these features can still contain relevant information. For instance, user-defined features related to network addresses can offer critical insights into the behavior and functionality of an application, as malware samples often communicate with external servers to exchange data. Furthermore, as new malware samples commonly reuse components from older versions, other user-defined features might also be informative.

Through this analysis, we hope to provide deeper insights into the dynamics of feature changes in Android malware detection and understand (i) which types of features change the most over time and (ii) whether and to what extent removing these features can affect classification performance. We first introduce the Android framework and discuss how ML-based techniques distinguish malware from goodware (Sect. 2). We then drive our analysis by inspecting the new features that appear over time in the dataset, showing how the user-defined categories are the ones that are most affected by this phenomenon (Sect. 3). Then, we provide evidence supporting our hypothesis and assess how removing these features affects the classifica-

tion performances, thus showcasing how the user-defined features, despite being highly volatile, are still required to separate the malware applications from the benign (Sect. 4). We then discuss related work and highlight our conclusions (Sect. 5).

2 Android Malware Detection

In this section, we provide essential background information on Android applications and learning-based techniques for detecting Android malware. In particular, we focus on the detecting techniques considered for this study. Furthermore, we describe the main issue of these approaches that we will address in this work.

Android OS Applications. The applications deployed on the Android Operating System are distributed and installed through Android Application Package (APK) files, which consist of an archive (with `.apk` extension) containing all the required application source code, resources, assets, and metadata. The main files included in an APK are an `AndroidManifest.xml` file that defines essential information needed by the OS to deploy the application like its name, version, required permissions, and main components; one or more `classes.dex` files, containing the Java or Kotlin source code of the application compiled to Dalvik bytecode; a `res` directory with additional `.xml` resource files, used to define elements of the user interface, constant strings, multimedia contents, etc.; a `lib` directory containing architecture-specific compiled code, such as native C/C++ libraries; an `assets` folder for generic files; a `META-INF` folder containing application metadata, including a certificate and a signature.

Machine Learning-based Android Malware Detection. The huge number of yearly released applications and malware samples demands high analysis and detection throughput. Classic signature-based approaches, which are widely applied as they are fast and easy to implement, are often evaded through several techniques (and their combinations) like obfuscation, app repackaging, dynamic code loading, encryption, and malware dropping. Additionally, they are ineffective against unseen threats and require constant updates. To address some of these limitations, ML has been used for Android malware detection based on features extracted with static [1,3] or dynamic analysis [4,5] or a combination thereof [6]. Although dynamic analysis may capture the application's behavior better and prevent some evasion techniques, static analysis remains a relevant detection mechanism in use thanks to its effectiveness and low computational requirements.

In this work, we use as a case study the popular Drebin malware detector [1], consisting of a linear Support Vector Machine (SVM) model trained on statically extracted binary features. The features are divided into 8 subsets:

- S_1: hardware components requested by the application;
- S_2: requested permissions;
- S_3: app components, i.e., activities, services, content providers, and broadcast receivers;
- S_4: filtered intents;
- S_5: restricted API calls;
- S_6: used permissions, i.e., the permissions that are requested to execute restricted API calls from S_5;
- S_7: suspicious API calls;
- S_8: network addresses.

S_1, S_2, S_3 and S_4 are extracted from the `AndroidManifest.xml`, while S_5, S_6, S_7 and S_8 are extracted from the `classes.dex` files. The feature extractor uses a vectorizer to represent the data through numerical vectors. The vectorizer V is a function that creates a set of features taken from $S_1, ..., S_8$, given a dataset. For a given dataset D, $V(D)$ represents the set of features extracted by the vectorizer. The feature set comprises all the d extracted features from the training data, and input samples are represented with a d-dimensional vector where each index position corresponds to a particular feature, and its value is 1 if the feature is present or 0 otherwise. Importantly, different datasets might produce different sets of features, thus creating vectors that are different in their elements.

Detectors performance over time. Despite the impressive performance reported for Drebin and other learning-based detectors, subsequent work has shown that their evaluations overlooked the temporal evolution that appears in both legitimate and malware applications. In fact, a new version of the Android OS is released approximately every year, bringing new functionalities and deprecating or removing other ones, resulting in changes in the previously defined features. In particular, some features might become obsolete and never reappear, and others might not be covered by the extracted features (thus, they do not impact the classification outcome).

In addition, even though the emergence rate of new malware families is relatively low, there is a significant proliferation of numerous variants within these families, especially for increasing the evasion capabilities with the aforementioned techniques. New families or variants might present patterns different from the seen malware training data and thus might not be represented by the distribution assumed by the ML model.

Both these phenomena can induce strong shifts in the data distribution, violating the Independent and Identical Distribution (I.I.D.) assumption between training and test data made by classical learning-based approaches. To assess the impact of the data drift, detectors should be tested on temporally-aware data rather than the usual random splits of the available dataset. That is, data should be split depending on the publication date of the applications (i.e., chronologically successive), using the first temporal split for the training phase and the remaining samples for the test. As shown in several works, the reported model's performance under temporal-aware evaluations presents a notable decay over time [2]. For this reason, it is necessary to design new techniques to make the classifiers more robust to data drifts or detect when classifiers become obsolete (thus rejecting unreliable decisions).

3 Selecting Stable Features

In this section, we characterize the variability of the feature sets generated with different datasets and relate them with the practical drifts appearing in the Android datasets.

First, we highlight that the 8 subsets of the Drebin feature set defined in Sect. 2 can be divided into two categories:

- *Framework-specific features.* These features are predefined in the Android OS framework and can be viewed as a closed set. This category includes Android OS API calls (S_5 and S_7), the permissions that might be associated with them (S_6), and hardware components (S_1). The features in this category might slightly change over time, depending on the updates affecting new Android OS versions.

- *User-defined features.* These features are based on user-defined content. Features in S_3 are the assigned names to the application's components (activities, services, receivers, and providers). In the following sections, we will separately consider each application component. Network addresses included in S_8 are constant strings in the application source code. Additionally, app developers can define custom permissions and intent filters, considered in S_2 and S_4, respectively. For our analysis, we include here S_2 and S_4 - although they also contain framework-specific features. The nature of features belonging to this category makes them highly variable over time.

We expect that the latter group changes more rapidly in the period considered for this study, as the Android framework undergoes release updates that happen yearly, while the rate of change for the other features can be much faster due to the user-defined nature of these.

To verify our hypothesis, we study how the features of the different families change from a dataset D_1 to another dataset D_2, which, in our context, is a dataset available later in time.

Thus, with two datasets we obtain with the vectorizer V two feature sets $F_1 = V(D_1)$, the features extracted from D_1, and $F_2 = V(D_2)$, the features extracted from D_2.

To analyze which features appear only in D_2 but not in D_1, we compare the two sets, i.e.,

$$\Delta F_{2,1} = F_2 \setminus F_1 \,, \tag{1}$$

where the subscript is to indicate the datasets being compared, and the operation is not commutative.

For example, given the two feature sets $F1 = \{f_a, f_b, f_c, f_d\}$, and $F2 = \{f_c, f_d, f_g\}$, we obtain that $\Delta F_{2,1} = \{f_d, f_g\}$, as in this example f_d and f_g are the new features that are only in D_2 and not in D_1. We then measure

$$\text{Diff}_{2,1} = \frac{|\Delta F_{2,1}|}{|F_2|} \,, \tag{2}$$

where the operator $|\cdot|$ measures the cardinality (size) of the sets. In other words, $\text{Diff}_{2,1}$ counts the percentage of features that appear only in D_2.

To select stable features, we split the data available at training time into two sets: the first, which we will call *training dataset* (D_{TR}), including the oldest samples, and the second, named in the following *validation set* (D_{VAL}), containing the remaining samples present in the training dataset. Then, we inspect the performance on the remaining dataset, which is composed of multiple *test sets* numbered in temporal order D_{TS1}, ... D_{TS9}.

4 Experiments

We now discuss our experimental setup and the findings gathered from the restults.

Dataset. The datasets are taken from the Android malware detection competition hosted in the ELSA Cybersecurity Use Case.[1] The applications are sampled from the AndroZoo [7] collection of Android Applications, which contains (at the time of writing) over 24 million samples collected from different sources. The sampling is performed based on analysis reports from VirusTotal, from which a timestamp (from the `first_submission_date` field) and a binary label are extracted. A negative label is assigned to samples that have no

[1] https://github.com/pralab/elsa-cybersecurity

TABLE 1. The considered data splits with their number of samples and temporal spanning. D_{TR} refers to the training set, D_{VAL} to the validation set, and D_{TSi} to the chronologically successive test sets.

Dataset	N. Samples	Sampling period	Quarter
D_{TR}	$75,000$	2017-01 - 2019-12	-
D_{VAL}	$6,250$	2020-01 - 2020-03	-
D_{TS1}	$6,250$	2020-04 - 2020-06	2020Q2
D_{TS2}	$6,250$	2020-07 - 2020-09	2020Q3
D_{TS3}	$6,250$	2020-09 - 2020-12	2020Q4
D_{TS4}	$6,250$	2021-01 - 2021-03	2021Q1
D_{TS5}	$6,250$	2021-04 - 2021-06	2021Q2
D_{TS6}	$6,250$	2021-07 - 2021-09	2021Q3
D_{TS7}	$6,250$	2021-09 - 2021-12	2021Q4
D_{TS8}	$8,820$	2022-01 - 2022-03	2022Q1
D_{TS9}	$3,680$	2022-04 - 2022-06	2022Q2

TABLE 2. Feature analysis. For each feature family, we show the percentage of features that differ between the validation and training dataset ($\text{Diff}_{val,tr}$) and the number of features belonging to that family.

Subset	Feature	$\text{Diff}_{val,tr}$ (%)	N. features
S_3	Activities	60.15	$13,370$
S_3	Services	53.58	$2,249$
S_3	Receivers	51.74	$1,351$
S_3	Providers	46.62	903
S_4	Intent Filters	46.19	$1,704$
S_8	URLs	33.74	$10,113$
S_2	Req. Perm.	32.72	703
S_5	API Calls	0.00	307
S_6	Used Perm.	0.00	59
S_1	Hardware comp.	0.00	32
S_7	Susp. Calls	0.00	25

detections from the VirusTotal antimalware engines, whereas a positive label is assigned to samples that are detected by at least 10 engines and can be uniquely assigned to a malware family by the *avclass* tool[2], effectively discarding any grayware application that has less than 10 detections or uncertain label. Moreover, a proportion of $9 : 1$ between legitimate and malware samples is kept, as suggested in previous works [2].

We rely on the provided training set and the test sets of the deployed "Track 3: Temporal Robustness to Data Drift", consisting of a total of $137,500$ samples, of which $123,750$ goodware and $13,750$ malware, sampled between January 2017 and June 2022.

Models. The experiments are conducted using a pre-trained Linear SVM model, taken from the ELSA competition, with Hinge Loss and C = 0.1.

Evaluation metrics. In order to evaluate the performance of the classification models, we employ the following metrics:

Precision. The precision measures the classifier's ability to classify as positive only the samples that are really positive. It is defined as the ratio of the true positives over the total positive predictions (true positives and false positives).

Recall. The recall, also known as True Positive Rate (TPR), measures the model's ability to identify all the positive samples. It is defined as the ratio of the true positives over the total number of positive samples (true positives and false negatives).

F1. The F1 score measures the harmonic mean of the precision and recall, which provides a metric capable of reliably computing the model's performance in the presence of unbalanced class distributions. It is defined as: $F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$.

Testing over time. To provide empirical support to our claims, we train the model on the dataset covering the first three years (D_{TR}). Then, we split the rest of the data into sequential sets spanning 3 months (quarters), keeping the first quarter as D_{VAL} and the successive ones as D_{TSi}. This allows us to visualize both the data drift across each test set and the model's performance in the presence of temporal drift. All the considered data splits and their temporal spanning are reported in Table 1.

Experimental Results. In the following, we present the experimental results. First, we discuss those of the analysis that allowed us to choose the more stable features. Then, we discuss the effect of this feature selection on the classifier performance.

Feature analysis. In Fig. 1, we show how the percentage of features different between the train and test datasets present in goodware and malware samples changes over time. We represent this difference with $\text{Diff}_{ts_i,tr}$, where i is an index that represents the test dataset. This figure clearly shows that the change in features over time is relevant. The goodware samples in the different considered test datasets present more different features with respect to the malware samples. This is probably because benign samples are constituted by a large variety of applications whereas malware is often created by applying simple modifications to existing malware to avoid signature-based detection. However, the features show a faster trend change over time in malware samples.

For each feature family, we show in Table 2 the percentage of features that differ between the validation and training dataset ($\text{Diff}_{val,tr}$) and the number of features belonging to that family. The features that present a more relevant change (i.e., are more unstable) are Activities, Services, Receivers, and Providers. Notably, as discussed in Sect. 3, they are all defined by the user, and in particular, all of them belong to the S_3 family (they are all names assigned to the application's components).

[2]https://github.com/malicialab/avclass

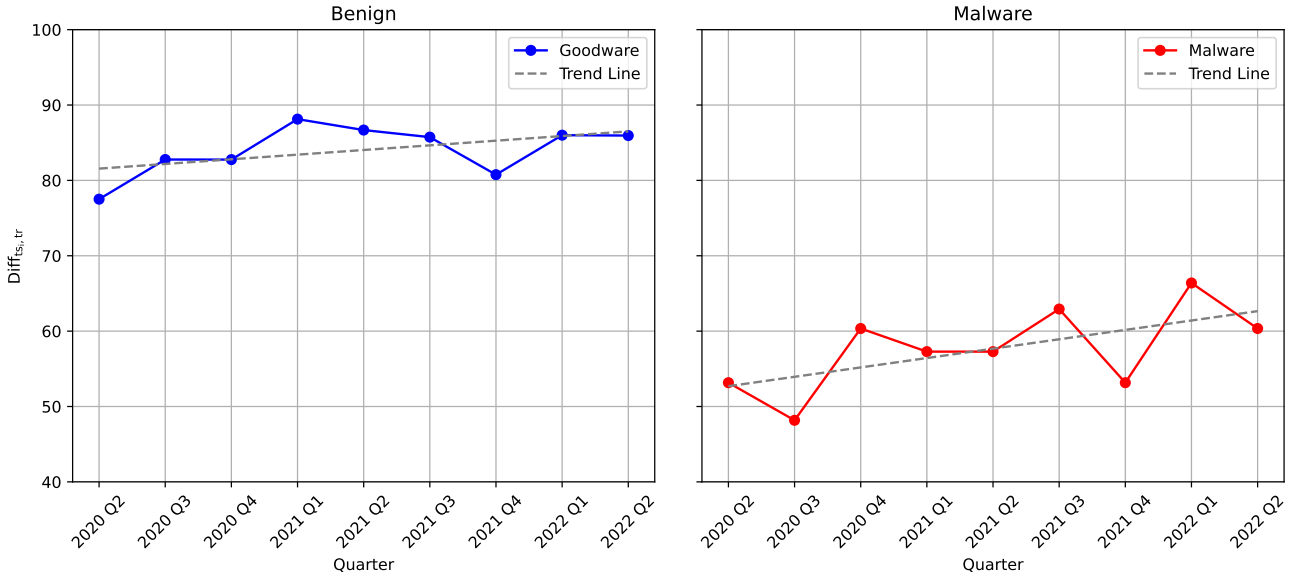

FIGURE 1. Percentage of features present in the test sets but not in the training set of the goodware (left) and malware (right) samples for each quarter. The gray dotted line shows the trend.

They can be thus easily changed when creating new malware.

Training on stable features. Once we identify the more unstable features, we try to understand if removing them may help the performance. In Fig. 2, we show the performance of the classifier trained on all the features and the classifier trained only on stable features. Whereas the reported Precision values are comparable, the latter's Recall and F1 score values are always lower along the entire temporal curve. We can conclude from it that using user-defined features positively influences the detector's performance. This can be explained by the previously presented findings, i.e., malware samples typically present small changes between their variants.

5 Conclusions

In this work, we focus on the problem of temporal drift in Android Malware. In designing security classifiers such as the Android malware detector considered in this work, it is important to consider the variability of the features as well as the capability of the attacker to easily modify those features. Highly-variable features, while being potentially informative, might pose several challenges: (a) highly-variable features can steer the classifier too much, leading to overfitting. These features might capture noise or transient patterns in the data, which can degrade the model's generalization performance (especially over time); and (b) highly-variable features in the considered scenario are also the easiest to modify by attackers. Malicious actors might exploit these features to manipulate the classifier's behavior, evading detection by introducing subtle yet significant changes to the input data without compromising the malicious behavior.

Therefore, it's often preferable to design more stable and robust features. Unfortunately, as shown in this work, training a classifier only on the more stable features over time is not an effective solution, as it substantially hinders classification performance. Instead, employing strategies that de-emphasize the importance of highly-variable features [8] may offer a more promising direction. In future work, we will compare the findings of this work with more complex approaches to counter the effects of temporal drift, such as [2, 8], investigating whether the features used are similar to the ones selected with the method presented in this work and whether they are more capable of withstanding drifts and attackers.

Acknowledgements

This work has been partly supported by the European Union's Horizon Europe research and innovation program under the project ELSA, grant agreement no. 101070617; by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU; by BMK, BMDW, and the Province of

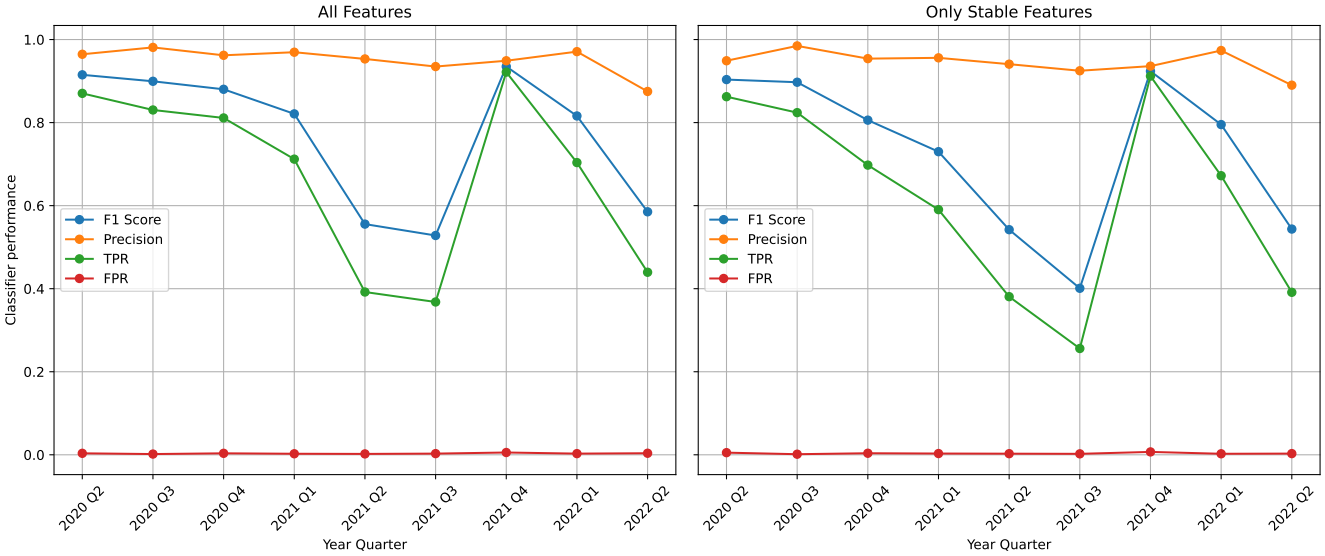

FIGURE 2. Performance of the classifier trained on all the features of the training dataset (left) and only with the sole stable features (all except for activities, services, receivers, and providers) (right).

Upper Austria in the frame of the COMET Programme managed by FFG in the COMET Module S3AI; and by Fondazione di Sardegna under the project "TrustML: Towards Machine Learning that Humans Can Trust", CUP: F73C22001320007.

References

[1] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

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

[3] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Trans. Priv. Secur.*, 22(2), apr 2019.

[4] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97, 2018.

[5] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14(6):1455–1470, 2019.

[6] Michael Spreitzenbarth, Thomas Schreck, Florian Echtler, Daniel Arp, and Johannes Hoffmann. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *Int. J. Inf. Secur.*, 14(2):141–153, apr 2015.

[7] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[8] Daniele Angioni, Luca Demetrio, Maura Pintor, and Battista Biggio. Robust machine learning for malware detection over time. In Camil Demetrescu and Alessandro Mei, editors, *Proceedings of the Italian Conference on Cybersecurity (ITASEC 2022), Rome, Italy, June 20-23, 2022*, volume 3260 of *CEUR Workshop Proceedings*, pages 169–180. CEUR-WS.org, 2022.