An update to this article is included at the end

Computers & Security 120 (2022) 102757

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Concept drift and cross-device behavior: Challenges and implications for effective android malware detection

Alejandro Guerra-Manzanares[a,*], Marcin Luckner[b], Hayretdin Bahsi[a]

[a] Department of Software Science, Tallinn University of Technology, Estonia
[b] Faculty of Mathematics and Information Science, Warsaw University of Technology, Poland

ARTICLE INFO

Article history:
Received 25 October 2021
Revised 4 February 2022
Accepted 13 May 2022
Available online 19 May 2022

Keywords:
Concept drift
Android
Malware detection
Android emulator
Real device
Smartphone
Mobile security

ABSTRACT

The large body of Android malware research has demonstrated that machine learning methods can provide high performance for detecting Android malware. However, the vast majority of studies underestimate the evolving nature of the threat landscape, which requires the creation of a model life-cycle to ensure effective continuous detection in real-world settings over time. In this study, we modeled the concept drift issue of Android malware detection, encompassing the years between 2011 and 2018, using dynamic feature sets (i.e., system calls) derived from Android apps. The relevant studies in the literature have not focused on the timestamp selection approach and its critical impact on effective drift modeling. We evaluated and compared distinct timestamp alternatives. Our experimental results show that a widely used timestamp in the literature yields poor results over time and that enhanced concept drift handling is achieved when an app *internal* timestamp was used. Additionally, this study sheds light on the usage of distinct data sources and their impact on concept drift modeling. We identified that dynamic features obtained for individual apps from different data sources (i.e., emulator and real device) show significant differences that can distort the modeling results. Therefore, the data sources should be considered and their fusion preferably avoided while creating the training and testing data sets. Our analysis is supported using a global interpretation method to comprehend and characterize the evolution of Android apps throughout the years from a data source-related perspective.

© 2022 Elsevier Ltd. All rights reserved.

1. Introduction

Threats originating from mobile malware create significant security incidents (Palmer, 2018; Yaswant, 2021) as mobile devices store an increasing amount of valuable data about individuals and enterprises. Android is the dominant operating system (OS) in the mobile OS market with a market share of 72% as of September 2021 (Statista, 2021). Due to its open nature and high prevalence, Android devices are constantly targeted by cybercriminals. For instance, according to Kaspersky, 98% of mobile banking attacks have been launched against these devices (Kaspersky, 2020). Malware spread remains a significant problem in Android OS despite the implementation of countermeasures by Google (2021) and Android original equipment manufacturers (OEMs) (Samsung, 2021). The detection of mobile malware is a challenging task with the traditional signature-based techniques used by most antivirus software due to rapid changes in the threat landscape and the emergence of new malware types. Machine learning approaches are seen as compelling solutions to address this bottleneck, especially when *zero-day* malware is considered (Fedler et al., 2013; Whitwam, 2021).

There exists a large body of research regarding the application of machine learning to mobile malware detection (Sharma and Rattan, 2021). However, the vast majority of the studies aim to prove the superiority of the proposed machine learning-based solutions on *static* data sets, neglecting the *dynamism* of the phenomenon and the obstacles that are inevitable to face in real settings (i.e., malware evolution and new trends). Such experiments with static data sets can give initial insights about the feasibility of machine learning algorithms to solve the problem but not deeper application-oriented knowledge.

In an organizational setting, machine learning models are incorporated into continuous processes that require a sustainable data analytics ecosystem. In this regard, a well-functioning data pipeline should be established, model life-cycles should be carefully managed (i.e., creating, updating, or replacing the models) and interpretation of model results should be shared among experts to generate trust between them and the machine learning models. Results could also be used for re-designing the model life-cycles if needed. When the malware detection problem is reviewed from this application perspective, for instance, considering the typical

* Corresponding author.
E-mail address: alejandro.guerra@taltech.ee (A. Guerra-Manzanares).

https://doi.org/10.1016/j.cose.2022.102757
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

setting of a malware scanner vendor, it is apparent that practitioners must solve various hindrances such as preventing the detrimental impact of variability in data sources on detection performance and adapting the models to the ever-evolving threat landscape. Such relevant aspects cannot be handled by just optimizing the learning models for a static data set.

A data analytics process addressing mobile malware detection should consider the data pipelines that are fed from heterogeneous resources such as real devices of customers, honeypots, threat intelligence feeds, and sandboxes. Dynamic features obtained from the same sample may vary according to the data collection environment (e.g., real device or emulated device), which may potentially cause a degenerative impact on the learning models when the distinct data sources are not considered in the training and testing steps of the machine learning workflow (Guerra-Manzanares et al., 2019a). In (Alzaylaee et al., 2017), emulator and real phone issues are addressed using dynamically collected API calls and intent filters as data features. A similar approach focused on detection performance is used in Guerra-Manzanares et al. (2019a,b). However, the data sets used in these studies are too small and do not consider the changes in data over time, a significant variable affecting the performance of malware detectors. No other studies in the literature have taken data source variation into consideration.

Malware behavior is prone to change over time due to the intrinsic evolving nature of the problem domain, revolving around the constant attack and defense battle between malicious actors and defenders. The possible transformation of legitimate samples should not be underestimated either. Therefore, concept drift handling should be integral to a solution aiming to provide continuous effective detection. A phenomenon that has been addressed by a limited number of proposed implementations (Cai et al., 2019; Xu et al., 2019; Zhang et al., 2020). Furthermore, the incorporation of a reliable *time* feature into the data set, which is the centerpiece of concept drift modeling, has not been analyzed nor discussed in the related literature yet. The fundamental and challenging issue regarding the definition of timestamp *reliability* within the context of mobile malware detection must be addressed so that the timestamp that best grasps the behavior of malware and benign software is utilized to model data *drift* effectively.

Acquiring knowledge from data is an iterative process between the creation of a machine learning model and its analysis. The process of malware detection necessitates the involvement of various experts such as malware analysts. Thus, characterization of malicious behavior via interpretability constructs can enhance human-machine interaction and create a bidirectional feedback loop between experts and learning models.

In this paper, we deeply investigate a dynamic feature set derived from Android apps (i.e., system calls) within a concept drift model. More specifically, we explored the impact of data sources by creating and comparing models induced from data sets collected on real devices and emulators. Due to the central importance of timestamps to concept drift modeling, in addition to data sources, we explored the effect of distinct timestamping options on detection performance. We also performed characterization of concept drift using a global interpretability method to shed more light on the changes in malware and benign samples over the years. For our experimental setup, the *KronoDroid* data set was used (Guerra-Manzanares et al., 2021), which provides timestamped data encompassing all years of Android history (i.e., 2008–2020). We applied a sequential workflow that starts with a data preprocessing stage and continues with two different procedures: *concept drift detection* and *concept drift modeling*. The former addresses the question of whether concept drift exists in the data and, if so, what type of drift occurs, utilizing one-class anomaly models based on the Isolation Forest algorithm (Gözüaçk and Can, 2020), whereas the latter addresses concept drift by dividing the whole study period into data chunks and induces an adaptive learning model that dynamically selects the best ensemble model from a pool of classifiers for each chunk (Guerra-Manzanares et al., 2022; Zyblewski et al., 2021). The last stage of the workflow applies the permutation feature importance technique (Breiman, 2001) to provide a chunk-based characterization of concept drift results.

It is worth emphasizing that the optimization of detection performance of the concept drift model is not the main aim of this research. Our focus is on the evaluation of the impact of distinct data sources and timestamps on model performance. In this regard, we created a working concept drift modeling solution complemented by a characterization step to comprehensively analyze the impact of data source variation and timestamp alternatives on the continuous detection of mobile malware throughout the years.

To the best of our knowledge, this study is the first work that explores the impact of timestamp alternatives on concept drift modeling in mobile malware detection, a critical issue to consider for *drifting* data. Moreover, the comparison of learning models induced from emulator and real device data sets has not been performed for system calls yet. The characterization of concept drift is another noteworthy contribution of our study as it may help security practitioners to better comprehend the behavioral changes of malware and legitimate apps leading to the observed concept drift.

This paper is structured as follows: Section 2 references the state-of-the-art in Android malware detection while Section 3 provides the methodological description of this study. The main results are detailed in Section 4. Section 5 outlines the discussion points and limitations of this research while Section 6 summarizes the study and future work.

2. Related work

Static and *dynamic* features extracted from Android *apps* are used to induce effective machine learning-based Android malware detection systems (Liu et al., 2020).

Static features are collected without running the app, generally from the source code or the *apk* bundle. Features such as security permissions, API calls, and intent filters lie inside this category. Static features are fast and easy to collect in an automated fashion. However, the detection systems built based on them are prone to be bypassed by *zero-day* and sophisticated malware, especially when obfuscation and encryption techniques are used.

The collection of dynamic features requires the app to be executed, allowing for the capture of the *real* behavior of the running app in a *live* environment. Features such as system calls and network flow data can be acquired using this approach. The acquisition of dynamic features is generally time-consuming and challenging but they tend to generate more robust and effective detection systems.

2.1. Real device vs. emulator

System calls are the most commonly used dynamic feature for Android malware detection (Liu et al., 2020). System calls are the mechanism used by running software to request a service from the kernel of the underlying OS. They allow collection of the behavior of the application by capturing the information flow between the distinct OS layers (Dimjašević et al., 2016). Due to their dynamic nature, the acquisition of system calls features requires the execution of the app in a live Android environment. *Real devices* and *emulators* are used as execution devices for such purpose. A *real device* is an actual physical phone running an Android OS version whereas an *emulator* is a software running on a computer that simulates almost all the capabilities of a real device (Android, 2021).

There is no clearly dominant execution platform in the recent related literature. While some researchers prefer the usage of real devices for their experimentation, either using single (Amin et al., 2016; Saracino et al., 2018; Xiao et al., 2019) or multiple real devices (Alzaylaee et al., 2020; Vidal et al., 2017; Wang and Li, 2021; Wei et al., 2022), others advocate for the exclusive usage of emulators to perform their operations, using either specialized sandboxes for dynamic analysis (Feng et al., 2018; Han et al., 2020) or general-purpose Android emulators (Casolare et al., 2021; Dimjašević et al., 2016; Guerra-Manzanares et al., 2019c; Jerbi et al., 2020; Lin et al., 2013; Surendran et al., 2020; Vinod et al., 2019; Zhang et al., 2021).

Table 1 outlines relevant and recent studies in the research domain. As can be observed, distinct Android platforms (i.e., device types), combined with different data sources, dynamic features and algorithms, have been used with significant success for malware discrimination purposes. In this regard, the single usage of any of the approaches shows advantages and limitations.

Emulators are easy to deploy, manage, and they fit perfectly in automated analysis and detection systems (Dimjašević et al., 2016), enabling the mimicry of almost all real device capabilities in a wide variety of *virtual* devices and Android versions without actually having each real device (Android, 2021). However, malware with anti-sandbox evasion techniques can deceive emulators (i.e., the malicious behavior would not be triggered if a sandbox environment is detected) (Lindorfer et al., 2015). Although some solutions provide enhancements on this issue (Naval et al., 2015; Vinod et al., 2019), they generally provide limited interaction (i.e., specific triggering events might not be possible such as SMS messages or SIM card detection (Feng et al., 2018)) and fail to install apps that do not support x86 or x86-64 architecture libraries.

Real devices are more difficult to manage and integrate into automated systems. For instance, restarting to run every sample in a *clean* device can be time-consuming, *rooting* can *brick* the device, and ensuring the exact same conditions for all tests might not be possible (Lin et al., 2013). However, they provide full interaction with the app, they are inherently immune to anti-sandbox techniques, and they show much fewer incompatibility issues (Guerra-Manzanares et al., 2021).

In any case, the main underlying *axiom* in these studies is that the behavior of applications is fully consistent across devices (Lin et al., 2013) and Android versions (Burguera et al., 2011; Vidal et al., 2017) and, consequently, that the nature of the devices (i.e., emulators or real devices) and OS versions used do not really matter. This axiomatic assumption explains the absence of homogeneity on the selection criteria and the wide variety of devices/versions and approaches used in research setups. However, the studies that have experimented with both devices (Alzaylaee et al., 2017; Guerra-Manzanares et al., 2019a; 2019b) challenge the *validity* of this *cross-device behavioral consistency* postulate. For instance, in Alzaylaee et al. (2017), when *API calls* and *intents*, usually analyzed as static features, were captured dynamically, real devices were found to provide more reliable and stable features for malware detection than emulators, thus leading to a more effective detection outcome. However, when system calls are used as features, as in Guerra-Manzanares et al. (2019b) and Guerra-Manzanares et al. (2019a), the results show that emulators may provide better detection outcomes than real Android devices.

As can be observed, both kinds of devices have been widely used for Android malware detection purposes. The selection criteria are mainly based on the available resources and required flexibility under the assumption that app behavior is consistent across devices. Emulators are usually preferred to perform such operations due to their comparatively lower analysis cost, flexibility, and easier integration in automated analysis. A small number of studies have considered both kinds of devices in their experimentation, and their outcomes challenge the validity of the *consistent behavior*

Table 1
Relevant Android malware detection studies using dynamic features.

Reference	Device type	Data set	Data set size	Features	Algorithm/s used	Accuracy	Task
Xiao et al. (2019)	Real	Drebin + Google Play	B: 3536 + M: 3567	System calls	LSTM	0.94	Binary
Amin et al. (2016)	Real	MalGenome + Google Play	B: 227 + M: 1260	System calls	Experimental *goodness* threshold	0.87	Binary
Saracino et al. (2018)	Real	4 data sources	B: 9804 + M: 2800	Behavioral	Features/Misbehaviors correlation	0.96	Binary
Alzaylaee et al. (2020)	Real	McAfee Labs	B: 19,620 + M: 11,505	API calls + Intents	D-NN	0.95	Binary
Vidal et al. (2017)	Real	MalGenome/Drebin	B: 570 + M: 5130	System calls	Statistical hypothesis testing	0.96	Binary
Wei et al. (2022)	Real	MalGenome + Xi'an University	B + M: 2000	System calls	Weighted features threshold	0.96	Binary
Wang and Li (2021)	Real	–	B: 1275 + M: 1275	Kernel-related	NB, DT, ANN, k-NN	0.98	Binary
Feng et al. (2018)	Emulator	Drebin/Androzoo + Google Play	B: 13,806 + M: 10,213	Behavioral	Ensemble + Meta-classifier	0.97	Binary
Han et al. (2020)	Emulator	–	B: 3535 + M: 25,134	Behavioral	Linear combination of classifiers	N/S	Binary
Zhang et al. (2021)	Emulator	PlayDrone + Drebin	B: 2707 + M: 2978	System calls	CNN-Bi-LSTM-Attention	0.97	Binary
Lin et al. (2013)	Emulator	–	B: 400 + M: 102	System calls	Bayes theorem	0.96	Multi
Vinod et al. (2019)	Emulator	7 data sources	B: 3130 + M: 11,514	System calls	RF, Rotation Forest, AdaBoost	0.99	Binary
Casolare et al. (2021)	Emulator	Drebin + Google Play	B: 3462 + M: 3355	System calls	RF, SVM, MLP, CNN	0.89	Binary
Guerra-Manzanares et al. (2019c)	Emulator	Drebin/VirusTotal + APKMirror	B: 1,000 + M: 2,000	System calls	kNN, LR, DT, SVM	0.97	Binary
Surendran et al. (2020)	Emulator	Drebin/AMD/Contagio + Google Play	B: 1,250 + M: 1,250	System calls	NB, SVM, DT, RF, ANN	0.99	Binary
Jerbi et al. (2020)	Emulator	AMD/DROIDCat + Google Play	B: 1000 + M: 2000	API calls	Genetic Algorithm	0.99	Binary

assumption, opening the door for further exploration of the phenomenon. This research gap is explored thoroughly in this research by analyzing the same set of applications on distinct Android platforms and assessing the cross-device detection performance of induced models.

2.2. Concept drift analysis

The vast majority of prior related studies built and tested their proposed solutions for Android malware detection using static *snaphots* of data from Android history, usually using the same data sets. In this regard, *MalGenome* (Zhou and Jiang, 2012) and *Drebin* (Arp et al., 2014) are the most used data sets for Android malware research. Despite their relatively small size and being composed of *outdated* data (i.e., their most recent samples date back to 2012), they are still used as the main sources of malware in recent publications (Sasidharan and Thomas, 2021). Even though some studies (Cai et al., 2021; Gao et al., 2021) complement their data with more recent and larger data sets, such as the *Android Malware Dataset* (AMD) (Wei et al., 2017), to mitigate data-related issues (i.e., Drebin duplication (Irolla and Dey, 2018)) and increase the representativeness of the data set, they still rely on incomplete, relatively *old* (i.e., AMD's most recent sample is from 2016 and it provides samples for just 71 malware families), and *short* snapshots of malware data from the whole Android historical timeline (i.e., from 2008 to 2021). Furthermore, when using data sets for machine learning purposes, the common practice is to mix all the data and then split it randomly into two disjoint sets (i.e., train/test sets), thus disregarding apps' location in the historical timeline. This fact undermines the *historical coherence* and yields *significantly biased* and *historically incoherent* results (Allix et al., 2015; Pendlebury et al., 2019).

As a result, these issues pose serious doubts about the *generalization* capabilities and effectiveness of these solutions to detect evolved and recent malware.

Only a limited number of the related studies considered the usage of distinct and *historically coherent* snapshots of Android history for the train/test split. However, as they show significant time gaps between them (Guerra-Manzanares et al., 2019a; 2019b; 2019c), concept drift and its degenerative impact are neglected. Consequently, the *time* variable and malware evolution over time have been purposely ignored in the vast majority of current Android malware research studies.

As provided in Table 2, only a few studies dealing with Android malware detection have considered the concept drift issue and proposed machine-learning solutions that *adapt* to changes in the data, and are able to minimize its detrimental effect over time. Even though some general approaches have been proposed to detect data *drift* (Barbero et al., 2020; Jordaney et al., 2017; Pendlebury et al., 2019), all the proposed solutions dealt with API calls (Cai, 2020; Cai et al., 2019; Lei et al., 2019; Narayanan et al., 2016; Onwuzurike et al., 2019; Xu et al., 2019; Zhang et al., 2020), an inherently static feature but one that can also be acquired dynamically. None of the studies have dealt with system calls, a *pure* dynamic feature that enables us to capture the real run-time behavior of the app and which is robust to obfuscation and encryption techniques that can bypass static API-based detection systems.

2.3. Timestamps: when time matters

The central elements behind concept drift analysis are *timestamps*. Timestamps enable the temporal placement of the sample, which aims to provide a *reliable* temporal context. However, due to the lagging nature of the malware discovery process, this is not always possible or generates reliability issues. Even though some concept drift-related studies did not provide information about

Table 2
Concept drift-related Android malware detection research.

Reference	Time-frame	Data set	Data set size	Features	Algorithm/s used	Timestamp	Performance
Narayanan et al. (2016)	2014	7 data sources	B: 44,347 + M: 42,910	Graph-kernel	Online classifier	Compilation date	Acc: 0.84
Onwuzurike et al. (2019)	2010–2016	Drebin/Virushare	B: 8,447 + M: 35,493	API calls	RF, k-NN, SVM	First seen	F1: 0.99-0.87
Cai et al. (2019)	2009–2017	5 data sources	B: 17,365 + M: 16,978	API calls	RF	First seen	F1: 0.97
Jordaney et al. (2017)	2010–2014	Drebin + MARVIN	B: 133,127 + M: 14,739	Static	Conformal Evaluation	-	F1: 0.82
Xu et al. (2019)	2011–2016	AndroZoo	B: 33,294 + M: 34,722	API calls	5 online classifiers	Compilation date	F1:0.95-0.85
Lei et al. (2019)	2012–2018	PlayDrone/Google Play + VirusShare	B: 14,956 + M: 28,848	API calls	ANN	First seen	F1: 0.99-0.84
Pendlebury et al. (2019)	2014–2016	AndroZoo	B: 116,993 + M: 12,735	Static	Evaluation framework	Compilation date	F1: 0.91-0.82
Barbero et al. (2020)	2014–2018	AndroZoo	B: 232,848 + M: 26,387	Static	Conformal Evaluator	Compilation date	F1: 0.90-0.70
Zhang et al. (2020)	2012–2018	5 data sources	B: 290,505 + M: 32,089	API calls	Evaluation framework	First seen	F1: 0.92-0.68
Cai (2020)	2010–2017	VirusShare/AndroZoo + Google Play	B: 13,627 + M: 12,755	API calls	RF	Compilation date	F1: 0.92-0.72

the timestamp approach they used (Onwuzurike et al., 2019), in the ones that reported these data, some common timestamp approaches, although differently named, can be observed.

The *compilation date* is an *internal* timestamp that relates to the creation or compilation time of the *apk* bundle. Despite being appointed as the most reliable timestamp in the past (Pendlebury et al., 2019) and used in related research (Barbero et al., 2020; Cai, 2020; Pendlebury et al., 2019; Xu et al., 2019), it has become an *unusable* approach as most of the apps released nowadays have it set at 1980 (Luxembourg, 2021). Another internal timestamp proposed lately is the *last modification* timestamp, which refers to the most recent modification timestamp found in any of the *apk* inner files (Guerra-Manzanares et al., 2021). This feature was introduced in Guerra-Manzanares et al. (2021) which discusses the feasibility of four distinct timestamp approaches for Android malware detection.

Even though internal timestamps could be deemed as *accurate* approaches, they are prone to third-party manipulation which could lead to temporal misplacement. In this regard, more robust temporal approaches can be achieved using *external* timestamps. *Virustotal*'s *first seen*, also referred as *appearance* or *submission* time in the literature, dates the application with the *datetime* it was first received by the VirusTotal scanning service. This timestamp has been used in relevant Android concept drift-related studies (Cai et al., 2019; Lei et al., 2019; Zhang et al., 2020) as being based on external and reliable services, making it easy to acquire and more robust to alterations. However, it is prone to significant delay and time misplacements due to the required proactive behavior from the user to *timestamp* the app (i.e., submission of the file).

As can be observed, the timestamp approach emerges as a critical issue to properly handle data *drift* for effective Android malware detection. Despite that, it has been neglected by all the concept drift-related studies in the problem domain. In this research, we address this research gap by considering and evaluating distinct timestamps.

2.4. Explainability in android malware detection

Our work aims to understand and evaluate the decision process behind the concept drift model utilized for Android malware detection. *Explainability* or *interpretability* methods have been used to understand the decision processes used by the machine learning-based detection systems on their predictions (i.e., XAI). In this regard, Scalas et al. (2019) stated that research regarding ransomware detection should focus attention on the explainability of the predictions to review the model outputs for the purpose of better detection. They used explainability methods to find the most discriminant features analyzing packages, classes, and methods used in Android applications. Kinkead et al. (2021) pointed out the lack of research regarding explainability behind the predictions made by Android malware detection systems. In their study, the LIME algorithm was used to find the most important features for the classification task, and LIME activations were analyzed for specific malware families.

Karn et al. (2021) explored the usage of explainability methods on models based on system calls to classify anomalous cloud containers. The authors compared XAI techniques and concluded that not all have practical applications for malware detection (i.e., SHAP and LIME are efficient but, the LSTM autoencoder is less amenable for automated explanation extraction because of convergence instability). Iadarola et al. (2021) proposed a novel method based on image representations of Android apps used as an input for an explainable deep learning model designed for Android malware detection and malware family recognition tasks. In this work, we applied a post-hoc explainability method to characterize and analyze the evolution of mobile malware over time.

2.5. Contribution to the field

As a result, even though the related literature does not consider any cross-device behavioral differences, several studies found that the dynamic behavior of an app might not be fully consistent across Android platforms. This fact may lead to a degenerative impact on the models when data from distinct sources are mixed or not properly used. Furthermore, the *time* variable is usually neglected in Android malware detection studies, which poses a severe concern regarding the generalization capabilities of the proposed solutions, trained on outdated data, against new malware. Finally, the timestamp approach, a critical variable for effective concept drift handling, has not been properly considered in the concept drift-related studies.

The main contribution of this study is to shed light on those significant research gaps by assessing the cross-device behavioral differences using system calls for Android malware detection under the consideration of the time variable (i.e., concept drift), the analysis of distinct timestamp approaches, and the assessment of their impact on the machine-learning-based models over an extended period of time.

3. Methodology

3.1. Data set

The data set used in this research is *KronoDroid* (Guerra-Manzanares et al., 2021), a hybrid-featured, timestamped, and labeled Android data set that includes malware and benign samples for all years of Android history (i.e., 2008–2020). Kronodroid is split into two data sets with different sizes, according to the acquisition device used for the dynamic features it provides (i.e., system calls). Therefore, emulator and real device-related data sets compose the full *KronoDroid* data set. This device-related split makes it an ideal data set to perform a behavioral comparison between emulators and real devices. However, as the sizes of the data sets are different, as not all apps were run in both devices, to perform a sound comparison of the dynamic profiles, just the intersection between the two data sets was selected using the *hash* attribute. As a result, the intersection data set, used in this research, was composed of 28,343 malware samples and 34,981 benign apps.

The KronoDroid data set provides four possible timestamps per record: *last modification*, *earliest modification*, *first seen VT*, and *first seen in the wild*. Based on their reliability and prevalence among the data points, two timestamps were selected for this study: *last modification* and *first seen VT*. The *earliest modification* and *first seen in the wild* timestamps were discarded due to the inaccurate nature of the former (i.e., many apps had a 1980 value) and the high ratio of missing data of the latter (i.e., not available for most of the apps) (Guerra-Manzanares et al., 2021). The *last modification* timestamp locates the app within the Android history timeline according to the most recent modification timestamp retrieved among the app *inner* files. In contrast, the *first seen VT* reports about the date and time when the app was submitted to *VirusTotal* for the first time.

The analysis of concept drift-related issues requires the usage of the *historical* context based on app timestamps. In this regard, there is no unambiguous approach to determine an app's temporal location with complete reliability and accuracy. Due to its generation mechanism, the *first seen* timestamp is prone to important delays as it depends on submission by users to VirusTotal. This timestamp heavily depends on the usability and popularity of the service to get timely notification of malware samples based on the users' proactive behavior. Therefore, it can be hypothesized that the reliability of the *last modification* timestamp, when it has not been tampered with, should be greater than the *first seen* timestamp in terms of accurately positioning the app in the historical

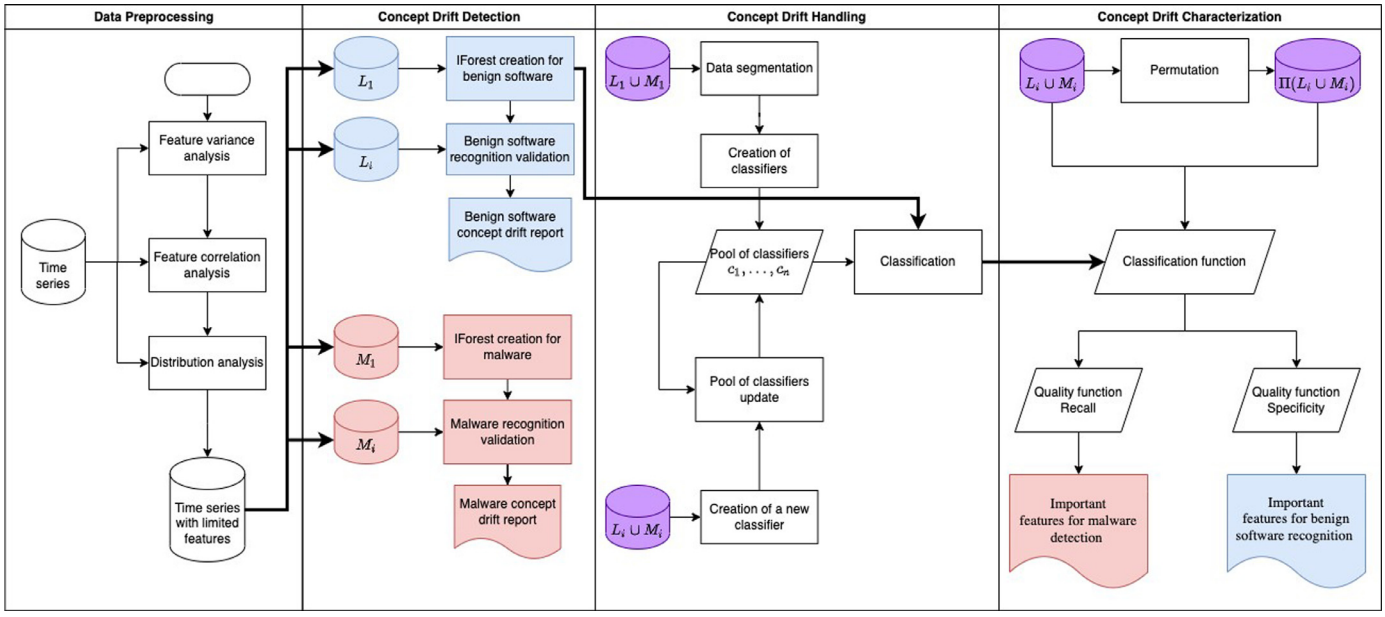

Fig. 1. Depiction of the methodological workflow followed in this research.

timeline. Despite that, in this study, we used and experimentally compared the reliability of these two approaches to deal with concept drift.

Lastly, the behavioral features collected per app in both data sets are system calls, also named as *kernel calls* or *syscalls* for short. The whole feature set is composed of 288 system calls. As Guerra-Manzanares et al. (2021) emphasizes, some of the system calls are device-related; thus the feature sets may not be consistent across Android platforms. For instance, the emulator feature set is composed of 212 system calls, whereas the real device feature set includes these features and more system calls, being extended to 288 features. Therefore, to perform a sound comparison, this comparative analysis uses both feature sets in the experimental approach, referenced distinctively as *emulator* or *reduced* feature set (i.e., 212 system calls) and *real* or *extended* feature set (i.e., 288 system calls). It is worth mentioning that when the extended feature set is used to characterize the emulator data, the values for the syscalls that belong exclusively to the real device set and which are not present in the emulator data (i.e., 76 features) are filled with zeroes. As provided by the Kronodroid data set, for each sample, the value for each feature provides the number of times the specific system call feature was used by the specific application at run-time. Thus, the input vector for the induced classifiers was composed of numeric values reflecting the absolute frequency of each system call invoked per application.

3.2. Workflow

The methodology used in this study is composed of 4 sequential phases. They are summarized in Fig. 1 and briefly explained as follows:

1) *Data Preprocessing*: zero-valued and redundant features were removed from the initial feature sets. The distributions of the remaining features were assessed using normality tests.
2) *Concept Drift Detection*: anomaly detection models were induced to assess the existence of concept drift in the data.
3) *Concept Drift Handling*: an existing solution for data streams (Zyblewski et al., 2021), was slightly customized as described in Guerra-Manzanares et al. (2022) and used to address the concept drift issue in Android malware data. Different combinations of training and testing data sets belonging to distinct data sources were evaluated with different timestamps.
4) *Concept Drift Characterization*: the analysis of changes in feature importance over time was used to characterize the observed concept drift.

A more thorough explanation of the stages is provided in the following paragraphs.

3.2.1. Data preprocessing

Machine learning heavily relies on data quality to build effective models. The removal of redundant and irrelevant features is an essential step to improve data quality within the machine learning workflow. This step aims to remove noisy, repeated, and unimportant features within the data set that may harm the classifier performance during the model's training. Three sequential steps were performed in this phase:

1) *Variance analysis*: sample variance was calculated for all features. A zero variance value, reporting no variability, was obtained for features that had constant or zero values for all samples and labels. Therefore, zero variance features were removed as they did not provide any relevant information to describe the data.
2) *Correlation analysis*: *Pearson's correlation coefficient* (r) was calculated pairwise for all features. Highly correlated features (i.e., $|r| \geq 0.80$) were dropped. This step aims to remove redundant data, a critical step for model building and the characterization methods used in this study. Correlated features may disturb the outcomes of perturbation-based global interpretability methods; thus, their removal is essential to have more reliable characterization results (Molnar et al., 2020).
3) *Distribution analysis*: the adherence of each feature distribution to the *Gaussian* distribution was assessed using statistical tests. The adherence of the feature distribution to normality is useful to assess the techniques used in posterior steps.

Once the data preprocessing step was concluded, the resulting feature sets (i.e., emulator and real device feature sets) were used in the following stages to tackle concept drift and analyze cross-device behavioral differences.

3.2.2. Concept drift detection

This phase aims to assess whether the use of concept drift handling is necessary, that is, if there is significant *drift* in the data.

In a continuous analytics process, each new observation can be represented by $c_i = (x_i, y_i)$, where $x_i = (x_i^1, x_i^2, \ldots, x_i^n) \in \mathbf{X}$ is the feature vector and $y_i \in \mathbf{Y}$ is the target label. The incoming observations are aggregated into chunks, probes of the same size, or collected during a similar period (i.e., six months time-frame). Let us assume that features from two chunks can be described by distributions F and F'. *Feature drift* is defined if the null hypothesis H_0 that F and F' are identical can be rejected (Lu et al., 2014), that is, they are *significantly different* distributions. Despite this clear statistics-based definition, feature drift can be hard to detect in real data using statistical methods. For instance, Mutz et al. (2006) and Ruiz-Heras et al. (2017) showed that Android system calls could not be modeled using Gaussian distribution. Moreover, feature drift detection is not very relevant from a practical point of view. A more relevant phenomenon is *concept drift*, which occurs when feature drift leads to a change in $\hat{y}$, the target estimation value provided by a predictive model.

Relying on the aforementioned definitions and reservations, concept drift can be detected experimentally. Let us take two series of data M_i and L_i ordered in n subsequent chunks. The series describe malware and benign software, respectively, with the value of i referring to the order in the sequence, $1 \leq i \leq n$.

Next, let us define the most important discriminators among the features using the following procedure. The data from the first chunk $M_1 \cup L_1$ was balanced using a random oversampling method (Seiffert et al., 2010) to avoid over-representation of any of the classes. Then, the classes were discriminated using the Random Forest (RF) algorithm (Breiman, 2001), a fast and reliable ML algorithm tested in similar scenarios showing outstanding performance (Guerra-Manzanares et al., 2019a; 2019b). The most relevant features of this initial classifier were selected using the permutation feature importance technique (Altmann et al., 2010). Only the features with positive mean importance were selected, thus generating the *important* feature set. In this regard, if the *important* set of features can obtain high performance on the initial set $L_1 \cup M_1$, a relevant question is whether the performance level can be kept for $L_i \cup M_i$ where $i > 1$.

To test this issue, *one-class* anomaly detection models trained separately on L_1 and M_1 were employed. The usage of one-class algorithms eliminates the class relations influence. The Isolation Forest algorithm proposed by Gözüaçk and Can (2020) was used as the detection algorithm. The detectors trained on initial-period data (i.e., $i = 1$) were tested on the subsequent L_i and M_i data sets described by the *important* feature set to calculate the ratio of observations recognized as part of the modeled class in the given chunk. The decrease in ratio signals the occurrence of concept drift, which occurs when the initially selected *important* features are not able to correctly model the analyzed phenomenon in the test data.

3.2.3. Concept drift handling

The concept drift problem is usually identified in data streams (Aggarwal, 2015; Margara and Rabl, 2018). However, Android malware detection shows related characteristics and faces similar issues; thus, a solution to handle emerging concept drift for data streams could be applied. In Zyblewski et al. (2021) an algorithm to address concept drift issues in data streams split into *data chunks* was proposed. The method uses a pool of classifiers trained on past data to make predictions about new data samples. During the prediction process, the best ensemble of classifiers is dynamically selected to perform accurate predictions. Furthermore, the pool is modified to introduce classifiers trained on new data and remove low-performance models, aiming to keep high performance over time by updating the pool of classifiers with new and evolved data. The pool update procedure enables the detection system to handle concept drift effectively.

To apply the original solution described in Zyblewski et al. (2021) for Android data analytics, the following changes were applied, as proposed in Guerra-Manzanares et al. (2022).

- The classifier pool was full and ready from the first data chunk. This fact avoids waiting for S chunks to gradually fill the classifier pool until its completeness (i.e., the S hyper-parameter refers to fixed pool size), as proposed in the original solution.
- The pool of binary classifiers is supported by an anomaly detection model to improve the recognition of benign software. This improvement was made on the basis of the experimental research that evidenced a more consistent profile over time in benign data than in malware data.

The proposed classification method, based on dynamic ensemble selection, is used in this research as a tool for concept drift handling and characterization.

3.2.4. Concept drift characterization

The main aim of this investigation is not the optimization of concept drift detection but to use the concept drift handling method to analyze changes and differences in Android malware detection when distinct data sources are used over time. For this purpose, the *permutation feature importance* technique (Breiman, 2001) was employed to analyze whether the *important* feature sets were significantly different among data chunks and for distinct data sources.

The permutation feature importance technique is an alternative method to the built-in Random Forest's importance estimation (Maimon and Rokach, 2005). The method is defined as follows. For a matrix of feature values $\mathbf{X}$ with rows $\mathbf{x}_i$ given each of N observations and corresponding response y_i, $\mathbf{x}_i^{\pi,j}$ is a vector achieved by randomly permuting the jth column of $\mathbf{X}$. The method determines the *importance* of a feature for the model by assessing the decrease in the model's performance after a random permutation for the specific feature is performed while keeping the other features unchanged. According to Altmann et al. (2010) and due to the stochastic nature of the technique, the permutation process should be repeated at least 50 times to achieve stable results. For a loss function L, the importance VI_j of the jth feature is defined as the difference between the loss calculated using pseudo-random values and the original data.

The concept drift characterization method uses the classification function f_t on data X_t from period P_t. Next, the analysis observations X are taken from the set $\biguplus_{l=t+1}^{l+h} X_j$ where h declares a time horizon for the analysis (e.g., 3 months). The procedure is summarized in the following equation:

$$VI_j^{\pi}(t) = \frac{1}{N} \sum_{\substack{i=1, \\ x_i \in \biguplus_{l=t+1}^{l+h} X_l}}^{N} Q(y_i, f_t(x_i)) - Q\left(y_i, f_t(\mathbf{x}_i^{\pi,j})\right), \tag{1}$$

where $Q(.) = 1 - L(.)$ is a quality function such as:

- *F1 score*, a comprehensive metric for malware detection performance on imbalanced data sets defined as:

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{2}$$

- *Specificity (True Negative Rate)*, which provides the benign software recognition performance (i.e., negative label) and it is calculated as:

$$TNR = \frac{TN}{TN + FP} \tag{3}$$

Table 3
Data preprocessing results.

Preprocessing stage	Results	
	Emulator	Real device
Initial Set	212 syscalls	288 syscalls
Variance Analysis	90 constant	160 constant
Correlation Analysis	28 high-correlated	31 high-correlated
Distribution Analysis	0 normal	0 normal
Final Set	94 syscalls	97 syscalls

- *Recall (True Positive Rate)*, a measure of the quality of malware detection (i.e., positive label) defined as:

$$TPR = \frac{TP}{TP + FN} \tag{4}$$

where *TP* (i.e., true positive) refers to the number of correctly recognized malware in the test set. *TN* (i.e., true negative) reflects the number of correctly recognized benign software in the test data. *FP* (i.e., false positive) provides the number of incorrectly recognized malwares in the test set, and *FN* (i.e., false negative) provides the number of incorrectly recognized benign data points in the test samples.

4. Results

4.1. Data preprocessing

The results obtained after the application of each preprocessing step are summarized in Table 3 and explained in the following paragraphs.

The initial feature sets, related to each device, were composed of 212 features for the emulator and 288 features for the real device. After *variance analysis*, 122 syscalls showed non-zero variance for the emulator case and 128 for the real device case. Thus, 90 features were removed from the emulator feature set and 160 from the real device feature set. The remaining features on each set were further processed and highly correlated features (i.e., $|r| \geq 0.80$) were removed. As a result, the final feature sets were composed of 94 features for the emulator and 97 for the real device.

The *normality* tests applied to the final sets of features showed that no feature was normally distributed. Figs. 2 and 3 show the distributions of features included on both final sets as illustrating examples (i.e., red for malware samples' values and green for benign samples).

As can be observed, both features (i.e., *getuid32* in the left graph and *ioctl* in the right) show a positively skewed distribution in both Android platforms, and, consequently, a non-normal distribution. Furthermore, there is a remarkable difference in the shape of the distributions for the same feature on each of the devices. Analogous differences were spotted for all syscalls distributions in both platforms. Therefore, these differences arise as initial support to challenge the assumption of cross-device consistent behavior.

4.2. Concept drift detection

The KronoDroid data set provides timestamped data for the whole Android history (i.e., 2008–2020). Initially, the data was split into 6-month data chunks for both timestamps. The first period with enough data to build a classifier for both timestamps corresponds to the second semester of 2011. Even though the KronoDroid data set provides data from previous years, the selected period was preferred in order to avoid biased results due to the small number of samples belonging to the previous periods in the data set. The scarcity of data samples for the 2008–2010 time frame is consistent with the actual threat landscape timeline, as the first Android malware was discovered in 2010 (Sophos, 2017). Therefore, the second semester of 2011 was selected as the *initial period* for both timestamps.

Table 4
Important feature sets ranges.

Timestamp	Data	Min	Max
Last Modification	Emulator	28	31
	Real Device	29	32
First Seen	Emulator	16	21
	Real Device	16	26

Table 5
Top-10 features ranking.

Emulator		Real device	
Last mod	First seen	Last mod	First seen
rt_sigprocmask	rt_sigprocmask	epoll_ctl	clock_gettime
fcntl64	getuid32	futex	SYS_329
futex	ioctl	SYS_329	writev
getuid32	recvfrom	clock_gettime	epoll_ctl
ioctl	read	writev	getuid32
write	futex	ioctl	write
read	write	write	close
writev	fcntl64	getuid32	gettimeofday
recvfrom	prctl	munmap	ioctl
pread64	fstatat64	read	connect

The initial period was composed of 8378 samples for the last modification timestamp (i.e., 6672 benign, 1706 malware) and 2595 instances for the first seen timestamp (i.e., 2133 benign, 462 malware). In both cases, the data was imbalanced towards the benign class. The data sets were balanced using a random oversampling technique, and the data was used to induce Random Forest classifiers with 300 estimators. Both classifiers provided an accuracy of over 0.95 on test data.

In order to select the most relevant features for each classifier, the permutation feature importance technique was applied to the training data (i.e., 500 permutations per feature). Only features with positive average importance were selected, as they reflect the actual impact on the model's performance. The results were averaged and ranked. Due to the stochastic nature of the permutation technique, a distinct amount of features might be part of the *important* feature sets on every trial. Therefore, ten trials were performed. Table 4 provides the ranges of the number of important features observed after the iterations.

As can be observed, a smaller set of important features was observed using the first seen timestamp for both devices' data. However, the range and *inner* variability in the selected sets were greater for the first seen timestamp than for the last modification timestamp. Besides, the last modification timestamp showed a much more consistent feature set selection across trials, thus showing greater stability on the number and the composition of the sets of important features. As a descriptive example, the top 10 features for each timestamp and device combination are provided in Table 5 in decreasing importance order. These results were obtained by averaging the importance ranking positions on each iteration. The two columns on the left in Table 5 provide the information about emulator data features sets for each timestamp. The two right-most columns show the same information for the real device data. For a better comparison, data related to the same timestamp are displayed in the same color (i.e., grey for last modification and white for first seen). Features observed in all feature sets are highlighted in blue.

As can be seen in Table 5, the feature sets differ not only between timestamps but more remarkably between Android platforms. More precisely, the usage of distinct timestamps in the same device produced relatively similar feature sets, mostly chang-

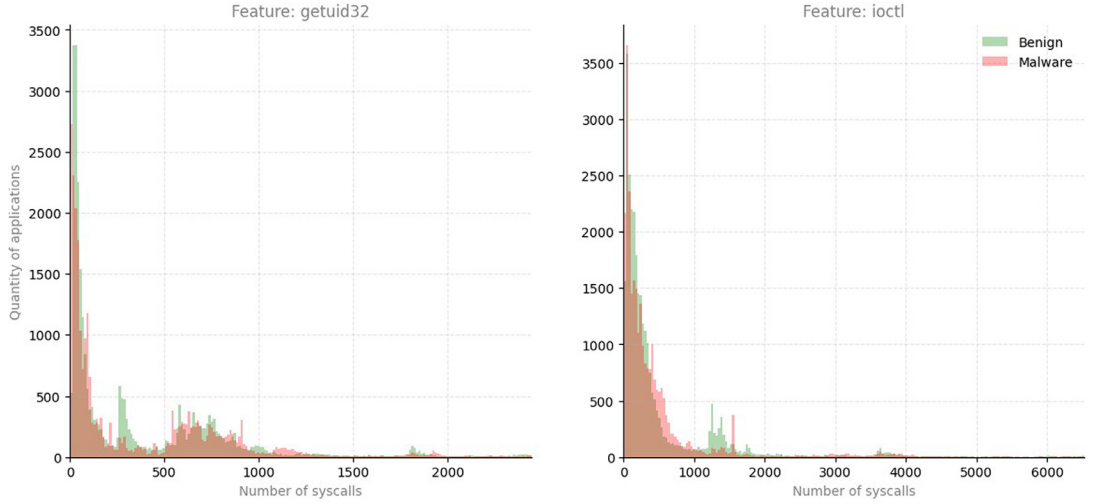

Fig. 2. Emulator features distributions.

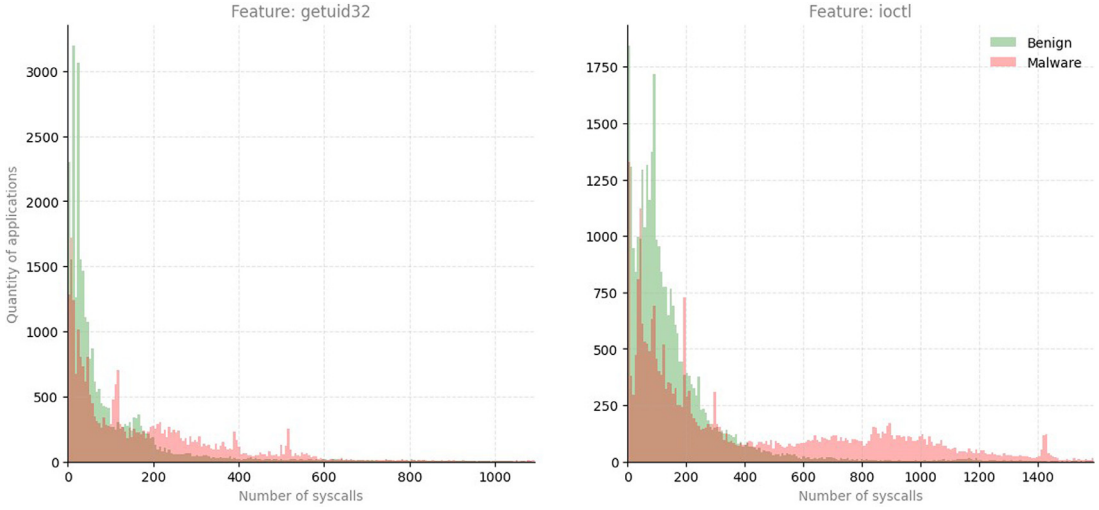

Fig. 3. Real device features distributions.

ing in order. But when the feature sets are compared between devices, the differences are significant. For instance, the most important feature in the emulator is *rt_sigprocmask* for both timestamps, whereas in the real device this feature is not found in the top 10 for any timestamp. Similarly, *clock_gettime* shows noticeable importance in the real device but not in the emulator. Three features are common in all rankings but located on distinct positions, thus showing different importance (i.e., discriminatory power). Furthermore, architecture-related syscalls appear to have notable importance in both cases, as it is evidenced by the high ranking of *SYS_329* for the real device (i.e., ARM architecture) and *fcnlt64* for the emulator (i.e., x86_64 architecture). Therefore, the results provided in Table 5 suggest that the timestamp selected might cause differences in the relevant feature set, mostly related to the order, but that more significantly, the data source can have a critical impact on the definition of the feature sets.

These initial differences are further explored by assessing concept drift in the data. In order to test the data changes, *one-class* anomaly detection models (i.e., one for malware detection and another for benign software detection) were built using the *minimal* important feature sets found using permutation feature importance as the feature selection technique. The *minimal* feature sets were constructed using the smallest *important* feature set among all iterations for each device and timestamp combination (i.e., the lower boundary (min) reported in Table 4). The rationale behind the anomaly detection test is explained as follows. If the phenomenon is stationary, meaning that the initial-period features, even with varying discriminatory power, could be consistently used to perform effective *class* discrimination in future data, the anomaly models built should show high performance over time. However, if the phenomenon evolves, meaning that important features for effective discrimination are prone to change, the anomaly model performance should drop or fluctuate significantly over time. Therefore, in these models, data *drift* is detected as an *anomaly* with respect to the initial data.

Anomaly detection models were induced using 5, 10, and all features of the minimal sets. The results of the initial-period anomaly models evaluated using 6-month data chunks from consecutive periods in the 2011–2020 time-frame are shown in Figs. 4 and 5 for emulator data and real device data respectively. For

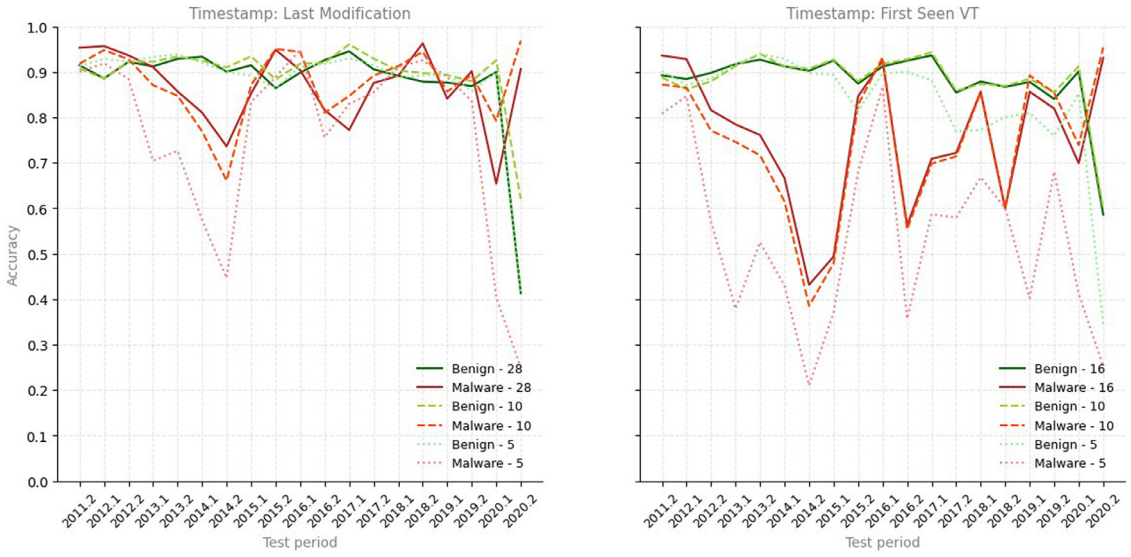

Fig. 4. Emulator anomaly detection models.

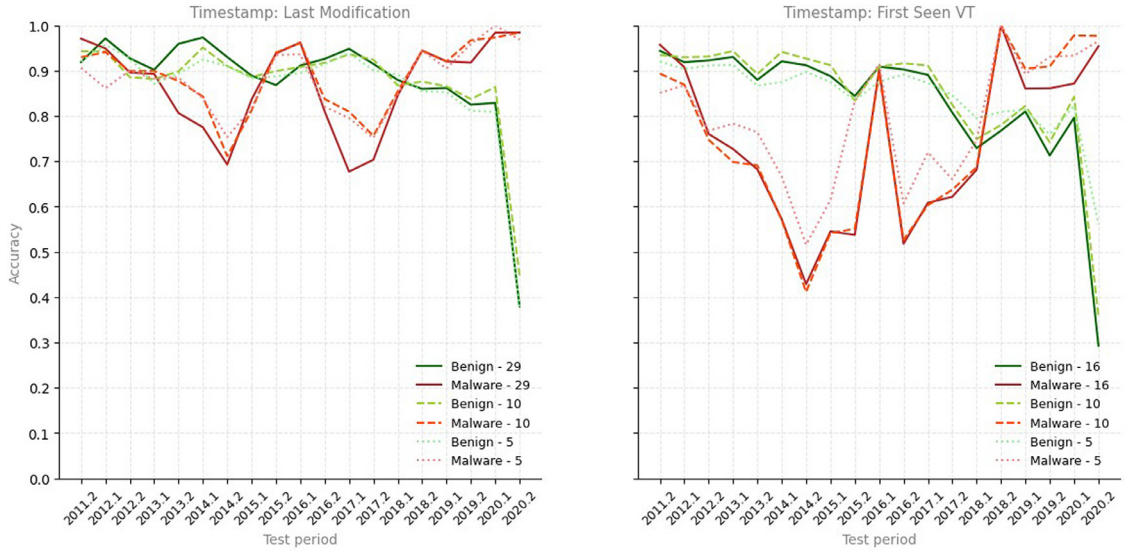

Fig. 5. Real device anomaly detection models.

each device-related graph, six lines are depicted, corresponding to anomaly models induced for each class using different feature sets (i.e., red for malware and green for benign software) under one timestamp. Model accuracy is provided in the vertical axis while the time-frame of the test data is provided in the horizontal axis. In this regard, the test periods correspond to 6-month consecutive chunks where the added suffix *.1* and *.2* to each year refer to the first six months and to the last six months respectively.

As can be observed in Fig. 4, the data obtained from the emulator does not show data *drift* for benign software. The accuracy fluctuates in a tight range for all test chunks, except for the last one. However, in the case of malware, concept drift is present in the form of *blips* (i.e., rapid decrease of accuracy deep under the reference level outlined by the benign software) (Ramírez-Gallego et al., 2017), for instance in the 2014.2 and 2020.1 periods. Furthermore, the blips are more pronounced for the first seen timestamp. The remarkable differences observed between both timestamps deserve an in-depth analysis, which is presented in Section 4.3.

In the case of emulator data, a reduced number of important features (i.e., 10) yielded similar results as the whole set of essential features. However, the reduction to five features decreased accuracy rapidly, especially for the observed *blips*. This situation is not observed in the real device data, where five features showed better performance than the whole minimal feature set.

For the real device data, the benign data shows a slightly decreasing tendency over time. Despite that, the performance is over 0.80 for most of the analyzed period. In the case of malware, data drift is observed in the form of repeated dips around 0.70 performance area for the last modification timestamp and reaching lower values for the first seen timestamp. Again, the performance dips are more pronounced for the first seen timestamp, analogously to the emulator case.

Substantial differences are observed between the emulator and real device data, depicted in Figs. 4 and 5 respectively. Firstly, the usage of distinct timestamps provides remarkably different performance for both cases. Overall, data blips for the real device malware data are deeper than for emulator data. However, emulator malware data show more blips. Despite that, in both cases, the last modification timestamp provides fewer and shallower dips than the first seen timestamp. Secondly, the reduction in the size of the feature sets increased accuracy for most of the chunks in the real device data but not in the emulator. Lastly, accuracy in the case of real device benign data shows a decreasing tendency, especially for the first seen timestamp. Such a tendency may obscure the analysis of concept drift in a classification task.

As a result and regardless of the data and timestamp used, the pronounced fluctuations in the accuracy performance for malware in all cases evidence that the initial set of important features did not maintain its discriminatory power over time. It lost it in some periods (i.e., blips, where some other features became important) to regain it in some others (i.e., peaks). This fact clearly manifests the existence of concept drift in the data. Furthermore, when using the first seen timestamp the observed *drifts* seem to be more significant (i.e., deeper dips) than when the last modification timestamp is used. This fact indicates the generation of a more pronounced drift in the data when the first seen timestamp is used, which might be caused by the delayed nature of this timestamp and the consequent data misplacement.

The observations extracted from Figs. 4 and 5 evidenced that concept drift emerges as a significant threat to the performance of detection models over time. The next stage in our workflow is to address the situation with a dedicated classifier and use it to perform a deeper analysis of the data.

4.3. Concept drift handling

The concept drift-handling detection system, based on the dynamic selection of the best ensemble of classifiers from a pool of classifiers and its constant update as explained in Section 3.2.3, was used to analyze malware and benign data.

In order to explore the phenomenon from all possible perspectives, the solution was applied using different combinations of training and testing sets that were described using both feature sets (i.e., reduced and extended feature sets). The usage of emulator and real device data allowed us to analyze differences between data sources and the usage of both feature sets enabled us to explore the effect of features in detection performance. Furthermore, both timestamps were also used for every feature set and data source combination. For instance, when emulator data was used as the *train* set with the last modification timestamp, 4 distinct combinations were tested by using the 2 possible feature sets as data descriptors and the 2 data sources as *test* set. These multi-testing scenario results are reported in Fig. 6. The analysis of the phenomenon taking all possible permutations of the variables into account (i.e., feature set, device, and timestamp) enriched the analysis of differences between data sources, the impact of feature sets, and the reliability of timestamps. More importantly, it enabled us to ensure unbiased results as no assumption was performed.

For the sake of deeper exploration of the phenomenon, a greater level of granularity was used to better capture emerging concept drift. The data was split into quarter-year data chunks limited to 4000 samples, thus data were analyzed for each quarter of the 2011–2018 time frame. F1 score performance metric was calculated for each period using the classification model, composed of a dynamic ensemble of n classifiers trained during previous periods (i.e., $n = 12$ yielded the best performance in our experimental setup).

The obtained results illustrate changes in the quality of malware detection among periods. The results for the last modification timestamp using emulator data as the train set are provided in Fig. 6. Fig. 7 shows the results obtained when real device data were used as the train set for the same timestamp. As can be observed, the 4 possible combinations of test data variables are plotted. Disregarding the train data and timestamp, test data are always defined by a data source (i.e, emulator or real device) and a data descriptor set (i.e., *reduced* or *extended* feature set used to describe the data). In the cases where the test data source differs from the train data source, it enables us to explore cross-device performance, whereas the usage of distinct feature sets provides information about the discriminatory capabilities of a larger feature set versus a reduced feature set. In the figures, the feature set is referred to as the original source of data it belongs to (i.e., they are architecture-related features). For instance, in Fig. 6, *Real* refers to the extended set of features and *Emu* to the reduced set of features. This denomination is preferred to properly relate the feature set impact to the data source. The same information is provided in Fig. 8 and Fig. 9 for the first seen timestamp. More precisely, Fig. 8 uses emulator data as the train set whereas Fig. 9 uses real device data to train the model. In these graphs, the horizontal axis reports the quarter analyzed, and the vertical axis the corresponding F1 performance of each test set on the trained model. It is worth noting that when using the first seen timestamp, a greater number of apps were dated in the period 2012–2013. The higher prevalence of malware for the first seen timestamp in this period might have been caused by the expansion and increased popularity of *VirusTotal* service during that time (VirusTotal, 2012). Due to the 4000 samples per chunk constraint, and to not miss emerging concept drift, more than one chunk was processed per quarter. Therefore, for this timestamp, in the 2012–2013 time frame the relation chunk-quarter was exceptionally bypassed to analyze all the available data. This is reflected by the greater separation in the horizontal axis for this specific time frame. These additional data chunks are provided for the sake of completeness and as evidence of the different temporal alignments for the same data sets when using different timestamps. For the sake of interpretability of the graphs, test data source and feature set are plotted distinctively. More specifically, the line color indicates the source of the test data (i.e, blue for emulator and yellow for real device) and the line style reflects the feature set used to describe the train/test sets (i.e., solid for the extended feature set and dashed for the reduced feature set).

When the last modification timestamp is considered, the performance of the proposed method to deal with concept drift is relatively stable. The method obtains over 0.80 F1 score in most of the studied timeframes especially when the model is tested with data from the same source as the training set as evidenced by the blue lines in Fig. 6 and the yellow lines in Fig. 7. The results obtained using different feature sets are similar. This fact indicates the goodness of the reduced feature set (i.e., emulator features) to provide similar performance as the extended feature set and that the zero-filling for missing data when the real device feature set is used to describe emulator data does not significantly harm the model performance. When the model is trained with real device data but tested with data from the emulator using real device features as descriptors (i.e., extended feature set), the results obtained are the worst for this timestamp. In this case, the feature set seems to notably impact the performance, as when real device data are used for training with emulator-based features (i.e., reduced feature set), better results are observed. This is confirmed by the fact that the performance *blip* observed in 2012-Q3 when the extended feature set is used it is not observed when the reduced feature set is used. More interestingly, the dip in 2012-Q3 is just observed

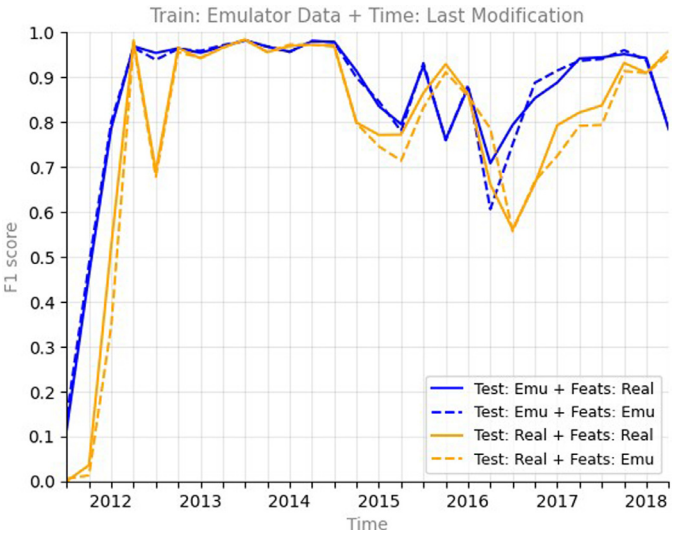

Fig. 6. Last Modification timestamp - Training Emulator.

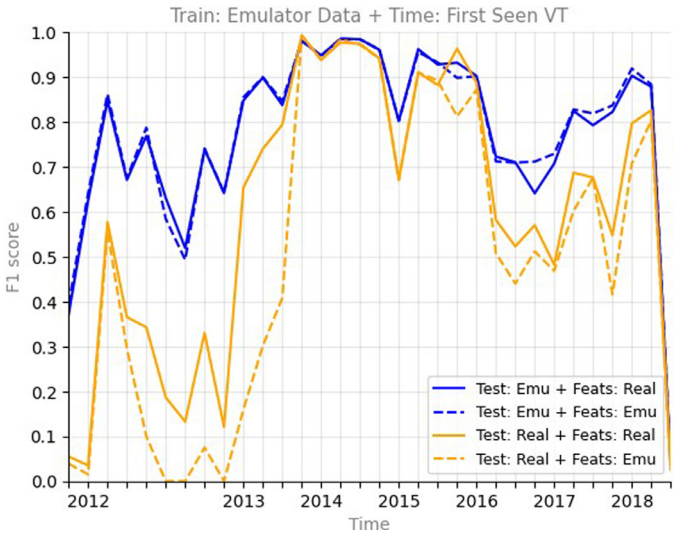

Fig. 8. First Seen timestamp - Training Emulator.

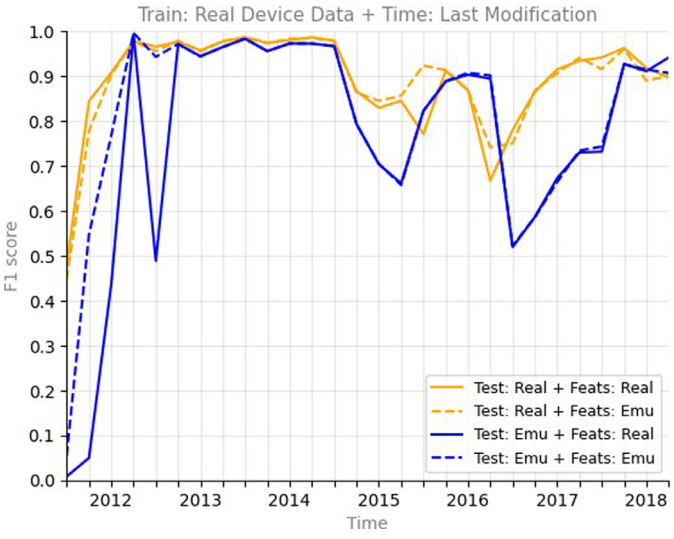

Fig. 7. Last Modification timestamp - Training Real Device.

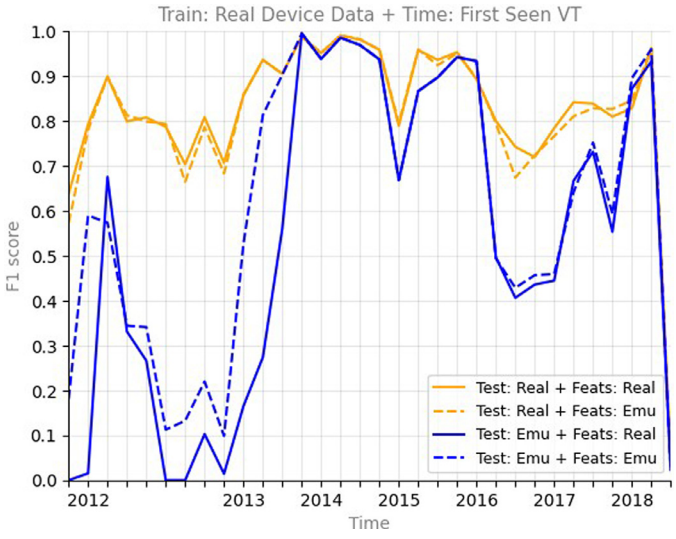

Fig. 9. First Seen timestamp - Training Real Device.

when the test data used does not belong to the same source as the training data. From 2016-Q3, a general worsening of the results is observed for all models, being especially pronounced when training and testing data sources differ. This fact is observed regardless of the feature set used. Overall, from the analysis of Figs. 6 and 7, it can be extracted that for the models built using the last modification timestamp, the feature set used does not have a relevant impact on the performance of the model, whereas the data source used has a significant decreasing impact on the model's outcome, especially if the training and testing set sources differ.

When the first seen timestamp is used to locate the same set of apps within the Android historical timeline instead of the last modification timestamp, the performance of the model is radically different as evidenced by Figs. 8 and 9. In this case, except for the period located between mid-2013 and the beginning of 2016 where the results are similar, the usage of distinct train and test sources yields very poor results. This fact is especially evident in the 2011–2013 time frame where the test detection performance using data distinct from the training source drops to null levels in two consecutive data chunks. Furthermore, the performance of the different feature sets allows the observer to draw contradictory conclusions in the first periods. For instance, as can be observed in Fig. 8, when the model is trained with emulator data and emulator-related features (i.e., reduced feature set) are used to describe the data, the testing with real device data provides worse performance than when real device-related features are used as descriptors (i.e., extended data set). However, the opposite situation is observed in Fig. 9 for the real device data trained models. In this case, when the extended feature set is used, testing with emulator data provides worse results than when the reduced feature set is used with the same testing data. This suggests that better results can be achieved when the model is trained with the *natural* feature set used to describe the testing source (i.e., the extended for real device data and the reduced for emulator data). Nevertheless, due to the aforementioned *zero-filling* imputation procedure for missing values when using the real device feature set to characterize emulator data, bias may be introduced to the model, and consequently, without proper knowledge about the source of

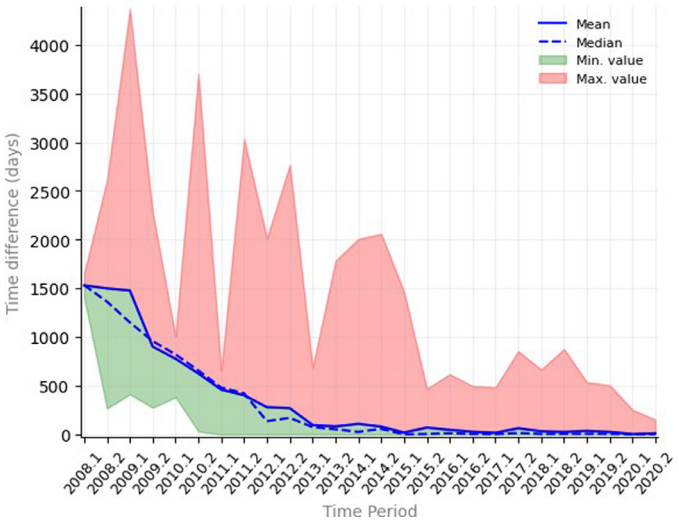

Fig. 10. Temporal differences between timestamps - Benign data.

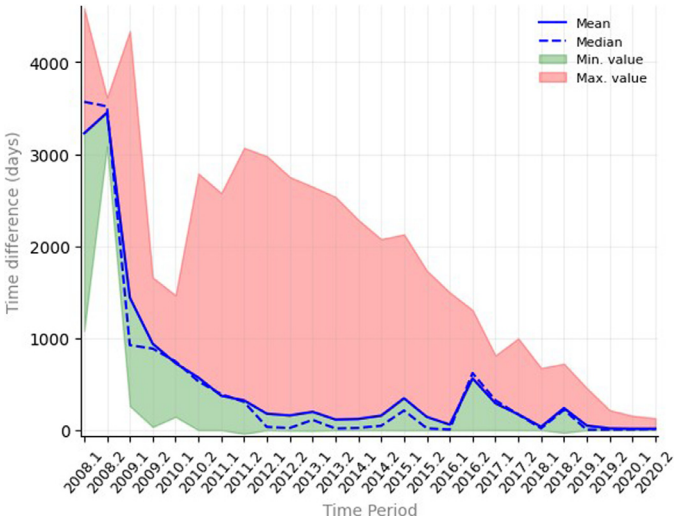

Fig. 11. Temporal differences between timestamps - Malware data.

the testing data, it is safer to use the reduced feature vector in all cases. The overall performance of the models built using the first seen timestamp is significantly worse than when the last modification timestamp is used. The overall performance is lower, especially when testing with a distinct data source, and the feature sets used seem to have a remarkable impact in the models. Furthermore, the performance varies abruptly from quarter to quarter indicating that sudden data drift occurs. This sudden drift could be caused by an *artificial* drift generated by the misplacement and the mix of *historically incoherent* data. Historical incoherence occurs when data belonging to different time-frames are blended together, thus generating an unnaturally occurring set of data. This is opposed to the overall smooth performance lines observed in the last modification timestamp, which may indicate a more naturally occurring drift in the data.

On the basis of the differences observed in the previous graphs, a deeper exploration of the timestamps becomes of interest. Every data sample can be located historically using either of the timestamps. The timestamps might converge or diverge. Timestamps converge if they provide similar timestamps, for instance, 2018 in both cases, whereas they diverge when their locations are significantly apart, for instance, one locates the sample in 2018 and the other in 2012. The temporal differences (i.e., divergence) between the timestamps can provide relevant insights for timestamp selection and analysis. Figs. 10 and 11 provide the temporal differences between both timestamps, computed for each sample and displaying benign and malware data separately. The individual differences in timestamps are grouped in six-month periods and located in the timeline based on the sample last modification timestamp, which is taken as reference. The individual differences for every period are averaged and reported in Figs. 10 and 11 for benign and malware samples, respectively. *Days* are used as the *difference* basic unit. More precisely, these graphs report the average value of the difference between the last modification timestamp and the first seen timestamp for the samples located in a specific period by the last modification timestamp. The initial expectation is that the last modification timestamp would place the sample more accurately in Android history (i.e., if not tampered), and earlier in time than the first seen timestamp. Therefore, it is chosen as the reference time. In Figs. 10 and 11, the blue line provides the average value for each period while the dashed line provides the median value. These two central tendency measures provide insights into the expected value of displacement of the samples for each period. The red and green areas provide the notion of the differences dispersion. More specifically, it is the range between the largest and the smallest difference found in that period (i.e., the maximum and minimum difference found in specific samples belonging to that chunk). The red area encompasses from the average value to the maximum value for each specific period, whereas the green area ranges from the average to the minimum value.

For both classes and all cases, a positive difference between both the timestamps is observed. This evidences that the first seen timestamp locates the samples later in time (i.e., delayed with respect to the last modification timestamp). The differences are especially pronounced in the early years of Android history, where differences average around 1500 days (i.e., four years) for benign applications and around 3500 days (i.e., over nine years) in the case of malware samples. For instance, a malware sample located in 2008.1 according to the last modification timestamp would be located in 2017.1 by the first seen timestamp (i.e., when VirusTotal first received the sample). This significant difference notably impacts the performance of the classifier and its adaptation to malware evolution when the concept drift issue is considered, as can be observed in Section 4.3.

However, as can be spotted in Figs. 10 and 11, these temporal differences have decreased over time. More precisely, they have monotonically decreased for benign instances and decreased significantly in the case of malware samples, thus making these timestamps more synchronized and closer in time. For example, for benign samples, 2020.1 and 2020.2 periods show a temporal difference average of just 4.88 and 12.37 days and a median of 2 and 3 days, respectively. In the case of malware samples, an average value of 15.98 and 16.45 days and 7 and 11 days are observed, respectively. As a result, the gap between both timestamp approaches has largely decreased over time, making them converge and increasing the reliability and accuracy of the first seen timestamp in the more recent years (i.e., 2019–2020).

The large differences observed in the early years of Android, with data misplaced about 4–8 years on average, have a significant impact on the models induced using the first seen timestamp, as evidenced in Figs. 8 and 9. In both cases, these initial periods provide the worst performance of all models induced for all tests cases, including when the test data source is the same as the training data source. Overall, as differences between timestamps decrease, thus first seen converges to the last modification

timestamp, the detection performance appears to improve, especially with regards to the test data from the same source as the train data. Therefore, it can be deduced that performance differences in the most recent years are mainly caused by apps' behavioral differences in different Android platforms rather than the impact of the timestamp, as the variation of the timestamp values minimizes.

The main derivation from these experiments indicates that the timestamp approach has a significant effect on the performance of concept drift models, especially for the first years of Android history. The last modification timestamp emerges as a more reliable temporal approximation to tackle the concept drift phenomenon, as evidenced by the high performance kept by the models in Figs. 6 and 7 over time and on both testing sets. Furthermore, the smooth transition between quarterly performances suggests that the data drift occurs more *naturally* and that the classifiers built on previous data chunks can be leveraged to accurately address emerging concept drift issues. The first seen timestamp performance transitions between adjacent quarters are abrupt and even extreme, thus indicating the existence of an *artificial* drift that can be hardly modeled using previous data knowledge.

Therefore, the last modification timestamp was selected to further explore the behavioral differences of Android apps across devices in the next section. Besides, data samples were described using the reduced feature set, which seems a more reasonable option for the selected timestamp.

4.4. Concept drift characterization

To perform a deeper analysis of the behavioral differences of the same set of apps in an emulator and a real device, *permutation feature importance* was calculated using *specificity* and *recall* as quality (Q) functions. In this exploration, for each analyzed case, the training and testing sources belonged to the same Android platform, thus enabling to describe the specific behavioral patterns for each device.

Feature importance calculated for *specificity* informs about the essential features to recognize benign software. The same function calculated for *recall* identifies the important features for malware recognition.

The graphs in Fig. 12 show the features with positive average feature importance for the benign and malware recognition task (i.e., specificity and recall). The permutation feature importance technique was calculated using 2000 permutations, and the experiment was repeated three times to ensure the stability of the results. For the real device data, 48 features were found important in at least one quarter. For the emulator, this number reached 65.

For each plot in Fig. 12, the colored lines/areas provide the important features for the related task (i.e., vertical axis) on each specific quarter (i.e., horizontal axis). The line/area color relates to specific features while the area width provides the *relative* feature importance of the specific feature. The relative importance of each system call on each specific chunk is reported in relative standing to the total importance of all the important system calls for that chunk. The relative measure of importance is preferred to the absolute importance value to provide a more comprehensive visual comparison of the important features over time. For the same reason, all features that showed a maximum relative importance value lower than five percent for any chunk in the whole analyzed period are aggregated in the *others* category. For each graph, the number of features aggregated in this category is provided within parentheses.

Fig. 12 a and b report on the important features for the benign software recognition task per quarter. As can be observed, a similar set of features are important on both devices for specificity performance. However, the proportion of features importance (i.e., relative importance) differs across devices. For instance, in the case of the emulator, the importance of *faccessat* feature is significant, especially in the first quarter, whereas, for the real device case, the importance is not remarkable in any period, thus it is included in the *others* category. On the other hand, *clock_gettime* and *read* have a remarkably greater importance for the real device than for the emulator. Besides, the first three periods for the emulator are described by a smaller set of important features, as evidenced by the thin lines and gaps observed in the importance plots. This might be a reason for the poor performance provided by the classifier in the initial periods when testing with distinct device data (see Fig. 7). Overall, similar sets are used but with varying proportions which makes the characterization substantially different. Furthermore, even though in the real device, a large set of features show importance, some of them show remarkably more importance than the others in the same chunk (e.g., *clock_gettime* in 2016–2017 time-frame and *recvfrom* in the 2014–2018 time frame) whereas, in the emulator, the importance is split among a larger number of features showing small individual importance values (with some exceptions like *recvfrom*).

Therefore, the usage of data from distinct sources as learning and testing sets for the benign recognition task may be biased and yield sub-optimal results but should be a relatively successful task due to the similar features describing effectively both sets.

In order to analyze deeper the differences between the emulator and real device data, the obtained feature importance, calculated for specificity as Q function, were compared in each quarter for the emulator and the real device using the *Wilcoxon signed-rank* test. To apply this statistical test, the same number of values must be present in the compared sets. However, some features noted missing values in quarters where they were not found as an important feature for the task. Due to the high ratio of missing values in the analyzed data: 64% and 68% for the emulator and the real device, respectively, the imputation of zero value might bias the comparison results, increasing the similarity of the vectors due to the high number of missing values replaced by zero. A better approach was to replace the missing values with the mean importance of the feature, calculated for the given data set and taken as the reference *negative* value for the test. Thus, vectors were compared using means when the number of missing values was high and using the distribution of importance otherwise. The last issue was the different sets of important features for distinct data platforms. To avoid future complications, the intersection of both feature sets was used. As a result, 45 features were compared.

Table 6 (see Appendix) summarizes the experimental results. The table presents features with a p-value < 0.005 which suggests a great difference between the compared vectors. The occurrence value refers to the number of non-missing values in the compared vectors. As a result, only 13 features among 45 are significantly different for the emulated and real data. These results confirm the relative similarity observed in the distributions in Fig. 12a and b.

The situation changes drastically when the malicious software recognition task is analyzed.

Fig. 12 c and d show the set of features with positive average feature importance calculated for the *recall* metric, in the same manner as it was performed for specificity.

As can be observed, in both cases, from 2011 to 2016-Q2, the importance of the features calculated for recall differs significantly from the results obtained for specificity. More precisely, the incidental spikes of feature importance for recall show that a reduced set of features are important in each quarter, contrary to the large set of features observed of specificity. For the malware recognition task, in most of the quarters until 2016, less than 10 features emerge as important. However, this situation changed over time. In the last periods, the bars become more similar, as a larger amount of features become important for recall, thus the plots for

Fig. 12. Relative feature importance evolution from 2011-Q3 to 2018-Q2.

both recognition tasks resemble each other but still keep identifiable differences. However, even though these similarities are *strong* enough to relate the distinct plots according to devices, the global picture is that malware recognition is a completely different task than benign software recognition, thus relying on different sets of important features.

Although Fig. 12c and d show the same set of important features, there are essential differences between them related to importance levels. For instance, the most important feature is different for both sets. Real device data are dominated by *clock_gettime* feature, which is the most important feature in thirteen periods. However, in the emulator data, this feature is not present in fourteen periods, and it is never the most important feature in a quarter. For the emulator data, the most important feature is *munmap* but with a lower dominance than the most important feature in the real device case. Besides, a visual comparison between both graphs enables the observer to easily spot the differences in important features between these sets. Their character is radically distinct in the vast majority of the chunks. Therefore, it can be concluded that the usage of the same set of features can not guarantee the same results for both sets as the proportions and dominance of important features vary significantly.

The observed differences in importance influence cross-device malware detection quality. For instance, in each quarter that the classification F1 performance drops (see Fig. 7, training with real device data and testing with emulator), the feature *clock_gettime* is not considered as important in the emulator data. Besides, it is worth mentioning that the emulator data promotes *mprotect* feature, which is not significant in real data. In the timeframe from 2016.Q3 to 2017.Q3, when differences between F1 performance calculated for real device and emulated testing set are the largest, even when the same set of features is used, *recvfrom* feature is critical for recall. This feature is irrelevant for the emulator data in the same period.

Analogously to the specificity case, Table 7 (see Appendix) presents the *Wilcoxon signed-rank* test results for recall calculated between the emulator and real device data. In this case, over 75% of features (i.e., 34 out of 45) statistically differ between data sources (i.e., p-value < 0.005). Thus significant differences in features are thrice more prevalent for recall than for the specificity metric. The experimental results backed by the statistical analysis prove that emulator data differ significantly from real device data and that the coincident usage of both data sets is not advisable in a malware detection task.

As the previously presented *Wilcoxon signed-rank* tests performed for specificity and recall are remarkably different (i.e., in the number of features with different distribution), we compared the *importances* calculated separately for emulator and real device data using various quality functions Q.

Table 8 (see Appendix) provides the results of the comparison between recall and specificity calculated just with emulator data. The results obtained for the real device data are presented in Table 9 (see Appendix).

In both cases, the test results confirm the high number of features that differ in importance for the specificity and recall metrics. The number of different features exceeds 50% for the emulator data (i.e., 34 from 65 features) and 75% for the real device data (i.e., 35 from 45 features). These results support the previous observation on the difference between importance distributions. Furthermore, these results prove that benign software and malware develop in a disparate way and are characterized by different features. More importantly, the only observed similarities between the emulator and real device data in this sense are restricted to the fact that just 17 features are common in both tables, which is about half of the listed features. For instance, the most important features are not shared. The shared features are emphasized in italics in Tables 8 and 9 (see Appendix).

To summarize the findings from the previous tests, it could be stated that there exist significant behavioral cross-device differences when using system calls. Furthermore, these differences also exist for the classes within the same data source, as malware and benign data are distinctly characterized. Thus, a description of malware based on data from one device (i.e., real or emulated) should be used with much care to model classification for another device because of the significant differences in their characterization. This fact was also evidenced by the performance of the proposed models when the test sets used were collected from a distinct source than the train sets used to induce the models. Consequently, both data sources should not be merged in the generation and evaluation of Android malware detection methods.

5. Discussion

The previous concept drift modeling works in the mobile malware domain did not draw attention to the timestamp selection approach and its impact on the detection model performance. Our work shows that accurately grasping the evolution of malware and legitimate samples, thus effective modeling of the concept drift, greatly depends on the timestamp approach used, as the timestamp value is the main determinant in positioning the instance in the suitable chunk.

Concept drift detection results given in Section 4.2 (i.e., Figs. 4 and 5) indicate that the first seen timestamp creates more and deeper performance *blips* during the evolution of malware regardless of the data source (i.e., real device or emulator). The results of our concept drift model given in Section 4.3 support the same fact, especially in the early years of the analysis period, as *first seen* led to very poor detection performance (i.e., Figs. 8 and 9). Thus, we determined that the last modification timestamp better enables the model to reflect the changes and evolution of the threat landscape.

The dynamic behavior of the same set of Android apps is compared between distinct Android platforms (i.e., an emulator and a real device), providing experimental and statistical evidence of the changing behavior of apps according to the execution device. Our findings suggest that data from different sources should not be merged in training and testing sets as the behavior of apps in different Android platforms is not consistent.

The operational implications of hybrid detection systems, where cloud data might be merged with data acquired in users' devices, are critical. If the model is trained in a cloud-based *backend* with data collected from emulators and deployed to users' devices, then the detection performance might be significantly hindered as demonstrated in Figs. 6 and 8. As evidenced by Figs. 7 and 9, the best scenario for detection performance requires that train and test data are collected from the same Android platform, which in this case implies using either only real devices or only emulators as collection devices. Real device data collection is more time-consuming than the collection on emulators, as the devices have to be manually reset and *root*, and the collection cannot leverage *snapshots* for cleaning and restarting. This could make the process of building an effective model significantly longer. Furthermore, there may be further implications with the usage of distinct real devices, as due to the myriad of different devices, Android OS versions, and chipsets found in Android devices, behavioral dissimilarities may also arise among them. Thus impacting the performance of the detection system. However, it could be more comfortable for the user as it enables on-device detection. If the selection is only emulators, the user should send the app to be processed on the cloud, and after all the processing, the detection result should be

provided to the user. This implies the need for connectivity to retrieve the result, as the model is not deployed in the device and, consequently, a slower detection process. However, emulator models might be faster to induce and device parameters are easier to control, which may generate more robust models. As can be seen, the selection of any platform requires the analysis of pros and cons. In any case, the main recommendation is the usage of the same platform data for the training and test sets. Those solutions using mixed data sources should trace the data source and avoid merging the data when high detection performance is the objective.

Therefore, not only does the timestamp become a critical issue when concept drift is explored, but the source of data also emerges as an essential variable, becoming an even more significant issue as the timestamps converge. Both critical factors have been overlooked by the existing body of Android malware research. This study is the first step in further exploration of the critical impact of timestamps and devices in the data collection step.

This paper performs an attempt to characterize concept drift findings. We advocate for the incorporation of interpretation methods into application-oriented machine learning studies to have a better understanding of the underlying aspects of the problem domain beyond detection accuracy. Our proposal is not exclusively for explaining black-box models, but also researchers can discuss the findings that can be obtained from inherently *interpretable* models. This can enhance and enrich the knowledge of the problem domain. Some studies leverage the combination of machine learning and interpretability to explore a phenomenon (Stachl et al., 2019; Zhao et al., 2020). We are aware of the possible limitations of these methods (Molnar et al., 2020; Rudin, 2019), and that, usually, interpretability findings may not explain causality. Nevertheless, such an endeavor can increase the body of the domain knowledge that can be derived from the data. Beyond the purpose of knowledge generation, we also acknowledge that cyber security community should consider interpretability more as a part of the machine learning-based solutions due to the fact that security analysts play key role in security operations.

5.1. Threats to validity

This research aims to emphasize and bring to light several issues that have been overlooked by the existing research and may affect the field of Android malware detection. However, this research is not free of limitations, which are summarized as follows.

- The data used in this research is specifically tailored to analyze concept drift-related issues. The same data set is analyzed on both Android platforms but issues such as malware *anti-sandbox* capabilities are not taken into account. Nevertheless, other third variables which may generate differences in behavioral profiles were controlled in the generation of the data set (i.e., using the same OS version, scripts, and debugging tool).
- The Android platforms used to run the applications were selected on the basis of user preferences and device popularity (i.e., Google Emulator and a Samsung device) (Guerra-Manzanares et al., 2021). However as they may change and evolve over time, other devices could have been used which might have provided distinct results.
- The characterization technique (i.e., permutation feature importance) has been widely used to characterize machine learning models. Nevertheless, due to its inherent randomization procedure, it might be prone to show distinct pictures of models, especially when feature randomization is embedded into the models, such as in the Random Forest algorithm.

Therefore, even though this study has limitations, they have been tested and minimized to provide the most complete approach to the phenomenon.

6. Conclusions and future work

The vast majority of literature in Android malware detection has neglected the detrimental impact of the *time* variable in the machine learning-based detection models and has not even considered the implications of merging data sources. Furthermore, the small number of concept drift-related studies have not paid attention to the timestamp selection, a central element to address concept drift effectively.

This research explores the emerging challenges and their implications when dealing with concept drift for Android malware detection using different timestamps and, at the same time, using data collected from distinct Android platforms (i.e., real device and emulator). Our results show the detrimental impact on machine learning classifiers caused by concept drift, especially when using *first seen* as the timestamp. The *last modification* timestamp appears to be a reliable and accurate source for the historical location of apps in the Android timeline, providing and keeping high-performance detection models over time, even when distinct data sources are considered. However, our experimental setup proves that the behavior of Android apps is not consistent across Android platforms and that data collected from different sources should not be used coincidentally.

An extensive body of research has previously focused on the optimization of proposed solutions in short and mostly outdated Android historical data sets. The focus on these solutions tailored for static data snapshots poses severe concerns about the generalization capabilities of such solutions to recent malware. To the best of our knowledge, this research is the first work addressing the challenges and implications of distinct timestamps for historical location, concept drift issues, and cross-device data for Android malware detection. This work aims to bring to light the significant impact of these underlying critical variables that have been overlooked by the specialized research.

In our future work, we plan to continue digging deeper into this approach and tackle the aforementioned limitations in a more restrictive way. This work should be understood as a pioneering step into a new exploratory direction that considers the impact of several variables on the performance of the learning models and challenges the assumptions of most of the previous research.

Declaration of Competing Interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

CRediT authorship contribution statement

Alejandro Guerra-Manzanares: Conceptualization, Methodology, Software, Validation, Investigation, Visualization, Writing – original draft, Writing – review & editing. **Marcin Luckner:** Conceptualization, Methodology, Investigation, Visualization, Writing – original draft, Writing – review & editing. **Hayretdin Bahsi:** Conceptualization, Methodology, Writing – review & editing.

Appendix A

Table 6
Features that differ between emulated and real device data in importance calculated for Specificity $p < 0.005$.

	Feature	*p*-value	max(Importance)	Occurrences
1	recvfrom	0.0036	0.2250	53
2	clock_gettime	0.0001	0.1423	43
3	setsockopt	0.0000	0.0755	39
4	uname	0.0000	0.0193	20
5	statfs64	0.0000	0.0218	17
6	exit_group	0.0000	0.0006	12
7	gettid	0.0001	0.0063	10
8	tgkill	0.0000	0.0001	6
9	nanosleep	0.0000	0.0000	3
10	sched_getscheduler	0.0000	0.0000	2
11	msync	0.0000	0.0141	2
12	listen	0.0000	0.0000	1
13	mremap	0.0000	0.0070	1

Table 7
Features that differ between emulated and real device data in importance calculated for Recall $p < 0.005$.

	Feature	*p*-value	max(Importance)	Occurrences
1	clock_gettime	0.0032	0.0825	32
2	getdents64	0.0002	0.0161	19
3	fsync	0.0011	0.0527	18
4	lseek	0.0050	0.0111	18
5	dup	0.0000	0.0375	17
6	fcntl64	0.0001	0.0477	16
7	socket	0.0001	0.1513	15
8	write	0.0000	0.0197	15
9	recvfrom	0.0016	0.0728	15
10	fchmod	0.0000	0.0058	14
11	getuid32	0.0000	0.0363	14
12	mprotect	0.0015	0.0484	14
13	sigaltstack	0.0000	0.0077	14
14	read	0.0000	0.0313	12
15	_llseek	0.0000	0.0263	11
16	fstatfs64	0.0000	0.0710	11
17	sched_yield	0.0000	0.0063	11
18	sendmsg	0.0000	0.0071	9
19	setsockopt	0.0000	0.0058	9
20	connect	0.0000	0.1851	8
21	futex	0.0000	0.0441	8
22	uname	0.0000	0.0018	6
23	wait4	0.0000	0.0059	6
24	exit_group	0.0000	0.0001	5
25	getpriority	0.0000	0.0022	5
26	flock	0.0000	0.0059	4
27	listen	0.0000	0.0000	4
28	getcwd	0.0000	0.0000	3
29	gettid	0.0000	0.1234	3
30	restart_syscall	0.0000	0.0000	2
31	tgkill	0.0000	0.0000	2
32	nanosleep	0.0000	0.0000	2
33	mremap	0.0000	0.0000	2
34	msync	0.0000	0.0000	2

Table 8
Features with different importance for Specificity and Recall for emulated data $p < 0.005$.

	Feature	*p*-value	max(Importance)	Occurrences
1	writev	0.0038	0.1808	35
2	*getdents64*	0.0003	0.0303	34
3	pread64	0.0001	0.0587	34
4	*recvfrom*	0.0000	0.1070	32
5	fstatfs64	0.0002	0.0545	31
6	epoll_create1	0.0000	0.0702	30
7	fcntl64	0.0009	0.0556	30
8	*lseek*	0.0028	0.0548	30
9	*fstatat64*	0.0000	0.1219	29
10	*write*	0.0002	0.1536	29
11	*futex*	0.0000	0.0696	28
12	*read*	0.0005	0.0994	28
13	dup	0.0039	0.0724	27
14	eventfd2	0.0011	0.0304	27
15	*flock*	0.0005	0.0189	26
16	*setsockopt*	0.0000	0.0058	19
17	pipe2	0.0001	0.0069	18
18	statfs64	0.0000	0.0090	17
19	process_vm_readv	0.0006	0.0001	12
20	restart_syscall	0.0000	0.0003	11
21	inotify_add_watch	0.0000	0.0001	7
22	ppoll	0.0000	0.0045	6
23	*exit_group*	0.0000	0.0006	6
24	*tgkill*	0.0000	0.0001	5
25	*nanosleep*	0.0000	0.0000	4
26	inotify_init1	0.0000	0.0001	4
27	msync	0.0000	0.0141	4
28	*gettid*	0.0000	0.0001	4
29	*mremap*	0.0000	0.0070	2
30	uname	0.0000	0.0000	1
31	*sched_getscheduler*	0.0000	0.0000	1
32	*listen*	0.0000	0.0000	1
33	*getcwd*	0.0000	0.0000	1
34	rt_sigsuspend	0.0000	0.0000	1

Table 9
Features with different importance for Specificity and Recall for real device data $p < 0.005$.

	Feature	*p*-value	max(Importance)	Occurrences
1	clock_gettime	0.0019	0.1423	47
2	*recvfrom*	0.0000	0.2250	36
3	rt_sigprocmask	0.0047	0.1716	34
4	*lseek*	0.0000	0.0550	33
5	prctl	0.0007	0.3801	33
6	close	0.0000	0.1269	32
7	*getdents64*	0.0000	0.0601	32
8	getuid32	0.0013	0.0751	32
9	*read*	0.0000	0.4032	32
10	*write*	0.0001	0.1145	32
11	connect	0.0000	0.1851	31
12	gettimeofday	0.0000	0.0755	31
13	mprotect	0.0000	0.3975	30
14	sigaltstack	0.0000	0.0265	30
15	sched_yield	0.0000	0.0248	30
16	*setsockopt*	0.0000	0.0755	29
17	*futex*	0.0000	0.2334	28
18	ioctl	0.0000	0.2304	28
19	socket	0.0000	0.1513	27
20	getpid	0.0014	0.0081	27

(*continued on next page*)

Table 9 (*continued*)

	Feature	*p*-value	max(Importance)	Occurrences
21	getpriority	0.0000	0.0519	27
22	sendmsg	0.0005	0.0432	26
23	*fstatfs64*	0.0004	0.0990	25
24	*flock*	0.0000	0.0371	23
25	wait4	0.0000	0.0220	21
26	*exit_group*	0.0001	0.0003	11
27	*gettid*	0.0000	0.1234	9
28	getegid32	0.0000	0.0002	6
29	getgid32	0.0000	0.0001	5
30	*listen*	0.0000	0.0000	4
31	*tgkill*	0.0000	0.0000	3
32	*getcwd*	0.0000	0.0000	2
33	*sched_getscheduler*	0.0000	0.0000	1
34	*nanosleep*	0.0000	0.0000	1
35	*mremap*	0.0000	0.0000	1

References

Aggarwal, C.C., 2015. Data Mining: The Textbook. Springer.

Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y., 2015. Are your training datasets yet relevant? In: International Symposium on Engineering Secure Software and Systems. Springer, pp. 51–67.

Altmann, A., Toloşi, L., Sander, O., Lengauer, T., et al., 2010. Permutation importance: a corrected feature importance measure. Bioinformatics 26 (10), 1340–1347. doi:10.1093/bioinformatics/btq134.

Alzaylaee, M.K., Yerima, S.Y., Sezer, S., 2017. Emulator vs. real phone: android malware detection using machine learning. In: Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics, pp. 65–72.

Alzaylaee, M.K., Yerima, S.Y., Sezer, S., 2020. DL-Droid: deep learning based android malware detection using real devices. Comput. Secur. 89, 101663.

Amin, M.R., Zaman, M., Hossain, M.S., Atiquzzaman, M., 2016. Behavioral malware detection approaches for android. In: 2016 IEEE International Conference on Communications (ICC), pp. 1–6. doi:10.1109/ICC.2016.7511573.

Android. Run apps on the android emulator. https://developer.android.com/studio/run/emulator; 2021.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C., 2014. DREBIN: effective and explainable detection of android malware in your pocket. In: Ndss, vol. 14, pp. 23–26.

Barbero F., Pendlebury F., Pierazzi F., Cavallaro L.. Transcending transcend: revisiting malware classification with conformal evaluation. arXiv preprint arXiv:201003856 2020.

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32. doi:10.1023/A:1010933404324.

Burguera, I., Zurutuza, U., Nadjm-Tehrani, S., 2011. Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 15–26.

Cai, H., 2020. Assessing and improving malware detection sustainability through app evolution studies. ACM Trans. Softw. Eng. Methodol. (TOSEM) 29 (2), 1–28.

Cai, H., Meng, N., Ryder, B., Yao, D., 2019. DroidCat: effective android malware detection and categorization via app-level profiling. IEEE Trans. Inf. Forensics Secur. 14 (6), 1455–1470. doi:10.1109/TIFS.2018.2879302.

Cai, L., Li, Y., Xiong, Z., 2021. Jowmdroid: android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters. Comput. Secur. 100, 102086.

Casolare, R., De Dominicis, C., Iadarola, G., Martinelli, F., Mercaldo, F., Santone, A., 2021. Dynamic mobile malware detection through system call-based image representation. J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl. 12 (1), 44–63.

Dimjašević, M., Atzeni, S., Ugrina, I., Rakamaric, Z., 2016. Evaluation of android malware detection based on system calls. In: Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics, pp. 1–8.

Fedler, R., Schütte, J., Kulicke, M., 2013. On the effectiveness of malware protection on android. Fraunhofer AISEC 45.

Feng, P., Ma, J., Sun, C., Xu, X., Ma, Y., 2018. A novel dynamic android malware detection system with ensemble learning. IEEE Access 6, 30996–31011. doi:10.1109/ACCESS.2018.2844349.

Gao, H., Cheng, S., Zhang, W., 2021. Gdroid: android malware detection and classification with graph convolutional network. Comput. Secur. 106, 102264.

Google. Google play protect. https://developers.google.com/android/play-protect; 2021.

Gözüaçk, O., Can, F., 2020. Concept learning using one-class classifiers for implicit drift detection in evolving data streams. Artif. Intell. Rev. (0123456789) doi:10.1007/s10462-020-09939-x.

Guerra-Manzanares, A., Bahsi, H., Nõmm, S., 2021. Kronodroid: time-based hybrid-featured dataset for effective android malware detection and characterization. Comput. Secur. 102399.

Guerra-Manzanares, A., Bahsi, H., Nõmm, S., 2019a. Differences in android behavior between real device and emulator: a malware detection perspective. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), pp. 399–404. doi:10.1109/IOTSMS48152.2019.8939268.

Guerra-Manzanares, A., Luckner, M., Bahsi, H., 2022. Android malware concept drift using system calls. Under Rev.

Guerra-Manzanares, A., Nõmm, S., Bahsi, H., 2019b. Time-frame analysis of system calls behavior in machine learning-based mobile malware detection. In: 2019 International Conference on Cyber Security for Emerging Technologies (CSET). IEEE, pp. 1–8.

Guerra-Manzanares, A., Nomm, S., Bahsi, H., 2019c. In-depth feature selection and ranking for automated detection of mobile malware. In: ICISSP, pp. 274–283.

Han, Q., Subrahmanian, V.S., Xiong, Y., 2020. Android malware detection via (somewhat) robust irreversible feature transformations. IEEE Trans. Inf. Forensics Secur. 15, 3511–3525. doi:10.1109/TIFS.2020.2975932.

Iadarola, G., Martinelli, F., Mercaldo, F., Santone, A., et al., 2021. Towards an interpretable deep learning model for mobile malware detection and family identification. Comput. Secur. 105, 102198. doi:10.1016/j.cose.2021.102198.

Irolla, P., Dey, A., 2018. The duplication issue within the Drebin dataset. J. Comput. Virol. Hack. Tech. 14 (3), 245–249.

Jerbi, M., Dagdia, Z.C., Bechikh, S., Said, L.B., 2020. On the use of artificial malicious patterns for android malware detection. Comput. Secur. 92, 101743.

Jordaney, R., Sharad, K., Dash, S.K., Wang, Z., Papini, D., Nouretdinov, I., Cavallaro, L., 2017. Transcend: detecting concept drift in malware classification models. In: 26th {USENIX} Security Symposium ({USENIX} Security 17), pp. 625–642.

Karn, R.R., Kudva, P., Huang, H., Suneja, S., Elfadel, I.M., et al., 2021. Cryptomining detection in container clouds using system calls and explainable machine learning. IEEE Trans. Parallel Distrib. Syst. 32 (3), 674–691. doi:10.1109/TPDS.2020.3029088.

Kaspersky. Mobile security: Android vs. iOS - which one is safer?https://www.kaspersky.com/resource-center/threats/android-vs-iphone-mobile-security; 2020.

Kinkead, M., Millar, S., McLaughlin, N., O'Kane, P., et al., 2021. Towards explainable CNNs for android malware detection. Procedia Comput. Sci. 184 (2019), 959–965. doi:10.1016/j.procs.2021.03.118.

Lei, T., Qin, Z., Wang, Z., Li, Q., Ye, D., 2019. Evedroid: event-aware android malware detection against model degrading for IoT devices. IEEE Internet Things J. 6 (4), 6668–6680. doi:10.1109/JIOT.2019.2909745.

Lin, Y.D., Lai, Y.C., Chen, C.H., Tsai, H.C., 2013. Identifying android malicious repackaged applications by thread-grained system call sequences. Comput. Secur. 39, 340–350.

Lindorfer, M., Neugschwandtner, M., Platzer, C., 2015. MARVIN: efficient and comprehensive mobile app classification through static and dynamic analysis. In: 2015 IEEE 39th Annual Computer Software and Applications Conference, vol. 2. IEEE, pp. 422–433.

Liu, K., Xu, S., Xu, G., Zhang, M., Sun, D., Liu, H., 2020. A review of android malware detection approaches based on machine learning. IEEE Access 8, 124579–124607.

Lu, N., Zhang, G., Lu, J., 2014. Concept drift detection via competence models. Artif. Intell. 209 (1), 11–28. doi:10.1016/j.artint.2014.01.001.

U. du Luxembourg. Androzoo - lists of APKs. https://androzoo.uni.lu/lists; 2021.

, 2005. In: Maimon, O., Rokach, L. (Eds.), Data Mining and Knowledge Discovery Handbook. A Complete Guide for Practitioners and Researchers. Springer, San Francisco, CA, USA.

Margara A., Rabl T.. Definition of Data Streams; Cham: Springer International Publishing. p. 1–4. doi:10.1007/978-3-319-63962-8_188-1.

Molnar C., König G., Herbinger J., Freiesleben T., Dandl S., Scholbeck C.A., Casalicchio G., Grosse-Wentrup M., Bischl B.. Pitfalls to avoid when interpreting machine learning models. arXiv preprint arXiv:200704131 2020.

Mutz, D., Valeur, F., Vigna, G., Kruegel, C., 2006. Anomalous system call detection. ACM Trans. Inf. Syst. Secur. 9 (1), 61–93. doi:10.1145/1127345.1127348.

Narayanan, A., Yang, L., Chen, L., Jinliang, L., 2016. Adaptive and scalable android malware detection through online learning. In: 2016 International Joint Conference on Neural Networks (IJCNN), pp. 2484–2491. doi:10.1109/IJCNN.2016.7727508.

Naval, S., Laxmi, V., Rajarajan, M., Gaur, M.S., Conti, M., 2015. Employing program semantics for malware detection. IEEE Trans. Inf. Forensics Secur. 10 (12), 2591–2604. doi:10.1109/TIFS.2015.2469253.

Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G., 2019. Mamadroid: detecting android malware by building Markov chains of behavioral models (extended version). ACM Trans. Privacy Secur. (TOPS) 22 (2), 1–34.

Palmer D.. Sophisticated android malware spies on smartphones users and runs up their phone bill too. https://www.zdnet.com/article/sophisticated-android-malware-spies-on-smartphones-users-and-runs-up-their-phone-bill-too/; 2018.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L., 2019. {TESSERACT}: eliminating experimental bias in malware classification across space and time. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 729–746.

Ramírez-Gallego, S., Krawczyk, B., García, S., Woźniak, M., Herrera, F., et al., 2017. A survey on data preprocessing for data stream mining: current status and future directions. Neurocomputing 239, 39–57. doi:10.1016/j.neucom.2017.01.078.

Rudin, C., 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. Nat. Mach. Intell. 1 (5), 206–215.

Ruiz-Heras, A., García-Teodoro, P., Sánchez-Casado, L., 2017. ADroid: Anomaly-based detection of malicious events in android platforms. Int. J. Inf. Secur. 16 (4), 371–384. doi:10.1007/s10207-016-0333-1.

Samsung. About knox. https://www.samsungknox.com/en/about-knox; 2021.
Saracino, A., Sgandurra, D., Dini, G., Martinelli, F., 2018. Madam: effective and efficient behavior-based android malware detection and prevention. IEEE Trans. Dependable Secure Comput. 15 (1), 83–97. doi:10.1109/TDSC.2016.2536605.
Sasidharan, S.K., Thomas, C., 2021. ProDroid—An android malware detection framework based on profile hidden Markov model. Pervasive Mob. Comput. 72, 101336.
Scalas, M., Maiorca, D., Mercaldo, F., Visaggio, C.A., Martinelli, F., Giacinto, G., et al., 2019. On the effectiveness of system API-related information for android ransomware detection. Comput. Secur. 86, 168–182. doi:10.1016/j.cose.2019.06.004.
Seiffert, C., Khoshgoftaar, T.M., Van Hulse, J., Napolitano, A., et al., 2010. RUSBoost: a hybrid approach to alleviating class imbalance. IEEE Trans. Syst., Man, Cybern. Part A 40 (1), 185–197. doi:10.1109/TSMCA.2009.2029559.
Sharma, T., Rattan, D., 2021. Malicious application detection in android—A systematic literature review. Comput. Sci. Rev. 40, 100373.
Sophos. Malware goes mobile: Timeline of mobile threats, 2004–2016. https://www.sophos.com/en-us/medialibrary/PDFs/marketing%20material/sophos-threat-infographic-ten-years-malware-mobile-devices.pdf; 2017.
Stachl C., Au Q., Schoedel R., Buschek D., Völkel S., Schuwerk T., Oldemeier M., Ullmann T., Hussmann H., Bischl B., et al. Behavioral patterns in smartphone usage predict big five personality traits2019;.
Statista. Mobile operating system market share worldwide, July 2020–July 2021. https://gs.statcounter.com/os-market-share/mobile/worldwide; 2021.
Surendran, R., Thomas, T., Emmanuel, S., 2020. Gsdroid: graph signal based compact feature representation for android malware detection. Expert Syst. Appl. 159, 113581.
Vidal, J.M., Orozco, A.L.S., Villalba, L.J.G., 2017. Malware detection in mobile devices by analyzing sequences of system calls. World Acad. Sci., Eng.Technol., Int. J. Comput., Electr., Autom., Control Inf. Eng. 11 (5), 594–598.
Vinod, P., Zemmari, A., Conti, M., 2019. A machine learning based approach to detect malicious android apps using discriminant system calls. Future Gener. Comput. Syst. 94, 333–350.
VirusTotal. An update from virustotal. https://blog.virustotal.com/2012/09/an-update-from-virustotal.html; 2012.
Wang, X., Li, C., 2021. Android malware detection through machine learning on kernel task structures. Neurocomputing 435, 126–150.
Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W., 2017. Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 252–276.
Wei, W., Wang, J., Yan, Z., Ding, W., 2022. EPMDroid: efficient and privacy-preserving malware detection based on SGX through data fusion. Inf. Fusion.
Whitwam R.. Android antivirus apps are useless - here's what to do instead. https://www.extremetech.com/computing/104827-android-antivirus-apps-are-useless-heres-what-to-do-instead; 2021.
Xiao, X., Zhang, S., Mercaldo, F., Hu, G., Sangaiah, A.K., 2019. Android malware detection based on system call sequences and LSTM. Multimed. Tools Appl. 78 (4), 3979–3999.
Xu, K., Li, Y., Deng, R., Chen, K., Xu, J., 2019. Droidevolver: self-evolving android malware detection system. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, pp. 47–62.
Yaswant A.. New advanced android malware posing as "system update". https://blog.zimperium.com/new-advanced-android-malware-posing-as-system-update/; 2021.
Zhang, N., Xue, J., Ma, Y., Zhang, R., Liang, T., Tan, Y.a., 2021. Hybrid sequence-based android malware detection using natural language processing. Int. J. Intell. Syst. 36 (10), 5770–5784.
Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., Zhang, M., Yang, M., 2020. Enhancing state-of-the-art classifiers with API semantics to detect evolved android malware. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 757–770.
Zhao, X., Lovreglio, R., Nilsson, D., 2020. Modelling and interpreting pre-evacuation decision-making using machine learning. Autom. Constr. 113, 103140.
Zhou, Y., Jiang, X., 2012. Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy, pp. 95–109. doi:10.1109/SP.2012.16.
Zyblewski, P., Sabourin, R., Woźniak, M., 2021. Preprocessed dynamic classifier ensemble selection for highly imbalanced drifted data streams. Inf. Fusion 66 (June 2020), 138–154. doi:10.1016/j.inffus.2020.09.004.

Alejandro Guerra Manzanares is a Ph.D. candidate at the Center for Digital Forensics and Cyber Security, Department of Software Science, Tallinn University of Technology (Estonia). He received a BA degree in criminology from the Autonomous University of Barcelona (Spain) in 2013, and a BS degree in ICT engineering from the Polytechnic University of Catalonia (Spain) in 2017. In 2018, he received a M.Sc. in cyber security from Tallinn University of Technology (Estonia). His research interests are in the application of machine learning techniques to digital forensics and cyber securityrelated issues, such as mobile malware and IoT botnet detection.

Marcin Luckner works at the Faculty of Mathematics and Information Science, Warsaw University of Technology. He received his Ph.D. degree from the System Research Institute of the Polish Academy of Science in 2010. His primary interests are in pattern recognition and artificial intelligence and their applications, as well as in anomaly detection. He has published about 40 papers. He has managed Research and Development projects for business on behalf of the academic research center, including several projects focused on cyber-security in cell networks and web-spam detection.

Hayretdin Bahsi is a research professor at the Center for Digital Forensics and Cyber Security at Tallinn University of Technology, Estonia. He has two decades of professional and academic experience in cybersecurity. He received his Ph.D. from Sabanc University (Turkey) in 2010. He was involved in many R&D and consultancy projects about cybersecurity as a researcher, consultant, trainer, project manager, and program coordinator at the National Cyber Security Research Institute of Turkey between 2000 and 2014. His research interests include machine learning and its application to cyber security and digital forensic problems.

Update

Computers & Security

Volume 124, Issue , January 2023, Page

DOI: https://doi.org/10.1016/j.cose.2022.102998

Computers & Security 124 (2023) 102998

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Corrigendum

Corrigendum to Concept drift and cross-device behavior: Challenges and implications for effective android malware detection Computers & Security, Volume 120, 102757

Alejandro Guerra-Manzanares*, Marcin Luckner, Hayretdin Bahsi

Tallinn University of Technology and Warsaw University of Technology

The authors regret that the citation corresponding to *Guerra-Manzanares, Luckner, Bahsi, 2022* is incomplete in the original article. We would like to correct the current, incomplete reference from:

Guerra-Manzanares, Luckner, Bahsi, 2022 A. Guerra-Manzanares, M. Luckner, H. Bahsi ***Android malware concept drift using system calls*** *Under Rev (2022)*

to the complete, correct reference, which is detailed as follows:

Guerra-Manzanares, Luckner, Bahsi, 2022 A. Guerra-Manzanares, M. Luckner, H. Bahsi ***Android malware concept drift using system calls: detection, characterization and challenges*** *Expert Syst. Appl. (2022), p. 117200,* 10.1016/j.eswa.2022.117200

Declaration of Competing Interest

None.

DOI of original article: 10.1016/j.cose.2022.102757

* Correspondence author.
E-mail address: alejandro.guerra@taltech.ee (A. Guerra-Manzanares).

https://doi.org/10.1016/j.cose.2022.102998
0167-4048/© 2022 Elsevier Ltd. All rights reserved.