Expert Systems With Applications 206 (2022) 117200

Contents lists available at ScienceDirect

Expert Systems With Applications

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

Android malware concept drift using system calls: Detection, characterization and challenges☆

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

Keywords:
Concept drift
Android malware
System calls
Mobile malware
Malware characterization
Malware detection
Malware evolution
Malware behavior

ABSTRACT

The majority of Android malware detection solutions have focused on the achievement of high performance in old and short snapshots of historical data, which makes them prone to lack the generalization and adaptation capabilities needed to discriminate effectively new malware trends in an extended time span. These approaches analyze the phenomenon from a stationary point of view, neglecting malware evolution and its degenerative impact on detection models as new data emerge, the so-called *concept drift*. This research proposes a novel method to detect and effectively address concept drift in Android malware detection and demonstrates the results in a seven-year-long data set. The proposed solution manages to keep high-performance metrics over a long period of time and minimizes model retraining efforts by using data sets belonging to short periods. Different timestamps are evaluated in the experimental setup and their impact on the detection performance is compared. Additionally, the characterization of concept drift in Android malware is performed by leveraging the inner workings of the proposed solution. In this regard, the discriminatory properties of the important features are analyzed at various time horizons.

1. Introduction

Android operating system (OS) leads the mobile OS market since 2012. At present, over 71% of smartphones are powered by this open-source, highly customizable, and versatile OS (Statista, 2021c). Its ubiquity combined with the open-source nature of the OS, the high prevalence of devices running outdated OS versions, the poor end-user security awareness (e.g., *sideloading* and running over-privileged apps), and the wealth of data stored in these devices make Android users an attractive target for cyber attackers (Rafter, 2021). Despite the security enhancements introduced in the OS regular upgrades, Android is still the most targeted mobile operating system by malware, accounting for over 98% of the mobile cyber attacks (Kaspersky, 2020). These attacks are carried out using a wide variety of attack vectors over the large attack surface exposed by mobile devices (Townsend, 2020). In 2020, an average of 482,579 new Android malware samples were discovered per month (Statista, 2021a). Mostly *trojans* and *adware*, the most predominant malware types nowadays (Statista, 2021b; Chebyshev, 2021). However, the threat landscape is not static but subject to continuous change. For instance, ransomware Trojans were the most predominant type of Trojans in 2017, whereas, in 2020, banking Trojans were significantly more prevalent (Unuchek, 2018; Chebyshev, 2021). New malware trends have emerged over time and more sophisticated malware samples have been discovered, evidencing the non-stationary attribute of the threat, featured by constant evolution and innovation (Microsoft, 2020). As a result, the figures regarding detected mobile malware may only reflect a small portion of the total malware *in the wild*, with new and more sophisticated malware variants remaining undetected (Broersma, 2020).

Notwithstanding the dynamic nature of the phenomenon, the specialized research has overlooked the evolution and changes in Android malware over time. In this regard, despite the vast body of literature available on the optimization of detection methods, the change in malware features over time and its degenerative impact on the machine learning-based detection models, the so-called *concept drift*, has not been explored thoroughly. Most machine learning-based models for malware detection are based on the assumption of *consistent* data, thus requiring the properties of the testing data distribution to approximately

☆ The code (and data) in this article has been certified as Reproducible by the CodeOcean: https://codeocean.com/capsule/6162888/tree/v1. More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail addresses: alejandro.guerra@taltech.ee (A. Guerra-Manzanares), mluckner@mini.pw.edu.pl (M. Luckner), hayretdin.bahsi@taltech.ee (H. Bahsi).

https://doi.org/10.1016/j.eswa.2022.117200
Received 12 September 2021; Received in revised form 7 April 2022; Accepted 7 April 2022
Available online 21 April 2022

match the characteristics of the training data distribution. However, due to the constant battle between attackers and defenders, malware evolve to exploit new vulnerabilities and improve its hiding capabilities in response to enhanced defenses, thus generating new malware variants that may use a distinct attack vector and behave differently but pursue the same ends. As a result, the incoming data distribution may diverge significantly from the model's original training distribution, thus, generating concept drift and, consequently, harming the model's performance over time. Despite that, the myriad of machine learning-based solutions proposed for Android malware detection are generally built, validated, and tested using relatively small data sets, collected in short time-frames, generally far from the present time. For instance, the most used Android data sets in research studies were gathered between 2010 and 2012 (i.e., MalGenome (Zhou & Jiang, 2012) and Drebin (Arp et al., 2014)), a decade ago, when malware capabilities and characteristics were significantly distinct to the present ones (e.g., the first Android ransomware was detected in 2014 and had its major outbreak in 2016, ergo not represented in these data sets). Even so, they are still being used in recent studies as the main malware references and frequently as the only malware source (Zhao et al., 2021; Reddy et al., 2021). Therefore, most of the proposed solutions have been optimized for malware detection at specific snapshots of the Android history, built and tested on *static* and *partial* data sets containing specific malware trends at a particular time, thus lacking the generalization capabilities needed to address the constant evolution of Android malware, its non-stationary character, and its challenges.

The majority of studies on Android malware detection based on machine learning techniques lack *historical coherence*, mainly caused by splitting randomly the available data into the *testing* and *training* sets without respecting the *historical timeline*. More specifically, the testing set should always be composed of *future* or *posterior* data regarding the training set (Allix et al., 2015; Arp et al., 2020). In these solutions, concept drift is neglected and the optimized models provided by such approaches yield *significantly biased* and *historically incoherent* results, a critical issue when it comes to real Android malware detection (Allix et al., 2015; Pendlebury et al., 2019; Arp et al., 2020). In this regard, a critical challenge to deal with concept drift is how to locate the samples within the Android historical timeline in a reliable way. Our research uses and compares two timestamping approaches that may result in good approximations in the search for temporal accuracy. Furthermore, in production setups, the number of samples processed may rarely be evenly split between the classes. For instance, a malware outbreak may cause the processed data to be imbalanced towards the positive class (i.e., malware label), whereas, in the absence of an outbreak, the majority of the new samples should be benign. This fact is usually not considered by the related research, assuming and working with evenly split data sets. Our study addresses the concept drift issue in the presence of imbalanced data, providing a more realistic scenario and reliable results. Thus, the proposed solution can effectively handle the additional challenge of imbalanced data towards any of the classes. Lastly, the small number of studies that proposed solutions considering Android malware concept drift issues did not provide any insights on the changes in the data, that is, the characterization of concept drift. This is a distinctive point of this study, which not only addresses concept drift but also aims to understand the evolution of features over time in the analyzed context. The characterization of concept drift allows understanding the direction of changes, enabling the expansion and enhancement of the knowledge about the threat while providing useful insights to improve the detection systems.

1.1. Novelty and contribution

In this paper, we address significant research gaps in Android malware detection studies, by exploring, addressing, and characterizing the phenomenon of concept drift in Android malware detection using dynamic features (i.e., system calls) on imbalanced data sets. Despite the existence of methods to detect *drifting* data (Yang et al., 2021; Jordaney et al., 2017) and a large body of research regarding malware detection (Liu et al., 2020), just a few studies related to Android malware detection have considered concept drift in their detection solutions (Xu et al., 2019; Onwuzurike et al., 2019; Cai et al., 2018; Hu et al., 2017; Narayanan et al., 2017). All these studies used API calls, a static feature sensitive to code obfuscation and encryption techniques (Kaspersky, 2021). Unlike these previous works, our study uses system calls, runtime data features that are robust to code obfuscation and encryption methods. System calls enable us to capture the real behavior of the app and are the most used dynamic features for Android malware detection (Liu et al., 2020). Besides, no previous study in the field has provided characterization of concept drift nor compared the performance of distinct timestamps when dealing with emerging concept drift, which are unique contributions of this research. Lastly, the usage of the *KronoDroid* data set (Guerra-Manzanares et al., 2021) enables us to overcome the limitations of other data sets and explore concept drift as it provides labeled and timestamped data for the whole Android history (i.e., 2008–2020). In this regard, our analysis spans a seven-year-long continuous time frame, whereas previous works used either discontinued data sets (Hu et al., 2017; Narayanan et al., 2017) or encompassed a shorter time period (Onwuzurike et al., 2019; Xu et al., 2019; Cai et al., 2018). Our workflow is composed of three stages where we analyze and demonstrate the presence of concept drift in Android malware detection, propose a solution to handle it, and characterize its behavior. Furthermore, the performance of the proposed solution is compared with the state-of-the-art solutions, outperforming all of them.

The main novel points of this research are: (1) the usage of system calls as features in an Android concept drift-related study, (2) the proposed solution, which addresses the impact of concept drift on the classification model, enabling the detection system to sustain high detection performance over an extended period of time, even when imbalanced data are present, (3) the characterization of concept drift, which allows the overall understanding of its behavior and direction, and (4) the evaluation of distinct timestamping approaches to effectively deal with concept drift issues.

The paper is structured as follows: Section 2 provides background information and a summary of related research studies in the field. Section 3 explains the methodology followed in this study and introduces the proposed solution to address and characterize concept drift in Android malware detection. Section 4 describes the results of the experimentation using the proposed solution and the main outcomes of this research. Section 5 provides a discussion of the main results and outlines future work. Finally, Section 6 summarizes the study.

2. Background information and related work

2.1. Background information

A *data stream* can be defined as a *countably infinite sequence of elements* that become available over time (Margara & Rabl, 2018). Due to their cumulative, continuous, rapid, and evolving nature, data streams, usually referenced under the umbrella term of *big data*, pose a variety of challenges such as one-pass constraint, concept drift, resource restriction, and massive-valued features (Aggarwal, 2015). Despite not facing all these challenges, Android malware detection shows issues related to data stream processing such as large data volume, continuous release of apps, and evolving data. Consequently, Android malware concept drift may be effectively handled when tackled from a data stream perspective.

This paper performs a novel attempt to demonstrate, handle and characterize Android data concept drift using system calls as model features and from a data stream perspective. For this purpose, state-of-the-art algorithms are leveraged and customized in our study to tackle effectively concept drift issues in Android malware detection. The following paragraphs introduce their basics.

Gözüaçık and Can (2020) proposed an implicit (i.e., unsupervised) learning algorithm called *One-Class Drift Detector* (OCDD), which uses a one-class learner with a sliding window to detect concept drift. As the data analyzed in our work do not show normal distribution characteristics, it was not possible to apply statistical analysis to detect changes in the features over time. Therefore, the *OCDD*'s central idea was leveraged to analyze the impact of concept drift in the observed data. This approach was implemented in our work using the *Isolation Forest* algorithm. Isolation Forest (iForest) is an anomaly detection technique proposed by Liu et al. (2012). The algorithm uses binary decision trees to detect anomalous data based on path length. More precisely, in randomly generated binary trees, where instances are recursively partitioned, these trees produce noticeable shorter paths for anomalies. In the regions occupied by anomalies fewer partitions are observed (i.e., shorter paths in the tree structure). For a specific sample, the received path length is compared to the average path length of unsuccessful searches in the binary search tree to obtain a universal anomaly measure. This measure is used by the *iForest* algorithm to detect anomalies based on the results obtained on several trees.

Zyblewski et al. (2021) proposed a novel framework employing stratified bagging to train base classifiers, integrating data preprocessing, and using dynamic ensemble selection methods for imbalanced data stream classification. The experimental results showed that dynamic ensemble selection coupled with data pre-processing could outperform state-of-art methods for highly imbalanced data streams. In our work, we analyze Android app data in an extended time frame, where the ratio of malware to benign applications varies over time, thus causing imbalanced data issues. The high-level framework proposed by Zyblewski et al. (2021) was the point of departure of our algorithm and its application to the Android data issue. The original algorithm, designed for data streams split into *data chunks*, uses a pool of classifiers trained on past data to classify new data samples in upcoming data chunks. The best combination of classifiers for the new data (i.e., an ensemble of classifiers) are dynamically selected using the previous data chunk. The classifier pool is constantly purged and updated with new classifiers trained on data from each new chunk. Concept drift is addressed by the constant update of the pool of classifiers, while the ensemble selection mechanism enhances the classification results. The original algorithm was modified in our work to address the particularities of Android data and enhance the detection results, as reported in Section 3.2.2 and Section 4.3.

2.2. Related work

A large variety of malware detection approaches have been proposed since the early years of Android OS (Liu et al., 2020). Most of these solutions were optimized and tested on static snapshots of Android malware historical data, using *old* and short-time data sets (Zhou & Jiang, 2012; Arp et al., 2014). As a result, these solutions disregard the evolution and change in data over time and its potentially harmful effect on the detection system's performance.

The phenomenon of *concept drift*, where the statistical characteristics of the incoming data change over time, is visible in long-term Android data (Ramirez-Gallego et al., 2017). Neglecting the changes in malware data patterns over time has a significant detrimental impact on the classifiers' performance, as models built using *old* data tend to make poor and ambiguous decisions when tested on *new* data (Jordaney et al., 2017). Thus, adapting to the rapid evolution of Android malware is critical for an effective detection system (Hu et al., 2017). Consequently, concept drift should be considered in all ML-based detection methods aiming to provide high and reliable performance over time. However, even though an increasing number of studies recognize the importance of addressing concept drift in Android malware detection models (Suarez-Tangil et al., 2017; Hu et al., 2017), only a reduced number of studies have taken its impact into account. These solutions are briefly discussed in the following paragraphs.

Hu et al. (2017) proposed the usage of an ensemble of classifiers to analyze data within a sliding window and dynamic adjustments to address concept drift on static features (i.e., permissions, actions, and selected API calls). The authors reported 96% accuracy in a relatively small, imbalanced, and discontinued in time data set. The time range of the data set and the source of the majority of the samples are not reported, which generates concerns about the results and the actual existence of concept drift in the data, which is assumed but not proved. In our study, the first stage aims to prove the existence of drifting data within the data set. After, the proposed solution is tested on a large and time-extended data set addressing imbalanced data issues.

In *DroidOL* (Narayanan et al., 2016), online algorithms were used to deal with concept drift. The solution was built and tested on a static-featured data set spanning 8 months (i.e., using inter-procedural control-flow graphs as features). The authors reported 84% accuracy on a balanced data set. Even though the usage of an online learning algorithm can have benefits over batch learning algorithms for concept drift handling purposes, it is questionable that the time span of the data set might be too short for the emergence of concept drift, which was assumed but not proved in the study. The usage of online algorithms for concept drift handling was enhanced in *DroidEvolver* (Xu et al., 2019), where a pool of 5 online classifiers was used to build the detection system. Raw API calls were extracted from the source code and used to generate the input vector. After an initialization step, the pool of classifiers was used to label every new instance. Next, based on a *drift* indicator, the detection models and feature sets were updated, if needed. The update of the feature sets (i.e., done incrementally by including all the new API calls) and the update of aging classifiers are intended to provide resilience against concept drift. Besides, the usage of a pool of classifiers aims to avoid the bias of a single classifier and generate more reliable detection results. The usage of a pool of classifiers to improve the detection performance is also leveraged in our proposed solution. However, the distinctive elements of our proposed solution that uses enhanced classifier dynamics (i.e., *dynamic ensemble selection*) instead of online learning algorithms which require constant retraining, the usage of dynamic features instead of static features, and a reduced and stable feature set as opposed to the incremental cost of an ever-growing feature set, provide increased and more stable long-term detection performance, as it is shown in Section 5.

TRANSCEND (Jordaney et al., 2017) framework used statistical metrics to identify when a classification model was consistently misclassifying new data, thus signaling the emergence of concept drift and the aging of the model. In the study, the framework was merely used as a *drift* indicator, not proposing any solution to handle concept drift distinct from data relabeling and model retraining once *drifting* was identified. The limitations of this work were addressed in *TRANSCENDENT* (Barbero et al., 2020), a model agnostic rejection framework composed of conformal evaluators. In the experimental setup, the framework helped to extend the effectiveness of *Drebin* classifier (Arp et al., 2014), doubling its lifespan, keeping an F1 score over 80% for two years.

MaMaDroid (Onwuzurike et al., 2019) used static analysis to extract sequences of API calls from the call flow graph, and abstracted each API call to three distinct higher abstraction levels (i.e., family, package, or class). The sequences of abstracted API calls were used to build the feature vectors, represented using Markov chains. Concept drift was tackled under the assumption that the representation of the sequences of API calls in the higher level of abstraction changes less over time than the raw API calls, thus being more robust and resilient than the approaches that use directly the API calls, such as DroidEvolver (Xu et al., 2019), which need constant retraining, especially after new Android API releases and API changes, to address concept drift.

API-Graph (Zhang et al., 2020), aims to enhance API call-based detection systems by leveraging API semantics and similar API usages among malware. The framework builds an API-level relation graph by extracting entities such as APIs and permissions and establishing their

relations into five *meta*-categories. The authors showed that the usage of *API-Graph* may improve the generalization capabilities and robustness against performance decay of existing solutions such as *MaMaDroid* and *DroidEvolver*.

Event groups semantics were employed in *EveDroid* (Lei et al., 2019). In this study, API call graphs were used in conjunction with event grouping techniques (i.e., clustering) to train neural network models for Android malware detection in IoT devices. The ability of neural networks to extract semantics from the input features was leveraged to provide robustness against performance decay. Even though the study results reported high performance, the two data sets used encompassed short and discontinued time-frames (i.e., 2013–2014 and 2017–2018). More importantly, the models were trained using random selection of samples, thus disregarding the chronological order and mixing the data in both the training and testing sets. Furthermore, legitimate and malware samples did not belong to the same exact time-frames (e.g., legitimate samples were from 2014 and 2018, whereas malware was from 2013, 2014, 2017 and 2018-Q1). As a result, the robustness of the solution against concept drift poses severe doubts.

Even though API calls are considered static features in malware analysis, they can also be collected dynamically (i.e., at run-time). This was the approach taken in *DroidSpan* (Cai, 2020) to overcome the limitations of static features (e.g., code obfuscation and encryption). In this work, particular API calls, related to sensitive data access and operations, invoked at run-time were collected. The data was used to generate an input vector composed of 52 features. Although the solution aimed for long-term stability and reported better results than MaMaDroid, it did not provide any adaptive measures to maintain long-term performance (i.e., the model is never updated), making it prone to concept drift-related performance decay over time. A similar approach was used in Fu & Cai (2019) and Cai et al. (2018).

Pendlebury et al. (2019) emphasize the significant impact of *spatial* and *temporal bias* in the Android malware detection research literature, which has consistently yielded not representative and overly inflated unrealistic performances. More specifically, *temporal bias* is caused by incorrect temporal splits on the training and testing sets (i.e., neglecting concept drift and providing *impossible* temporal configurations), whereas *spatial bias* is caused by unrealistic distributions on the training and testing sets (e.g., class-balanced data sets). The proposed tool, *TESSERACT*, did not provide any novel strategy to tackle concept drift (i.e., incremental retraining, active learning, or classification with rejection), but aimed to assess the robustness of other solutions to performance decay by removing the spatio-temporal bias from the evaluations, thus revealing their true performance. The framework was tested with samples from *AndroZoo* data set (Université du Luxembourg, 2021) which were temporally located using the *dex date* timestamp, suggested by the authors as the most reliable timestamp and comparable to VirusTotal's *first seen* timestamp. The *dex date* timestamp informs about the compilation date of the *apk* (i.e., app's archive file). However, according to Université du Luxembourg (2021), the *dex date* timestamp is no longer a reliable timestamp as the vast majority of the apps released nowadays have a 1980 *dex date*, thus being not usable for temporal location purposes. Our proposed solution uses a distinct and more reliable timestamp, the *last modification*, which is compared in this study to VirusTotal's *first seen* timestamp, taking advantage of the timestamps provided by the *KronoDroid* data set.

As can be observed, the vast majority of proposed solutions focused on static features which suffer from proven limitations to counter-detection techniques, such as code-obfuscation, packing, and encryption (Aghakhani et al., 2020). Although more complex and time-consuming to acquire, dynamic features are more robust against deception techniques, thus preferred for an effective detection solution. System calls, the most used dynamic features for Android malware detection are used in our study (Liu et al., 2020). Besides, none of the studies that dealt with concept drift provided further exploration of the phenomenon (i.e., characterization). In this regard, the characterization of concept drift may provide a better understanding of the phenomenon and assist malware specialists to understand changes in malware over time, detect trends and build more effective detection systems while expanding the knowledge regarding Android malware behavior. An effective solution for Android malware detection should consider the emergence of concept drift and have adaptive skills to change its inner structure according to the detected data changes. In addition, other challenges overlooked by the specialized research should be addressed such as imbalanced data sets and high-dimensionality problems (i.e., ever-growing feature sets). All these issues are considered in this study and addressed by the proposed solution to build a long-lasting, effective, and robust Android malware detection system.

3. Material and methods

3.1. Data set

The data set used in this research is *KronoDroid* (Guerra-Manzanares et al., 2021). This data set is composed of two device-related sub-datasets (i.e., emulator and real device data sets) containing both benign and malware samples. Every sample in the data set (i.e., Android *apps*) is labeled and characterized by 289 dynamic features and 200 static features (i.e., including timestamps). The data set provides data covering the whole historical timeline of Android OS, from 2008 to 2020. In this regard, four timestamp features provide the possible *temporal context* of the apps, which makes this data set the only publicly available Android data source suitable for the investigation of concept drift in Android malware. For this research, just the real device data set, composed of 78,137 samples, was used due to its larger size (i.e., 41,382 malware samples belonging to 240 malware families and 36,755 benign apps).

To analyze the phenomenon of concept drift in Android malware detection from a dynamic perspective, just the dynamic features provided by the *KronoDroid* data set were used along with the class labels and timestamp features. The dynamic features provided by the data set are *kernel* or *system* calls (*syscalls* for short). Syscalls are the most used dynamic features in Android malware detection systems (Liu et al., 2020). The feature set is composed of 288 numeric variables, whose values provide the absolute frequency of each system call invoked by the app during the execution time (i.e., 1 min, no user interaction). Regarding the label, *Kronodroid* provides two labels for each data sample: the *hard label* and the *soft label*. The only difference between them is the labeling approach used, as they are both defined as non-probabilistic binary labels (i.e., $y \in \{0,1\}$). More precisely, the *soft label* relies on the data source to assign the class to an instance without any further verification. The *hard label* is based on a stricter labeling technique as it imputes the class of a sample according to the detection results from *VirusTotal*'s antivirus scanner disregarding the data source, as explained in Guerra-Manzanares et al. (2021). Due to this stricter class imputation, in this study, the *hard label* was selected as it increases the certainty about the class of the samples (i.e., benign or malware), thus increasing the reliability of the concept drift analysis results.

Finally, as the study of concept drift explores the evolution of data over time, the timestamp used to date the samples becomes of critical importance. In this regard, from the four possible timestamps, the *last modification* and *first seen* timestamps were selected as they provide more data coverage, reliability, and accurate location of the apps within the Android historical timeline. The *last modification* timestamp provides the date of the most recent modification of any file inside the app, while the

first seen timestamp reports about the date when the app was submitted for the first time to *VirusTotal*'s antivirus engine.

As a result, 78,137 Android apps described by 288 system calls numeric variables, the label, and two distinct timestamps were used as input data for this research. The following paragraphs explain the workflow followed to effectively detect, handle and characterize concept drift across Android OS history.

3.2. Workflow

The methodology used in this study is composed of three sequential stages related to concept drift: *detection, handling* and *characterization*. At each stage, different techniques are used, which are explained in detail in the following sections. Briefly, the *concept drift detection* stage aims to prove the existence of concept drift in the data using state-of-the-art techniques. Upon confirmation of the presence of concept drift, the following stages are set to address the related issues. In the *concept drift handling* stage, distinct techniques are used to effectively handle concept drift over time. The algorithm used in this stage to reduce the classification error caused by concept drift provides also the information used for *concept drift characterization*, the last stage of the workflow. It is worth emphasizing that the *handling* and *characterization* stages use a distinct methodology than the *detection* stage as the only goal of the latter is to prove the need for concept drift handling techniques in the Android malware detection case.

3.2.1. Concept drift detection

The concept drift detection process consists of three sequential phases, namely *data pre-processing*, *feature selection*, and *drift detection*. The whole process is depicted in Fig. 1 and explained as follows.

The *initial* set of features was pre-processed using a sequential procedure to remove features that might disturb or not provide any significant input data to the machine learning algorithm (i.e., null-valued and redundant features). The outcome was a *refined* set of features obtained after performing the following steps.

1. *Feature variance analysis*: Homogeneous (i.e., zero variance) and zero-valued features (i.e., not invoked by any app) were removed from the initial feature set.
2. *Feature correlation analysis*: *Pearson's correlation coefficient (r)* was calculated pairwise for all variables. Highly correlated features were removed from the feature set.
3. *Feature distribution analysis*: statistical normality tests were performed on all features that remained in the feature set after the previous steps.

The goal of these sequential steps was to select relevant features, remove redundant variables and assess the data distribution to select the best techniques for the posterior steps.

Concept drift detection can be formulated as follows. Each new observation is represented by $c_i = (x_i, y_i)$, where $x_i = (x_i^1, x_i^2, .., x_i^n) \in \mathbf{X}$ is the feature vector and $y_i \in \mathbf{Y}$ is the target label. Given two data chunks (i.e., probes of the same size or collected during similar periods) taken from distributions F and F', respectively, the following applies. For the null hypothesis, H_0, that F and F' are identical, the aim is, as stressed by Lu et al. (2014), to refuse H_0, and identify some local regions of the problem space where H_0 does not hold, and quantify the difference between F and F'. However, as shown by Mutz et al. (2006) and Ruiz-Heras et al. (2017), Android system calls' distributions must be modeled and Gaussian distributions cannot be assumed. For this reason, the analysis of F and F' distributions is hindered.

In addition, the difference between F and F' might be statistically important but not change the decision of the malware detection system. More precisely, concept drift changes the classification decision by changing the conditional probability $p(y|\mathbf{X})$. However, it is possible to change the probability $p(\mathbf{X})$ without changing $p(y|\mathbf{X})$. In such a case, the change in the feature space does not affect the classification outcomes and is called *feature drift* or *virtual drift* (Ramirez-Gallego et al., 2017). The main difference is that under *real concept drift*, the restructuring of the learning model is required, whereas, under virtual drift, the old knowledge is extended with additional data from the same environment (Gözüaçık & Can, 2020; Elwell & Polikar, 2011). Consequently, concept drift is only observed if and only if the input data changes affect the classification results (i.e., the conditional probability $p(y|\mathbf{X})$ is affected).

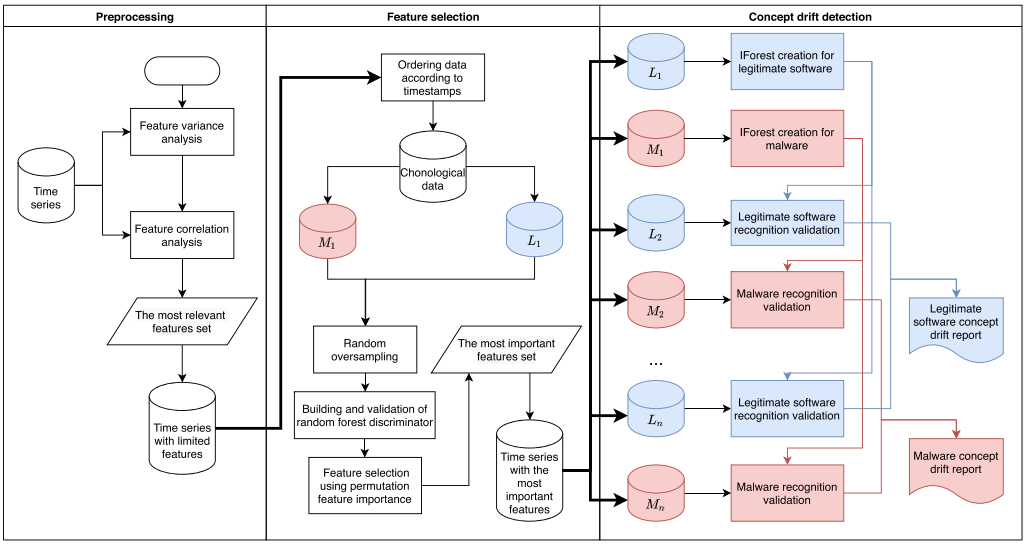


Fig. 1. Schema of the concept drift detection stage.

To assess the existence of concept drift, the data set must be chronologically ordered and divided into consecutive periods. For n periods, two series of subsets are generated. In the first series, the set M_i consists of records describing malware and labeled with the timestamp (i.e., *last modification* or *first seen*) from the i-th period. The set L_i, defined for the second series, is labeled analogously but composed of just benign apps.

Then, the following procedure was performed to define the most discriminatory features for the first period (i.e., feature selection).

The data from the first period $M_1 \cup L_1$ was balanced using a random oversampling method to avoid over-representation of L_1 or M_1 (Seiffert et al., 2010). Next, *Random Forest* (RF) algorithm, a decision tree-based ensemble classification algorithm proposed by Breiman (2001), was induced to discriminate between L_1 and M_1. The rationale behind the selection of RF algorithm is the superior performance shown by this algorithm over other classification algorithms in analogous cases on previous research (Guerra-Manzanares et al., 2019a; Guerra-Manzanares et al., 2019b; Guerra-Manzanares et al., 2019c). The most relevant features of this initial classifier were selected using the *permutation feature importance* technique applied to the training data (see Section 3.2.3 for further details). This enabled the selection of the most relevant features for this baseline classifier. In this regard, only the features with positive mean importance for the model were selected, defining the *important* feature set.

If the initial-period classification model provides high performance (e.g. accuracy $>$ 95%), it is reasonable to assume that the selected features are significant to recognize classes L_1 and M_1. However, an open challenge is whether the same features can be successfully used to recognize classes L_i and M_i for $n \geq i > 1$. To address this issue, One-Class Drift Detection Models (Gözüaçık & Can, 2020) were used to analyze the impact of concept drift in the generated series. Therefore, based on the fact that the selected *important* features could be used successfully for the classification task on the $M_1 \cup L_1$ set, one-class anomaly detectors were induced using L_1 and M_1 separately, to assess data drift. The Isolation Forest (iForest) algorithm was used to induce the anomaly detection models. This approach allows the observer to analyze the concept drift for malware and legitimate software in a more controlled way eliminating the class relations influence.

Next, the iForest detection models induced were tested separately with L_i and M_i sets, where $n \geq i > 1$, described by the *important* feature set to calculate the ratio of samples recognized as part of the modeled class in each period (i.e., negative detection rate). The resulting ratios enable us to assess the emergence of concept drift on the data over time. More specifically, if the initially selected *important* features are not able to describe the modeled class effectively in a given period, the ratio will drop, thus indicating data drift. The observed drops may be qualified as concept drift as the ratio of correctly recognized observations decreases. Therefore, the change affects the classification results.

The workflow performed for concept drift detection is described in Fig. 1 for further reference.

The emergence of concept drift in the data requires the implementation of an adaptive detection solution capable of handling it effectively. This issue is addressed in the next section.

3.2.2. Concept drift handling

The proposed solution to handle concept drift in Android malware is a modification of the algorithm proposed by Zyblewski et al. (2021). Although the original algorithm may provide good performance, the modifications performed address Android concept drift particularities, thus boosting the detection performance. The suggested adaptations are detailed in the following paragraphs.

The first modification yields *a complete pool of classifiers available from the initial stage*. At the initial stage, the original algorithm starts with a single classifier in the pool. For every subsequent data chunk processed, a new classifier is added to the pool until the pool is full. Only then the full pool is available for prediction. For instance, if the size of the pool of classifiers is set to contain 10 elements (i.e., S parameter), then 10 chunks must be processed to have available the full pool of classifiers. Thus, these 10 *initial* data chunks are all tested with an incomplete pool of classifiers. After the pool is completed, in the subsequent data chunks (i.e., from the 11th chunk), each classifier is tested and the worst performer is eliminated (i.e., pool purge). Then, a new classifier trained on the new data chunk is added (i.e., pool update). Therefore, only after the initial S chunks are processed, the pool is guaranteed to be always composed of S elements. Our experimentation has shown that it may be beneficial to generate the full pool from the initialization phase (i.e., first chunk), as the usage of a larger variety of classifiers to process the initial data chunks enhances the performance on the initial stages. As a result, the proposed solution eliminates the delay on the generation of the full pool by splitting the initial chunk into S ordered subchunks and training a classifier on each subset, thus generating the full pool in the initial step. Then, after the first chunk is processed (i.e., initialization stage), the whole pool, containing S elements, is used to process all the subsequent data chunks (i.e., from the 2nd chunk). The classifier pool is updated using the same *purge-update* mechanism as the original solution when a new data chunk is processed.

The second modification performs a *refinement on the predictions using a supportive anomaly detection model*. The original solution produces its predictions according to the classifiers' pool assignment. The proposed modification adds a refinement step to the prediction process by using an anomaly detection model, induced on a subset of data (i.e., benign data), aiming to improve the predictions provided by the dynamic ensemble selection of classifiers. The rationale behind the addition of an anomaly detection model trained on legitimate data is due to the observation of more concept drift resilient features in this subset of data during our experimentation, as shown in Fig. 3 and Fig. 9a. More specifically, concept drift in legitimate data appears to be less significant than in malware data, showing a more robust performance over time and keeping as *important* a broader and more consistent subset of features. Therefore, the addition of an anomaly detector trained on just benign data can help in dubious cases where the prediction probability of the classifier pool may not give enough confidence to the assigned label (e.g., $p(y|x) \approx 0.5$). In such cases, the usage of the additional knowledge from the anomaly detection model might help. However, the benefits of this modification heavily rely on the prediction probabilities output by the classification model and the specific set of rules and thresholds used by the particular implementation.

As an example, in our experimental setup, a *reassignment rule* was applied for borderline predictions as follows: if $0.55 \geq p_{pool}(benign|x) \geq 0.5$, where x refers to a given sample and p_{pool} to a particular prediction probability from the pool of classifiers, then the sample was assigned to the class suggested by the anomaly model. In any other case, the class assigned to x was the one suggested by p_{pool}. This simple rule yielded from 1% to 4% improvement in detection performance in some time periods, especially in the initial chunks. It is worth mentioning that, in our experimental setup, no rule optimization was performed, thus there is room for improvement in this regard by the particular implementations of the proposed solution.

The proposed solution is provided in Algorithm 1 and Algorithm 2. For the sake of comprehension and similarly to the original formulation by Zyblewski et al. (2021), the algorithm is split into *training* and *prediction* phases. However, as these phases are applied sequentially for each data chunk, they should be embedded in the actual implementation of the solution. More specifically, Algorithm 1 provides the *pseudo-code* implementation of the *training* phase while Algorithm 2 defines the steps performed in the *prediction* phase.

Algorithm 1: Training phase of the proposed framework

Input:

$Stream$ - Data stream

S - Fixed size of the classifier pool

$\Pi \leftarrow \varnothing$ - Pool of classifiers (initially empty)

$\lambda \leftarrow -1$ - Sample size of the anomaly detector

Symbols:

DS_k - Data chunk

Ψ_k - Bagging classifier

L_k - Legitimate data portion of the data chunk

Φ_k - Anomaly detector

1 **foreach** k, DS ***in*** $Stream$ **do**
2 **if** $k == 0$ **then** // first data chunk
3 $IDS \leftarrow splitInitialDataset(DS_k, |\pi|)$ // split data chunk
4 **for** $i \leftarrow 0$ *to* $S-1$ **do**
5 $\Psi_k \leftarrow trainClassifier(IDS_i)$ // train classifier
6 $\Pi \leftarrow \Psi_k$ // add classifier to the pool
7 **end**
8 $\Phi_k \leftarrow trainAnomalyDetector(L_k, \lambda)$ // train anomaly
9 **else** // rest of the data
10 $\Pi \leftarrow pruneWorstClassifier(\Pi)$ // purge pool
11 $\Psi_k \leftarrow trainClassifier(DS_k)$
12 $\Pi \leftarrow \Psi_k$
13 $\Phi_k \leftarrow trainAnomalyDetector(L_k, \lambda)$
14 **end**
15 **end**

Algorithm 2: Prediction phase of the proposed framework

Input:

$Stream$ - Data stream

Π - Pool of classifiers

Φ_k - Anomaly detector

Symbols:

DS_k - Data chunk

$y_{k_{pred}}$ - Predicted labels for the samples in the current chunk

Π_{Dk} - Ensemble of classifiers selected using a DES algorithm

$DSEL$ - Dynamic ensemble selection data set

1 **foreach** k, DS ***in*** $Stream$ **do**
2 **if** $k == 0$ **then** // first data chunk
3 $DSEL \leftarrow preprocess(DS_k)$ // store DSEL for next step
4 **else** // rest of data chunks
5 $\Pi_{Dk} \leftarrow dynamicSelection(\Pi, DSEL, DS_k)$ // DES step
6 $y_{k_{pred}} \leftarrow predict(DS_k, \Pi_{Dk})$ // prediction step
7 $y_{k_{pred}} \leftarrow anomalyDetector(y_{k_{pred}}, \Phi_k)$ // refinement step
8 $DSEL \leftarrow preprocess(DS_k)$
9 **end**
10 **end**

Before providing a detailed explanation of the whole system, some general considerations must be addressed. They are provided as follows:

- For the initialization of the system, the following hyper-parameters must be selected: the size of the classifier pool (i.e., $|\pi|$, S), the chunk size (i.e., n, sample size) and the anomaly detector sample size (i.e., -1 by default, using all L_{k-1} data).
- The first data chunk (i.e., $k = 0$) is used only for training purposes and not for testing purposes. All subsequent data chunks are processed sequentially, applying first the *predictive* phase and then the *training* phase. This is a modification to the original algorithm in which distinct steps are performed for the first and second chunks.
- As all data chunks are assumed to be imbalanced, a data balancing technique (e.g., random oversampling) is applied to the learning set every time a new classifier is generated (i.e., inside the *trainClassifier ()* function of Algorithm 1).

Furthermore, as can be noticed in Algorithm 1 and Algorithm 2, the proposed solution requires the selection of a few hyper-parameters for its deployment. They are described in the following items:

- *Chunk size (n)*: the number of samples that compose each data chunk. It is a hyper-parameter that is not included in the algorithm description. However, it is assumed that all chunks have the same size and are processed in chronological order. Note that the chunk size may affect the concept drift detection capabilities of the system.
- *Classifier pool size (S)*: the number of classifiers in the pool. It may impact the concept drift adaptation capabilities of the system.
- *Anomaly detector's sample size*: the size of the data set that is used to induce the anomaly detection model. In this regard, the whole legitimate set of the new chunk can be used, a portion of it or even a bigger set by generating a cumulative data set from consecutive chunks. It may affect the accuracy of the predictions.
- *Dynamic Ensemble Selection (DES) algorithm*: the algorithm used to select the best ensemble of classifiers from the pool to predict the class of the new data. Note that the usage of distinct *DES* algorithms can provide significantly different performances, thus it is a very important hyper-parameter of the system. The *DSEL* may also have a significant impact on the prediction performance but it is not a hyper-parameter as it strictly depends on data.
- *Data set balancing method*: the technique used to balance the classes in the training set may have an impact on the detection accuracy. In this regard, different over-sampling and under-sampling techniques might be used.

Despite that the selection of hyper-parameters might have a significant impact on the performance of the detection system, the proposed solution can be implemented with some degree of flexibility, thus enabling the usage of different configurations to achieve high performance on Android malware detection in the presence of concept drift. In this regard, our experimentation evidenced that good performance metrics can be obtained with the vast majority of possible configurations. The results and hyper-parameters used in our specific implementation are provided in Section 4.3.

After setting the general considerations and describing the main variables of the proposed solutions, the following paragraphs provide a detailed explanation of the intricacies and inner workings of the proposed solution.

As specified in Algorithm 1, when the first data chunk is received (i.e., $k = 0$), the whole chunk is processed by the *splitInitialDataset()* function which takes the chunk as input, splits its n elements into S ordered and equal-sized data chunks (i.e., each composed of n/S samples), and outputs the *IDS* data set. Then, each subset is used to train a new classifier (i.e., *trainClassifier()* function) which is added to the pool (i.e., lines 4–7 of Algorithm 1). As a result, a full pool of classifiers is generated after processing the first chunk, thus available for the testing

phase of all the subsequent data chunks (i.e., *first modification* to the original algorithm). In the next step, the set of legitimate samples from the initial data chunk (i.e., all by default, but a different sample size could be used) are used to train an anomaly detection model (i.e., *trainAnomalyDetector()*). Finally, the last processing step of the initial chunk involves the storage of the whole initial chunk as the dynamic ensemble selection data set (i.e., *DSEL*) for the next chunk (i.e., line 3 of Algorithm 2). As previously explained, the *DSEL* is used to select the best classifier ensemble from the classifier pool for each data sample in the new data chunk. This selection may vary according to the dynamic ensemble selection algorithm used in the particular implementation (Ko et al., 2008).

This concludes the processing of the initial data chunk, which is used for initialization purposes, and it is the only data chunk with distinct processing steps in our proposed solution. For all subsequent data chunks, the same *first-testing-then-training* procedure is applied, described as follows.

After the first chunk is processed, when a new data chunk is received, the prediction phase is applied first, as outlined in Algorithm 2. Thus, upon the arrival of the new data chunk, the dynamic ensemble selection algorithm is fit with the previously stored *DSEL*, the classifier pool, and the new data chunk (i.e., input of *dynamicSelection()* function, line 5 of Algorithm 2). This step aims to select the best ensemble of classifiers to predict the labels for each sample in the new data chunk. Once Π_{Dk} is fit, this dynamic ensemble model is used to forecast the class of the n elements of DS_k, thus generating the initial predictions (line 6 of Algorithm 2). These initially assigned labels are then refined based on *custom* rules, included inside the *anomalyDetector()* function, and using the anomaly detector forecasts for each sample in DS_k (line 7 of Algorithm 2). The outcome of this step is the final prediction for all the samples of the new data chunk. As mentioned before, the anomaly detector helps to support or challenge the class prediction assigned by the classifier pool in borderline cases where the anomaly model may provide more accurate results. Finally, in the last step of Algorithm 2, the new data chunk is stored as *DSEL* for the next chunk.

This concludes the first processing step of the new data chunk, the *predictive* phase. The next step for the new chunk is the *training* phase, as described in Algorithm 1.

The training phase uses the whole data chunk and the outcome of the previous phase to update the pool of classifiers and generate a new anomaly detector. More specifically, the worst-performing classifier on the new data chunk is removed from the classifier pool (i.e., *pruneWorstClassifier()*). Then, a new classifier is induced using the samples from the new chunk and their predicted labels. The new classifier is added to the pool (line 12 in Algorithm 1), which is again composed of S classifiers. The removal of an *aging* classifier and the insertion of a *new* classifier keeps the pool at the specified size while updating its capabilities to accurately forecast on new data, thus being able to adapt and react to emerging concept drift. Finally, the legitimate portion of new data (i.e., L_k) is used to generate a new anomaly detector that will be used in the predictive step of the next data chunk. This last step concludes the processing of the new data chunk, using it as a training set to update the forecasting capabilities of the system.

The described *first-testing-then-training* cycle is repeated for all the subsequent data chunks in the data stream, enabling the system to address concept drift issues effectively and efficiently without major changes in the solution.

3.2.3. Concept drift characterization

The proposed solution is able to detect and adapt effectively to concept drift in Android malware detection but also provide relevant insights about its character. The characterization of concept drift can provide useful knowledge and insights about the changes in Android malware, its direction, and expectations. It can also help to enhance the trust of analysts in the detection system. The inner workings of the proposed solution can be leveraged to explore thoroughly the phenomenon of concept drift by analyzing the influence of data changes on classification quality measures in various time horizons.

For concept drift characterization and appraisal of its influence, *permutation feature importance* analysis was employed. This method, proposed by Breiman (2001), is model-agnostic and applicable to the discussed case of binary classification (i.e., malware detection) which can be evaluated by quality measures related to the classification results. The permutation feature importance technique is explained 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, $x_i^{\pi,j}$ is a vector achieved by randomly permuting the j-th column of $\mathbf{X}$. Given a loss function L, the importance VI_j of the j-th feature is defined as the difference between the loss calculated using pseudo-random values and the original data, as it is expressed by the following equation:

$$VI_j^{\pi} = \frac{1}{N}\sum_{i=1}^{N} L\left(y_i, f\left(x_i^{\pi,j}\right)\right) - L(y_i, x_i) \tag{1}$$

It is worth mentioning that Random Forest algorithm offers an alternative assessment based on *Gini coefficient* or *entropy* (Maimon & Rokach, 2005), called *feature importance*. However, such calculated importance is based on the training data used to create the classification model. In the discussed case, a more important issue is how the model works on new data, which might be affected by concept drift, rather than how well the features were used to discriminate the learning set. Therefore, the application of the permutation feature importance technique to the test data is preferred. In this regard, due to its stochastic nature, the permutation feature importance score may vary significantly among iterations. Hence, for the sake of results' stability, it is recommended to repeat the permutation procedure at least 50 times and average the results (Altmann et al., 2010).

The concept drift characterization method used in this study adopts Eq. (1) by the creation of the classification function f_t using data X_t from period P_t. Next, observations X are taken from the set $\bigcup_{l=t+1}^{l+h} X_j$ where h declares an analysis time horizon. In this work, we discuss the following time horizons: *short-term* (i.e., 3 months), *mid-term* (i.e., 6 months) and *long-term* (i.e., 12 months). The usage of several time horizons brings an opportunity for better characterization of the changes in the importance of features. The whole procedure is summarized by the following equation:

$$VI_j^{\pi}(t) = \frac{1}{N}\sum_{\substack{i=1,\\ x_i \in \bigcup_{l=t+1}^{l+h} X_l}}^{N} L\left(y_i, f_t\left(x_i^{\pi,j}\right)\right) - L(y_i, x_i) \tag{2}$$

The procedure can be used to evaluate the influence of features on various quality functions $Q(.) = 1 - L(.)$ such as:

- *F1 score*, which is a more comprehensive metric for malware detection performance on imbalanced data sets than the overall accuracy and it is defined as:

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{3}$$

- *Specificity (True Negative Rate)*, which describes the quality of benign software recognition (i.e., negative label). It is calculated as:

$$TNR = \frac{TN}{TN + FP} \tag{4}$$

- *Recall (True Positive Rate)*, which describes the quality of malware detection (i.e., positive label) and it is defined as:

$$TPR = \frac{TP}{TP + FN} \quad (5)$$

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

The analysis of permutation feature importance scores of chronologically ordered data chunks allows the exploration of changes and the observation of the evolution of important features in the data, which enable the detection of trends and the characterization of emerging concept drift.

4. Results

4.1. Data pre-processing

After the application of each sequential data pre-processing step, the results reported in Table 1 were obtained.

As can be observed, after the first pre-processing step, 128 syscalls were found to be non-zero and not constant valued. This filtered subset of features was further processed and highly correlated features (i.e., $|r| > 0.80$) were removed. More specifically, 31 features were found to be over the specified threshold, thus showing a *strong* linear correlation with another feature. The resulting feature set was composed of 97 features.

To assess the adherence to the normal distribution of the feature distributions, four normality tests were applied to every feature, including the *Shapiro-Wilk* normality test, the most powerful normality test according to Mohd Razali & Bee Wah (2011). The results of the statistical analysis proved that no feature showed Gaussian distribution characteristics, as evidenced graphically by the plots of feature distributions in Fig. 2. Therefore, the final feature set was composed of 97 non-normally distributed syscalls.

4.2. Concept drift detection

The initial period selected for concept drift detection was the second semester of 2011. As this study compares the performance of two timestamps for concept drift detection and handling, it was critical to select a period where effective models could be induced for both timestamps. Thus, this period was preferred as it provided enough data to build effective classification models for both timestamps (i.e., *accuracy* $>$ 95%). Data from prior periods were discarded and not further analyzed. Random Forest classification models were induced using the whole feature set (i.e., 97 syscalls) and the class labels of the samples corresponding to each timestamp.

The most relevant features of the classification models were selected using the permutation feature importance technique (i.e., feature selection). In our experimentation, 500 permutations per feature were used, which is significantly over the empirically recommended quantity for results' stability (Altmann et al., 2010). From the obtained results, only the features that showed positive mean importance on the initial-period model for each timestamp were selected and ranked in descending order of importance. The following results were obtained for each timestamp:

- *Last modification* timestamp: 32 features were found *important* from the whole feature set (i.e., 97). This selected feature set is referenced as *initial-lm-set*. The data set for this period was composed of 9,288 samples (i.e., 6,916 legitimate apps and 2,372 malware apps), and the accuracy of the RF classification model on the testing set was 0.9870.
- *First seen* timestamp: 17 features were found *important* from the whole feature set. This selected feature set is referenced as *initial-fs-set*. The data set for this period was composed of 2,677 samples (i.e., 2,124 legitimate apps and 553 malware apps), and the accuracy of the RF classification model on the testing set was 0.9859.

After feature selection, the resulting feature sets were used to build one-class anomaly detection models using the Isolation Forest algorithm (i.e., 300 estimators per model). As a result, for each class (i.e., malware and benign) in each timestamp-related data set, a one-class anomaly model was generated, using the corresponding feature set as model features (i.e., *initial-hm-set* or *initial-fs-set*). Then, the malware and benign data belonging to *posterior* time frames were split into 6 months periods (i.e., from 2012 to 2020) and used as testing sets for each corresponding timestamp-class model. Besides, for every timestamp-class combination, 3 anomaly models were induced using distinct subsets of features from the *important* feature sets (i.e., best 5 features, best 10 features, and all features).

The accuracy metrics of all the induced anomaly models on their respective testing sets were retrieved. The results are provided in Fig. 3. The line graph on the left shows the results related to the *last modification* timestamp, while the line graph on the right provides the data related to the *first seen* timestamp. The 6 anomaly models generated for each timestamp are reported with different colors and line styles. The color reflects the data class (i.e., red for malware and green for benign apps). The line style informs about the subset of features that was used to build and test each specific anomaly model. More precisely, solid line is used for all features, dashed line for the 10 *most important* features, and dotted line for the 5 *most important* features. The horizontal axes provide the test period, whereas the vertical axes provide the *accuracy* value retrieved for each specific period. The horizontal axes are split into 6-months periods. The *.1* value attached to the year number informs about data belonging to the first semester of that year (e.g., 2012.1), whereas *.2* reflects the data regarding the second semester (e.g., 2012.2). As a result, six anomaly detection models were built and tested per timestamp (i.e., three per class) encompassing the whole 2012–2020 time frame.

The anomaly detection results provided in Fig. 3 demonstrate the existence of concept drift in the data, thus proving that the same set of features and values are not useful in all time frames to recognize either one of the classes. In this regard, according to the concept drift typology proposed by Ramirez-Gallego et al. (2017), the following behaviors are observed.

In benign applications, an *incremental drift* dominates. The number of recognized observations slightly goes down over time to dip in the last period in a *sudden drift*. However, this deep dip in performance might have been caused by the scarcity of samples available for this last period in the data set (e.g., over 1000 are available for 2020.1, whereas less than 40 for period 2020.2). Thus, except for the last period, the observed behavior is typical for an AI system with a static learning set tested on data evolving over time, as in Luckner (2019).

The data *drift* is especially evident in malware data. The graphs depict that the initial model, trained on any feature set, shows remarkably distinct accuracy values from period to period, suggesting that the importance of features for the classification models might have changed significantly, thus evidencing concept drift. Both timestamps provide a similar picture of the phenomenon, with the initial models performing well on data belonging to closer periods and losing discriminatory power over time. In 2016.1 and 2019.1, the initial set of important features seems to become relevant again, reaching accuracy levels similar to benign data, but losing its importance in the subsequent periods, thus leading again to data *drift* and poor discrimination. It could be related to a *recurrent* threat emerging in the initial, 2016.1 and 2019.1 periods.

In any case, the analysis of the line graphs in Fig. 3 evidences the presence of concept drift in Android behavioral data, which is especially pronounced in the malware case. In consequence, to build long-lasting

and robust Android malware detection solutions, the detection systems must be able to adapt and learn from the changes in the data to keep high and stable performance over time.

4.3. Concept drift handling

The proposed solution, detailed in Section 3.2.2, was applied to KronoDroid data set, described in Section 3.1. As the data set is not a real data stream and encompasses a long period, the data was divided into time-constrained data chunks. This operation enabled us to simulate a realistic scenario where the data flow is constant and in a great volume. In this regard, the samples in each data chunk are likely to be similar until *drifting* emerges in any of its forms. Besides, as the 6 months periods used in Section 4.2 might be too wide to accurately detect emerging concept drift, a shorter temporal constraint was established. Thus, a maximum of 3 months of data were included in every chunk (i.e., referenced as year quarter, Q). In addition, to train the models, a fixed chunk size of 4000 samples was established. In the case that the incoming data for a specific time frame did not contain enough data to cover the 4000 samples per chunk of the training phase, prior data were used respecting the chronological order.

The data set used provided a large number of samples to cover the years from 2011 to 2018, but there were not enough timestamped data to cover effectively the requirement of 4000 samples per quarter in the years before and after this time frame (i.e., 2008–2010 and 2019–2020). Consequently, just the data from 2011.Q3 to 2018.Q2 were used in the experimental setup, thus covering 7 full years of Android history.

Based on experimental testing, the values of the hyper-parameters selected for our implementation were: 4000 samples per training data chunk, pool size of 12 classifiers, and *Random Forest* models with 300 estimators as classification models. The dynamic ensemble selection algorithm used was *META-DES* (Cruz et al., 2015). *Isolation Forest* algorithm was used to induce the anomaly detection models. The data balancing technique used was *random oversampling*.

Although no hyper-parameter optimization procedure was performed, the selected hyper-parameters provided high and stable performance. A different selection of hyper-parameters may yield similar performance metrics as the solution is robust and allows certain degree of flexibility in the selection of the hyper-parameters. The implementation was coded in Python programming language, leveraging the functionality of *scikit learn*, *imblearn* and *deslib* libraries.

The performance of the proposed solution when the *last modification* timestamp data was used is reported in Fig. 4, while Fig. 5 provides the performance for the *first seen* timestamp data.

Fig. 4 provides the F1 score performance of the proposed solution, using the provided hyper-parameters, and its comparison with two naive solutions and the original algorithm. More precisely, the *initial classifier* line (i.e., dotted grey line) provides the performance results of a classifier generated using the data of the first chunk and tested on all the subsequent chunks. This approach simulates the scenario where a detection model is generated at a specific time (i.e., 3rd quarter of 2011 in this case) and is never updated, thus neglecting concept drift. As the initial data chunk was significantly imbalanced towards the legitimate class (i.e., 98% of the data points were benign apps), another *naive solution* is provided as a reference (i.e., dashed grey line), using data from the second chunk, where the data were more balanced (i.e., 65% legitimate, 35% malware). As can be observed, the two naive solutions, which are never updated, yield poor detection performance as time passes. On the contrary, the proposed solution (i.e., solid blue line) provided a detection performance of over 90% in almost all periods, showing robustness against concept drift, reacting, and updating its knowledge when it emerged. Further, its performance was superior to the performance shown by the original algorithm in most periods, especially in the first ones.

As can be seen in Fig. 5, when the *first seen* timestamp is used, the high, stable, and smooth performance line provided by the solution in the previous case is not observed. The performance line performs sudden dips and boosts that might have been caused by a general temporal misplacement of the data which led to improper concept drift handling. This timestamp does not seem as reliable as the *last modification* timestamp to locate the data samples in their correct period and, consequently, the changes in data features do not emerge naturally but artificially, likely caused by temporal displacement. Despite that, the solution still shows good performance and adaptation over time.

It is worth noticing that, in this case, the horizontal axis starts and ends a period later than in Fig. 4. This difference is due to the distinct temporal

Table 1
Data pre-processing results.

Preprocessing step	Results
Feature Variance Analysis	160 constant or zero-valued
Feature Correlation Analysis	31 highly correlated
Feature Distribution Analysis	0 normal
Final feature set	97 non-normal syscalls

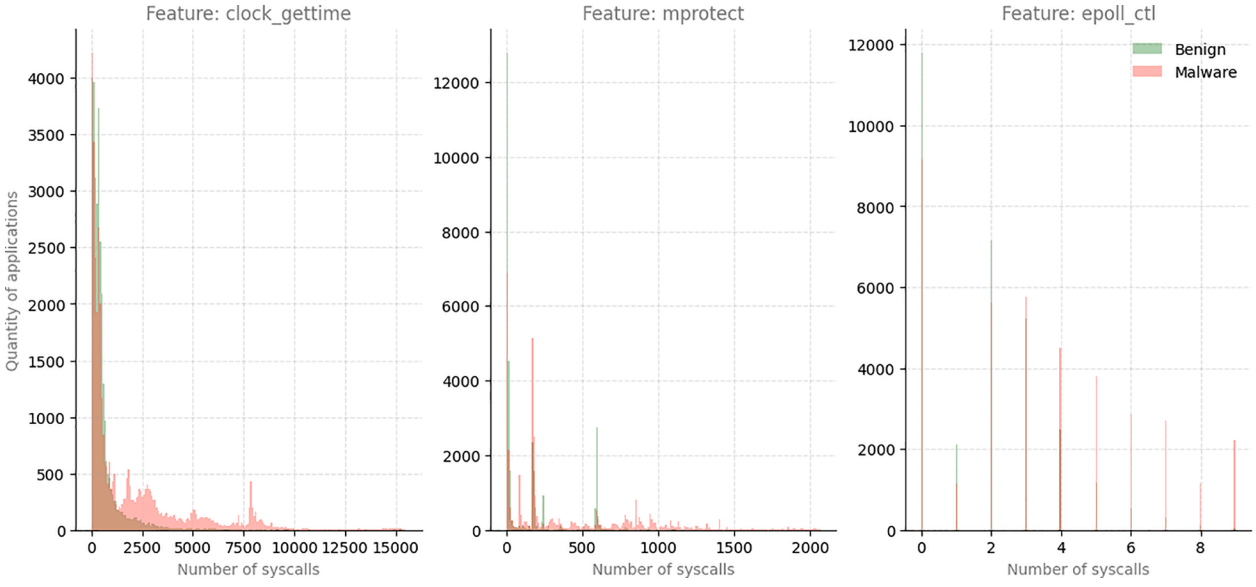


Fig. 2. Distributions of example features.

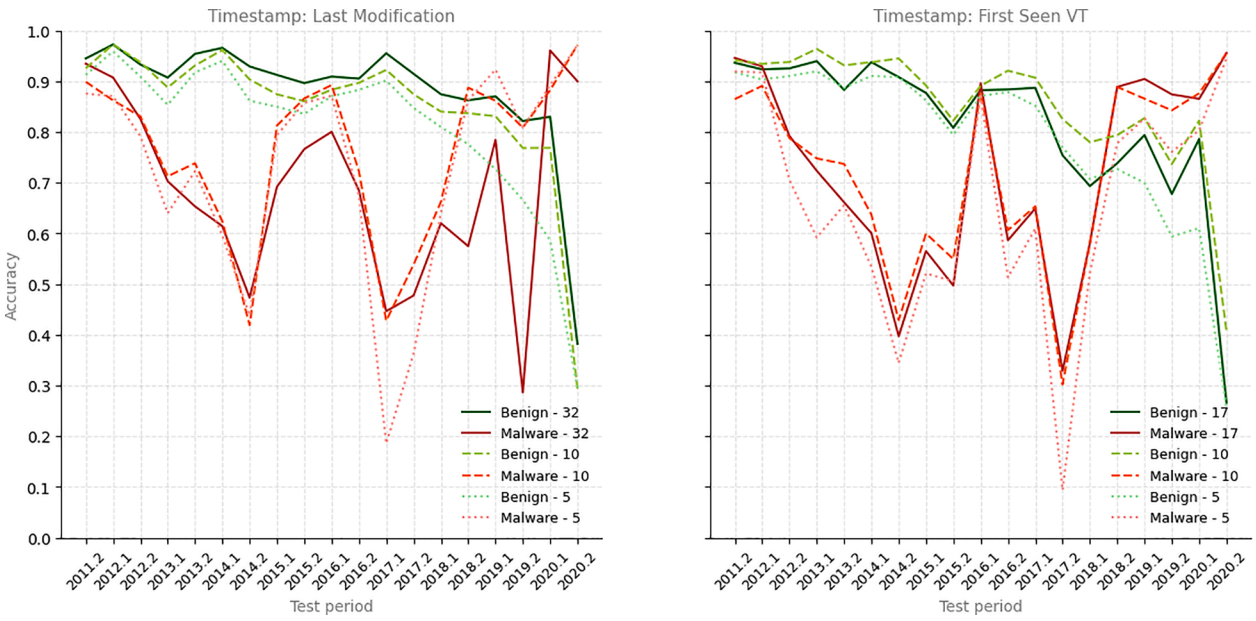


Fig. 3. Performance of the one-class anomaly detection models.

distribution generated by this timestamp, which allowed generating the initial classifier on 2011.Q3 but not a test set for that period. Therefore, the time series data is displaced one quarter with respect to Fig. 4. This fact supports the aforementioned differences between these two timestamps regarding the location of data samples within the historical timeline.

As can be observed in Fig. 5, when the performance of the proposed solution is compared with the other approaches, it significantly outperforms the *initial classifier* and the *naive solution*, reaching high-performance values in almost all periods. Again, it outmatches the performance of the original algorithm, especially in the first periods. Despite that, when the *first seen* timestamp is used, the performance of the system is less smooth and remarkably lower than when the *last modification* timestamp is used. More precisely, the displacement of data samples over the historical timeline overrides the emergence of a *natural* concept drift, thus hindering its proper handling.

In conclusion, the usage of single period classifiers, applied over time with no update, proved to be inefficient and showed poor and degenerative performance as time passed. These solutions become obsolete and ineffective in a short time span. Contrarily, the proposed solution is robust and resilient to changes in data over time, especially when the *last modification* timestamp is used, keeping high-performance metrics on Android malware detection under the challenge of constant data evolution. These results also demonstrate that system calls can be used to achieve effective, long-lasting, and robust Android malware detection even when concept drift threatens the performance of the detection system.

4.4. Concept drift characterization

The following paragraphs explore the concept drift phenomenon using several instruments for its characterization. More specifically, the impact of the *pool size* and the *evolution of the importance of features* are analyzed.

4.4.1. Impact of the pool size

The analysis of the proposed solution according to the number of classifiers present in the pool of classifiers brings some interesting findings on the concept drift analysis and the modeled data.

Different pool sizes for the proposed solution, ranging from 2 to 20 classifiers, were assessed. The experiment was repeated 20 times per pool size and the results were averaged. In this regard, Fig. 6 provides relevant information regarding the experimental results such as the *average lifetime of a classifier* (i.e., how long, on average, a classifier was in the pool before it was removed), the *average lifetime of the classifiers from the initial pool* (i.e., in what period, on average, all the classifiers created in the first period were removed), and the *quality of the new classifiers* (i.e., how many times, on average, the most recent classifier was the best performer) calculated for both timestamps. The color ribbons surrounding the lines in Fig. 6 provide the standard deviation for each reported value.

As can be observed, regardless of the timestamp, the average lifetime of a classifier inside the pool is linear according to the pool size in a ratio of 0.8–0.9. This fact shows that the *oldest* classifier in the pool is not always the one removed when the poorest model is purged from the pool. Recurring threats might cause old classifiers to perform relatively well in later periods. Furthermore, the results show that regardless of the pool size, a single classifier is never the best performer for more than 5 periods, thus demonstrating the dynamism of the phenomenon.

More interestingly, the number of periods in which the newest classifier is the best slightly decreases when the pool size increases. As can be observed, in all models the number of times a classifier is the best performer is different from 1. This shows that there are periods where an older model is repeatedly the best performer and suggests that there might be gaps between periods where the same classifier is the best performer, thus reinforcing the existence of concept drift in the analyzed data. Besides, it should be noted that newly created classifiers are valid for a longer time for the *first seen* timestamp than for the *last modification* timestamp. Hence, *natural* concept drift handling seems to generate more specific and better classifiers with reduced lifetimes, as shown in Fig. 4, whereas misplaced data generate more generic and worse performer classifiers, but with longer lifetimes, as displayed in Fig. 5.

In the case of the *last modification* timestamp, the classifiers from the initial pool are always removed as soon as possible (i.e., in the first S periods, where S refers to the pool size). This shows that the initial data cannot be used effectively to discriminate new data, so the related classifiers are rapidly removed. This observation concurs with the results obtained on the anomaly detection models (see Fig. 3). For the *first seen* timestamp, the lifetime of the initial pool substantially increases for seven and more pool components. This suggests that the knowledge from the initial periods is useful for later periods, which might be caused by a general misplacement of the data samples along the timeline, provoking *artificial* drift in the data.

Although this thorough analysis yielded relevant insights about the

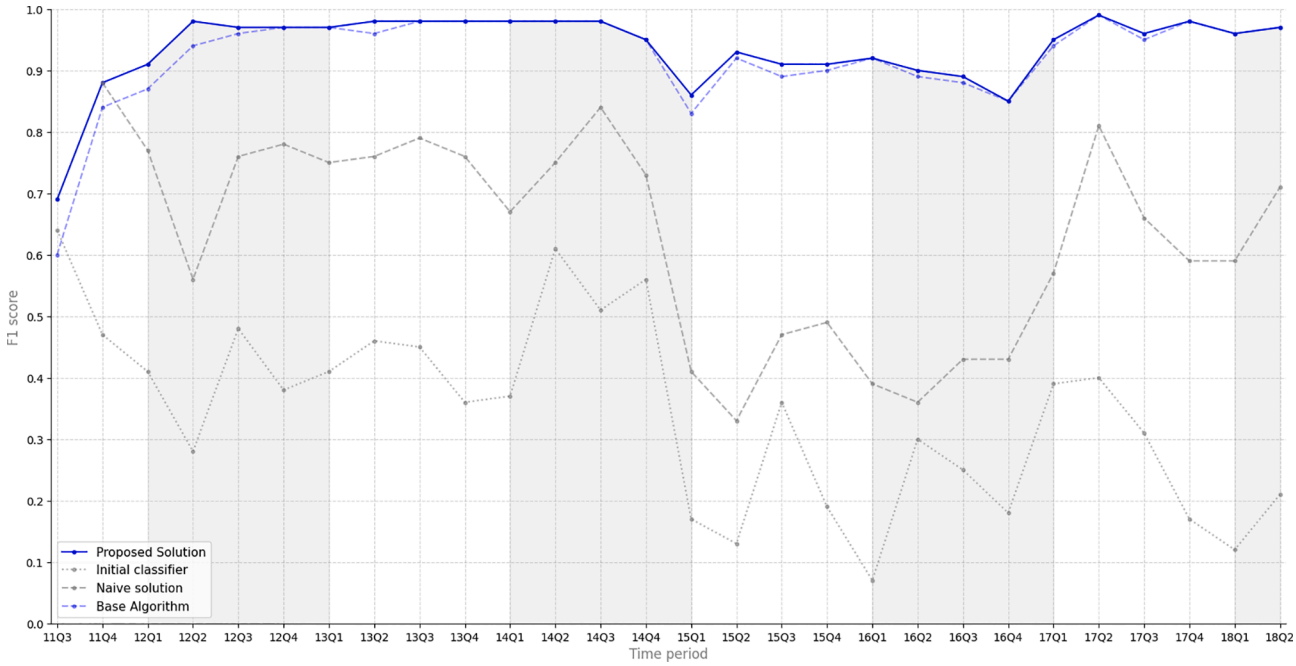


Fig. 4. Proposed solution performance using the *last modification* timestamp.

impact of the pool size on concept drift and data modeling, it did not provide any hint about the optimal pool size, a relevant question that remains unsolved. The following paragraphs address this issue.

Fig. 7 shows the classification results (i.e., F1 score) obtained using the proposed solution with pools of various sizes. The usage of boxplots enables us to easily compare the distributions of the classification results for distinct pool sizes. The dark blue boxplots provide the results for the *last modification* timestamp while the light blue boxplots provide the distributions of the classification results for the *first seen* timestamp. The body of the boxplots reflects the range where the central 50% of data is located, also referenced as *interquartile range* (*IQR*). The IQR is calculated as $IQR = Q3 - Q1$, where $Q3$ and $Q1$ are the 75% and 25% percentiles, respectively (i.e., the borders of the box). The orange line crossing the bodies references the median while the average is provided by the red rhombus. *Outliers* or extreme values are reported as grey dots, located further than the minimum and maximum *whiskers* which are calculated as $minimum = Q1 - 1.5 * IQR$ and $maximum = Q3 + 1.5 * IQR$.

The boxplots in Fig. 7 reflect that even though the average quality diminishes for larger pools in both timestamps, in general, the pool size does not seem to influence substantially the classification quality. However, the pool size of 12 classifiers for the *last modification* timestamp shows distinctive properties. First, the average and median values are nearly the same, defining a relatively *symmetrical* distribution. Thus, the deviations of all terms from the median cancel out. This distribution is different from the other distributions, which are skewed, thus making the median a better central measure than the average and relating to the existence of extreme values. Second, the number of outliers in the distribution is minimal. The other pool sizes generated classifiers with

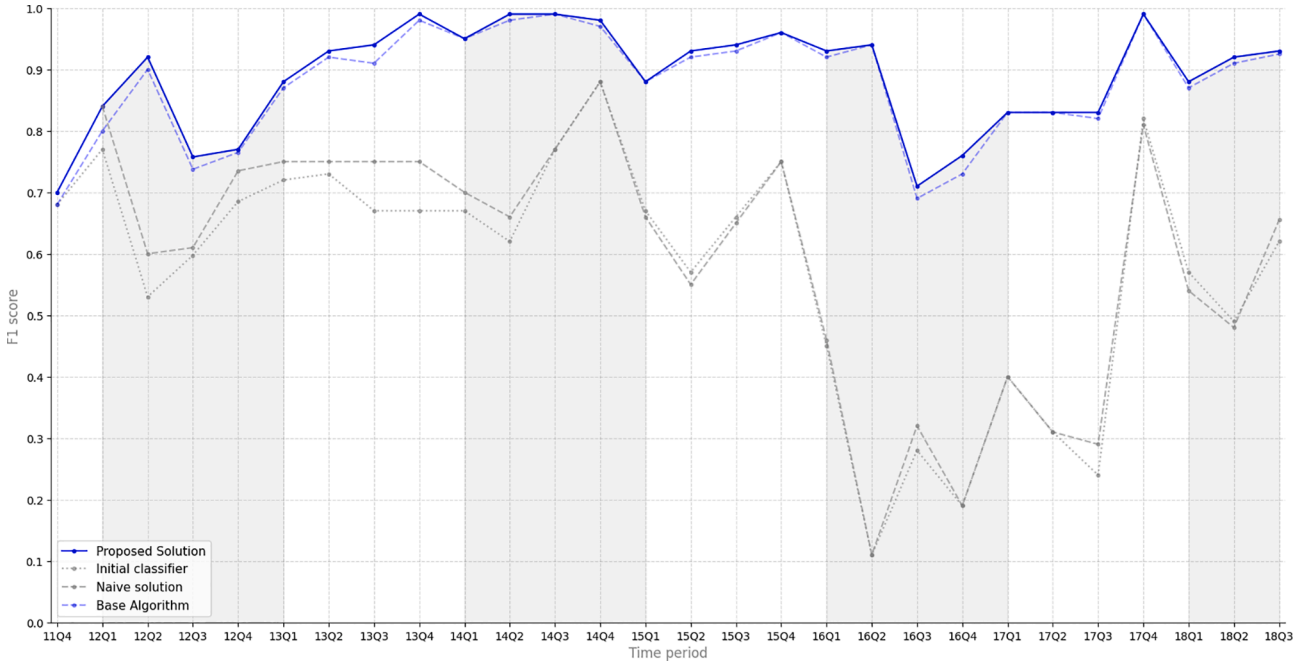


Fig. 5. Proposed solution performance using the *first seen* timestamp.

worse and more spread results. Third, even though the pool size of 13 shows similar properties, the whiskers for the pool size of 12 are shorter, thus concentrating the data in a shorter range, with less variability. Besides, as a smaller number of classifiers is needed, it is a more efficient approach than the pool size of 13 components. As a result, all these observations make the pool size of 12 the preferred option for optimal results.

As can be seen in Fig. 7, in the case of the *first seen* timestamp, the obtained results for all pool sizes are remarkably worse than for the *last modification* timestamp.

The combination of these results with the previous findings from Fig. 6 enables us to confirm that the system relied excessively on old classifiers when using the *first seen* timestamp. Two aspects that stress it are the delay in replacing the initial pool and the extended lifetime of the classifiers. Due to the discretionary nature of the generation of the *first seen* timestamp, which depends on users proactive submissions, it seems to lag behind the *last modification* timestamp unpredictably, thus being prone to displace the samples within the historical timeline and provide a relatively inaccurate temporal location. Therefore, the obtained results suggest that the *first seen* timestamp provides a less realistic approximation to the *real* concept drift and, consequently, a less accurate solution and characterization of it.

Based on these results, the following experiments were performed using the pool size of 12 components and the *last modification* timestamp.

4.4.2. Evolution of features importance

The permutation feature importance technique was used to analyze the evolution of the importance of features over time, thus enabling to characterize concept drift. The procedure and main results of this stage are explained as follows.

For each period P_i, the best classifier was selected according to the results obtained in Section 4.3 and detailed in Fig. 4. The permutation feature importance technique applied to the classifier was calculated using Eq. (2) with F1 score as loss function. The *importance* was calculated separately for three test sets (i.e., time horizons). The first set was the subsequent period to P_i, thus P_{i+1}. The second set consisted of two successive periods, $\cup_{j=i+1}^{i+2} P_j$, and the third set contained the four subsequent periods, $\cup_{j=i+1}^{i+4} P_j$. As defined, the sets were built incrementally, thus corresponding to three, six, and twelve months data horizons. The results were calculated for all possible periods of the data set in the range $P_1,\ldots,P_{n-4}$.

The usage of three incremental test sets enabled us to observe how the importance of features varied in short, medium, and long-term time horizons. In this regard, Fig. 8 provides the distributions of feature importance using boxplots, calculated for all periods and including all syscalls that reported a non-null importance in at least one period. The box color indicates the time span or horizon (i.e., darker colors reference longer time-frames). The orange line crossing the body indicates the median and the green triangle provides the average value. The horizontal axis informs about the system call name, while the vertical axis reflects increasing scores of permutation feature importance (i.e., a larger score directly relates to greater importance).

The results provided in Fig. 8 were obtained from 20 tests of 500 permutations each. Even though the results slightly varied among iterations, the main findings described in the following analysis were common for all tests.

As can be observed in Fig. 8, no feature was found useful or *important* in all tests as all boxplots start near the zero value. A fact that stresses the existence of concept drift. More interestingly, based on these results three types of features can be distinguished. The first type of features includes those features that are not useful in any time horizon like *getgid32* or *restart_syscall*. These features might have provided a low importance score in some periods due to the stochastic nature of the technique or a non-random positive importance but with a negligible impact on the task. The second group of features is related to features that are more important in longer time frames (i.e., medium and long term) than in the short-term. These features are not very good at recognizing sporadic threats, but they constitute a solid base in a time-extended threat detection system. Features like *clock_gettime* and *flock*, which lie inside this category, show a relatively stable discriminatory power over time. Lastly, the third type of feature shows the opposite situation, the feature is a relatively good discriminator in the short term but is not as useful in longer time frames. Due to the larger number of distinct threats present in longer time frames (i.e., more families and malware variants), these features are not so useful for overall discrimination as in the short time frame, where a smaller variety of threats is present. Consequently, these features might work well to distinguish specific malware families. Features such as *write* or *SYS_317* are included in this category.

To perform a deeper analysis of the importance of features for specific recognition tasks, the permutation feature importance was calculated using *specificity* and *recall* as loss functions. The results for

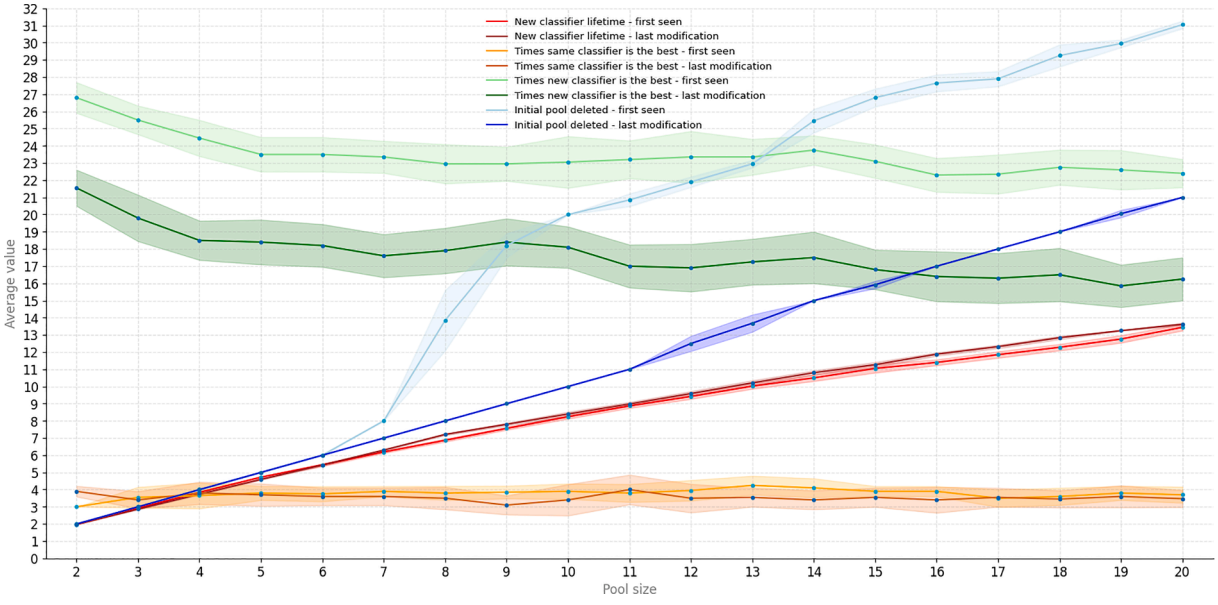


Fig. 6. Classifier pool statistics for both timestamps.

specificity provide information about important features to recognize benign software, whereas for *recall*, also called *sensitivity*, they describe important features for the malware detection task.

The obtained results are depicted in Fig. 9, showing features with positive importance in benign and malicious software recognition tasks and their evolution over time. The presentation is limited to features with positive mean importance estimation obtained using Eq. (2) and that, for each task, were found in the top 3 most important features in any period. The horizontal axis provides the timeline, split into quarters or periods. Regarding the vertical axis, the color relates to specific features, while the colored areas (i.e., vertical range) in each stacked bar provide the importance score of the specific features relative to the total importance of each specific period of time (i.e., the total importance of a period is the sum of the importance scores of all the important features in that period). Consequently, the larger the vertical range or area spanned by a feature in a bar, the greater the importance of the feature in the period.

In the case of benign software recognition, presented in Fig. 9a, the importance of features appears to be locally stable. Several features like *read* and *mprotect* have similar influence for extended periods of time (i.e., from 2011-Q4 to 2014-Q2 in the case of *read* and from 2012-Q1 to 2017-Q1 in the case of *mprotect*). Besides, quarters with clearly outstanding features are rare (e.g., 2011-Q4, 2017-Q2). So, despite that some trends can be spotted, with some features gaining and others losing importance in some periods of time (e.g., *SYS_310* increases from 2016-Q1 to 2017-Q1 and *flock* increases from 2016-Q1 to 2018-Q1), the overall picture shows stability and that the same set of features is relevant in all time frames with no distinctive changes in relative influence and no new *important* features emerging over time (i.e., the bars have either a small portion of grey area or no grey area, meaning that most of the important features for each period are included in the bars).

The results are drastically different for the malware recognition task. Fig. 9b shows the changes in feature importance calculated for the recall function. As can be noticed, for most quarters, the dependencies observed in a specific period are not repeated in the following periods. Besides, even when a feature shows an extremely high importance in one period (e.g., *pread* in 2014-Q2), no consistency is observed and the importance of the feature dramatically decreases in the next periods. The only remarkable exception is *clock_gettime* feature, which is a very important discriminatory variable for several years (i.e., from 2012-Q3 to 2015-Q3). However, even in this case, there are quarters in this extended time frame where the feature loses completely its discriminatory power for malware detection (i.e., from 2014-Q2 to 2015-Q1).

Based on these observations, it is worth analyzing how two relevant features, *clock_gettime* and *pread*, were represented in the time horizon analysis performed previously. In this regard, Fig. 8 shows that *clock_gettime*, the feature found important for an extended period of time, is more important for the medium and long-term time horizons than in the short-term. In contrast, *pread*, the feature that was found critically important for a single period, obtains similar results in all horizons. Therefore, the horizon analysis supports the previous observations. In any case, it should be stressed that any importance score (i.e., local or periodical) influences all three horizons, but that the relationship among the levels gives additional information about the character of the importance.

Another issue observed in the malware recognition case is the existence of periods where the total importance of the features included in the bar is far from reaching the top (i.e., 2014-Q4, 2018-Q2). In those periods, none (e.g., 2014-Q4) or few of the included features (e.g., 2018-Q2) were found important for the malware recognition task. In the former case, it suggests that the set of features was not large enough to model all malware types observed in the data, whereas in the latter case, new features, not important in other periods, emerged as important.

Finally, even though important features seem to vary dramatically among quarters for the malware recognition task, some general patterns can be spotted. For instance, as mentioned before, *clock_gettime* is critically important from 2012-Q2 to 2015-Q2 but not so much after (i.e., more recent years). The internet-related system calls (i.e., *socketpair*, *recvfrom*, *setsockopt* and *getsockopt*) appear to have more importance in the recent years, from 2015-Q4 to 2017-Q3. More interestingly, the bars from 2012-Q1 to 2016-Q1 show clear dominance of small subsets of features (i.e., mainly *clock_gettime*), whereas in the latter years, the bars are composed of more features, looking more similar to the bars of the benign recognition task. In this regard, it is worth noting that, when comparing Fig. 9a with Fig. 9b, the segmentation of the bars is a major difference between them. Specifically, for the benign recognition task, the bars are dense, composed of many features, and show stability (i.e., the same set of features shows similar importance over years). On the contrary, the bars for the malware recognition task are mostly composed of a small subset of features, showing clear dominance of some of them

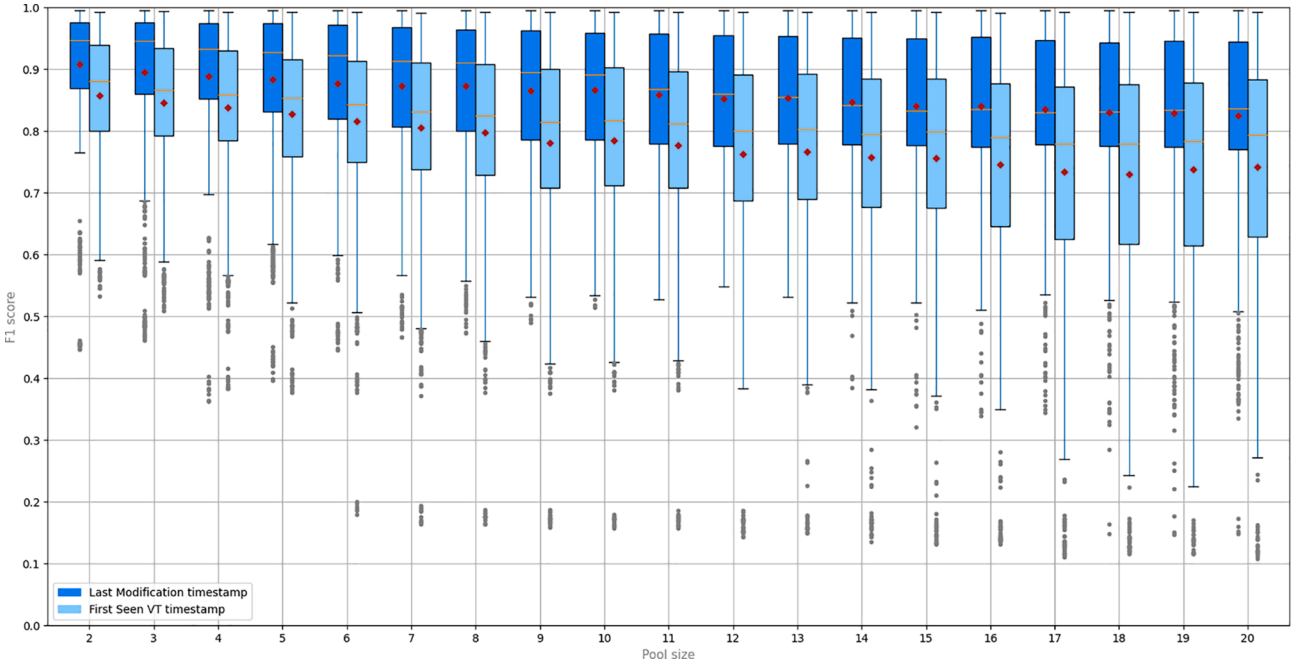


Fig. 7. Classification performance boxplots for both timestamps.

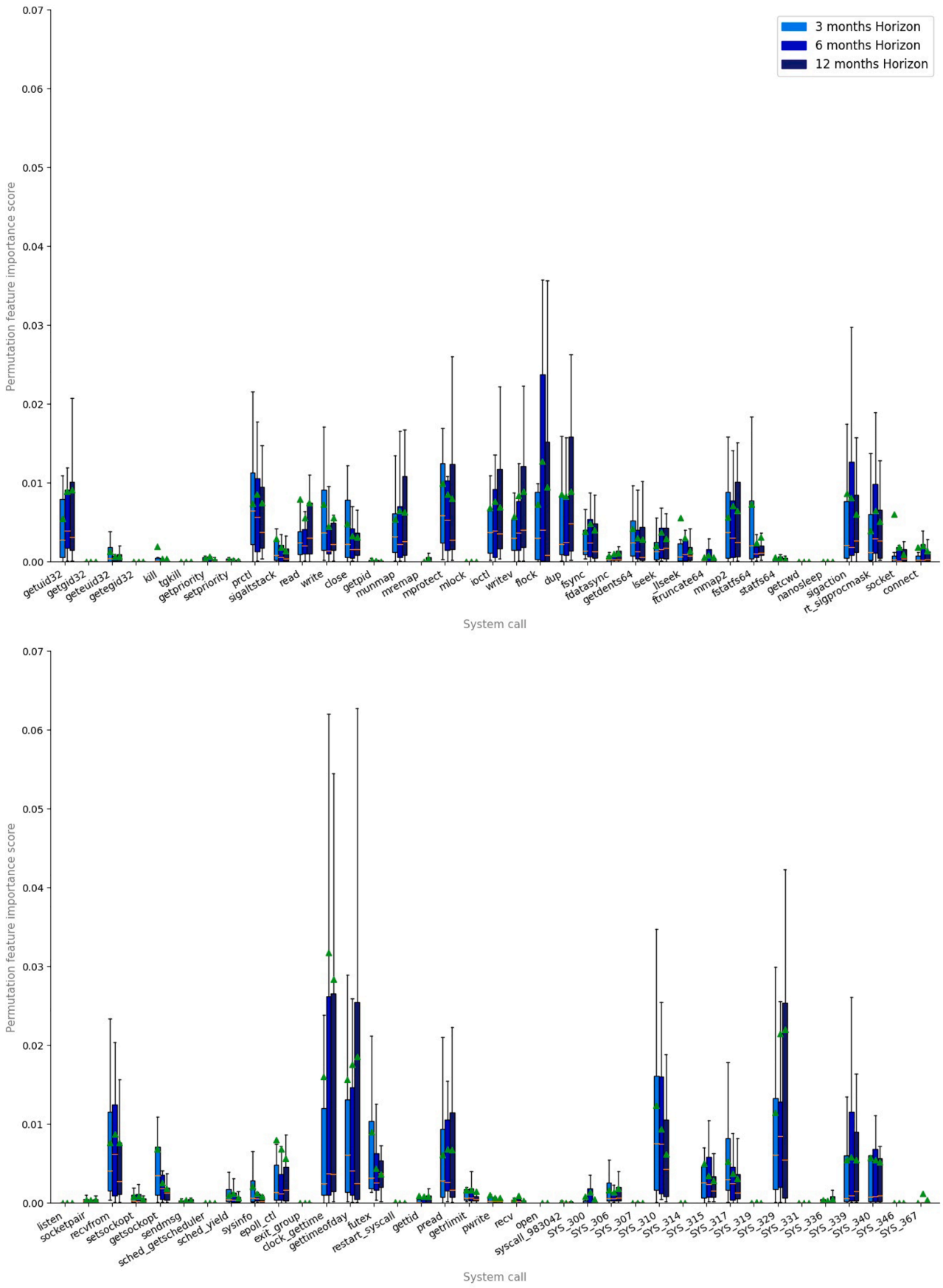
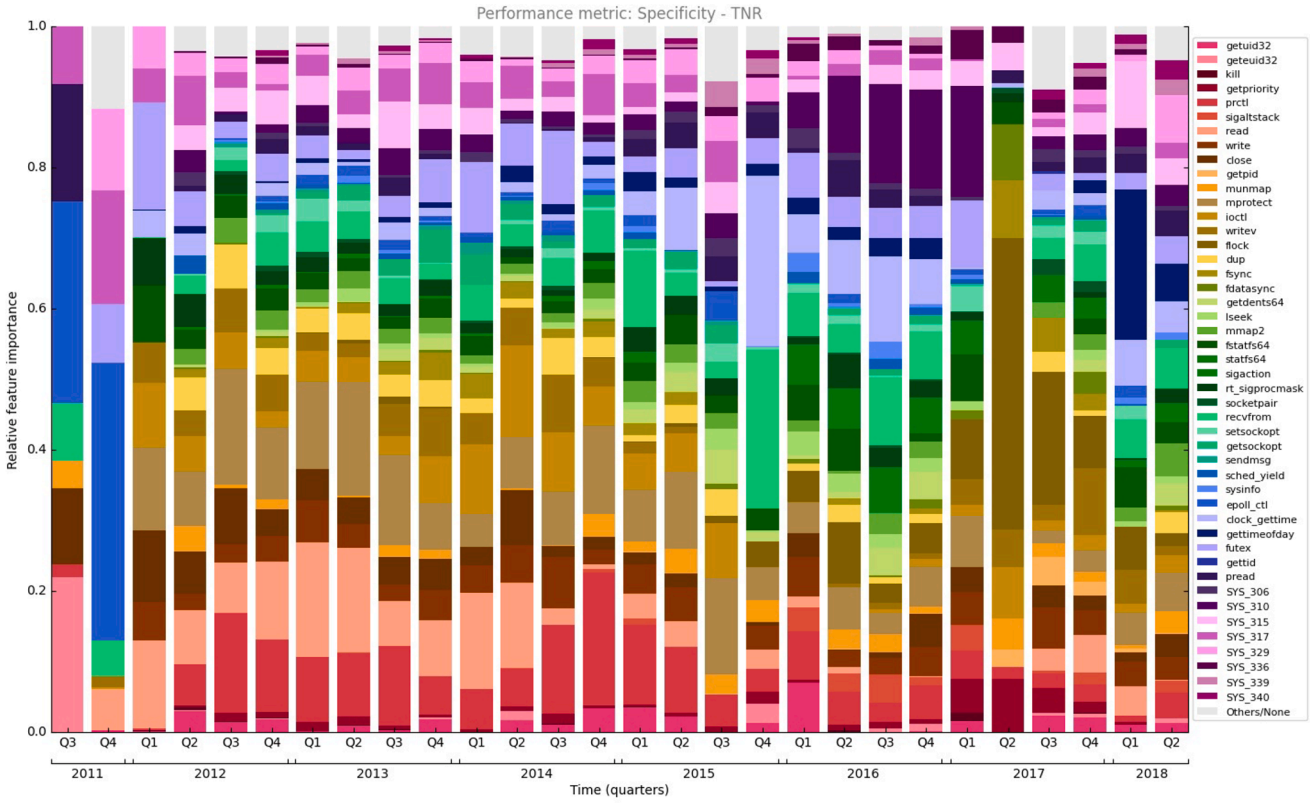
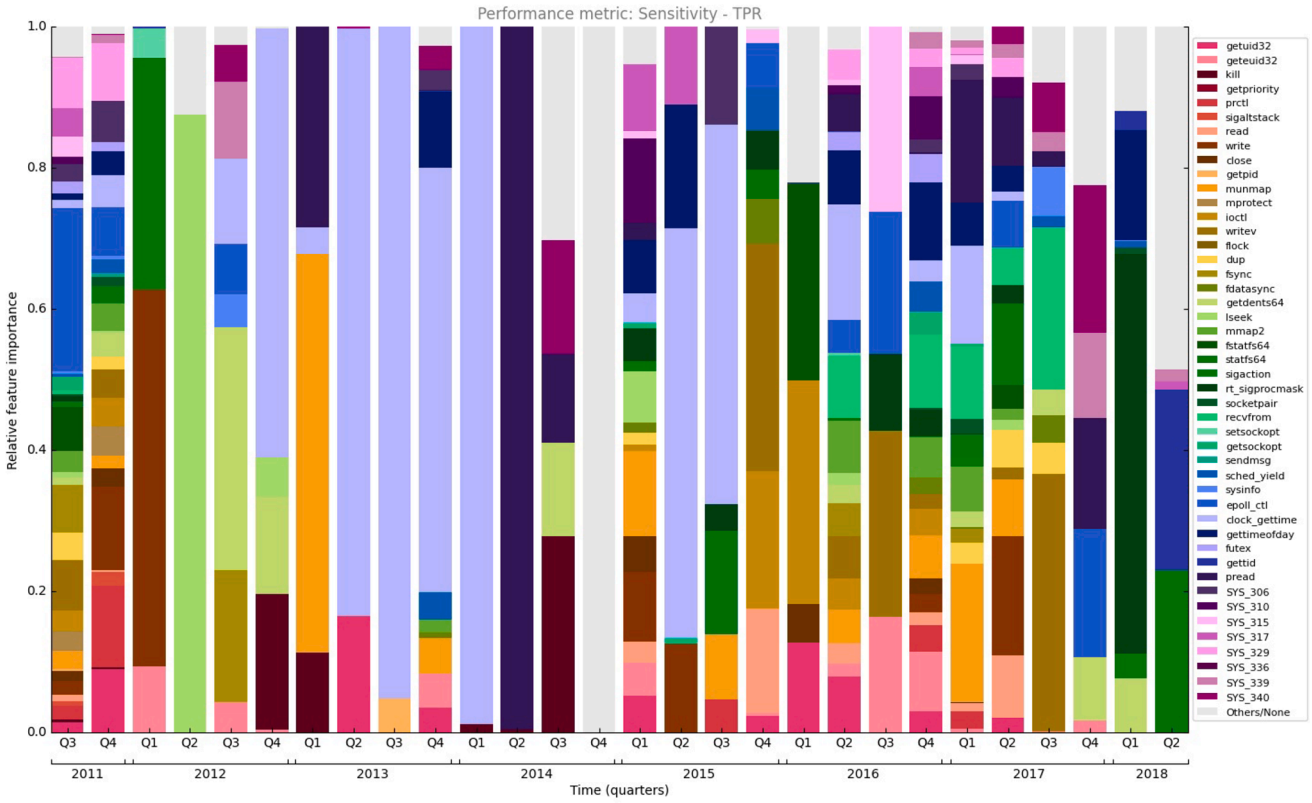


Fig. 8. Boxplots providing feature importance distributions calculated for *short*, *medium* and *long-term* time horizons.



(a) Specificity



(b) Recall

Fig. 9. Feature importance calculated quarterly for specificity (a) and recall (b) tasks.

over the rest. Consequently, the malware recognition task appears to be significantly more complex and changing more rapidly than the benign software recognition task.

To sum up, in this section we performed a thorough analysis of the evolution of the importance of relevant features. This analysis provides relevant insights about the data evolution, assisting to characterize concept drift which, in turn, can help to understand malware changes and their direction to design and implement better detection systems.

5. Discussion

The phenomenon of concept drift in Android malware detection has been neglected by most of the specialized research in the domain, which has overlooked the degenerative impact of *time* in the machine learning-based malware detection systems. The reduced number of studies that considered the impact of time in their detection systems proposed solutions to address the issue mainly based on static approaches (i.e., API calls) and did not provide any characterization of the phenomenon.

To the best of our knowledge, the solution proposed in this study is the first to tackle the concept drift issue in Android malware using dynamic features and achieving long-term high performance. The previous solutions focused on static features (i.e., API calls) and spanned shorter time frames. In this regard, *DroidEvolver* (Xu et al., 2019) used API calls as features and obtained significant results in a 6 year-long time frame (i.e., 2011–2016), outperforming *MaMaDroid* (Onwuzurike et al., 2019), a prior solution. As shown in Fig. 10, the solution proposed in this study outperforms the state-of-the-art solution, *DroidEvolver*, both in time and detection performance (i.e., F1 score). More specifically, when the training period is excluded (i.e., 2011), our proposed solution achieves an average F1 score of 94.05% in the 2012–2016 time frame, whereas *DroidEvolver* averages 89.56% in the same period of time. When the longest period is considered, from 2012 to 2018, our proposed solution's average F1 score increases to 94.65%. To assess the statistical significance of the difference between both solutions, Wilcoxon's signed-rank test for paired scores (Japkowicz & Shah, 2011) was used. The results confirmed that our solution, in the time frame from 2012 to 2016, performs significantly better than DroidEvolver at the confidence level of 0.05 (p-*value* = 0.048). However, as our solution could not be trained with full 2011 data (i.e., just samples from Q3 and Q4), the statistical significance of the results could not be confirmed at confidence level 0.05 (p-*value* = 0.197) when the training period was included (i.e., 2011–2016).

Besides, is it worth noting that the data features used to build our approach are distinct from the ones in the related literature, including *DroidEvolver*. *DroidEvolver*, as most related solutions, uses API calls (i.e., static features), whereas our proposed solution uses system calls (i.e., dynamic features). In addition, none of the previous studies that dealt with Android concept drift provided any characterization of it, which hinders the interpretability of the results and the comprehension of the phenomenon. The solution proposed in this study has been proved effective to address concept drift in Android malware detection and characterize it.

The related solutions focused on F1 score performance, not providing any other performance metric. As a result, the comparison between solutions is restricted to the F1 score metric. For the sake of completeness and better comparison of other solutions with this work, a summary of other relevant performance metrics of our proposed solution is provided as follows. The proposed solution averaged 95.17% precision, 94.14% recall, and 89.49% specificity in the 2012–2018 time frame. These metrics emphasize the goodness of the proposed solution to effectively tackle concept drift while keeping high-performance metrics for the whole study period.

A distinctive point of this study is the evaluation of distinct timestamps to date the apps and the assessment of their impact on concept drift detection and handling. The *KronoDroid* data set enabled the usage of distinct timestamps on our evaluation, thus providing results based on relevant timestamps (i.e., *last modification* and *first seen*). To the best of our knowledge, no previous study in the field has evaluated distinct timestamps for concept drift detection and handling. More precisely, the concept drift-related studies in the literature do not usually provide details about the used timestamp or justify the usage of a specific approach. But, if they do, they do not assess the reliability of the timestamp. In this study, we address such issues by providing and comparing two relevant and useful timestamps for Android malware detection concept drift handling. The systematic usage of an *internal* timestamp (i.e., *last modification*) rather than *external* timestamps (e.g., *first seen*) has proved to be reliable and accurate to handle and characterize the phenomenon. Besides, the usage of the last modification timestamp may help to avoid errors and data misplacement caused by human-related techniques (e.g., user submission delay), thus enhancing historical accuracy. As shown in Section 4.3, the proposed solution using the *last modification* timestamp proved to be more accurate and reliable than when the *first seen* timestamp was used to locate applications in the Android historical timeline, thus generating a more effective detection solution.

The features used in this study (i.e., system calls) have demonstrated great effectiveness and consistency to deal with concept drift. In this regard, just a small subset of the whole feature set was used to build an

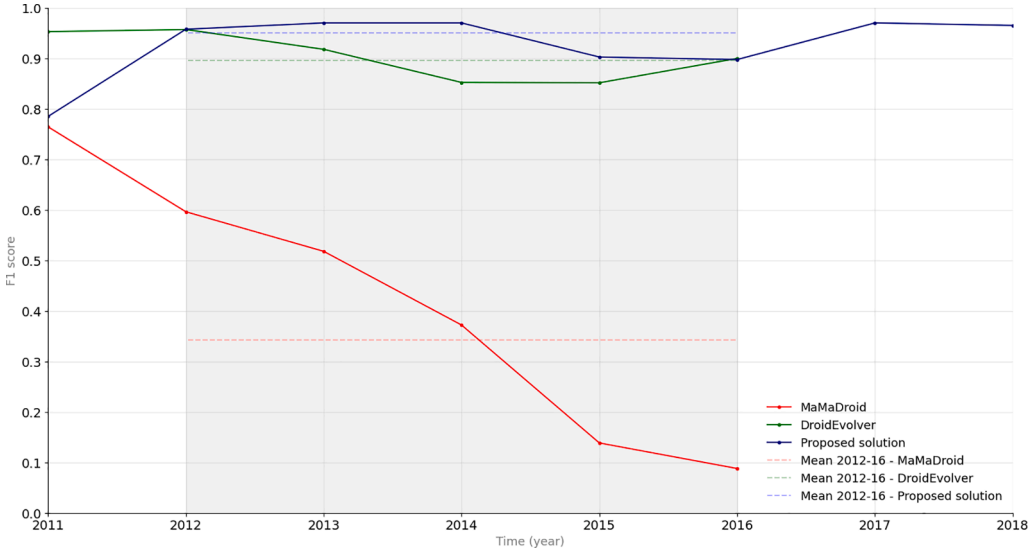


Fig. 10. Comparative performance of the proposed solution with the state-of-the-art solutions.

effective solution (i.e., 97 features). The system calls feature set is consistent, as system calls are rarely modified at kernel level which provides long-time stability for the feature set. This is radically different from the changing nature of API calls, which are prone to suffer constant modifications and the addition of new features on every new Android framework release. This constant modification of the features generates the need to constantly update the feature set to keep the models updated, which may end up in an increasingly large, ever-growing feature set. For instance, in *DroidEvolver*, the API calls feature set grew from 14,327 features in 2011 to 52,001 features in 2016 (Xu et al., 2019). On the contrary, our solution kept the same feature set constant, composed of just 97 features, for a longer period of time (i.e., 2011 to 2018). Furthermore, the usage of highdimensional feature sets may harm the performance of machine learning-based detection systems, a phenomenon called the *curse of dimensionality* (Aggarwal, 2015). Therefore, our proposed solution simplifies the learning process, based on a small subset of features, avoiding high-dimensional data issues in the long run. As a result, although the acquisition of dynamic features is generally more complex and time-consuming than the collection of static features, they have proved to be more reliable, efficient, robust, and consistent over time, thus enabling the generation of a more effective detection system.

Finally, our characterization results demonstrate that the relevant system calls to discriminate benign applications do not show rapid variations in consecutive periods, whereas dominant feature sets for malware samples suffer radical changes. These results may help malware analysts get a general idea about the evolution of benign and malware samples and understand the reason behind concept drifts, thus improving the trust of the experts in the learning models. However, despite the advantages shown by system calls to generate an effective detection model, an expert may not derive a clear understanding of what type of app behavior is induced by each feature as an individual system call can be associated with different system functions. Static features such as permissions or API calls can benefit more from our characterization approach due to a more comprehensible mapping between these features and the behavior of the application. We consider the application of our methods to those features as one of the main directions in our future work.

6. Conclusions

The evolving nature of Android malware has been neglected by the majority of the machine learning-based detection methods proposed in the related literature, thus disregarding the degenerative impact of feature changes over time (i.e., concept drift). The reduced number of solutions that considered the impact of the *time* variable focused on the usage of API calls as input features. API calls can be used effectively to discriminate malware and provide a relatively good representation of its behavior. However, system calls, the most used dynamic features for Android malware detection, which allow capturing the real behavior of the apps at run-time, and are robust to obfuscation and encryption techniques, have not been considered in concept drift solutions.

This experimental study proposes a method that uses system calls data gathered on real Android devices to detect, characterize, and handle Android malware concept drift effectively.

Our proposed method minimizes model retraining and uses a pool of classifiers trained with recent data to adapt effectively to malware evolution. The experimental results evidence that system calls can effectively discriminate malware in the presence of concept drift using the proposed method, providing high-performance metrics for an extended period of time. More precisely, in a 7 year-long test, the proposed solution averaged 94.65% F1 score, 95.17% precision, 94.14% recall, and 89.49% specificity, proving the goodness of our solution to adapt and react to the concept drift issues that affect Android malware detection while keeping high-performance metrics. The proposed solution outperforms the state-of-the-art solutions for Android malware detection under concept drift conditions.

A critical issue to deal effectively with concept drift is the timestamp used to date the apps. In this study, distinct timestamps are analyzed and compared regarding concept drift-related performance. To the best of our knowledge, this is the first study on Android malware detection to perform such a comparison.

Lastly, the proposed solution allows the characterization of the changes in the data by analyzing the important features on the best classifiers. The observation of concept drift in different time horizons was used to describe the important features and determine their evolution and usefulness over time. In this regard, some features were found to have a prolonged (i.e., long-term) influence on the model performance, whereas others showed an impact limited to the short term (i.e., specific periods). This fact evidenced the existence of concept drift and provided insights into its character. More specifically, when the malware recognition task was analyzed (i.e., recall), it was observed that a small number of features had importance in each period, showing significant concept drifts and rapid feature importance changes. These facts were not observed for the benign software detection task (i.e., specificity).

The usage of the proposed method in combination with other relevant data features for Android malware detection, such as security *permissions*, remains part of our future work.

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, Writing – review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Aggarwal, C. C. (2015). *Data mining: The textbook*. Springer.

Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., et al. (2020). When malware is packin'heat; limits of machine learning classifiers based on static analysis features. *In Network and Distributed Systems Security (NDSS) Symposium 2020*.

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* (pp. 51–67). Springer..

Altmann, A., Tolo¸si, L., Sander, O., & Lengauer, T. (2010). Permutation importance: A corrected feature importance measure. *Bioinformatics, 26*, 1340–1347. https://doi.org/10.1093/bioinformatics/btq134

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., & Rieck, K. (2020). Dos and don'ts of machine learning in computer security. *arXiv preprint arXiv:2010.09470*, .

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, 14*, 23–26.

Barbero, F., Pendlebury, F., Pierazzi, F., & Cavallaro, L. (2020). Transcending transcend: Revisiting malware classification with conformal evaluation. *arXiv preprint arXiv: 2010.03856*, .

Breiman, L. (2001). Random Forests. *Machine Learning, 45*, 5–32. 10.1023/ A: 1010933404324.

Broersma, M. (2020). Android hit by 'incredibly sophisticated' malware. https://www.silicon.co.uk/workspace/android-sophiticatedmalware-344222.

Cai, H. (2020). Assessing and improving malware detection sustainability through app evolution studies. *ACM Transactions on Software Engineering and Methodology (TOSEM), 29*, 1–28.

Cai, H., Meng, N., Ryder, B., & Yao, D. (2018). Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security, 14*, 1455–1470.

Chebyshev, V. (2021). Mobile malware evolution 2020. https:// securelist.com/mobile-malware-evolution-2020/101029/.

Cruz, R. M., Sabourin, R., Cavalcanti, G. D., & Ren, T. I. (2015). Metades: A dynamic ensemble selection framework using meta-learning. *Pattern recognition, 48*, 1925–1935.

Elwell, R., & Polikar, R. (2011). Incremental learning of concept drift in nonstationary environments. *IEEE Transactions on Neural Networks*, *22*, 1517–.

Fu, X., & Cai, H. (2019). In *On the deterioration of learning-based malware detectors for android* (pp. 272–273). IEEE.

Guerra-Manzanares, A., Bahsi, H., & Nõmm, S. (2021). KronoDroid: Time-based hybrid-featured dataset for effective android malware detection and characterization. *Computers & Security, 110*, 102399.

Gözüaçık, Ö., & Can, F. (2020). Concept learning using one-class classifiers for implicit drift detection in evolving data streams. *Artificial Intelligence Review, URL:*. https://doi.org/10.1007/s10462-020-09939-x. 10.1007/s10462-020-09939-x

Guerra-Manzanares, A., Bahsi, H., & Nõmm, S. (2019a). Differences in Android Behavior Between Real Device and Emulator: A Malware Detection Perspective. In *Proceedings of the Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)* (pp. 399–404). IEEE.

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)* (pp. 1–8). IEEE.

Guerra-Manzanares, A., Nõmm, S., & Bahsi, H. (2019c). In-depth feature selection and ranking for automated detection of mobile malware. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP* (pp. 274–283). INSTICC, SciTePress.

Hu, D., Ma, Z., Zhang, X., Li, P., Ye, D., & Ling, B. (2017). The concept drift problem in android malware detection and its solution. *Security and Communication Networks, 2017*. https://doi.org/10.1155/2017/4956386

Japkowicz, N., & Shah, M. (2011). *Evaluating Learning Algorithms: A Classification Perspective*. New York, NY, USA: Cambridge University Press.

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}*.

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

Kaspersky (2021). Machine learning for malware detection. https: //media.kaspersky.com/en/enterprise-security/Kaspersky-LabWhitepaper-Machine-Learning.pdf.

Ko, A. H., Sabourin, R., & Britto, A. S., Jr (2008). From dynamic classifier selection to dynamic ensemble selection. *Pattern recognition, 41*, 1718–1731.

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 of Things Journal, 6*, 6668–6680. https://doi.org/10.1109/JIOT.2019.2909745

Liu, F. T., Ting, K. M., & Zhou, Z. H. (2012). Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data, 6*, 1–44. https://doi.org/10.1145/2133360.2133363

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. *Artificial Intelligence*, *209*, 11–28. URL: http://dx.1016/.

Luckner, M. (2019). Practical web spam lifelong machine learning system with automatic adjustment to current lifecycle phase. *Security and Communication Networks, 2019*. https://doi.org/10.1155/2019/6587020

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

Margara, A., & Rabl, T. (2018). Definition of data streams. In S. Sakr, & A. Zomaya (Eds.), *Encyclopedia of Big Data Technologies* (pp. 1–4). Cham.

Microsoft (2020). Sophisticated new android malware marks the latest evolution of mobile ransomware. https://www.microsoft.com/security/ blog/2020/10/08/ sophisticated-new-android-malware-marks-thelatest-evolution-of-mobile-ransomware.

Mohd Razali, N., & Bee Wah, Y. (2011). Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. Journal of Statistical Modeling and Analytics, *2*, 21–33.

Mutz, D., Valeur, F., Vigna, G., & Kruegel, C. (2006). Anomalous system call detection. *ACM Transactions on Information and System Security, 9*, 61–93. https://doi.org/10.1145/1127345.1127348

Narayanan, A., Chandramohan, M., Chen, L., & Liu, Y. (2017). Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence, 1*, 157–175. https://doi.org/10.1109/TETCI.2017.2699220

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

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 Transactions on Privacy and Security (TOPS), 22*, 1–34.

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).

Rafter, D. (2021). Android vs. ios: Which is more secure? https://us.norton.com/internetsecurity-mobile-android-vs-ios-whichis-more-secure.html.

Reddy, R., Swamy, M. K., & Kumar, D. A. (2021). Feature and sample size selection for malware classification process. In *ICCCE 2020* (pp. 217–223). Springer.

Ramirez-Gallego, S., Krawczyk, B., Garcia, S., Wózniak, M., & Herrera, F. (2017). A survey on data preprocessing for data stream mining: Current status and future directions. *Neurocomputing, 239*, 39–57. https://doi.org/10.1016/j.neucom.2017.01.078

Ruiz-Heras, A., Garćia-Teodoro, P., & Śanchez-Casado, L. (2017). ADroid: Anomaly-based detection of malicious events in Android platforms. *International Journal of Information Security, 16*, 371–384.

Seiffert, C., Khoshgoftaar, T. M., Van Hulse, J., & Napolitano, A. (2010). RUSBoost: A hybrid approach to alleviating class imbalance. *IEEE Transactions on Systems, Man, and Cybernetics Part A: Systems and Humans, 40*, 185–197. https://doi.org/10.1109/TSMCA.2009.2029559

Statista (2021a). Development of new android malware worldwide from june 2016 to march 2020. https://www.statista.com/statistics/680705/ global-android-malware-volume.

Statista (2021b). Distribution of leading android malware types in 2019. https://www.statista.com/statistics/681006/share-of-androidtypes-of-malware.

Statista (2021c). Mobile operating system market share worldwide february 2021. statcounter..

Suarez-Tangil, G., Dash, S. K., Ahmadi, M., Kinder, J., Giacinto, G., & Cavallaro, L. (2017). Droidsieve: Fast and accurate classification of obfuscated android malware. In *In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (pp. 309–320).

Townsend, K. (2020). How smartphones have become one of the largest attack surfaces. https://blog.avastcom/smartphones-and-increasingmobile-threats-avast.

Université du Luxembourg (2021). Androzoo - lists of apks. https://androzoo.uni.lu/lists.

Unuchek, R. (2018). Mobile malware evolution 2017. https://securelist.com/mobile-malware-review-2017/84139.

Xu, K., Li, Y., Deng, R., Chen, K., & Xu, J. (2019). In *Droidevolver: Self-evolving android malware detection system* (pp. 47–62). IEEE.

Yang, L., Guo, W., Hao, Q., Ciptadi, A., Ahmadzadeh, A., Xing, X., et al. (2021). CADE}: Detecting and explaining concept drift samples for security applications. *In 30th {USENIX Security Symposium ({USENIX} Security 21)*.

Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., et al. (2020). Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In *In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (pp. 757–770).

Zhao, L., Wang, J., Chen, Y., Wu, F., Liu, Y. et al. (2021). Droidmfc: A novel android malware family classification scheme based on static analysis. *arXiv preprint arXiv: 2101.03965*, .

Zhou, Y., & Jiang, X. (2012). In *Dissecting android malware: Characterization and evolution* (pp. 95–109). IEEE. https://doi.org/10.1109/SP.2012.16.

Zyblewski, P., Sabourin, R., & Wózniak, M. (2021). Preprocessed dynamic classifier ensemble selection for highly imbalanced drifted data streams. *Information Fusion, 66*, 138–154. URL: .org/.