Hybrid Multilevel Detection of Mobile Devices Malware Under Concept Drift

A. Augello, A. De Paola, G. Lo Re

Journal of Network and Systems Management
DOI: https://doi.org/10.1007/s10922-025-09906-3

Hybrid Multilevel Detection of Mobile Devices Malware under Concept Drift

Andrea Augello[1*], Alessandra De Paola[1] and Giuseppe Lo Re[1]

[1]Department of Engineering, University of Palermo, Palermo, Italy.

*Corresponding author(s). E-mail(s): andrea.augello01@unipa.it;
Contributing authors: alessandra.depaola@unipa.it; giuseppe.lore@unipa.it;

Abstract

Malwares are a major threat to the security of mobile devices, and Machine Learning (ML) is a widespread approach to automatically detect them. However, running ML analysis pipelines can be excessively burdensome for energy-constrained mobile devices. On the other hand, completely off-loading all the analysis to a remote server can introduce unacceptable communication overheads and delays in the detection process. In this paper, we propose a multilevel approach for malware detection on mobile devices that combines a lightweight local analysis of static features with a more computationally expensive remote analysis of dynamic features, through the adoption of ML methods. However, the effectiveness of automatic malware detection systems based on ML is often limited by unforeseen variations in the statistical characteristics of the observed data. This phenomenon, known as concept drift, can lead to a degradation of the performance of ML models over time. The proposed malware detection system is equipped with self-evaluation capabilities, enabling it to detect the occurrence of periods when its predictions become unreliable due to concept drift so that appropriate response strategies can be activated. In particular, when such critical events occur, the self-evaluation agent triggers the execution of an additional layer of analysis, hosted by a remote server, which allows the system to react to the unexpected reduction in its detection capabilities. The computational cost of the detection process is minimized by limiting the remote analysis to only those samples for which the analysis performed on-board the mobile device is likely to incorrectly classify the app.

Keywords: Android malware detection, mobile security, concept drift, machine learning

1 Introduction

Mobile devices are becoming increasingly popular, and they are now used for a wide range of activities, from communication to banking and shopping. However, the increasing popularity of mobile devices has also made them an attractive target for malware developers. The increasing spread of malware for mobile devices poses a serious threat to the security of mobile devices [1] and the privacy of their users [2]. Thus, developing effective detection methods is of paramount importance.

Machine learning (ML) has been successfully applied to malware detection, both in mobile [3] and desktop environments [4]. Typically, ML-based systems analyze app metadata, source, or behavior in order to extract a set of representative features. This information is then processed by one or more classifiers that determine whether the app is malicious or not. The features employed by the classifiers can be divided into two categories: static and dynamic. Static features are extracted from the app without executing it, and include information such as the requested permissions, the declared activities and services, the libraries used, and other metadata. Dynamic features are extracted from the execution of the app [5], and include information such as API calls, system calls, and the generated network traffic.

Extracting static features is more computationally efficient than extracting dynamic features, as it does not require the execution of the app. Static features, however, are also more susceptible to code obfuscation techniques [6], dynamic code loading [7], and runtime fetching of malicious payloads [8]. On the other hand, dynamic features are more robust to code obfuscation techniques, but their extraction has a greater computational cost, requiring the execution of the app in a virtualized or sandboxed environment, which prevents their use in on-device detection. Thus, it can be advantageous to combine static and dynamic features for classification [9, 10].

Despite the benefits of adopting machine learning in malware detection [11], another major challenge in its application in this domain is the problem known as *concept drift*. Concept drift occurs when the statistical properties of the target variable, or the relationship between input variables and the target variable, change over time [12]. As a consequence of concept drift, machine learning models trained on historical data experience a decrease in their accuracy and effectiveness over time [13]. In the context of malware detection for mobile devices, concept drift can occur due to various reasons, such as the evolution of malwares, the introduction of previously unseen malicious functionalities, or modifications made by malware developers to avoid detection [14]. Analysis based on dynamic features is more robust to concept drift than analysis based on static features, as it relies on features extracted from the execution of the app, which is not as affected by code obfuscation techniques [6]. However, it is not feasible to rely entirely on dynamic features for malware detection, both for computational and communication reasons. A system that relies entirely on a mobile device would quickly consume the memory and energy resources if it had to perform the extraction of dynamic features locally, rendering task offloading to a remote server necessary. On the other hand, a system that always relies on a cloud system to perform the dynamic analysis could generate excessive communication overhead [15], as well as excessively overload the remote server.

To overcome these limitations, this paper proposes a two-level approach for the detection of malware on mobile devices that combines static and dynamic feature analysis making use of a self-evaluation agent to decide whether to run the remote analysis or not. The objective of the agent is to classify most of the samples with a less computationally expensive local static analysis on the mobile device. When the local analysis is likely to incorrectly classify the app, the self-evaluation agent triggers the execution of the more robust analysis on a cloud server. The proposed architecture can be applied to any mobile context but, in this work, the focus is on the Android operating system due to its widespread adoption [16] and the availability of a large number of malware samples [17].

The most relevant novel features of the proposed work are as follows:

- Differently from most of the existing approaches, which need both static and dynamic features for every sample to be classified, this work proposes a less computationally expensive two-level approach where the dynamic analysis is triggered on a case-by-case basis.
- A self-evaluation agent on the mobile device learns to recognize concept drift at the local analysis level and to trigger the execution of the more robust second level on the cloud server.
- The computational cost of the detection process is minimized by the self-evaluation agent, which triggers the execution of the second level only when the local analysis is likely to incorrectly classify the app.

The remainder of this paper is organized as follows. Section 2 presents related works. Section 3 describes the proposed architecture. Section 4 presents the experimental evaluation. Finally, Section 5 concludes the paper.

2 Related works

The landscape of malware detection is marked by the constant evolution of malicious techniques to avoid detection. Traditional signature-based methods, while lightweight, fail against unknown malware due to their susceptibility to evasion through simple modifications by malware developers [18]. In response, the field has seen a burst of research focusing on machine learning-based approaches [19], particularly in the realm of Android malware detection [11]. However, the computational burden of dynamic analysis, wherein each app must be executed in a simulated environment, has led to solutions involving specialized hardware or remote server execution [20]. This has prompted the exploration of hybrid approaches, combining static and dynamic features through cascades of classifiers [21, 22, 9]. Despite these advancements, relying exclusively on static analysis confidence as an indicator for triggering dynamic analysis is not foolproof, as static features can be affected by concept drift, resulting in false negatives [23].

2.1 Hybrid malware analysis

While signature-based approaches are prevalent in commercial malware detection solutions and are lightweight, they prove ineffective against previously unknown malware

samples [18]. Malware developers can easily evade signature-based detection by altering their software, introducing polymorphism, and obfuscating their code using compression techniques [24].

Machine learning techniques can generalize to previously unseen malware samples. Thus, most research effort in malware detection for mobile devices has focused on machine learning-based approaches [11].

Several solutions employ a set of characteristics consisting of both static and dynamic features to perform the classification [25]. The static features are extracted from the app without executing it and can range from immediately available information such as the permissions requested, the declared activities and services, and the libraries used [26], to more complex features such as the control flow graph of the app [27, 28]. These features can be obtained from the *AndroidManifest.xml* file, which is included in every Android app, or by disassembling the app and analyzing the source code [29]. Dynamic features are extracted from the execution of the app [30], and include information such as API calls, system calls, generated network traffic [31], and the memory contents [32].

Obtaining dynamic features is computationally expensive, as it requires the execution of the app in a simulated environment for every app to be classified.

Dynamic analysis can incur such a high computational cost that the introduction of dedicated specialized hardware modules have been proposed to speed up the process [20], although in the context of desktop malware detection. The limited energy resources of mobile devices makes this approach unfeasible in the context of Android malware detection. Therefore, the dynamic analysis has to be performed on a remote server, which introduces additional latency in the detection process.

To tackle this significant shortcoming, some studies have proposed hybrid approaches that use a cascade of classifiers using first static and then dynamic features [21, 22, 9]. For mobile malware detection, Gharib et al., [21] proposed a hybrid classifier that first uses static features to classify the app. If the classification probability is below a threshold, the app is executed in a sandbox and monitored until a final decision is made based on the sequence of API calls made by the suspected malware. Ding et al., [22], instead, analyze the network traffic generated by the apps whose static features are classified with low confidence.

2.2 Concept drift in malware detection

The aforementioned approaches assume that the confidence of the static analysis is a reliable indicator of the accuracy of the classification. However, such an assumption is not always true, as the static features can be affected by concept drift, especially sudden drifts (Fig. 1). Thus, a classifier can mistakenly label malware samples as benign with high confidence [23].

Misclassification due to concept drift is a common issue as malware developers frequently modify their apps to avoid detection. In many cybersecurity domains, this phenomenon hinders the ability of automatic detection systems to distinguish between benign and malicious samples [33]. Automated detection systems are particularly vulnerable to adversarial attacks, where an attacker modifies the malware to evade detection by the classifier. For instance, Ami et al., [34] proposed a semi-automated framework that can generate Android malware samples with malicious functionalities,

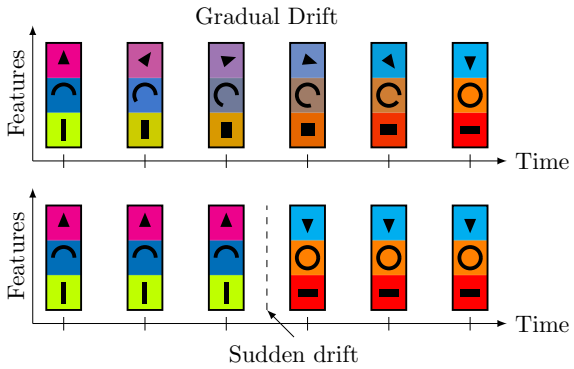

Fig. 1 Under gradual concept drift, the distribution of the data changes slowly over time, while under sudden concept drift, the distribution changes abruptly.

such as leaking sensitive data, and modifies them to evade detection from static analysis tools. Additionally, new families of malware are constantly being developed, and the features of these new malware families are not present in the training set. Detecting the emergence of new malware families is a challenging task and requires constant attention from the security community [35]. In the presence of concept drift, systems that rely exclusively on the classification confidence of the static analysis would not trigger the dynamic analysis, leading to false negatives.

Thus, in spite of the widespread use of machine learning in malware detection, concept drift can negatively affect the performance of machine learning models in this domain [13, 36]. The presence of concept drift can cause the performance of ML models to degrade rapidly if the incoming data is not independently and identically distributed with respect to the training data [37]. Yizheng et al., [14] have shown that after training an Android malware classifier on one year's worth of data, the F1-score dropped by 23 percentage points after just six months of concept drift. Even more advanced dynamic behavioral analysis tools such as MaMaDroid are not immune to performance drops over time [38]. This highlights the importance of continuously updating and retraining models to maintain their accuracy and effectiveness in detecting Android malware [39]. To address concept drift in Android malware detection, researchers have proposed various techniques, such as periodically updating the classifiers if concept drift is detected [40]. Moreover, some authors have focused on developing methods to detect and effectively address concept drift in Android malware detection using system calls [41] and adaptive classifiers [42].

The use of dynamic features can offer more robustness to concept drift compared to employing static ones [6]. Despite this advantage, classifiers relying on dynamic features have a lower accuracy compared to those trained on static features. The lower effectiveness of dynamic features is attributable to the capabilities of many types of malware to detect when they are being run in a simulated environment and alter their behavior accordingly [22, 43]. This, coupled with the higher computational cost of the dynamic analysis, makes it unfeasible to only use dynamic features for classification.

Despite these efforts, concept drift remains a notable challenge in the detection of malware on mobile devices. Recent works have focused on developing methods to

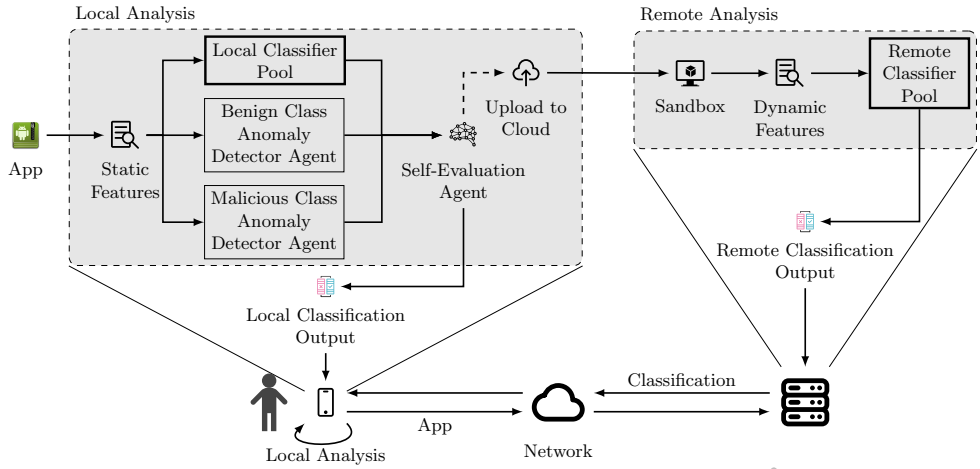

Fig. 2 Hybrid Multi-level Architecture: an incoming sample is first analyzed by the local classifier pool, and the self-evaluation agent decides whether to trigger the remote analysis. The remote analysis is performed on a cloud server and the final classification is returned to the user.

detect concept drift in order to update the classifiers exploited by malware detection systems focusing on desktop ransomware detection [44]. Nevertheless, the problem of providing effective solutions to maintain the accuracy and effectiveness of machine learning models when a drift is ongoing, before a new model can be trained, is still open and largely unaddressed, especially in the context of Android malware detection. To the best of the authors' knowledge, no existing approach is able to effectively detect the occurrences when sudden concept drift is present and trigger a further analysis level to maintain acceptable detection performance. More effort is needed to develop robust and efficient solutions to maintain the accuracy and effectiveness of machine learning models in this domain.

3 Hybrid Multi-level Architecture

The architecture presented in this paper is composed of two levels, as shown in Figure 2. Part of the system resides on the mobile Android device, and part of it is deployed on a remote cloud server. The invocation of the remote analysis is triggered on-demand by the mobile device. Such multi-tier architecture allows the system to provide a quick response to the user in most cases, using only the computational capabilities of the mobile device. Whenever a more in-depth analysis is required the cloud system is invoked. This necessity can be caused either by an application whose classification is more challenging or by periods in which the system is experiencing concept drift. The cloud system can employ more computational and hardware resources, without overburdening the limited resources of the mobile device.

In the first level, the app is analyzed by exploiting static features, and a classifier is used to determine whether the app is benign or malicious. These features can be extracted without executing the software, on the mobile device itself. Furthermore, the

mobile device hosts a self-evaluation agent that accepts as input the output of the static analysis and the anomaly scores of the app to determine whether the classification is reliable or not. If the classification is deemed reliable, the analysis terminates at this stage. Only if the self-evaluation agent determines that the classification is unreliable, the second level is invoked, and the app is sent to the cloud server for further analysis together with the context required to perform the analysis. On the second level, the app is executed in a simulated environment on a remote server, and the classification is performed on the dynamic features extracted from the execution. Since some user data might be needed to perform the dynamic analysis, the remote execution environment is customized for each user, and the user's data is not shared with other users, as mobile devices often contain sensitive information.

This two-level approach ensures that the end user can obtain high-quality classification results without excessive computational overhead for the mobile device. At the same time, by limiting the number of dynamic analyses performed, the system can reduce the communication overhead, the latency, and the computational cost of the detection process.

3.1 Concept drift resilience

Since one of the main objectives of the system is to be resilient to concept drift, the samples are classified by an ensemble of classifiers, instead of a single classifier. The classifier pool can be updated periodically to ensure that the classifiers are able to detect the most recent malware families. Incrementally updating a classifier pool helps the system to adapt to gradual concept drift [45]. At the same time, by maintaining some of the older classifiers in the pool, the system can retain the ability to detect older malware families, which may still be present in the wild.

3.1.1 Classifier pool ensemble selection

The classifier pool, denoted as Π, is periodically updated to ensure that the system is able to detect the most recent malwares, reducing the impact of gradual concept drift. In this work, base classifiers in the pool are implemented through random forests with 300 trees each.

As shown in Figure 3, whenever a sample needs to be classified, only a subset of the classifiers in the pool is selected to evaluate the sample. The outputs of the selected classifiers are then combined to obtain a final prediction.

The selection of the optimal subset of classifiers is performed through the k-Nearest ORAcles-Union (KNORA-U) algorithm [46]. The KNORA-U dynamic classifier selection algorithm is a method for selecting multiple models (referred to as "oracles") that perform well on the neighborhood of a test sample. The predictions of the selected oracles are then combined using a majority voting mechanism to obtain a final output prediction.

The selection algorithm relies on the availability of a training set, which is used to determine the area of competence of each classifier in the pool. Given a sample to classify, KNORA finds its nearest K neighbors in the training set. Then all the classifiers in the pool are tested on the neighbors of the sample. The classifiers which

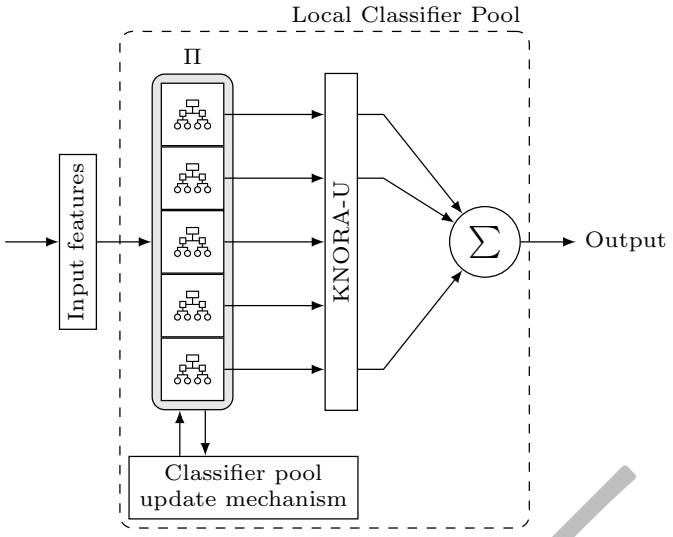

Fig. 3 Concept-drift resistant ensemble learning strategy for Classifier Pool management: only a subset of the classifiers in the pool is selected to evaluate the sample and their outputs are combined to obtain a final prediction.

correctly classify the neighbors of the incoming sample constitute the ensemble for classifying the sample.

The KNORA algorithm has two main variants:
KNORA-Union and KNORA-Eliminate. The KNORA-Eliminate (KNORA-E) version of the algorithm selects all the classifiers that correctly classify *all* the neighbors of the test sample. The KNORA-Union (KNORA-U) version of the algorithm selects all the classifiers that correctly classify *at least one* of the neighbors of the test sample.

More formally, given a sample x to classify, a pool of trained classifiers $\Pi = \{c_1, \ldots, c_m\}$, and a set $D = \{(x_1, y_1), \ldots, (x_k, y_k)\}$ of the nearest neighbors of x in the training set, the KNORA-U algorithm selects the classifiers $c_i \in \Pi$ that satisfy Eq. 1.

$$c_i \in \Pi \quad \wedge \quad \exists (x_j, y_j) \in D \quad \text{s.t.} \quad c_i(x_j) = y_j \tag{1}$$

On the other hand, the KNORA-E algorithm selects the classifiers $c_i \in \Pi$ that satisfy Eq. 2.

$$c_i \in \Pi \quad \wedge \quad \forall (x_j, y_j) \in D, \quad c_i(x_j) = y_j \tag{2}$$

In this work, the KNORA-U version of the algorithm is used, as Zyblewski et al. [47] have shown through extensive empirical evaluation that, in most cases, it outperforms the KNORA-E version.

3.1.2 Classifier pool update mechanism

The algorithm used to create and update the classifier pool is adapted from [41], and utilizes a dynamic ensemble selection algorithm to choose from a dynamically updated pool of classifiers [47].

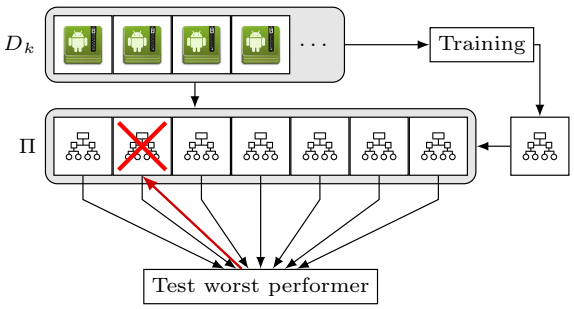

Fig. 4 Local classifier pool update mechanism: periodically, the worst classifier in the pool is removed. The removed classifier is replaced by a new classifier trained on the most recent data.

When the system is first deployed, the initial training dataset is partitioned into multiple splits. Each split is used to train a different classifier, and the classifiers are then added to a pool Π. Then, after each fixed period of time k, a new chunk of data D_k is collected. This data is used to update the pool of classifiers as shown in Figure 4.

First, all the classifiers in the pool are evaluated on the new data. Then, the classifier with the worst performance on the new data is removed from the pool according to Eq. 3.

$$c_{worst} = \arg\min_{c \in \Pi} \quad |\{(x, y) \in D_k \quad \text{s.t.} \quad c(x) = y\}| \tag{3}$$

Finally, a new classifier is trained to minimize the loss on the new data D_k and it is added to the pool Π to be used for future classifications. The pseudocode reported in Algorithm 1 details the classifier training algorithm more formally.

Algorithm 1 Classifier pool training algorithm.

Input: Π = pool of classifiers
D_k = data collected in the k-th chunk
n number of classifiers in Π

$\Pi \leftarrow \emptyset$
while True **do**
  if $k == 0$ **then**
    Split D_0 into n folds $D_0^1, \ldots, D_0^n$
    for $i \leftarrow 1$ **to** n **do**
      $\Pi \leftarrow \Pi \cup \{\text{train\_classifier}(D_0^i)\}$
    end for
  else
    $c_{worst} \leftarrow \arg\min_{c \in \Pi} \quad |\{(x, y) \in D_k \quad \text{s.t.} \quad c(x) = y\}|$
    $\Pi \leftarrow \Pi \setminus \{c_{worst}\}$
    $\Pi \leftarrow \Pi \cup \{\text{train\_classifier}(D_k)\}$
  end if
end while

By updating the classifier pool with the most recent data, the system always has at least one classifier in the pool that is suitable for the current data distribution, mitigating the effect of concept drift. The classifier pool lets the system also handle recurring concept drifts. Recurring drifts are a common type of concept drift, where the data distribution periodically returns to one of the previous states. Thus, discarding the classifiers trained on the previous data could cause the system to lose the ability to detect samples from one the previous data distributions [48]. Therefore, to ensure robustness against this form of concept drift, in our architecture, the system maintains a pool of classifiers instead of relying exclusively on the most recent one.

3.2 Local analysis

The first level of the system performs the analysis of the static features of the software, which are extracted from the app, without executing it. The static features are extracted from the *AndroidManifest.xml* file and entail information such as which permissions are requested, the declared activities and services, and the libraries used. These features are encoded as binary vectors, where each element represents the presence or absence of a specific feature. These relatively simple static features are used in place of more complex graph-based features, such as the control flow graph of the app, to reduce the computational cost of the analysis since the static features are extracted and analyzed on the mobile device. Not all the static features are used for classification. Activity names, for instance, are too specific and can be easily modified by malware developers to avoid detection [49]. Only the permissions requested by the app are used for classification.

The set of static features is fed to a pool of classifiers that adopt ensemble learning strategies to classify the app. At this level, the classification subsystem is supported by two anomaly detection agents, whose purpose is to allow the self-evaluation agent to detect whether the local analysis is undergoing a concept drift.

3.2.1 Anomaly detection agents

As previously stated, confidence of the classification on the static features is not a reliable indicator of whether the classification is correct or not. Thus, we introduce two additional agents to the system. These agents provide a richer context for the self-evaluation agent to base its decision on.

The two anomaly-detection agents depicted in in Figure 2 perform unsupervised anomaly detection on the static features of the app and provide two anomaly scores, one for each class. The anomaly scores are computed with two unsupervised anomaly detection algorithms, one trained on benign samples and one trained on malware samples. The anomaly detection agents are trained on the same features used by the static classifier pool. The two agents have the same architecture, relying on an isolation forest [50] to compute the anomaly scores.

The isolation forest is an anomaly detection algorithm designed to efficiently identify outliers or anomalies within a dataset. This algorithm capitalizes on the notion that anomalies are typically isolated instances, standing out from the majority of normal data points. The underlying principle of the Isolation Forest algorithm is rooted

in recursive partitioning in order to isolate anomalies through a process of random splitting. The algorithm begins by randomly selecting a feature and a threshold value, creating a split that separates the data into two subsets. This process is repeated recursively until the sample under analysis is in a partition by itself. Consequently, the path length to isolate an anomaly serves as a robust measure of its divergence with respect to normal samples, with shorter paths indicating more anomalous instances. This anomaly detection algorithm is chosen because it is fast to train and to compute, and it was shown to be effective in this context [41].

It is worth noting that it is not redundant to include two anomaly detection agents. One agent is trained exclusively on benign samples while the other one is trained on malware samples. Thus, they each detect those samples that are anomalous with respect to their respective label. For instance, an attacker might modify a malware sample to cross the classifier's decision boundary and causes it to be classified as benign with high confidence, but the perturbed sample might still be anomalous with respect to the benign samples. Conversely, a benign application could be mistakenly classified as malware, but it could still be anomalous with respect to the malware samples in the training dataset.

3.3 Self-Evaluation Agent

When trying to ascertain whether an app is malicious or benign, running the dynamic analysis for each sample would be computationally expensive, producing unacceptable delays in the detection process for the end-user. Thus, the system also includes an agent tasked with deciding whether to run the remote analysis or whether the local analysis is sufficient. The *self-evaluation agent*, tries to recognize misclassifications in the local analysis level and appropriately triggers the execution of the more robust second level. The self-evaluation agent is the core component of the architecture, as it connects the local and the remote stages of the system.

The self-evaluation agent bases its decisions on four input features: the binary output of the local classifier, the classification probability of the local classifier encoded as a continuous value in the range $[0, 1]$, and the two anomaly scores of the app also encoded as binary values. Provided with these inputs, the self-evaluation agent is tasked with deciding whether to run the remote dynamic features analysis or use the output of the local classifier as the final classification. For each received sample, the self-evaluation agent outputs one of two possible decisions:

- `reliable`: the static classifier's output is deemed reliable by the self-evaluation agent, and the invocation of the remote analysis is not needed. In this case, the classification obtained from the local classifier pool is used as the final classification and is immediately returned to the user.
- `unreliable`: the static classifier's output is unreliable, and the APK needs to be uploaded to the cloud server to undergo further dynamic analysis before a final, reliable, classification can be obtained.

The self-evaluation agent is implemented as a feedforward neural network with 2 hidden layers of 16 neurons each. The input layer has 4 neurons, one for the output of the local classifier, one for the classifier probability, and one for each anomaly score.

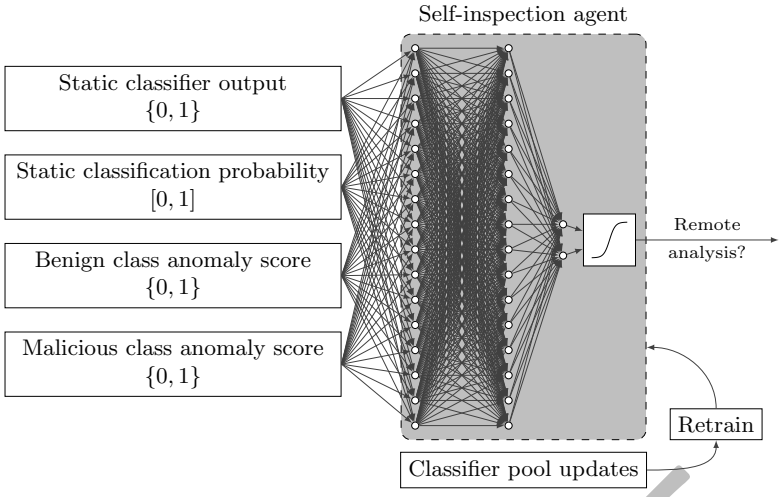

Fig. 5 The self-evaluation agent is implemented as a neural network that takes as input the output of the local classifier, the classification probability, the anomaly scores, and produces as output the decision to trigger the remote analysis. The self-evaluation agent is retrained when the classifier pool is updated.

Between layers, a Rectified Linear Unit (ReLU) activation function is used. The output layer has two neurons, one for each possible decision, and is followed by a Softmax activation function to obtain a probability distribution over the two possible decisions.

The agent learns its policy in a supervised fashion, using the samples in the training set as positive and negative examples to approximate the optimal policy. Since the local classifier is generally very accurate, the training dataset is imbalanced, with a large majority of samples being correctly classified. Thus, to avoid biasing the agent towards a specific decision, a balanced subset of the training dataset is used instead of the entire training set D_k. Out of the available training samples, stratified sampling is used to select a balanced dataset where the static classifier is wrong 50% of the time to avoid biasing the agent towards a specific decision. First, the set D_k^- of all the wrongly classified samples is extracted from the training set according to Eq. 4.

$$D_k^- = \{(x, y) \in D_k \quad \text{s.t.} \quad y \neq \Pi(x)\} \tag{4}$$

Then, an equal number of samples is randomly selected from the remaining correctly classified samples as per Eq. 5.

$$D_k^+ \subseteq D_k \setminus D_k^- \quad \text{s.t.} \quad |D_k^-| = |D_k^+| \tag{5}$$

The training dataset for the self-evaluation agent contains the output of the local classifier and the anomaly scores of the samples as features, labeled according to whether the local classifier's output is correct or not. The network is then trained to minimize the cross-entropy loss function between the output of the agent and the optimal policy. Every time the classifier pool is updated, the self-evaluation agent is also fine-tuned on the new samples.

In order to reduce the computational overhead of the dynamic features analysis on the remote server, an alternative, more conservative, approach is also presented. By default, the self-evaluation agent triggers the execution of the second level of the malware detection system whenever the local classifier's output is believed to be incorrect, for both false positives (FP) and false negatives (FN). Instead, the alternative approach only triggers the execution of the second level when the local classifier's output is believed to be incorrect for false negatives, while it does not trigger the execution of the second level for false positives, ignoring instances where the local classifier might be flagging a benign app as malware. This is a reasonable compromise since, as demonstrated in Section 4.1, the local classifier is more likely to misclassify malware samples as benign during sudden drifts, while the opposite is not true. Experiments in Section 4.4 show that this alternative approach does not excessively hamper the classification accuracy of the system, while it noticeably reduces the average number of remote analyses performed.

Thus, the alternative approach which only triggers the remote analysis for suspected false negatives would be more suitable in scenarios where the additional overhead of the dynamic analysis is not acceptable. For this approach to be viable, a minor reduction in the accuracy of the system should be tolerable. On the other hand, the default approach would be more suitable in scenarios where the slightly higher false positive rate is not acceptable. The default approach requires a cloud server with sufficient computational resources to extract the dynamic features of all the APKs that are suspected to be false positives to perform the additional analyses. The default approach would be more suitable, for instance, in a corporate environment where the security of the network is paramount. The alternative approach would be more suitable in a consumer environment to reduce the average cost of the detection process.

3.4 Remote analysis

For all those samples that the self-evaluation agent on the mobile device deems needing further analysis, the second level of the system is invoked. Such second level is located on a cloud server that is characterized by incomparably higher computational resources than the mobile device. A more expensive analysis of the dynamic features of the app can be performed exploiting such capabilities of the remote server.

To allow the remote server to analyze the app, the mobile device uploads the APK file to the cloud server together with the context required to perform the analysis. The app is executed in an isolated sandboxed environment. Sandboxes are a common technique to analyze potentially malicious software, as they allow the execution of the app in a controlled environment, preventing it from accessing sensitive data or performing malicious actions [51, 52]. Differently from other works [53], since in the proposed system the static features of the app have already been analyzed, the virtualized execution only extracts the dynamic features of the app. The virtualized execution is monitored, and all the API calls made by the app are recorded and provided as features for the ensemble of classifiers in the cloud server. Specifically, for each available API call, the number of times it is invoked is recorded as a feature. The dynamic features reflect aspects of the app's behavior that are not visible from the static features and thus prove a complementary source of information for the classification task.

The features provided by the local static analysis and the remote dynamic analysis do not overlap. Therefore, the remote pool is different from the local pool, and all the classifiers in the remote pool are trained separately. The classifier pool in the cloud server is created and updated in the same way as the local classifier pool, as described in Section 3.1.2. The classifiers in the remote pool are also implemented through random forests with 300 trees each. For each sample, the subset of classifiers selected to evaluate the sample is determined with the same KNORA-U algorithm described in Section 3.1.1.

Finally, the complete malware detection system is shown in Algorithm 2. The `FN_only` operation mode refers to the alternative approach described in Section 3.3.

Algorithm 2 Proposed malware detection system.

Input: Android APK file f, operation_mode
Output: Classification of f as `benign` or `malware`

Local analysis on the mobile device
$x \leftarrow$ static features of f
$y \leftarrow$ local_classifier(x)
$a_1 \leftarrow$ benign_anomaly_score(x)
$a_2 \leftarrow$ malware_anomaly_score(x)
if self-evaluation_agent(y, a_1, a_2) = `reliable` **then**
  return y
else if operation_mode = `FN_only` **and** y = `malware` **then**
  return y
else
  upload f to remote analysis environment
  Remote analysis on the cloud server
  execute f in a sandbox
  $x' \leftarrow$ dynamic features of f
  $y \leftarrow$ remote_classifier(x')
  Send y to the mobile device
  return y
end if

4 Experimental evaluation

In order to assess the effectiveness of the proposed system, the KronoDroid dataset [17] was adopted, as it contains both static and dynamic features for each sample, and contains samples collected over a long period of time from diverse sources, so that the effectiveness in mitigating the impact of concept drift could be evaluated. This dataset contains static and dynamic features for 28,745 malware samples and 35,256 benign samples, labeled with the corresponding timestamp, spanning from 2008 to 2020. This dataset contains examples of modern malwares that employ advanced evasion techniques, such as obfuscation, polymorphism, and anti-analysis mechanisms, and is

thus more suitable for evaluating the effectiveness of the proposed system in detecting modern malware [54] compared to other widely used datasets, such as the Drebin dataset [55]. Since the first and last years of the dataset contain a small number of samples, they are not used in the experiments. Consequently, the experiments cover a period from 2011 to 2018 as in [56].

The dataset has been split into discrete chunks representing contiguous 3-month-long time periods. The samples are split according to the last modification date of the APK, for comparability with [41]. Each chunk is labeled as `yy.q`, where `yy` is the last two digits of the year and `q` is the quarter of the year. The dataset presents two instances of sudden drift, in the 15.4 and 16.3 chunks, linked to abruptly emerging different patterns in permission usage and API calls in the malware samples [56]. The two sudden drifts are uncorrelated, with different sets of features acquiring greater relative importance in the classification of malicious samples.

The classifier pools contain twelve classifiers, that are initially trained on the samples belonging to the first chunk. The first chunk is not used for the evaluation of the system's performance to avoid temporal experimental bias. The other chunks have been subject to a prequential evaluation [57] *testing-then-training* procedure, where the samples are first classified with the current classifiers through Algorithm 2, and then the classifiers are updated with the new samples. With this procedure, even if the testing set and the training set are not fully independent, the model is never tested on the samples it is trained on, avoiding temporal snooping [58]. Temporal snooping is a common issue in malware detection evaluation and entails the use of a testing set that contains samples from the same period as the training set. In a real-world scenario, a system would be trained on a dataset collected up to a certain point in time and then tested on a dataset collected after that point. Without an appropriate experimental design, temporal snooping would lead to an overly optimistic estimate of a system's performance [58] and would hide the effect of concept drift.

The system proposed here is compared against several baselines. The first baseline is partially adapted by the approach proposed by Gharib et al. [21] and represents one of the most common approach to coordinate separate classifiers working on static and dynamic features, i.e., the adoption of a threshold on the static features classifier's output is used to decide whether to run the dynamic features classifier or not [22]. In the following, this baseline is referred to as *threshold*. The second baseline, referred to as *random*, is a system that randomly decides whether or not to perform dynamic analysis. The third baseline, named *ideal*, performs the dynamic analysis only when the local classifier's output is incorrect; it represents a theoretical upper bound on the system's performance, corresponding to a perfect self-evaluation agent. Moreover, the performance of the local and the remote classifiers are also reported.

The set of features used by the classifiers are constant over time, as they have little impact on the effect of concept drift [59], as long as the classifiers are trained on new data periodically [49].

The experimental evaluation takes into account the following metrics:

- **Precision**: the percentage of samples classified as `malware` that are actually malware;
- **Recall**: the percentage of malware samples that are correctly classified as `malware`;
- **F1-score**: the harmonic mean of precision and recall;

- **FRP**: the percentage of benign samples that are incorrectly classified as `malware`;
- **TPR**: the percentage of malware samples that are correctly classified as `malware`;
- **Dynamic calls**: the number of APKs that are classified by the remote analysis environment;
- **Switch accuracy**: the percentage of samples for which the self-monitoring agent correctly decides whether to run the remote analysis or not.

The average results over the 3-month chunks are given in the following sections. The statistical significance of the results is assessed with the two-tailed t-test with a significance threshold of $p < 0.05$ and a replication factor of 10.

4.1 Sudden drift characterization

The dynamical update of the classifiers relies on the assumption that the concept drift is gradual and that the classifiers trained on the dataset D_{k-1} are still suitable for the dataset D_k. The updating mechanism should provide resilience to gradual drifts but cannot handle sudden drifts [60], thus requiring the presence of the self-evaluation agent. In order to verify the validity of this assumption, concept drift in the dataset was first characterized. As shown in Figure 6, the precision of the local and remote classifiers is resistant to gradual drift and remains generally stable over time, while the recall of the local classifier is more heavily affected by sudden drifts in 15.4 and 16.3. The remote classifier is more robust to sudden concept drift than the local classifier, as it is trained on dynamic features that are less affected by concept drift compared to static features. In spite of these advantages, the remote classifier is on average less accurate than the local classifier. Additionally, the extraction of the dynamic features is more computationally expensive than the static analysis. For these reasons, an effective system cannot exclusively rely on the remote classifier. Moreover, even the remote classifier can be affected by sudden drifts, albeit less severely than the static one, as shown by the recall drop in 16.3.

Figure 7 shows the Receiver Operating Characteristic (ROC) curve of the local classifier in normal conditions and under sudden drift. The ROC curve is obtained by varying the threshold on the local classifier's output to label a sample as `malware` or `benign`. The true positive rate (TPR) is plotted against the false positive rate (FPR) for each threshold obtaining the ROC curve. As the threshold decreases, the classifier is more likely to label a sample as `malware`, thus increasing the number of true positives and false positives. A perfect classifier would have a ROC curve that passes through the top left corner of the plot, thus having a TPR of 1 and a FPR of 0 for any threshold.

When the concept drift is gradual, the ROC curve is quite close to the ideal scenario, and the proposed system matches the performance of the local classifier. However, when sudden drift occurs, a much higher false positive rate is required to detect the same percentage of malware samples. Such a high false positive rate is not acceptable in a real-world scenario. Indeed, the exhibited behavior indicates that, during sudden drifts, the local classifier wrongly classifies malware samples as benign with high confidence. For this reason, threshold-based approaches are expected to perform poorly during sudden drifts. A threshold high enough to avoid false negatives would also be triggered by many false positives, leading to a high number of dynamic analyses, with

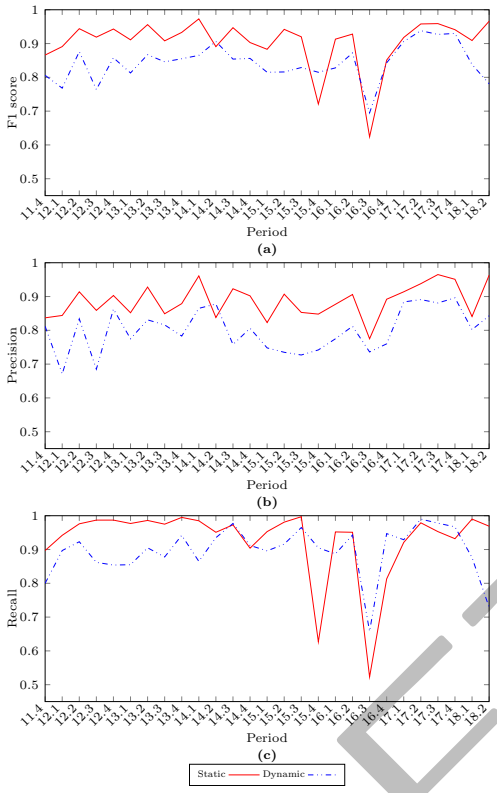

Fig. 6 Performance of the local and remote classifiers. When sudden drifts occur, the recall of the local classifier using static features is more heavily affected than the classifier using dynamic features.

a consequent unacceptable increase in the average computational cost of the detection process. On the other hand, setting a threshold low enough to avoid the majority of the false positives would also miss most of the false negatives, leading to a decrease in the overall accuracy of the system and usefulness of the cloud component of the system. On the contrary, the system proposed here exhibits similar performance in normal conditions and during sudden drifts, as also shown in Figure 7, which compares the ROC curve of a system performing only the local analysis on the static features (the red line) with the performance of the proposed system (the black dot).

4.2 Performance comparison

The performance over time of the different approaches compared here is shown in Figure 8, which reports the F1-score averaged over the 3-month chunks. In normal conditions, the proposed system performs similarly to the local classifier, as expected. Excluding the sudden drift at 16.3, the performances of the two classifiers are statistically indistinguishable (p-value> 0.35), with a small (0.4%) relative difference in favor of the proposed architecture. During the first sudden drift, at 15.4, all the approaches

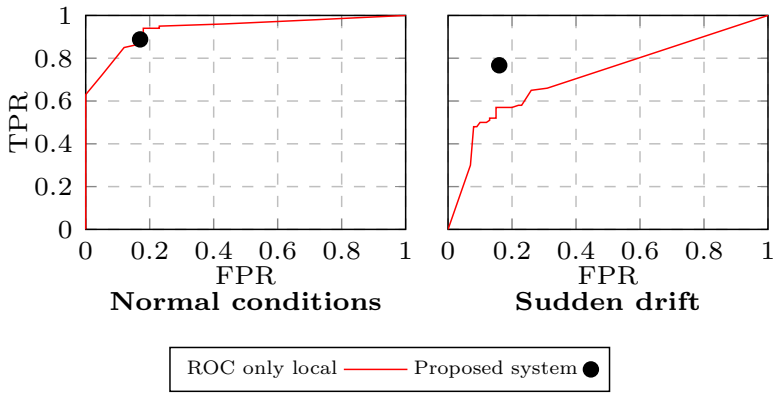

Fig. 7 ROC curve of the local classifier in normal conditions and under sudden drift. During drifts, the local classifier achieves a much lower TPR for the same FPR. The proposed system is less affected by sudden drifts, as shown by the black dot.

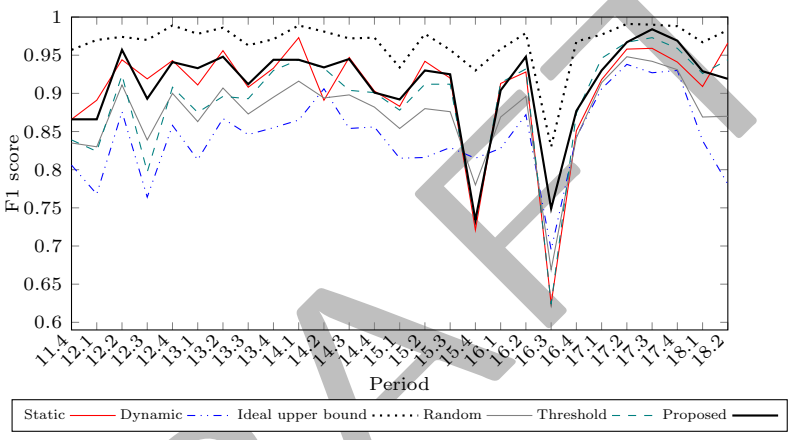

Fig. 8 F1-score of the different evaluated approaches. The proposed system performs similarly to the local classifier in normal conditions and after learning from the sudden drift at 15.4, it shows great resilience to the sudden drift at 16.3.

except the ideal upper bound and the remote classifier suffer a sharp drop in performance. In particular, the improvement of the proposed system compared to the static classifier, albeit significant ($p<0.05$), is only 2.3%. This result is not surprising, as the self-evaluation agent is not yet trained to recognize sudden drifts and thus cannot accurately trigger the remote analysis. However, the proposed system recovers quickly, and in the next sudden drift, at 16.3, the system maintains high performance, while the other approaches suffer a sharp drop in performance, with a significant relative improvement over the static classifier of 16.7% ($p<0.005$).

The presented approach, together with the ideal upper bound, is the only one able to outperform the remote classifier in this sudden drift. The ability of the system to outperform the classifier which only relies on dynamic features demonstrates that the

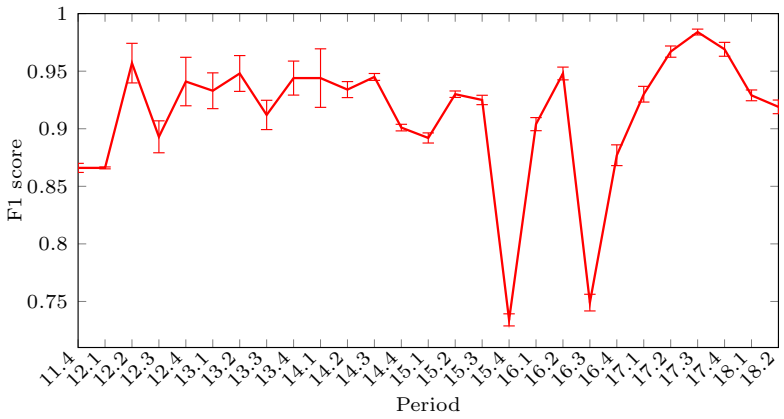

Fig. 9 Variance of the F1-score of the proposed system over the performed experiments.

remote analysis is triggered only for misclassified samples, and it is not being overused for all the input samples.

The ideal strategy, which represents the theoretical upper bound on the proposed system's performance, is the best-performing approach, with an average F1-score of 0.967. The ideal strategy can perfectly recognize when the local classifier's output is incorrect and trigger the remote analysis only for those samples, corresponding to a perfect self-evaluation agent in the proposed system. This result confirms that an intelligent mechanism to trigger the analysis of the dynamic features can be very effective in improving the performance of an automated malware detection system. At the same time, there is still a margin of improvement for developing a more effective mechanism to trigger remote analysis, leaving room for future research.

Across all the performed experiments, the proposed system exhibits stable performance, with an average variance of the F1-score of 0.008, as shown in Figure 9.

4.3 Recurring drifts

The dataset used for experimental evaluation contains only two instances of sudden drift, and there are no comparable publicly available datasets that span a longer period of time and contain more instances of sudden drift. Therefore, to further evaluate the effectiveness of the self-evaluation agent in recognizing sudden drifts and appropriately triggering the remote analysis, additional artificial recurring drifts were introduced in the dataset through the following procedure partially inspired by [61]: a slice of the dataset is replicated to re-introduce samples from a previous distribution, simulating a sudden drift.

Clearly, the sudden drifts introduced in this way are not as realistic as the ones present in the dataset, and the testing procedure could be affected by spatial and temporal snooping. The classifiers in the pool should not have been trained on malware families that are not available before the sudden drift. To limit the contamination of the tested samples and the ones used for training, the slice of the dataset used to

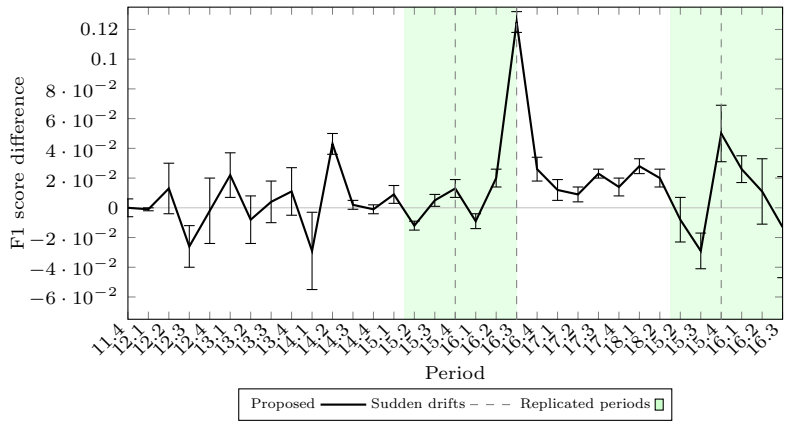

Fig. 10 F1-score difference between the proposed system and the static classifier in the presence of additional artificial drifts. The self-evaluation agent, once trained, is able to recognize further drifts.

introduce the artificial drift starts from a chunk that is old enough for the classifier pool to not contain any classifiers trained on that data.

To avoid reporting results inflated by this unavoidable bias, instead of the F1 score, the difference in F1 score between the proposed system and the static classifier is reported in Figure 10. This metric is less affected by the bias introduced by the artificial drifts and better reflects the true effect of the self-evaluation agent in recognizing sudden drifts.

Compared to the first time the system encounters the drift at 15.4, the system is able to recognize the drift more effectively the second time, with an improvement in the F1-score ($p < 0.0002$) 2.86 times higher than the first time. Further artificial drifts do not show a significant performance improvement as the static classifier pool now contains classifiers trained on the distribution under test and is not affected by drifts anymore.

4.4 Reduction of the dynamic analyses

This section assesses the effectiveness of the self-evaluation agent in limiting the number of dynamic analyses performed. In these experiments the more conservative approach is also evaluated, where the dynamic analysis is triggered only when the local classification is likely to be a false negative. Figure 11 shows the effectiveness of the different approaches in terms of F1-score in part (a) and the number of dynamic analyses performed by the different approaches in part (b).

The difference in F1-score between the normal self-evaluation agent and the conservative one is negligible, and they both outperform the threshold baseline, especially so in the second sudden drift. Compared to the ideal upper bound baseline, the number of dynamic analyses performed by the conservative version of the proposed system is statistically indistinguishable (p-value> 0.2) from the optimal policy, with only 8.51% of the samples requiring the dynamic analysis performed by the cloud server. The difference in the average F1-score between the conservative version of the proposed

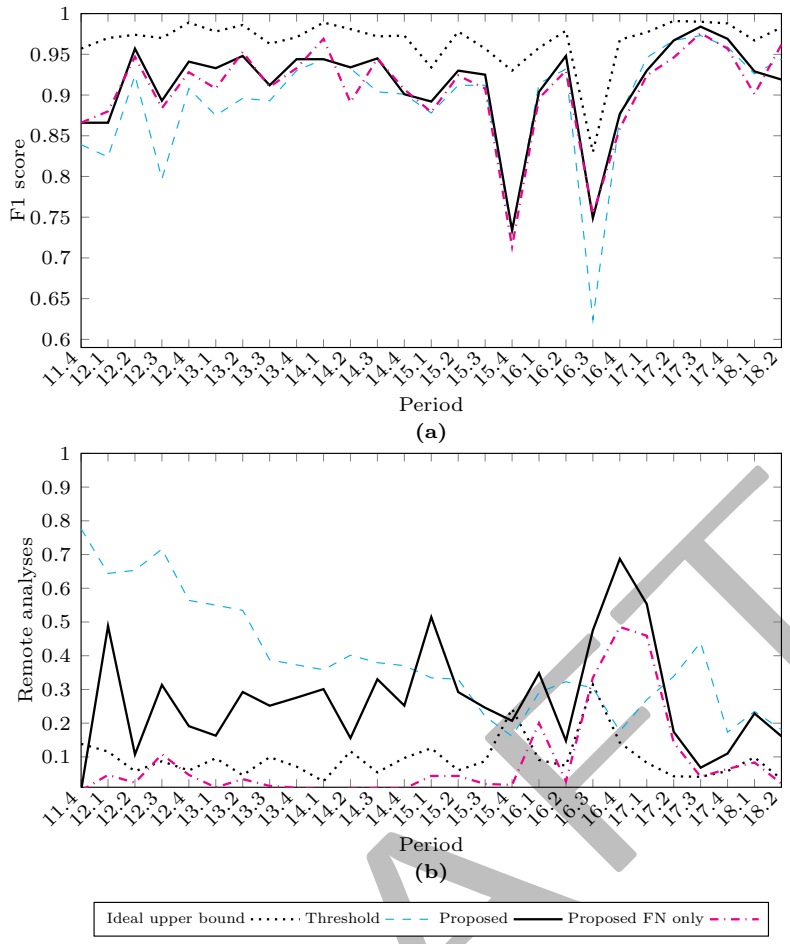

Fig. 11 F1-score and number of dynamic analyses performed by different approaches. Limiting the remote analysis to suspected false negatives with the FN only strategy significantly reduces the number of remote analyses with negligible impact on the performance.

system and the ideal upper bound is also quite small, only 6.1%, confirming the effectiveness of the proposed approach. The normal self-evaluation agent also displays an effectiveness similar to that of the ideal upper bound (5.6% lower F1-score), albeit with almost three times as many dynamic analyses required. Even considering this wider margin from the ideal upper bound, the system is still preferable to the threshold baseline, which requires 30% more dynamic analyses than the proposed system while having a lower F1-score.

Finally, the effectiveness of the self-evaluation agent in detecting errors of the local classifier is also evaluated. Figure 12 shows the accuracy of the different approaches in correctly switching to the remote classifier when the local classifier is wrong.

The self-evaluation agent is able to detect errors of the local classifier with high accuracy, up to 91.6%, and an average accuracy of 75.3% (standard deviation 7.7%).

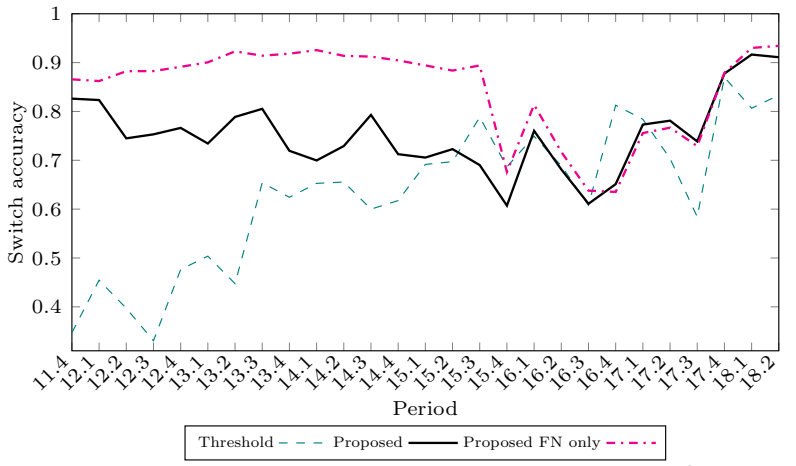

Fig. 12 Accuracy of the different techniques to detect local classifier's misclassifications and switch to the remote classifier. The self-evaluation agent effectively detects errors of the local classifier while the threshold approach is barely better than a random approach.

The revised version of the self-evaluation agent, which only switches to the remote classifier if the suspected error is a false negative, gives an additional 19% relative increase in accuracy compared to the normal self-evaluation agent. Additionally, there is a 33% accuracy increase with respect to the threshold baseline ($p < 0.05$). On average, in the test scenario, the threshold detects mistakes of the local classifier with an accuracy of 63% (14% standard deviation). The threshold approach, thus, is better than the random one but is not reliable enough to be used as a self-evaluation agent, especially considering that its accuracy can get as low as 34.7% and triggers an excessive number of dynamic analyses.

On the basis of these results, the conservative version of the proposed system appears to be suitable for most scenarios, as it is able to maintain high performance while significantly reducing the number of dynamic analyses performed, reducing the computational cost of the detection process. However, in critical scenarios where the minor reduction in the accuracy of the system caused by the additional false positives is not acceptable, and the cloud server has sufficient computational resources to perform the necessary analyses, the default version of the proposed system can be used instead.

4.5 Overall Performance Comparison

Finally, the overall performance of the different approaches is evaluated. Table 1 shows the Accuracy, Precision, Recall, F1-score, fraction of samples requiring remote analysis, and average classification latency of the different approaches, averaged over the test period, and during the sudden drift at 16.3. The second sudden drift at 16.3 is the one considered for this comparison as in this period the self-evaluation agent has had the chance to learn to recognize sudden drifts, whereas in the first sudden drift at 15.4 it is still untrained and unable to react properly.

The proposed system is able to maintain a high performance over time, with an average accuracy of 0.913. Employing the two-tier architecture is thus beneficial

Table 1 Average performance of the different approaches during the test period and in the sudden drift at 16.3. The underlined values indicate the best performance in each column (ideal upper bound excluded), while the bold values indicate the second-best performance. For the fraction of analyses requiring remote analysis, the "Static" and "Dynamic" rows are not considered when highlighting the best and second best values.

Approach	Average					
	Accuracy	**Precision**	**Recall**	**F1**	**Remote analyses**	**Latency**
Static	**0.905**	**0.887**	**0.929**	0.904	0.000	2.00
Dynamic	0.833	0.800	0.896	0.843	1.000	10.33
Random	0.896	0.841	0.912	0.873	0.499	7.15
Threshold	0.892	0.874	0.916	0.891	0.388	5.99
Proposed	0.913	0.906	0.924	0.913	**0.271**	5.58
Proposed (FN only)	0.901	0.866	0.954	**0.906**	0.085	**2.87**
Ideal upper bound	0.966	0.953	0.981	0.967	0.095	2.97

Approach	Sudden drift (16.3)					
	Accuracy	**Precision**	**Recall**	**F1**	**Remote analyses**	**Latency**
Static	0.685	0.775	0.523	0.624	0.000	**2.00**
Dynamic	0.711	0.736	0.658	0.695	1.000	10.33
Random	0.704	0.758	0.598	0.669	0.505	7.22
Threshold	0.691	0.796	0.513	0.624	0.305	5.15
Proposed	0.767	0.813	**0.694**	**0.749**	0.476	6.92
Proposed (FN only)	**0.760**	**0.776**	0.733	0.754	**0.335**	5.46
Ideal upper bound	0.842	0.900	0.770	0.830	0.314	5.24

as it improves the performance of the system even in normal conditions. Even the conservative version of the proposed system, which only triggers the remote analysis 8.5% of the time, has a slightly higher average F1 score than the static classifier. This version of the proposed system, being more lax with possible false positives, manages to correctly classify 95.4% of all the malware samples.

During the sudden drift at 16.3, the proposed system (in both versions) is the least affected, increasing the fraction of remote analyses by 0.203 and 0.250 respectively, to offer a more robust classification. Even the ideal upper bound, which is the theoretical upper bound for this system, needs to increase the fraction of remote analyses by 0.209 to maintain a high performance. Thus, the required increase in remote analyses is comparable to the optimal policy. On the other hand, the threshold-based approach actually reduces the fraction of remote analyses by 0.083, demonstrating that it is not able to effectively detect misclassifications of the local classifier. The additional communication overhead introduced by the remote analysis is proportional to the fraction of samples requiring remote analysis.

To quantify the improvement in performance of the proposed approach in terms of latency for the end-user, Table 1 also reports the average analysis time for the different approaches. In this computation, the average time for the on-device extraction of the static features is considered to be 2 minutes, while the upload and dynamic analysis is, on average, 10.3 minutes. These values are based on results in the literature [62]. The proposed system has an average latency of 5.58 minutes, which is a 179% increase compared to the static classifier, bust still 45.9% lower than always relying on the

dynamic analysis and 24.6 seconds faster than the less effective threshold approach. During the sudden drift at 16.3, the average latency is 6.92 minutes, which is a moderate 24.0% increase compared to normal conditions. In normal conditions, the conservative version of the proposed system has an average latency of 2.87 minutes, which is just a 42.5% increase compared to the static classifier. During drifts, when more remote analyses are required, the average latency is still 47.1% lower than always relying on the dynamic analysis, ad just 4.2% increase over the ideal performance.

As a final consideration, it is worth noting that the ideal upper bound's remarkable performance, even during the conceptual drift, confirm that the design choice of relying primarily on static feature-based analysis, requiring a more in-depth analysis, through dynamic features, only when the former provides an incorrect answer, is a very promising approach. The room for improvement with respect to the ideal upper bound's performance suggests that there is still an open research question on the autonomous ability of such systems to recognize their own errors.

4.6 Comparison with State-of-the-Art Methods

In order to assess the competitiveness of the proposed approach from a classification performance standpoint, a comparison with state-of-the-art methods is presented in Figure 13/Table 2. It is worth noting that none of these methods aim to reduce the cost of the detection process, as they all rely on hybrid feature sets, including both static and dynamic features. Nevertheless, despite being more constrained in the choice of features, the presented approach maintains competitive, and in some cases superior, performance compared to the state-of-the-art methods at a fraction of the cost.

The works presented by Guerra-Manzanares et al. in [41] is the most similar to ours, as it also leverages ensemble learning with periodic replacement of outdated classifiers. A key difference is that Guerra-Manzanares et al. train their classifiers on the entire feature set without an intelligent agent to avoid unnecessary computations, requiring the extraction of both static and dynamic features for each sample. Comparing their performance to the one of the presented system it is clear that there is no obvious advantage in always using the entire feature set, as the proposed system performs similarly to the state-of-the-art method. Additionally, after the self-evaluation agent is trained to detect sudden drifts, the proposed system is able to consistently outperform their approach.

AlSobeh et al. [63] is a more recent work that presents a novel approach to select the most appropriate features considering the concept drift issue. Their choice of features, however, is fixed for all the samples, and they do not consider the possibility of dynamically utilizing different feature sets for different samples. While more effective than the approach of Guerra-Manzanares et al. [41], their approach it is still on average outperformed by the proposed system which has greater flexibility in the choice of features on a case-by-case basis.

Aurangzeb et al. [54] also use ensemble voting, analyzing the impact of different feature sets on the classification performance, especially against adversarial obfuscation techniques. Nevertheless, their ensemble approach still requires the availability of all the dynamic features to classify the samples. The average F1-score achieved by their approach slightly outperforms the proposed system, however it is worth noting that,

Table 2 F1-score comparison with state-of-the-art methods on the KronoDroid dataset.

Method	F1-score
Guerra-Manzanares et al. [41]	0.873
AlSobeh et al. [63]	0.897
Aurangzeb et al. [54]	0.926
Neural Network [64] + finetuning	0.871
Proposed	0.913
Ideal upper bound	0.967

since they require the extraction of both static and dynamic features for each sample, their approach might not be feasible in a real-world scenario. Moreover, the ideal upper bound of the presented system still outperforms their strategy.

Finally, since all the approaches tested so far rely on tree-based classifiers, we also evaluate a neural network-based approach. The network architecture is inspired by the one in [64] which is trained using both statically and dynamically extracted features. Since the original work does not consider the concept drift issue, instead of directly reporting their results, we adapt the approach by fine-tuning the neural network on new samples with the same frequency as that used to update the classifier pool in the proposed system. This approach has the worst performance among the compared methods, showing that the presence of an ensemble mechanism offers a significant advantage when tackling concept drift in malware detection.

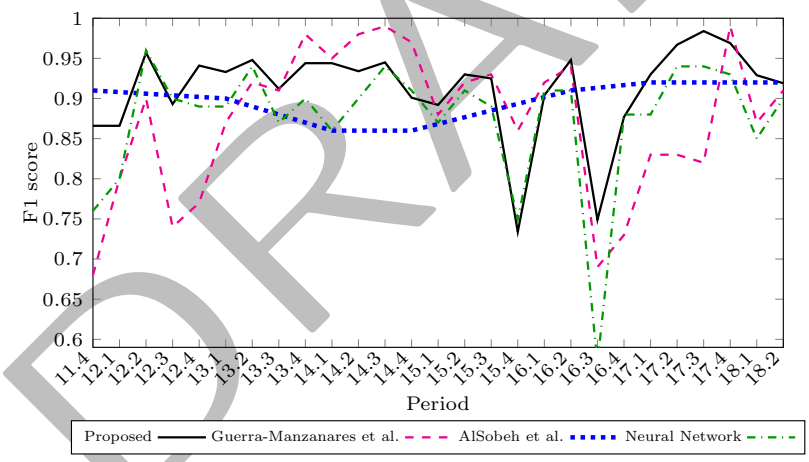

Fig. 13 Comparison of the presented approach with state-of-the art methods on the KronoDroid dataset.

5 Conclusion

Malware detection is a crucial component of mobile security. However, the limited computational and energy resources of mobile devices make it unfeasible to perform a thorough and exhaustive analysis of the apps on the mobile devices. On the other hand, the use of remote analysis on cloud servers, characterized by more computational resources, introduces additional latency and communication overhead in the detection process which might be unacceptable for the end-users.

A possible solution could be using machine learning methods on the mobile devices, relying exclusively on features whose extraction is not computationally expensive, such as the static features that can be extracted directly from the metadata contained in the APKs. However, this approach is significantly affected by the concept drift problem. Malware developers frequently modify their code to avoid detection, and the constant influx of new malware samples leads to performance degradation in machine learning models over time. Furthermore, sudden drifts can occur when malware developers rapidly release numerous new samples, causing models trained on static features to deteriorate quickly. Effectively managing concept drift in malware detection is crucial for maintaining the effectiveness of detection systems in this field.

To address this issue, this paper introduces a two-level malware detection approach for mobile devices that integrates local and remote analysis. An agent learns to recognize sudden concept drift in the local analysis level in order to trigger, on a case-by-case basis, the execution of the more robust second level which relies on drift-resilient dynamic features. Both the local and remote analysis levels are supported by ensembles of classifiers, which are trained and updated dynamically to adapt to the changing nature of the malware samples. Moreover, the computational cost of the detection process was reduced by limiting the execution of the second level only to suspected false negatives. The experimental results show that the self-evaluation agent can detect errors of the local classifier with high accuracy and can effectively mitigate the impact of concept drift on the overall performance of the system. The presented approach is able to maintain stable performance over time, even in the presence of sudden drifts, while executing most of the analyses locally, thus reducing the computational cost and the communication overhead of the detection process.

Statements and Declarations

The authors have no conflicts of interest to declare that are relevant to the content of this article.

Author contributions

Alessandra De Paola conceived the presented architecture. Andrea Augello carried out the data analysis and performed the fine-tuning of the AI-based modules. Andrea Augello and Alessandra De Paola conceived and planned the experiments. Giuseppe Lo Re supervised the project and took lead in writing the manuscript. All authors reviewed the manuscript.

Funding

This work was partially funded by the European Union Next-Generation EU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.3) – Progetto "Future Artificial Intelligence - FAIR" PE00000013, CUP J73C24000060007

Data Availability Statement

The data used in the experimental evaluation described in Section 4 are publicly available in the following github repository, made available by Guerra-Manzanares et al., [17]: `https://github.com/aleguma/kronodroid`.

References

[1] Mayrhofer R, Stoep JV, Brubaker C, Kralevich N. The Android Platform Security Model. ACM Trans Priv Secur. 2021 apr;24(3).

[2] Ansari AM, Nazir M, Mustafa K. Smart Homes App Vulnerabilities, Threats, and Solutions: A Systematic Literature Review. Journal of Network and Systems Management. 2024;32(2):1-62.

[3] Qiu J, Zhang J, Luo W, Pan L, Nepal S, Xiang Y. A survey of android malware detection with deep neural models. ACM Computing Surveys (CSUR). 2020;53(6):1-36.

[4] De Paola A, Favaloro S, Gaglio S, Lo Re G, Morana M. Malware detection through low-level features and stacked denoising autoencoders. In: ITASEC 2018. Italian Conf. on Cyber Security. CEUR-WS. Milan, Italy: CEUR-WS.org; 2018. .

[5] Li HW, Wu YS, Huang Y. On the Feasibility of Anomaly Detection with Fine-Grained Program Tracing Events. Journal of Network and Systems Management. 2022;30(2):28.

[6] Khalid S, Hussain FB. Evaluating Dynamic Analysis Features for Android Malware Categorization. In: 2022 Int. Wireless Communications and Mobile Computing (IWCMC). Dubrovnik, Croatia: IEEE; 2022. p. 401-6.

[7] Or-Meir O, Nissim N, Elovici Y, Rokach L. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. ACM Comput Surv. 2019 sep;52(5).

[8] Lachtar N, Ibdah D, Khan H, Bacha A. RansomShield: A Visualization Approach to Defending Mobile Systems Against Ransomware. ACM Trans Priv Secur. 2023 mar;26(3).

[9] De Paola A, Gaglio S, Re GL, Morana M. A hybrid system for malware detection on big data. In: IEEE INFOCOM 2018 - IEEE Conf. on Computer Communications Workshops (INFOCOM WKSHPS). Honolulu, HI, USA: IEEE; 2018. p. 45-50.

[10] Mahdavifar S, Alhadidi D, Ghorbani AA. Effective and efficient hybrid android malware classification using pseudo-label stacked auto-encoder. Journal of network and systems management. 2022;30(1):22.

[11] Kambar MEZN, Esmaeilzadeh A, Kim Y, Taghva K. A survey on mobile malware detection methods using machine learning. In: 2022 IEEE 12th Annual Computing

and Communication Workshop and Conf. (CCWC). IEEE. Las Vegas, NV, USA: IEEE; 2022. p. 0215-21.
[12] Khamassi I, Sayed-Mouchaweh M, Hammami M, Ghédira K. Self-adaptive windowing approach for handling complex concept drift. Cognitive Computation. 2015;7:772-90.
[13] Yang L, Guo W, Hao Q, Ciptadi A, Ahmadzadeh A, Xing X, et al. CADE: Detecting and Explaining Concept Drift Samples for Security Applications. In: 30th USENIX Security Symp. (USENIX Security 21). Vancouver, BC, Canada: USENIX Association; 2021. p. 2327-44.
[14] Chen Y, Ding Z, Wagner D. Continuous Learning for Android Malware Detection. In: 32nd USENIX Security Symp. (USENIX Security 23). Anaheim, CA: USENIX Association; 2023. p. 1127-44.
[15] Sherawat A, Nath SB, Addya SK. Optimizing Completion Time of Requests in Serverless Computing. Journal of Network and Systems Management. 2024;32(2):28.
[16] Stats SG. Mobile Operating System Market Share Worldwide; 2023.
[17] Guerra-Manzanares A, Bahsi H, omm SN. KronoDroid: Time-based Hybrid-featured Dataset for Effective Android Malware Detection and Characterization. Computers & Security. 2021;110:102399.
[18] Razeghi Borojerdi H, Abadi M. MalHunter: Automatic generation of multiple behavioral signatures for polymorphic malware detection. In: ICCKE 2013. Mashhad, Iran: IEEE; 2013. p. 430-6.
[19] Tsimenidis S, Lagkas T, Rantos K. Deep learning in IoT intrusion detection. Journal of network and systems management. 2022;30(1):8.
[20] Botacin M, Moreira FB, Navaux POA, Grégio A, Alves MAZ. Terminator: A Secure Coprocessor to Accelerate Real-Time AntiViruses Using Inspection Breakpoints. ACM Trans Priv Secur. 2022 mar;25(2).
[21] Gharib A, Ghorbani A. DNA-Droid: A Real-Time Android Ransomware Detection Framework. In: Yan Z, Molva R, Mazurczyk W, Kantola R, editors. Network and System Security. Cham: Springer Int. Publishing; 2017. p. 184–198.
[22] Ding C, Luktarhan N, Lu B, Zhang W. A Hybrid Analysis-Based Approach to Android Malware Family Classification. Entropy. 2021 Aug;23(8):1009.
[23] Molina-Coronado B, Mori U, Mendiburu A, Miguel-Alonso J. Efficient concept drift handling for batch android malware detection models. Pervasive and Mobile Computing. 2023;96:101849.
[24] Alkhateeb E, Ghorbani A, Habibi Lashkari A. A survey on run-time packers and mitigation techniques. Int J of Information Security. 2023:1-27.
[25] Sugunan K, Gireesh Kumar T, Dhanya K. Static and dynamic analysis for android malware detection. In: Advances in Big Data and Cloud Computing. Springer. Singapore: Springer; 2018. p. 147-55.
[26] Zyout M, Shatnawi R, Najadat H. Malware classification approaches utilizing binary and text encoding of permissions. Int J of Information Security. 2023:1-26.
[27] Liu P, Wang W, Luo X, Wang H, Liu C. NSDroid: efficient multi-classification of android malware using neighborhood signature in local function call graphs. Int J of Information Security. 2021;20:59-71.

[28] Alzaidi A, Alshehri S, Buhari SM. DroidRista: a highly precise static data flow analysis framework for android applications. Int J of Information Security. 2020;19(5):523-36.
[29] Balikcioglu PG, Sirlanci M, A Kucuk O, Ulukapi B, Turkmen RK, Acarturk C. Malicious code detection in android: the role of sequence characteristics and disassembling methods. Int J of Information Security. 2023;22(1):107-18.
[30] Bernardi ML, Cimitile M, Distante D, Martinelli F, Mercaldo F. Dynamic malware detection and phylogeny analysis using process mining. Int J of Information Security. 2019;18:257-84.
[31] Hamzenejadi S, Ghazvini M, Hosseini S. Mobile botnet detection: a comprehensive survey. Int J of Information Security. 2023;22(1):137-75.
[32] Wang R, Gao J, Huang S. AIHGAT: A novel method of malware detection and homology analysis using assembly instruction heterogeneous graph. Int J of Information Security. 2023:1-21.
[33] Sen S. Using instance-weighted naive Bayes for adapting concept drift in masquerade detection. Int J of Information Security. 2014;13:583-90.
[34] Ami AS, Kafle K, Moran K, Nadkarni A, Poshyvanyk D. Systematic Mutation-Based Evaluation of the Soundness of Security-Focused Android Static Analysis Techniques. ACM Trans Priv Secur. 2021 feb;24(3).
[35] Concone F, De Paola A, Lo Re G, Morana M. Twitter analysis for real-time malware discovery. In: 2017 AEIT Int. Annual Conf.; 2017. p. 1-6.
[36] Jiang Y, Li G, Li S, Guo Y. BenchMFC: A benchmark dataset for trustworthy malware family classification under concept drift. Computers & Security. 2024;139:103706.
[37] Barbero F, Pendlebury F, Pierazzi F, Cavallaro L. Transcending transcend: Revisiting malware classification in the presence of concept drift. In: 2022 IEEE Symp. on Security and Privacy (SP). IEEE. San Francisco, CA: IEEE; 2022. p. 805-23.
[38] Onwuzurike L, Mariconti E, Andriotis P, Cristofaro ED, Ross G, Stringhini G. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). ACM Trans Priv Secur. 2019 apr;22(2).
[39] Xu K, Li Y, Deng R, Chen K, Xu J. DroidEvolver: Self-Evolving Android Malware Detection System. In: 2019 IEEE European Symp. on Security and Privacy (EuroS&P). Stockholm, Sweden: IEEE; 2019. p. 47-62.
[40] Darem AA, Ghaleb FA, Al-Hashmi AA, Abawajy JH, Alanazi SM, Al-Rezami AY. An Adaptive Behavioral-Based Incremental Batch Learning Malware Variants Detection Model Using Concept Drift Detection and Sequential Deep Learning. IEEE Access. 2021;9:97180-96.
[41] Guerra-Manzanares A, Luckner M, Bahsi H. Android malware concept drift using system calls: Detection, characterization and challenges. Expert Systems with Applications. 2022 Nov;206:117200.
[42] Ceschin F, Botacin M, Gomes HM, Pinagé F, Oliveira LS, Grégio A. Fast & Furious: On the modelling of malware detection as an evolving data stream. Expert Systems with Applications. 2023;212:118590.
[43] Alecakir H, Can B, Sen S. Attention: there is an inconsistency between android

permissions and application metadata! Int J of Information Security. 2021:1-19.
[44] Fernando DW, Komninos N. FeSAD ransomware detection framework with machine learning using adaption to concept drift. Computers & Security. 2024;137:103629.
[45] Elsayed MA, Zincir-Heywood N. BoostSec: Adaptive Attack Detection for Vehicular Networks. Journal of Network and Systems Management. 2024;32(1):6.
[46] Ko AH, Sabourin R, Britto Jr AS. From dynamic classifier selection to dynamic ensemble selection. Pattern recognition. 2008;41(5):1718-31.
[47] Zyblewski P, Sabourin R, Woźniak M. Preprocessed dynamic classifier ensemble selection for highly imbalanced drifted data streams. Information Fusion. 2021;66:138-54.
[48] Agate V, Drago S, Ferraro P, Lo Re G. Anomaly Detection for Reoccurring Concept Drift in Smart Environments. In: 2022 18th Int. Conf. on Mobility, Sensing and Networking (MSN); 2022. p. 113-20.
[49] Daoudi N, Allix K, Bissyandé TF, Klein J. A Deep Dive Inside DREBIN: An Explorative Analysis beyond Android Malware Detection Scores. ACM Trans Priv Secur. 2022 may;25(2).
[50] Liu FT, Ting KM, Zhou ZH. Isolation Forest. In: 2008 Eighth IEEE Int. Conf. on Data Mining. Pisa, Italy: IEEE; 2008. p. 413-22.
[51] Liu S, Feng P, Wang S, Sun K, Cao J. Enhancing malware analysis sandboxes with emulated user behavior. Computers & Security. 2022;115:102613.
[52] Cui Y, Sun Y, Lin Z. DroidHook: a novel API-hook based Android malware dynamic analysis sandbox. Automated Software Engineering. 2023;30(1):10.
[53] Patil R, Dudeja H, Modi C. Designing in-VM-assisted lightweight agent-based malware detection framework for securing virtual machines in cloud computing. Int J of Information Security. 2020;19(2):147-62.
[54] Aurangzeb S, Aleem M. Evaluation and classification of obfuscated Android malware through deep learning using ensemble voting mechanism. Scientific Reports. 2023;13(1):3093.
[55] Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K, Siemens C. Drebin: Effective and explainable detection of android malware in your pocket. In: Ndss. vol. 14; 2014. p. 23-6.
[56] Guerra-Manzanares A, Bahsi H, Luckner M. Leveraging the first line of defense: A study on the evolution and usage of android security permissions for enhanced android malware detection. J of Computer Virology and Hacking Techniques. 2023;19(1):65-96.
[57] Gama J, Sebastiao R, Rodrigues PP. Issues in evaluation of stream learning algorithms. In: Proc. of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining; 2009. p. 329-38.
[58] Arp D, Quiring E, Pendlebury F, Warnecke A, Pierazzi F, Wressnegger C, et al. Dos and don'ts of machine learning in computer security. In: 31st USENIX Security Symp. (USENIX Security 22); 2022. p. 3971-88.
[59] Chen Z, Zhang Z, Kan Z, Yang L, Cortellazzi J, Pendlebury F, et al. Is It Overkill? Analyzing Feature-Space Concept Drift in Malware Detectors. In: Proc. of The 6th Deep Learning Security and Privacy Workshop (DLSP), in conjunction with IEEE

Symp. on Security and Privacy (IEEE SP). San Francisco, CA: IEEE; 2023. .
[60] Street WN, Kim Y. A streaming ensemble algorithm (SEA) for large-scale classification. In: Proc. of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining; 2001. p. 377-82.
[61] Minku LL, White AP, Yao X. The impact of diversity on online ensemble learning in the presence of concept drift. IEEE Trans on knowledge and Data Engineering. 2009;22(5):730-42.
[62] Faruki P, Bhandari S, Laxmi V, Gaur M, Conti M. Droidanalyst: Synergic app framework for static and dynamic app analysis. Recent Advances in Computational Intelligence in Defense and Security. 2016:519-52.
[63] AlSobeh AM, Gaber K, Hammad MM, Nuser M, Shatnawi A. Android malware detection using time-aware machine learning approach. Cluster Computing. 2024:1-22.
[64] Singh J, Singh J. Malware classification using multi-layer perceptron model. In: International Conference on Innovative Computing and Communications: Proceedings of ICICC 2020, Volume 2. Springer; 2021. p. 155-68.