Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 11, No. 04, December 2025.

# TAB-DROID: A FRAMEWORK FOR ANDROID MALWARE DETECTION USING THE TABPFN CLASSIFIER

Ahmed M. Saeed[1], Sameh A. Salem[1,2], Shahira M. Habashy[1] and Hadeer A. Hassan[1]

*(Received: 7-Jul.-2025, Revised: 13-Sep.-2025, Accepted: 17-Sep.-2025)*

## ABSTRACT

*The Android operating system is considered as a leading global mobile OS, with its open-source nature driving widespread use across critical daily activities like banking, communication, entertainment, education and healthcare. Therefore, Android is a primary target and attractive ground for cyber-threats. In this paper, a novel malware-detection framework, which is called TAB-DROID, is introduced. The proposed framework leverages advanced feature selection, compression, and classification techniques applied to real-world datasets. Firstly, the Conditional Mutual Information Maximization (CMIM) and Joint Mutual Information (JMI) algorithms are used concurrently for feature selection. Each algorithm independently selects relevant features from the datasets. Moreover, product quantization (PQ) for feature compression is applied separately to the outputs of both CMIM and JMI to enhance storage and accelerate subsequent processing without compromising critical information. Subsequently, the Tabular Prior data Fitted Network (TabPFN) classifier is integrated into pipelines to perform the classification task. By applying 5-fold cross-validation, the results demonstrate that the optimized pipeline using CMIM achieved superior detection performance compared to the pipeline using JMI. According to CMIM-based pipeline configuration, the accuracy, AUC, precision, recall, and F1-score metrics reach 99.2%, 99.9%, 99.6%, 98.7%, and 99.2%, respectively. In addition, integrating PQ with CMIM reduced testing time by 44.4% and memory usage by 42.8%, highlighting the framework's efficiency alongside its high detection accuracy. Furthermore, the results are compared to other competing techniques, showing that the proposed framework achieved significantly enhanced performance, where the TAB-DROID has improved the accuracy up to 1.52% and precision up to 2.69%, while also reducing the feature space by 73%.*

## KEYWORDS

*Android malware, Malware detection, Machine learning, Conditional mutual information maximization, Product quantization, TabPFN classifier.*

## 1. INTRODUCTION

Mobile devices have become integral to modern life, with global smartphone subscriptions approaching 7 billion in 2023 and projected to exceed 7.7 billion by 2028 [1]. As for the 1st quarter (Q1) 2025, Android operating system has maintained a dominant market share of approximately 71.88% [2], with Google Play hosting over 2.26 million applications [3] and surpassing 102 billion downloads in 2024 [4]. Despite their benefits, mobile devices and third-party applications introduce significant security and privacy risks. The pervasive uses of Android applications across sectors such as banking, commerce, and education amplify the exposure of sensitive data to potential threats. Android's open-source nature, while promoting flexibility, also permits relatively unrestricted third-party application installation, increasing the platform's susceptibility to malware and cyber-attacks.

As the most widely used mobile platform, Android has become a primary target for malware due to its open architecture and extensive user base [5]. In Q2 2024, Kaspersky has reported 367,418 malicious installation packages targeting Android devices, underscoring the persistent and large-scale nature of these threats [6]. Many malware variants employ advanced encryption and obfuscation techniques to evade detection, complicating identification and mitigation efforts [7]-[8].

The growing expansion of Android malware has intensified the need for robust detection and defence frameworks. Research highlights the raised risk posed by applications distributed outside the official

1. A. M. Saeed, S. A. Salem, S. M. Habashy and H. A. Hassan are with Department of Computer and Systems Engineering, Faculty of Engineering, Helwan University, Cairo, Egypt. Emails: {ahmed_mohamed_saeed, Sameh_salem, Shahira_heikal, Hadeer_ahmed}@h-eng.helwan.edu.eg
2. S. A. Salem is with Egyptian Computer Emergency Readiness Team (EG-CERT), National Telecom Regulatory Authority (NTRA), Cairo 12577, Egypt.

Google Play Store [9]-[10], with approximately 13% of Android application installations originating from third-party sources that often lack rigorous security vetting. These alternative channels facilitate the spread of malware, often disguised as legitimate applications. Many such applications delay malicious behavior through dynamic code loading or remote payload retrieval, further complicating detection. This threat is made worse by limited user awareness: only 35% of users review application permissions before installation, and just 23% have declined applications due to excessive permissions [11]-[12]. These trends emphasize the urgent need for automated, scalable, and effective detection frameworks to counter increasingly sophisticated Android malware.

Traditional signature-based methods have proven inappropriate against the rapidly evolving nature of Android malware, prompting increased interest in machine learning (ML) and deep learning (DL) techniques for dynamic and intelligent detection. These approaches enable automated analysis of application behaviors to uncover malicious patterns beyond static signatures. However, advanced ML models, particularly neural networks, often require substantial computational resources, limiting their real-time applicability on mobile devices [13]-[15]. Furthermore, their adaptability to emerging threats remains a challenge. To address these limitations, this study proposes TAB-DROID, a lightweight and resource-efficient classification framework designed to deliver robust, timely malware detection with minimal computational overhead. Its efficiency has formed a 73% reduction in feature space and significant reductions in testing time by 44.4% and memory usage by 42.8%. The primary contributions of this research are as follows:

- A novel resource-efficient classifier framework for Android malware detection is proposed, with efficiency validated through serious reductions in feature dimensionality, testing time, and memory usage, which addresses the computational challenges with novel existing ML-based approaches.
- The framework employs two parallel pipelines featuring effective feature-selection and compression stages. The first pipeline uses CMIM, while the second uses JMI for feature selection. Both pipelines are followed by a PQ step to reduce the feature space while retaining discriminative information.
- The two pipelines are subsequently completed using the TabPFN classifier, which is a pre-trained single transformer-based classifier, enabling fast few-shot inference and strong generalization without hyper-parameter tuning, making it ideal for real-time use on resource-limited devices.
- The framework is evaluated on two datasets: TUANDROMD (241 features), which despite its small size, includes diverse and modern Android threats, and Malgenome (215 features).
- Comprehensive experimentation is applied, and 5-fold cross-validation is used to demonstrate that the pipeline using the CMIM feature-selection technique achieved optimal performance with a reduced feature set attribute, compared to the proposed framework using JMI, therefore significantly enhancing detection accuracy and AUC. Moreover, comparative evaluations against other techniques collectively validate the efficacy of the proposed TAB-DROID framework.

The rest of this paper is structured as follows: Section 2 presents the related works, reviewing recent advancements in Android malware detection. Section 3 details the proposed TAB-DROID framework, including its main components: CMIM and JMI feature selection, PQ, and the TabPFN classifier. Section 4 describes the experimental setup and the evaluation metrics and presents the results. Finally, Section 5 concludes the paper and discusses its limitations and potential directions for future work.

## 2. RELATED WORKS

In this section, the use of ML techniques for Android malware detection is examined, which focuses on static, dynamic, and hybrid analysis approaches [16]-[17]-[18]-[19]. Each method exhibits inherent limitations: static analysis involves the inspection of the application's code without the need for execution, which is computationally efficient. Although, it is more vulnerable to evasion techniques, such as code obfuscation and binary packing. On the other hand, dynamic analysis observes the application's behaviors as it executes, which is more accurate due to its ability to monitor runtime activity; however, it is a more time-consuming and resource-intensive usage.

Finally, the hybrid analysis leverages the strengths of both static and dynamic techniques, aiming to provide a broader and more accurate understanding of malware characteristics. A comprehensive understanding of these techniques is vital for improving the performance and reliability of ML-driven malware-detection systems. In addition to these approaches, recent studies have explored federated and privacy-preserving learning frameworks [20], which enable collaborative malware detection without directly exposing the sensitive information of the users, thereby addressing growing privacy concerns in Android threat analysis.

## 2.1 Static-analysis Techniques for Detecting Android Malware

Numerous studies have focused on static analysis, highlighting its role in malware detection. Pathak et al. [21] utilized static analysis and reverse engineering to build an Android permission-based dataset. Using 48 features and a Random Forest (RF) classifier, they achieved 97.5% accuracy in malware detection. Soi et al. [22] proposed an Android malware-detection method using API-based features from the DEX Call Graph, enabling interpretable models and malware-family correlation, with 87.3% accuracy and F1-score.

Manh et al. [23] developed a five-step ML/DL-based approach to detect malicious APKs by generating and embedding Directed API Call Graphs (DACGs) with Graph2Vec, achieving 98% accuracy. Sivaprakash [24] proposed a static-analysis approach for Android malware detection using APK disassembly and bytecode inspection. The method leveraged LSTM and functional API-based DL models, utilizing features such as API calls, opcode sequences, and n-grams.

Zhao et al. [25] introduced AppPoet, an LLM-powered Android malware-detection framework using static analysis, custom prompts, and a deep neural network (DNN) classifier. It combined multiple views for accurate detection and provided human-readable diagnostic reports. Hero et al. [26] analyzed 50 Google Play applications using ESET Security and VirusTotal, revealing that 40 applications showed malware signs of 86% adware and 14% trojans, visualized *via* a Python tool.

[27] evaluated XGBoost, RF, Support Vector Machine (SVM), and Decision Tree (DT) for Android malware detection, with RF achieving the highest accuracy of 0.99, followed by SVM at 0.96, highlighting the effectiveness of ML in enhancing Android security.

## 2.2 Dynamic-analysis Techniques for Detecting Android Malware

Fallah et al. [28] used ML techniques to detect Android malware through network-traffic analysis across multiple malware families, achieving around 90% F1-score, but showed limited effectiveness in detecting both known and novel malware families. Amel et al. [29] proposed a dynamic-analysis framework for Android malware detection using system calls, debug logs and network activity. Combining these sources improved accuracy, aided by an MQTT-based intermediary to optimize database load and learning efficiency.

Sathyadevi et al. [30] developed an Android malware-detection system using diverse data sources, such as permissions, network traffic, API calls, opcodes, system calls, binary features and behavioral logs, combined with advanced ML techniques. The framework was evaluated using standard performance metrics. Zhu et al. [31] introduced XDeepMal, an explainable DL–based malware-detection framework. It featured XTracer+, a dynamic tool that captured real-time execution traces, and an interpretation module that isolated key behavior segments influencing model decisions. Empirical results showed that XDeepMal provided robust, interpretable insights into DL-based detection.

Prathapaneni et al. [32] proposed an ensemble of ML classifiers for multi-class dynamic feature classification using the CCCS-CIC-AndMal-2020 dataset. The ensemble notably improved malware-family prediction, boosting Adware detection from 33.84% to 81.96% and Backdoor from 67.42% to 83.71%. Ciaramella et al. [33] combined DL with the Longest Common Subsequence algorithm to classify Android applications *via* system-call sequences converted into images. Using a dataset of 13,570 samples, four convolutional neural networks (CNNs) achieved up to 89% accuracy. Class Activation Mapping techniques highlighted key regions to interpret system-call patterns, distinguishing malicious from benign behaviors.

In [34], the authors analyzed 1,535 malicious Android applications and found that 18.31% used anti-analysis techniques. To counter this, they introduced DOOLDA, a dynamic-analysis framework that detected and neutralized anti-analysis behaviors by injecting targeted instrumentation across bytecode and native layers. DOOLDA successfully defeated all known anti-analysis techniques.

## 2.3 Hybrid-analysis Techniques for Detecting Android Malware

Nasser et al. [35] proposed DL-AMDet, a DL–based Android malware-detection system using hybrid-analysis features. It combined deep autoencoders for anomaly detection with a CNN-BiLSTM model for static features, achieving 99.935% accuracy across two datasets. In [36], the authors proposed HGDetector, a hybrid malware-detection method combining static function-call graphs and dynamic network-traffic features *via* graph embedding. This method improved detection accuracy by around 4% with informative traffic and up to 26% when compensating for weaker features, demonstrating the effectiveness of hybrid feature fusion.

[37] introduced MPDroid, a multi-modal pre-training approach for Android malware detection using API and function-call graphs. It leveraged graph convolutional networks and modality fusion to reduce bias and enable efficient single-modality detection. MPDroid achieved 98.3% accuracy and 97.6% F1-score while reducing training and inference time. Mesbah et al. [38] proposed LongCGDroid, an image-based malware-detection method using semantic API call graphs from control and data-flow graphs. They evaluated model performance over time as APIs evolved, finding that CNNs with abstract API features remained the most robust despite general accuracy declines.

Mercaldo et al. [39] evaluated whether images generated by deep convolutional generative adversarial networks (DCGANs) from Android malware data could be distinguished from real ones. Using static and dynamic analysis to create datasets, ML classifiers achieved an F1-score of approximately 0.8 in differentiating synthetic from real malware images. In [40] the authors explored ML techniques for Android malware detection using hybrid features from Androguard and Droidbot. k-Nearest Neighbors (KNNs) achieved the highest accuracy at 99%, followed by DT at 98%, RF at 92%, and Naive Bayes (NB) at 86%, highlighting KNNs' effectiveness.

[41] introduced TAML, a time-aware ML framework for Android malware detection using the KronoDroid dataset. It built time-aware and time-agnostic models, identifying LastModDate as a key feature. TAML achieved a 99.98% F1-score in the time-agnostic setting and up to 99% annually in time-aware evaluations over 12 years. Aledam et al. [42] proposed a hybrid Android malware-detection model combining static and dynamic analysis with PCA-based feature reduction. Evaluated on real and synthetic datasets, the model improved detection accuracy and efficiency, enhancing smartphone security.

Waheed et al. [43] enhanced the KronoDroid dataset with malware-category labels and dynamic features from real devices. Using ExtraTree for feature selection, they trained several ML models, with RF achieving 98.03% accuracy for detection and 87.56% for classifying 15 malware types.

## 2.4 Federated Learning and Privacy-protection Approaches for Detecting Android Malware

Federated Learning (FL) is a decentralized paradigm that enables a model to be trained across multiple devices without sharing or exposing the raw data of the users. Each device is trained locally by utilizing its own data and then shares only the model parameters to a central FL server, while ensuring that the user data remains in the device. Then, the FL server aggregates these model parameters to create a global model.

Hus et al. [44] proposed a private preserving FL (PPFL), which is an Android malware-detection system based on SVM. The model was trained using static analysis and secured with the Secure Multi-Party Computation (SMPC). Their results claimed that their system achieved a higher detection rate compared to the local model, and the results showed that accuracy increased with an increase in the number of clients. Mahindru et al. [45] introduced DNNdroid, which is based on the principle of FL. This model collected features from the users' devices without prior knowledge of where an application is installed. The results revealed that the model achieved a 97.8% F1-score with a false positive rate of 0.95 using one million Android applications with 500 users and 50 rounds of federation. Moreover, Taheri et al. [46] presented Fed-IIoT, which is an architecture of FL that consisted of two parts: first, the data-collection part through dynamic attack based on Generative Adversarial Network (GAN) and Federated GAN; and second, the server part, where the parameters are monitored, and by utilizing the A3GAN to avoid anomaly in aggregation and monitoring the parameters. The results showed high accuracy with 8% higher than the existing solutions. Çıplak et al. [47] proposed FEDetect, an FL model, and compared it with non-FL models by building 22 variants using LSTM and feedforward neural networks. The results showed that FL achieved 99% accuracy in binary classification and 84.5% in multi-class classification.

In this paper, TAB-DROID is proposed, which is a new framework for Android malware detection that leveraged CMIM and JMI for feature selection, employed a PQ technique for feature compression and utilized TabPFN classifier. The system is evaluated using two datasets, the first consists of hybrid features, which have demonstrated superior performance compared to static or dynamic features mentioned above, thereby enhancing the overall detection accuracy and robustness of the model. The second dataset comprises only static features.

## 3. TAB-DROID FRAMEWORK

The proposed TAB-DROID framework is comprised of four sequential and interrelated stages, as illustrated in Figure 1, each contributing to the overall malware-detection process. The first stage is data pre-processing, where the data is cleaned and balanced. The second stage performs feature reduction using two concurrent advanced selection methods: CMIM and JMI, with selection guided by framework-accuracy performance. In the third stage, the selected features are compressed using PQ to reduce computational complexity while retaining discriminative capability. Finally, the fourth stage employs the TabPFN classifier for efficient and accurate malware detection. Figure 1 presents the overall methodology of the TAB-DROID framework.

### 3.1 Dataset Overview

This sub-section explores the real-world datasets employed in the proposed work. The first dataset utilized is TUANDROMD, comprising 4,464 Android APK samples (3,565 malware and 899 benign) and 241 behavioral features stored in CSV format. The dataset construction followed three phases: phase 1: application collection; phase 2: static and dynamic analysis using tools, like APKAnalyser [48], Androguard, and Smali-CFGs [49]; phase 3: feature extraction. Extracted features are categorized as permission-based (i.e., captured during installation and runtime) and API call–based (i.e., reflecting functional behavior). The resulting hybrid feature set combined static and dynamic attributes, enabling comprehensive behavioral profiling for robust machine learning–based malware detection.

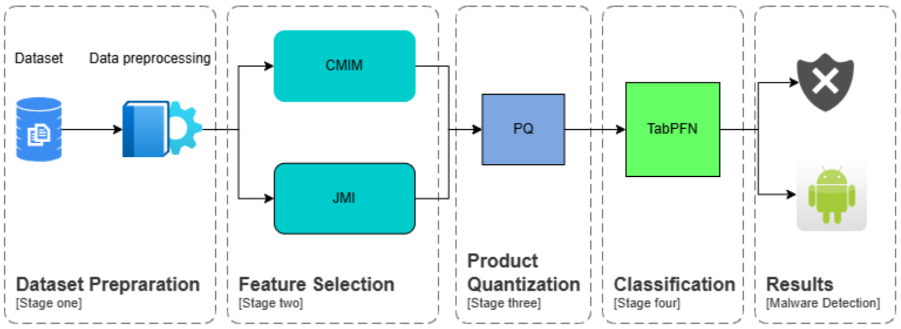

Figure 1. The proposed TAB-DROID framework.

The second dataset used is Malgenome, comprising 3,799 Android APK samples (2,539 benign and 1,260 malwares) sourced from the widely used Android Malware Genome Project [50]. It includes 215 features extracted *via* an automated static-analysis tool developed in Python. After decompiling the manifest files using AXMLPrinter2, the tool extracted permissions and intents, while API calls are retrieved by reverse engineering .dex files using the Baksmali disassembler. The analyzer also detected potentially dangerous Linux commands and checked for embedded files, such as: .dex, .jar, .so, and .exe, enabling comprehensive static feature extraction.

### 3.2 Data Pre-processing

Data pre-processing is an essential step to ensure the quality and suitability of the dataset before applying the proposed framework, as poor data quality can significantly impair model performance. The pre-processing process involves handling missing values, removing nulls and outliers and discarding features with zero or very low variance, which contribute a little to learning due to their minimal informational value. These steps ensure that the final dataset is clean, structured and ready for modelling.

As described earlier, the dataset exhibited class imbalance, with benign samples representing only a quarter of the malware samples. Such an imbalance can bias the model toward the majority class and degrade detection performance. To address this, two resampling strategies are typically employed: over-sampling (i.e., increasing minority-class instances) and under-sampling (i.e., reducing majority-class instances).

In this paper, the Synthetic Minority Oversampling Technique (SMOTE) is applied to over-sample the benign class. SMOTE is chosen for its ability to generate diverse synthetic samples, thereby preserving pattern variability and improving model generalization. This resulted in a balanced dataset with 3,565 samples per class for TUANDROMD dataset and 2539 per class for Malgenome dataset, subjected to complete pre-processing for subsequent analysis.

## 3.3 Feature Selection

Feature selection plays a critical role in pattern recognition and machine-learning systems, serving as an essential pre-processing step to identify or prioritize features based on their relevance to the target task. In this paper, feature generation, as illustrated in Figure 1, yields many features. Consequently, applying feature selection techniques becomes imperative to reduce dimensionality, thereby decreasing model training time and enhancing the overall detection performance and classification accuracy by emphasizing the most informative features. Various feature-selection strategies are developed to isolate an optimal sub-set of features, facilitating the construction of more efficient and effective models. Generally, these techniques are categorized into three primary groups: filter methods, wrapper methods and embedded methods.

Filter methods rank features based on their intrinsic statistical properties without involving model training. They are simple, fast and well-suited for large datasets. However, they ignore feature inter-dependencies, potentially selecting individually relevant features that may not improve model performance. Wrapper methods select features by iteratively training a model and evaluating performance. These approaches yield high relevance, but are computationally expensive and time-consuming. They also risk overfitting, especially with small datasets. The embedded methods integrate feature selection within model training, combining the efficiency of filters and the accuracy of wrappers. This process reduces overfitting by considering feature relevance during training, but depends heavily on the chosen algorithm and involves higher computational complexity.

In this paper, two filter techniques are utilized as concurrent feature-selection strategies because of the advantages of filter methods over wrapper and embedded methods. These techniques are CMIM [51] and JMI [52]. Both seek to identify the features that exhibit the maximum relevance to the target variable while minimizing redundancy with the already selected features. This ensures that the chosen features contribute to the most informative content without introducing unnecessary overlap, leading to more efficient and effective models.

The final selection between these two techniques is determined based on which technique yields superior performance for the TAB-DROID framework. Both techniques are based on information theoretic principles, relying on entropy and mutual information [53]-[54]. Entropy serves as a fundamental concept in the information theory of uncertainty in a random variable, denoted as $E(F)$, which quantifies the amount of uncertainty or randomness associated with the distribution of $F$. It is defined as shown in Equation (1).

$$E(F) = -\sum_{f\in F} \mathcal{P}(f) \log \mathcal{P}(f) \tag{1}$$

where $\mathcal{P}(f)$ is the probability mass function of the random variable $F$. $f$ represents the value of all possible values of $F$. If the distribution of $F$ is heavily skewed towards one particular event, indicating minimal uncertainty regarding the outcome, the entropy $E(F)$ is low. In contrast, events are likely equal, implying maximum uncertainty about the result, and the entropy reaches its maximum value. This characteristic of entropy helps quantify the unpredictability or the level of disorder inherent in the system represented by $F$. Let $Y$ be another event; the entropy can be conditioned on that event. This can be denoted as shown in Equation (2).

$$E(F|Y) = -\sum_{y\in Y} \mathcal{P}(y) \sum_{f\in F} \mathcal{P}(f|y) \log \mathcal{P}(f|y) \tag{2}$$

This can be interpreted as the amount of uncertainty that remains in $F$ after the outcome of $Y$ is known.

In other words, it quantifies how much information $F$ still contains after accounting for the information provided by $Y$. This is central to understanding the relationship between two random variables in terms of shared information. Now, mutual information between $F$ and $Y$ can be formally defined, which quantifies the amount of information shared between these two variables. It is defined as shown in Equation (3) (for more details, see appendix A).

$$MI(F;Y) = E(F) - E(F|Y) \tag{3}$$

Equation (3) represents the difference between: E($F$), which is the uncertainty about $F$ before knowing $Y$ and $E(F|Y)$, which is the uncertainty about $F$ after knowing $Y$, Mutual Information can be interpreted as the amount of uncertainty in $F$ that is eliminated by knowing $Y$. Thus, it corresponds to the intuitive interpretation of mutual information as the quantity of information that one variable reveals about another.

One of the properties of mutual information is symmetry, which means that $MI(F;Y) = MI(Y;F)$. Additionally, when $f$ and $y$ variables are statically independent, this means the mutual information equals zero, indicating that their joint probability distribution is factorized as $\mathcal{P}(f,y) = \mathcal{P}(f)\mathcal{P}(y)$.

Alternatively, Equation (3) according to Fleuret [50] can also be expressed as in Equation (5).

$$MI(F;Y) = E(F) + E(Y) - E(F,Y) \tag{4}$$

Similar to Equation (2), mutual information can also be conditioned, meaning that the amount of shared information can be measured between two random variables $F$ and $Y$ while accounting for the influence of a third variable $Z$. The conditional mutual information is defined as in Equation (5) (for more details, see appendix A).

$$MI(F;Y|Z) = E(F|Z) - E(F|YZ) \tag{5}$$

Alternatively, according to Fleuret [50] Equation (5) can also be expressed as in Equation (6) (for more details, see appendix A).

$$MI(F;Y|Z) = E(F,Z) - E(Z) - E(F,Y,Z) + E(Y,Z) \tag{6}$$

After establishing a foundational understanding of entropy, mutual information and their conditional forms, the JMI and CMIM techniques are introduced and discussed, along with their key differences. Assume that a dataset of features set $F$, where $F = \{f_1, f_2, ... f_n.., f_{N-1}, f_N\}$, and $N$ is the total number of features, while $S$ is another set of selected features chosen according to their scores, where $S = \{s_1, s_2, ... s_k., s_{K-1}, s_K\}$ while $K$ is the number of features needed to be selected and the label of the class is $L$. The objective function of JMI is represented as in Equation (7) (for more details, see appendix A).

$$JMI(f_n) = MI(f_n, L) - \frac{1}{|S|}\sum_{s_k \in S}[MI(f_n, s_k) - MI(f_n, s_k|L)] \tag{7}$$

Equation (7) computes the JMI between the pair of features $(f_n, s_k)$ and the class label $L$. By using the sum operator in the equation, it quantifies how much additional and combined information the new candidate feature $f_n$ can provide together with each previously selected feature $s_k$ in predicting the class label $L$. The objective function of CMIM is represented in Equation (8) (for more details, see appendix A).

$$CMIM(f_n) = MI(f_n; L) - \{max_{s_k \in S}[MI(f_n; s_k) - MI(f_n; s_k|L)]\} \tag{8}$$

Equation (8) computes the conditional mutual information between $f_n$ and the target class $L$, conditioned on each feature $s_k$ that has already been selected. The minimum of these values is then used as the score for $f_n$, which means that $f_n$ has maximum relevance to the class label and minimum redundancy to the features in the $S$ set.

The CMIM algorithm exists in two variants: the standard and optimized implementations. The fast implementation is utilized in this paper, which is the optimized version. The Fast-CMIM version exploits the observation that, as the selection process progresses, the score vector can only decrease, allowing for the omission of unnecessary score updates. Importantly, this optimization introduces no approximations.

In the fast CMIM approach, for each feature $f_n$, a partial score $Ps[n]$is maintained, representing the minimum of the conditional mutual information values as defined in Equation (8). Additionally, a vector $m[n]$ is used to track the index of the last selected feature considered in the computation of $Ps[n]$. The detailed procedure for the fast CMIM implementation is outlined in the pseudo-code presented in Algorithm 1.

The principal distinction between JMI and CMIM lies in their selection strategies. JMI emphasizes the selection of features that, in combination with the already selected features, share a significant amount of information with the class label $L$, as described in Equation (7). In other words, JMI favours features that demonstrate strong collaborative effectiveness. In contrast, CMIM focuses on selecting features that provide a novel information about the class label $L$, which has not yet been captured by the previously selected features, as described in Equation (8). Thus, CMIM prioritizes features that are individually strong and minimally redundant. Due to this difference, JMI may tolerate a certain degree of redundancy if the additional features contribute positively to the overall model performance, whereas CMIM may exclude features that, despite being informative, are redundant with already selected ones.

**Algorithm 1.** Fast-CMIM.

```
1.  For ($n = 1\ to\ N$) Do:
2.      $Ps[n] = MI(n)$
3.      $m[n] = 0$
4.  End For
5.  For ($k = 1\ to\ K$) Do:
6.      $q^* = 0$
7.      For ($n = 1\ to\ N$) Do:
8.          While ($Ps[n] > q^*$) & ($m[n] < K - 1$) Do:
9.              $m[n] = m[n] + 1$
10.             $Ps[n]$ = min ($Ps[n]$, $conditional\_MI$ ($n, nu[m[n]]$))
11.         End While
12.         If ($Ps[n] > q^*$) Then:
13.             $q^* = Ps[n]$
14.             $nu[k] = n$
15.         End If
16.     End For
17. End For
```

**Note**: $\boldsymbol{conditional\_MI(n,m)}$= $\boldsymbol{MI(f_n; L|s_k)}$, $\boldsymbol{MI(n)}$ = $\boldsymbol{MI(f_n; L)}$, and $\boldsymbol{nu[m[n]]}$ is conditioned on selected features.

## 3.4 Product Quantization

In the data-science domain, especially with high-dimensional datasets, the constraints of the resources of memory and computation present great challenges. As the size and dimensionality of data rise, the associated processing and storage requirements can quickly become infeasible. To address these restrictions, this sub-section presents PQ [55]-[57] as an effective data-compression technique that preserves main patterns and structure within the data while safely reducing size demands.

Before illustrating PQ, it is important to understand the concept of quantization and its associated benefits. Quantization is a compression technique used for high-resolution data that maps it to smaller discrete values. This data precision is slightly reduced, but it keeps the key characteristics of the original data, thereby enabling efficient compression of data without substantial loss of information. To formally describe the PQ process, some notations will be introduced. Let the dataset be composed of high-dimensional vectors in an $N$-dimensional space. For simplification, a vector $V \in \mathcal{R}^N$ is utilized. This vector $V$ is partitioned into $I$ sub-vectors ($SV$) as in Equation (9).

$$V = \{\overbrace{v_1, v_2, .., v_{\frac{N}{I}}}^{SV_1}, ...., \overbrace{v_{N-\frac{N}{I}+1}, .... v_N}^{SV_I}\} = \{SV_1, SV_2, ..., SV_i, .... SV_I\} \quad (9)$$

$SV_i \in \mathcal{R}^{N/I}$, for $i \in \{1,2, ..., I\}$, representing the $i^{th}$ sub-vector of $V$. Each $SV_i$ is quantized independently, which allows for efficient compression. During the training phase of PQ, a separate sub-codebook is constructed for each $SV_i$. Each sub-codebook is denoted as in Equation (10).

$$\mathcal{C}_i = \{ c_q^i\}_{q=1}^{Q} \quad (10)$$

where $c_q^i \in R^{N/I}$ represents the $q^{th}$ sub-codeword in the $i^{th}$ sub-codebook and $Q$ denotes the number of codewords per sub-codebook and the parameter $Q$ is specified by the user during the quantization.

Each sub-codebook $C_i$ is obtained by applying k-mean clustering algorithm to the $i^{th}$ sub-space of the training vectors. To encode a high-dimensional vector $V$ using PQ, each sub-vector $SV_i$ is independently encoded into an identifier of its nearest codeword in $C_i$ using sub-encoder function, which is denoted as in Equation (11).

$$\ell^i(\,SV_i) = arg\;min_{q\in\{1,..Q\}} \left\|SV_i - c_q^i\right\|^2 \tag{11}$$

After encoding all the sub-vectors, the original high-dimensional vector $V$ is compactly represented as a sequence of $I$ discrete identifiers, one for each sub-vector. At the end, by concatenating these $I$ identifiers, a PQ-code is created, which defined as in Equation (12).

$$\ell(V) = \{\ell^1(SV_1), \ldots.,\ell^I(SV_I)\} \tag{12}$$

where an encoder function is defined as $\ell: R^{\mathrm{N}} \longrightarrow \{1,\ldots,Q\}^I$. Although conceptually straightforward, the implementation of PQ is also relatively simple. This is illustrated in Algorithm 2, which presents the corresponding pseudocode.

**Algorithm 2.** Product Quantization Pseudocode (PQ)

```
1.  # a sample vector
2.  $V$
3.  # The dimension per sub-vector Ds = N / I
4.  I = 66
5.  # The dimension per sub-vector
6.  Ds = TNF/I
7.  # Creating empty array for codewords
8.  CW = (I, Q, Ds)
9.  **For (**i = 1 to I**) Do:**
10.     Sub_vec = V [i * Ds to (i+1) * Ds]
11.     CW [i] = k-means (Sub_vec, Q)
12. **End For**
13. #creating empty array for PQ identifier
14. PQC = [I]
15. #encoding each sub-vector into an identifier
16. **For (**i = 1 to I**) Do:**
17.     Sub_vec = V [ i * Ds to (i+1) * Ds]
18.     PQC[i] = VQ (sub_vec, CW[i])
19. **End For**
```

**V** is a sample vector, **I** = number of sub-vectors, **Ds** = dimension per sub-vector (N/I), **TNF** = total number of features, **CW** = 3D array to store codewords, **k-means** = is applying the k-mean clustering where the **Q** is the number of clusters, **PQC** = array for product-quantization code, **VQ** = nearest codeword in CW[i].

## 3.5 TabPFN Classifier

TabPFN is a type of classifier specifically designed for tabular datasets [58], built on a generative transformer-based neural network [59] and pre-trained or self-learned on a large synthetic dataset before being used. Since it is pre-trained, TabPFN requires neither any retraining tasks nor fine tuning during inference, which makes it computationally efficient. TabPFN does not stick with or tailor to a specific tabular dataset, which enhances its applicability to real-world data.

TabPFN consists of three phases: The first phase is the data-generation phase, where various synthetic tabular datasets are generated utilizing structural causal models (SCMs) [60] which encode diverse targets and feature relationships. These datasets are built to capture a broad spectrum of potential scenarios. SCMs provide a framework that represents the structural relationships and generative process that the dataset relies on. The construction of these datasets starts with sampling high-level hyper-parameters (i.e., the number of features, data size, and the level of difficulty) to control the properties of the datasets generated. Afterward, a cyclic graph is constructed, which allocates the causality structure that the dataset relies on. Each sample per dataset is generated by propagating random noise on the causal graph root node. These samples are produced by sampling from a uniform distribution or random normal distribution and applying a varied set of computational mapping as these samples pass through the edges of the graph. At each edge, Gaussian noise is added. After constructing the causal graph, the sample features and target values are extracted from the feature and target nodes, which are sampled.

The second phase is the pre-training phase, where a single transformer is trained offline using a large collection of synthetic datasets. The objective of this training is to learn to predict unknown targets within synthetic datasets. This phase is performed only once at the beginning and enables the model to acquire a generalized inference model. This approach is referred to as Prior Data Fitted Networks (PFNs) [61], where the learned algorithm can be applied to new datasets without additional retraining. In the final phase, after comprehensive pre-training on synthetic datasets, the model is applied to real-world data (i.e., an unseen dataset from the model's point of view). This step leverages in-context learning (ICL) [62], wherein a set of labelled instances is provided as context. The model utilizes this contextual information to infer the labels of unseen samples without requiring re-training or fine tuning. ICL enables the model to generalize effectively to new tasks by drawing on the algorithmic priors learned during the pre-training phase.

## 4. Experimental Results

This section outlines the setup for the experiment, the evaluation metrics employed and the results. There are five evaluation metrics employed to assess the performance of the proposed framework. These include accuracy, recall, precision, F1-score and the area under the Receiver Operating Characteristic (ROC) curve (AUC) as presented in Equations (13) to (16), respectively [63].

$$Accuracy = \frac{T_P+T_N}{T_P+T_N+F_P+F_N} \tag{13}$$

$$Recall = \frac{T_P}{T_P+F_N} \tag{14}$$

$$Precision = \frac{T_P}{T_P+F_p} \tag{15}$$

$$F1 = \frac{T_P}{T_P+\frac{1}{2}(F_p+F_N)} \tag{16}$$

Here, $T_P$ (True Positive) denotes the number of malicious APKs correctly classified as malicious, while $T_N$ (True Negative) represents the number of benign APKs correctly identified as benign. $F_P$ (False Positive) refers to benign APKs incorrectly labelled as malicious, and $F_N$ (False Negative) corresponds to malicious APKs that are incorrectly classified as benign.

After preparing the datasets as described in the data pre-processing sub-section, each one is partitioned into training (80%) and testing (20%) sub-sets. These sub-sets are processed through two parallel pipelines. The first pipeline consists of Fast-CMIM for feature selection, followed by PQ and finally, TabPFN. The second pipeline follows the same structure, but begins with JMI instead of Fast-CMIM.

Each feature-selection technique is independently applied to the training data to identify the most informative features. The selected features are then used to replace the original feature sets in both training and testing sub-sets. PQ is subsequently applied to compress these datasets, after which TabPFN is trained on the compressed training data and evaluated on the corresponding compressed test sets. To optimize the PQ, the number of sub-vectors $I$ and the number of codewords per sub-vector are empirically tuned. The best configuration is selected based on maximum classification accuracy with the lowest-feature sub-set. To reduce computational complexity, each feature vector is divided into sub-vectors containing exactly two features. Each sub-vector is then quantified using k-means clustering with a codebook size of two.

TabPFN is evaluated using 5-fold cross-validation, with performance metrics averaged to assess generalization capability. Experimental results demonstrate that the CMIM-based pipeline consistently outperformed the JMI-based pipeline. For the TUANDROMD dataset, the optimal configuration is achieved using 66 selected features, grouped into 33 sub-vectors, each quantized with 2 codewords, resulting in a final input dimensionality of 33. Similarly, for the Malgenome dataset, the best results are obtained with 88 features, 44 sub-vectors and 2 codewords per sub-vector, yielding a reduced dimension of 44.

Figure 2 illustrates the high performance of the CMIM-based pipeline across both datasets. It achieves peak accuracy at 66 features (TUANDROMD) and 88 features (Malgenome), while the JMI-based pipeline shows comparatively lower accuracy, even as the number of features increases. The CMIM-based pipeline also consistently outperforms JMI across all evaluation metrics, including accuracy, AUC, precision, recall, and F1-score, indicating its effectiveness in selecting highly discriminative features. These results confirm that CMIM not only reduces feature dimensionality, but also enhances the overall classification performance.

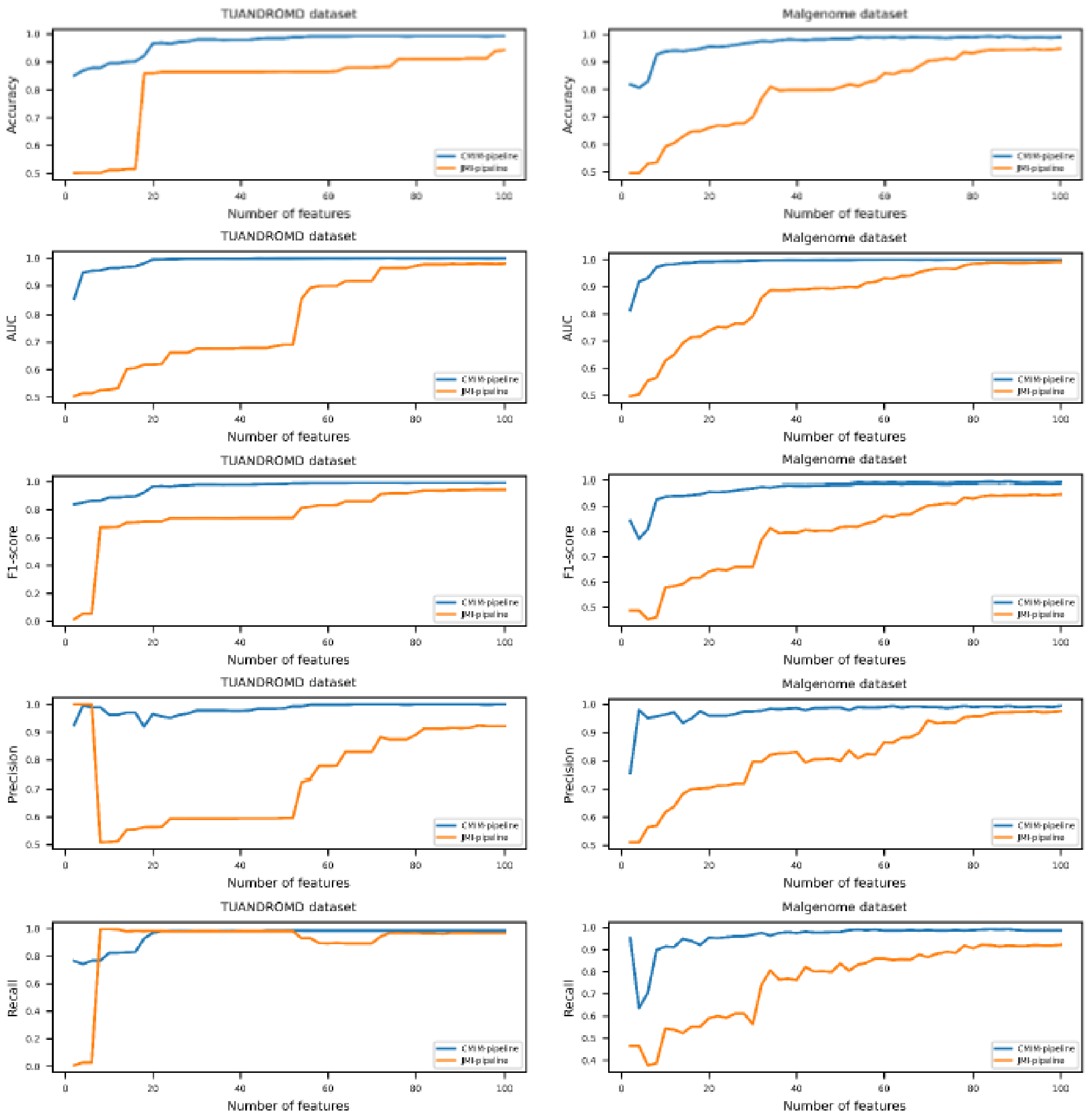

Figure 2. Comparison of evaluation metrics across different feature counts for CMIM and JMI-based pipelines on both datasets.

The results in Table 1 demonstrate the greater performance of the CMIM-based pipeline over the JMI-based pipeline, across all evaluated performance metrics. These results indicate that individual features in the dataset exhibit strong relevance to the target classes, which is consistent with the CMIM approach that prioritizes the individual contribution of each feature to the labelled class. In contrast, JMI emphasizes the joint relevance of feature combinations, which may overlook individually informative features. Based on the experimental findings, the TAB-DROID is constructed by integrating the CMIM feature-selection technique, PQ for dimensionality reduction and the TabPFN classifier, which is selected due to its superior performance across all evaluation metrics. Also, it is shown that incorporating PQ significantly reduces both training and testing times, particularly testing time, as well as overall memory usage, as shown in Table 2. This highlights PQ's effectiveness in optimizing computational efficiency within the proposed framework.

Table 1. Performance evaluation of the proposed CMIM-based pipeline and JMI-based pipeline.

| **Framework** | **Dataset Name** | **Number of Features** | **Accuracy (%)** | **AUC (%)** | **Precision (%)** | **Recall (%)** | **F1-score (%)** |
|---|---|---|---|---|---|---|---|
| CMIM-based Pipeline | TUANDROMD | 66 | 99.22 | 99.92 | 99.69 | 98.76 | 99.22 |
| JMI-based Pipeline | | 100 | 94.57 | 97.62 | 98.09 | 90.90 | 94.36 |
| CMIM-based Pipeline | Malgenome | 88 | 98.62 | 99.81 | 99.19 | 98.01 | 98.60 |
| JMI-based Pipeline | | 100 | 94.09 | 98.48 | 97.13 | 90.86 | 93.88 |

Table 2. Comparison of the CMIM-based pipeline with and without PQ in terms of training time, testing time and memory usage.

| **CMIM-based Pipeline** | **Dataset Name** | **Training Time (sec)** | **Testing Time (sec)** | **Memory Usage (MB)** |
|---|---|---|---|---|
| With PQ | TUANDROMD Dataset (66-feature) | 33.70 | 0.004 | 15.2 |
| Without PQ | | 33.82 | 0.009 | 35.5 |
| With PQ | Malgenome Dataset (88-feature) | 6.91 | 0.006 | 21.6 |
| Without PQ | | 8.657 | 0.01 | 34.1 |

Figure 3 compares the performance of traditional classifiers, such as LR, SVM, NB, and Gradient Boosting (GB), against the TabPFN classifier within the CMIM-based pipeline, using the same feature sub-set selected by the CMIM technique for both datasets. The results in the graphs demonstrate that the proposed framework achieves superior accuracy, along with higher precision, F1-score and recall. Although the AUC remains comparable across models, TabPFN consistently outperforms classical classifiers, confirming its effectiveness in the proposed detection system.

To assess the performance and reliability of the proposed TAB-DROID framework, a comparative analysis is carried out against other recent Android malware detection frameworks. As shown in Table 1, the TUANDROMD dataset consistently yields higher performance compared to the Malgenome dataset, which can be attributed to its hybrid feature composition, offering richer behavioral insights. Unlike Malgenome, which includes only static features, TUANDROMD integrates recent attributes, providing greater data diversity and enhancing its suitability for real-time and obfuscation-resilient [64] detection. A detailed comparison of the two datasets' main aspects is summarized in Table 3. Therefore, all frameworks are evaluated on the TUANDROMD dataset to ensure consistency and fairness in comparison, leveraging its comprehensive information for more robust evaluation.

Table 3. Comparison of TUANDROMD and Malgenome characteristics.

| **Aspect** | **Malgenome Dataset** | **TUANDROMD Dataset** |
|---|---|---|
| Data Diversity | Focuses on early Android threats of 49 malware families. | Modern dataset captures the recent spectrum of 71 malware families. |
| Real-time Testing | Less realistic, as samples do not reflect today’s threats. | Most realistic, as the samples exhibit current attack behaviors. |
| Obfuscation & Evasion | Susceptible to evasion attacks. | Employs advanced obfuscation and morphing techniques. |

The comparative results are summarized in Table 4. for TAB-DROID *versus* those state-of-the-art frameworks [64]-[70] utilizing the TUANDROMD dataset, Wajahat, Ahsan et al. [66] utilized the smallest feature set, emphasizing feature economy. T. Kacem et al. [67] achieved the highest accuracy and F1-score, which are nearly equivalent to those attained by the proposed TAB-DROID framework. Furthermore, TAB-DROID reduced the feature space by 73% of the original 241 features while maintaining superior performance. It outperformed all other frameworks in terms of precision and achieved an AUC value approaching 100%, highlighting its robustness and high classification confidence.

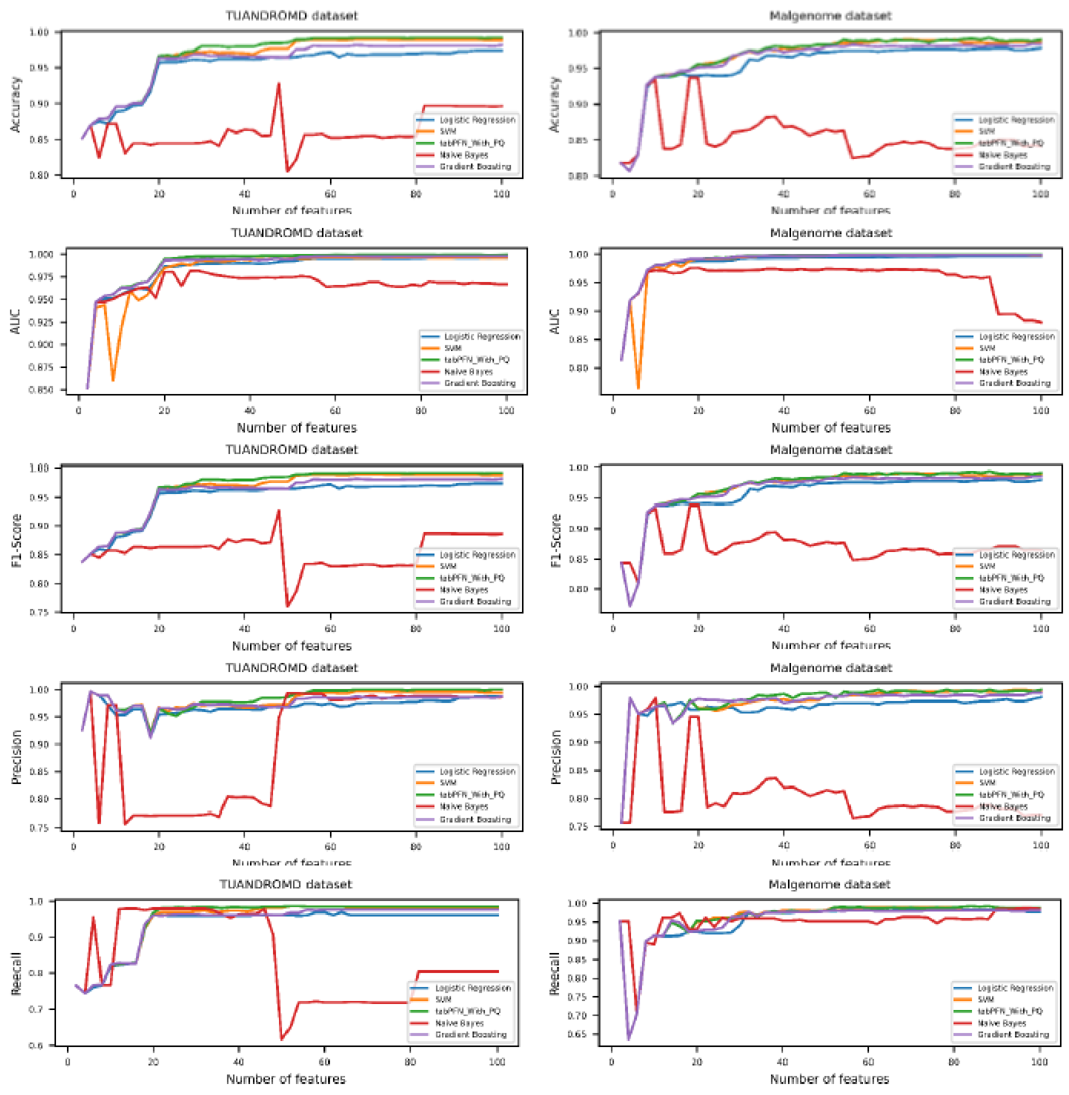

Figure 3. Comparison of evaluation metrics for CMIM-based pipeline *versus* classical classifiers on both datasets.

Despite its strong performance, TAB-DROID has limitations, including reliance on specific feature sets, challenges in detecting more advanced evasion techniques and the need for PQ hyper-parameter tuning. Additionally, employing a lightweight alternative classifier could make the system more suitable for deployment on mobile devices.

Table 4. TAB-DROID performance *versus* recent state-of-the-art frameworks.

| Ref. | Year | Model | Features (%) | Accuracy (%) | AUC (%) | Precision (%) | F1-score (%) |
|---|---|---|---|---|---|---|---|
| [64] | 2024 | SVC | 241 | 98.9 | 100 | 97 | 84.2 |
| [65] | 2024 | DNDF | 241 | 98.4 | 99 | 98.8 | 98.94 |
| [66] | 2024 | RFE+RF | 40 | 98 | - | 99 | 98.8 |
| [67] | 2025 | Transformer | 241 | 99.25 | 98.76 | 99.26 | 99.26 |
| [68] | 2025 | Decision Tree | 241 | 99.1 | - | 99 | 99 |
| [69] | 2024 | Transformer+CNN | 241 | 97.7 | - | 97.5 | 97.25 |
| [70] | 2024 | RF+PCA | - | 98 | - | 98 | 98 |
| **Proposed TAB-DROID** | | **CMIM+PQ+TabPFN** | **66** | **99.22** | **99.92** | **99.69** | **99.22** |

The entire system is implemented and executed using Python version 3.8 within a Jupyter Notebook environment. Feature selection using the JMI method is conducted utilizing the publicly available scikit-feature library [71], which includes a range of widely used feature-selection algorithms. Similarly, the Fast CMIM technique is adopted from the same repository, with modifications introduced to enhance computational efficiency, as outlined in algorithm 2. For vector quantization, PQ is implemented using the nanopq library [72]. The TabPFN classifier is integrated using its official open-source repository [73]. The experimental setup utilizes essential Python libraries, including Pandas (v1.2), NumPy (v1.23), Scikit-learn (v1.2) and Matplotlib (v3.7). The experiment is conducted using Google Colab, a cloud-based Jupyter notebook environment that provides free access to computational resources. Google Colab allows users to write and execute Python code directly in the browser without local configuration. It also offers access to hardware accelerators, including GPUs, which significantly enhances computational efficiency. In this paper, a T4 GPU is utilized to accelerate experimental processes.

## 5. CONCLUSION

In this paper, TAB-DROID is introduced, which is a novel and efficient Android malware-detection framework integrating CMIM, PQ and the TabPFN classifier. The framework demonstrated superior performance, where the accuracy, AUC, precision, recall and F1-score metrics reached 99.2%, 99.9%, 99.6%, 98.7% and 99.2%, respectively. CMIM reduced the feature space by 73%, while PQ decreased testing time and memory usage by 44.4% and 42.8%, confirming its resource efficiency. Compared to recent approaches, TAB-DROID improved accuracy by up to 1.52% and precision by up to 2.69%, achieving near-perfect classification. Moreover, TAB-DROID is considered general and scalable. Regarding generalization, it utilized TabPFN, which avoided overfitting to specific datasets and enabled robust detection across diverse malware samples. In terms of scalability, TAB-DROID applied feature reduction and compression, which lowered computational complexity, testing time and memory usage.

For future work, the TAB-DROID can be extended for deployment on cloud-based solutions, enabling mobile devices to leverage the framework without heavy local computation. Additionally, it is planned to evaluate it by integrating additional datasets that incorporate more advanced evasion techniques and a broader set of behavioral features, allowing the system to identify the most informative features and maintain high detection performance as Android malware rapidly evolves.

## APPENDIX A

### A1. Proof of Equation (3)

Let $F \ and \ Y$ be two random variables, the mutual information between $F$ and $Y$ by definition of Kullback-Leibler divergence is defined as follows: $MI(F;Y) = \sum_{f\in F}\sum_{y\in Y}\mathcal{P}(f,y)\log\frac{\mathcal{P}(f,y)}{\mathcal{P}(f)\mathcal{P}(y)}$

Expand the log, which separates the *MI* into three summation terms, each of which corresponds to an entropy as follows: $\sum_{f\in F}\sum_{y\in Y}\mathcal{P}(f,y)\,[\log\mathcal{P}(f,y) - \log\mathcal{P}(f) - \log\mathcal{P}(y)]$

Evaluate each term using the entropy definition:

The first term: $-\sum_{f\in F}\sum_{y\in Y}\mathcal{P}(f,y)\log\mathcal{P}(f,y) \ = \ E(F,Y)$

The second term: $-\sum_{f\in F}\sum_{y\in Y}\mathcal{P}(f,y)\log\mathcal{P}(f) \ = \ \sum_{f\in F}\left(\sum_{y\in Y}\mathcal{P}(f,y)\right)\log\mathcal{P}(f)$

According to the Marginal distribution, the second term can be rewritten as follows:

$$-\sum_{f\in F}\sum_{y\in Y}\mathcal{P}(f,y)\log\mathcal{P}(f) \ = \ \sum_{f\in F}\mathcal{P}(f)\log\mathcal{P}(f) \ = \ E(F)$$

The third term is similar to the second term: $-\sum_{f\in F}\sum_{y\in Y}\mathcal{P}(f,y)\log\mathcal{P}(y) = \sum_{f\in F}\mathcal{P}(f)\log\mathcal{P}(f) = \ E(Y)$

Substitute back the three terms into the *MI* equation: $MI(F;Y) = \ E(F,Y) - E(F) \ - \ E(Y)$

By using the chain rule, where $MI(F;Y) = E(\mathrm{Y}) + E(F|Y) \ = \ E(F) + E(Y|F)$, rearrange to obtain the following formula:

$$MI(F;Y) = \ E(F) - E(F|Y)$$

### A2. Proof of Equation (5)

The conditional mutual information between $F \ and \ Y \ given \ Z$ definition is as follows:

$$MI(F;Y|Z) = \sum_{f\in F}\sum_{y\in Y}\sum_{z\in Z}\mathcal{P}(f,y,z)\log\frac{\mathcal{P}(f,y|z)}{\mathcal{P}(f|z)\mathcal{P}(y|z)}$$

This definition measures the dependence between $F \ and \ Y$ once the variable $Z$ is known.

Expand the log by splitting the *MI* into three summation terms, each of which corresponds to a conditional entropy as follows: $\sum_{f\in F}\sum_{y\in Y}\sum_{z\in Z}\mathcal{P}(f,y,z)[log\,\mathcal{P}(f,y|z) - log\,\mathcal{P}(f|z) - log\,\mathcal{P}(y|z)]$
Evaluate each term using the conditional entropy definition:
The First term: $\sum_{f\in F}\sum_{y\in Y}\sum_{z\in Z}\mathcal{P}(f,y,z)\,log\,\mathcal{P}(f,y|z) = -\,E(F,Y|Z)$
The second term: $\sum_{f\in F}\sum_{y\in Y}\sum_{z\in Z}\mathcal{P}(f,y,z)\,log\,\mathcal{P}(f|z) = -\,E(F,|Z)$
The third term is similar to the second term: $\sum_{f\in F}\sum_{y\in Y}\sum_{z\in Z}\mathcal{P}(f,y,z)\,log\,\mathcal{P}(y|z) = -\,E(Y,|Z)$
Substitute back the three terms into the conditional mutual information equation:

$$MI(F;Y|Z) = E(F,Y|Z) - E(F|Z) - E(Y|Z)$$

By recalling the chain rule for conditional entropy:

$$E(F,Y|Z) = E(F|Y,Z) + E(Y|Z)$$

therefore, by substituting the chain rule into the expression, we obtain:

$$MI(F;Y|Z) = E(F|Z) - E(F|Y,Z)$$

### A3. Proof of Equation (6)

By recalling the definition of conditional entropy:

$$\begin{aligned} E(F|Z) &= E(F,Z) - E(Z) \\ E(F|Y,Z) &= E(F,Y,Z) - E(Y,Z) \end{aligned}$$

By substitution in Equation (5):

$$MI(F;Y|Z) = E(F,Z) - E(Z) - E(F,Y,Z) + E(Y,Z)$$

### A4. Proof of Equation (7)

The joint mutual information can be written as follows:

$$JMI(f_n) = \sum_{s_k\in S} MI(f_n, s_k\,; L) = \sum_{s_k\in S}[MI(s_k; L) + MI(f_n, L|s_k)]$$

Neglecting the term $MI(s_k; L)$ as it is constant with respect to $f_n$, thus the joint mutual information reduces as follows:

$$\begin{aligned} &= \textstyle\sum_{s_k\in S}[MI(f_n, L|s_k)] \\ &= \textstyle\sum_{s_k\in S}[MI(f_n, L) - MI(f_n, s_k) + MI(f_n, s_k|L)] \\ &= |S| \times MI(f_n, L) - \textstyle\sum_{s_k\in S}[MI(f_n, s_k) - MI(f_n, s_k|L)] \end{aligned}$$

$$JMI(f_n) = MI(f_n, L) - \frac{1}{|S|}\sum_{s_k\in S}[MI(f_n, s_k) - MI(f_n, s_k|L)]$$

### A5. Proof of Equation (8)

The Conditional Mutual Information has a very similar procedure. The original and its rewriting are as follows:

$$\begin{aligned} CMIM(f_n) &= min_{s_k\in S}[MI(f_n; L|s_k)] \\ &= min_{s_k\in S}[MI(f_n; L) - MI(f_n; s_k) + MI(f_n; s_k|L)] \\ &= MI(f_n; L) + min_{s_k\in S}[MI(f_n; s_k|L) - MI(f_n; s_k)] \\ CMIM(f_n) &= MI(f_n; L) - max_{s_k\in S}[MI(f_n; s_k) - MI(f_n; s_k|L)] \end{aligned}$$

## AVAILABILITY OF DATA AND MATERIALS

The TUANDROMD dataset used during the current study is available in the [UCI Machine Learning] repository, [https://doi.org/10.24432/C5560H].

The Malgenome dataset used during the current study is available in the [figshare] repository, [https://doi.org/10.6084/m9.figshare.5854590.v1]

## ACKNOWLEDGEMENTS

The authors would like to thank the Egyptian Computer Emergency Readiness Team (EG-CERT) and National Telecom Regulatory Authority (NTRA) for providing helpful ideas and suggestions.

## REFERENCES

[1] Statista.com, "Number of Smartphone Mobile Network Subscriptions Worldwide from 2016 to 2023, with Forecasts from 2023 to 2028," [Online], Available: http://statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[2] Statista.com, "Market Share of Mobile Operating Systems Worldwide from 2009 to 2025, by Quarter," [Online], Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[3] Statista.com, " Number of Available Apps in the Google Play Store from 2nd Quarter 2015 to 2nd Quarter 2024," [Online], Available: https://www.statista.com/statistics/289418/number-of-available-apps-in-the-google-play-store-quarter/.

[4] Businessofapps, "App Downloads Data (2025)," https://www.businessofapps.com/data/app-statistics/.

[5] J. M. Arif, M. F. Ab Razak et al., "A. Android Mobile Malware Detection Using Fuzzy AHP," Journal of Information Security and Applications, vol. 61, p. 102929, 2021.

[6] Securelist.com, "IT Threat Evolution in Q2 2024-Mobile Statistics," [Online], Available: https://securelist.com/it-threat-evolution-q2-2024-mobile-statistics/113678/.

[7] Ö. A. Aslan and R. Samet, "A Comprehensive Review on Malware Detection Approaches," IEEE Access, vol. 8, pp. 6249-6271, 2020.

[8] S. Aurangzeb and M. Aleem, "Evaluation and Classification of Obfuscated Android Malware through Deep Learning Using Ensemble Voting Mechanism," Scientific Reports, vol. 13, p. 3093, 2023.

[9] V. Rastogi et al., "Catch Me If You Can: Evaluating Android Anti-malware against Transformation Attacks," IEEE Trans on Information Forensics and Security, vol. 9, no. 1, pp. 99–108, 2013.

[10] P. Kotzias, J. Caballero and L. Bilge, "How Did that Get in My Phone? Unwanted App Distribution on Android Devices," Proc. of 2021 IEEE Symposium on Security and Privacy (SP2001), pp. 53-69, 2021.

[11] M. M. Alani, "Android Users Privacy Awareness Survey," Int. Journal of Interactive Mobile Technologies, vol. 11, no. 3, 2017.

[12] A. Wajahat et al., "Outsmarting Android Malware with Cutting-edge Feature Engineering and Machine Learning Techniques," Computers, Materials & Continua, vol. 79, no. 1, pp. 651, 2024.

[13] W. Wang, M. Zhao and J. Wang, "Effective Android Malware Detection with a Hybrid Model Based on Deep Autoencoder and Convolutional Neural Network," Journal of Ambient Intelligence and Humanized Computing, vol. 10, no. 8, pp.3035-3043, 2019.

[14] S. I. Imtiaz et al., "DeepAMD: Detection and Identification of Android Malware Using High-efficient Deep Artificial Neural Network," Future Generation Computer Systems, vol. 115, pp. 844-856, 2021.

[15] M. K. Alzaylaee, Y. Y. Suleiman and S, Sakir, "DL-Droid: Deep Learning-based Android Malware Detection Using Real Devices," Computers & Security, vol. 89, p.101663, 2020.

[16] A. Dahiya, S. Sukhdip and S. Gulshan, "Android Malware Analysis and Detection: A Systematic Review," Expert Systems, vol.42, no. 1, p. e13488, 2025.

[17] G. D'Angelo et al., "Privacy-preserving Malware Detection in Android-based IoT Devices through Federated Markov Chains," Future Generation Computer Systems, vol. 148, pp. 93-105, 2023.

[18] S. Kumar, A. Prachi and J. Sahni, "IOT Malware Detection Using Static and Dynamic Analysis Techniques: A Systematic Literature Review," Security and Privacy, vol. 7, no. 6, p. e444, 2024.

[19] S. Sehrawat and D. D. Singh, "Malware and Malware Detection Techniques: A Survey, " Int. Journal for Research in Applied Science and Engineering Technology, vol. 10, no. 5, pp. 3947–3953, 2022.

[20] B. McMahan, E. Moorem et al., "Communication-efficient Learning of Deep Networks from Decentralized Data," Artificial Intelligence and Statistics (PMLR), vol. 54, pp. 1273-1282, 2017.

[21] A. Pathak et al., "Static Analysis Framework for Permission-based Dataset Generation and Android Malware Detection Using Machine Learning," EURASIP J. on Infor. Sec., vol. 2024, Art. no. 33, 2024.

[22] D. Soi et al., "Enhancing Android Malware Detection Explainability through Function Call Graph APIs," Journal of Information Security and Applications, vol. 80, p. 103691, 2024.

[23] M. Vu Minh et al., "A Static Method for Detecting Android Malware Based on Directed API Call," Int. Journal of Web Information Systems, vol. 21, no. 3, pp. 183-204, 2025.

[24] P. Sivaprakash et al., "Autonomous Android Malware Detection System Based on Static Analysis," Proc. of the 2024 IEEE Int. Conf. on Integration of Emerging Technologies for the Digital World (ICIETDW), pp. 1-6, DOI: 10.1109/ICIETDW61607.2024.10939283, 2024.

[25] W. Zhao, J. Wu and Z. Meng, "Apppoet: Large Language Model-based Android Malware Detection *via* Multi-view Prompt Engineering, "Expert Systems with Applications, vol. 262, p. 125546, 2025.

[26] H. Wintolo et al., "Visualization of Malware on Android Applications Using Static Analysis," Proc. of the 2024 IEEE Int. Conf. of Adisutjipto on Aerospace Electrical Engineering and Informatics (ICAAEEI), pp. 1-5, Yogyakarta, Indonesia, 2024.

[27] M. A. Haq and M, Khuthaylah, "Leveraging Machine Learning for Android Malware Analysis: Insights from Static and Dynamic Techniques," Engineering, Technology & Applied Science Research, vol. 14, no. 4 pp. 15027–15032, 2024.

[28] S. Fallah and A. J. Bidgoly, "Benchmarking Machine Learning Algorithms for Android Malware Detection," Jordanian J. of Computers and Inform. Tech. (JJCIT), vol. 5, no. 3, pp. 216-230, 2019.

[29] A. Boudrega, S. Benzouaoua, P. Ea, O. Salem and A. Mehaoua, "Conception of an Autonomous Dynamic Analysis System for Android Malwares," Proc. of the 2024 IEEE Asian Conf. on Communication and Networks (ASIANComNet), pp. 1-6, Bangkok, Thailand, 2024.

[30] G. Sathyadevi, J. Abishek and B. S. Shakthieiswaran, "DynaShield: Android Malware Detection Using Dynamic Analysis of Network Traffic," Proc. of the 2024 IEEE Int. Conf. on System, Computation, Automation and Networking (ICSCAN), pp. 1-6, Puducherry, India, 2024.
[31] H. Zhu et al., "A Dynamic Analysis-powered Explanation Framework for Malware Detection," IEEE Trans. on Knowledge and Data Engineering, vol. 36, no. 12, pp. 7483-7496, 2024.
[32] N. Prathapaneni et al., "Dynamic Behaviour Analysis and Interpretation of Malware in Android Devices Using Ensemble Machine Learning," Proc. of the 2024 3rd IEEE Int. Conf. on Artificial Intelligence for Internet of Things (AIIoT), pp. 1-6, Vellore, India, 2024.
[33] G. Ciaramella, F. Mercaldo and A. Santone, "Dynamic Analysis for Explainable Fine-grained Android Malware Detection," Proc. of the Int. Workshop on Security and Trust Management, pp. 110-127, Part of the Book Series: Lecture Notes in Computer Science, vol. 15235, Springer, 2024.
[34] S. Lee et al., "Hybrid Dynamic Analysis for Android Malware Protected by Anti-analysis Techniques with DOOLDA," Journal of Internet Technology, vol. 25, no. 2, pp.195-213, 2024.
[35] A. R. Nasser, A. M. Hasan and A. J. Humaidi, "DL-AMDet: Deep Learning-based Malware Detector for Android," Intelligent Systems with Applications, vol. 21, p. 200318, 2024.
[36] J. Feng et al., "HGDetector: A Hybrid Android Malware Detection Method Using Network Traffic and Function Call Graph," Alexandria Engineering Journal, vol. 114, pp. 30-45, 2025.
[37] S. Zhang et al., "MPDroid: A Multimodal Pre-training Android Malware Detection Method with Static and Dynamic Features," Computers & Security, vol. 150, p. 104262, 2025.
[38] A, Mesbah, I. Baddari and M. A. Riahla, "LongCGDroid: Android Malware Detection through Longitudinal Study for Machine Learning and Deep Learning," Jordanian Journal of Computers and Information Technology (JJCIT), vol. 9, no. 4, pp. 328-346, Dec. 2023.
[39] F. Mercaldo, F. Martinelli and A. Santone, "Deep Convolutional Generative Adversarial Networks in Image-based Android Malware Detection," Computers, vol. 13, no. 6, p. 154, 2024.
[40] Bhooshan, Prashant and Nidhi Sonkar, "Comprehensive Android Malware Detection: Leveraging Machine Learning and Sandboxing Techniques through Static and Dynamic Analysis," Proc. of the 2024 IEEE 21st Int. Conf. on Mobile Ad-Hoc and Smart Systems (MASS), pp. 580-585, Seoul, Korea, 2024.
[41] A. M. AlSobeh et al., "Android Malware Detection Using Time-aware Machine Learning Approach," Cluster Computing, vol. 27, pp. 12627–12648, 2024.
[42] F. M. M. Aledam et al., "Enhanced Malware Detection for Mobile Operating Systems Using Machine Learning and Dynamic Analysis," Int. J. of Safety & Security Eng., vol. 14, no. 2, pp. 513-521, 2024.
[43] M. Waheed and S. Qadir, "Effective and Efficient Android Malware Detection and Category Classification Using the Enhanced KronoDroid Dataset," Security and Communication Networks, vol. 2024, Article ID 7382302, 2024.
[44] R.H. Hsu, Y.C. Wang et al., "A Privacy-preserving Federated Learning System for Android Malware Detection Based on Edge Computing," Proc. of the 2020 15th IEEE Asia Joint Conf. on Information Security (AsiaJCIS), pp. 128-136, Taipei, Taiwan, 2020.
[45] A. Mahindru and H. Arora, " Dnndroid: Android Malware Detection Framework Based on Federated Learning and Edge Computing," Proc. of the Int. Conf. on Advancements in Smart Computing and Information Security, Cham: Springer Nature Switzerland, vol. 17, no. 12, pp. 96-107, 2020.
[46] R. Tageri et al., "FED-IIoT: A Robust Federated Malware Detection Architecture in Industrial IoT," IEEE Transactions on Industrial Informatics, vol. 17, no. 12, pp. 8442-8452, 2020.
[47] Z. Çıplak et al. "FEDetect: A Federated Learning-based Malware Detection and Classification Using Deep Neural Network Algorithms," Arab. J. for Sci. and Eng., DOI: 10.1007/s13369-025-10043-x, 2025.
[48] M. Robnik-Šikonja and I. Kononenko, "Theoretical and Empirical Analysis of ReliefF and RReliefF," Machine learning, vol. 53, pp. 23-69, 2003.
[49] F. Wei, Y. Li, S. Roy, X. Ou and W. Zhou, "Deep Ground Truth Analysis of Current Android Malware," Proc. of the 14th Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2017), vol. 14, pp. 252-276, 2017.
[50] S.Y. Yerima and S. Sezer, "Droidfusion: A Novel Multilevel Classifier Fusion Approach for Android Malware Detection," IEEE Transactions on Cybernetics, vol. 49, no. 2, pp. 453-466, 2018.
[51] F. Fleuret, "Fast Binary Feature Selection with Conditional Mutual Information," Journal of Machine Learning Research, vol. 5, pp. 1531-1555, 2004.
[52] H. Yang and J. Moody, "Data Visualization and Feature Selection: New Algorithms for Non-Gaussian Data," Advances in Neural Information Processing Systems, vol. 12, pp. 687-693, 1999.
[53] X. Gu et al., "A Feature Selection Algorithm Based on Equal Interval Division and Conditional Mutual Information," Neural Processing Letters, vol. 54, no. 3, pp. 2079-2105, 2022.
[54] G. Brown et al., "Conditional Likelihood Maximisation: A Unifying Framework for Information Theoretic Feature Selection," Journal of Machine Learning Research, vol. 13, no. 1, pp. 27-66, 2012.
[55] R. Gray, "Vector Quantization, " IEEE ASSP Magazine, vol. 1, no. 2, pp. 4–29, 1984.
[56] H. J´egou, M. Douze and C. Schmid, "Product Quantization for Nearest Neighbor Search," IEEE TPAMI, vol. 33, no. 1, pp. 117–128, 2011.

[57] Y. Matsui et al., "A Survey of Product Quantization," ITE Transactions on Media Technology and Applications, vol. 6, no. 1, pp. 2-10, 2018.

[58] N. Hollmann et al., "Accurate Predictions on Small Data with a Tabular Foundation Model," Nature, vol. 637, no. 8045, pp. 319-326, 2025.

[59] A. Vaswani et al., "Attention Is All You Need," Neural Information Processing Systems, vol. 30, 2017.

[60] J. Pearl, Causality, 2nd Edn., ISBN 0-521-77362-8, Cambridge University Press, USA, 2009.

[61] S. Müller et al., "PFNs4BO: In-context Learning for Bayesian Optimization," Proc. of the Int. Conf. on Machine Learning (PMLR), pp. 25444-25470, 2023.

[62] T. Brown et al., "Language Models Are Few-shot Learners," Advances in Neural Information Processing Systems, vol. 33, pp. 1877–1901, Honolulu, USA, 2020.

[63] T. Fawcett, "ROC Graphs: Notes and Practical Considerations for Researchers, " Machine Learning, vol. 31, no. 1, pp. 1-38, 2004.

[64] A. Wajahat et al., "Outsmarting Android Malware with Cutting-edge Feature Engineering and Machine Learning Techniques," Computers, Materials & Continua, vol. 79, no. 1, pp. 651-673, 2024.

[65] A. Wajahat et al., "An Effective Deep Learning Scheme for Android Malware Detection Leveraging Performance Metrics and Computational Resources," Intell. Deci. Tech., vol. 18, no. 1, pp. 33-55, 2024.

[66] A. Wajahat et al., "Securing Android IoT Devices with GuardDroid Transparent and Lightweight Malware Detection," Ain Shams Engineering Journal, vol. 15, no. 5, p. 102642, 2024.

[67] T. Kacem and S. Tossou, "Trandroid: An Android Mobile Threat Detection System Using Transformer Neural Networks," Electronics, vol. 14, no. 6, p. 1230, 2025.

[68] H. Shah et al., "A Comparative Analysis for Android Malware Detection Using Machine Learning Models," Proc. of the 2025 6th IEEE Int. Conf. on Mobile Computing and Sustainable Informatics (ICMCSI), pp. 1040-1047, Goathgaun, Nepal, 2025.

[69] N. G. Ambekar, S. Thokchom and S. Moulik, "TC-AMD: Android Malware Detection through Transformer-CNN Hybrid Architecture," Proc. of the 2024 IEEE Int. Conf. on Advanced Networks and Telecommunication Systems (ANTS), pp. 1-6, Guwahati, India, 2024.

[70] T. Bhandari et al., "Unveiling Machine Learning Paradigms for Robust Malware Detection in Personal Data Security," Proc. of the 2024 6th IEEE Int. Conf. on Computational Intelligence and Communication Technologies (CCICT), pp. 226-231, Sonepat, India, 2024.

[71] J. Li and H. Liu, "Challenges of Feature Selection for Big Data Analytics," IEEE Intelligent Systems, vol. 32, no. 2, pp. 9-15, 2017.

[72] Y. Matsui, "Nano Product Quantization," [online], Available: https://github.com/matsui528/nanopq.

[73] Github.com, "TabPFN: Foundation Model for Tabular Data," [Online], Available: https://github.com/PriorLabs/TabPFN?tab=readme-ov-file.

## ملخص البحث:

يُعد نظام التّشغيل أندرويد في طليعة أنظمة التّشغيل حول العالم لأنّه يعتمد على بيئة تقوم على المصادر المفتوحة عبر الانشطة المختلفة مثل الخدمات المصرفية والاتصالات والترفيه والتعليم والرعاية الصّحية؛ لذلك فإنّه هدف أساسي وأرضية جاذبة للتّهديدات السّيبرانية.

في هذه الورقة، نقترح نظاماً مبتكراً للكشف عن برامج أندرويد الضّارّة يُسمّى (تابْ-دْرويْد) ويعتمد على تقنيات انتقاء السِّمات وضغطها وتصنيفها الّتي تُطبّق على مجموعات بياناتٍ في العالم الحقيقي. ويُعدّ النّظام المقترح إطار عملٍ ملائماً للكشف عن برامج أندرويد الضّارّة؛ فقد جرى تقييمه بواسطة عددٍ من مؤشرات الأداء، وبرهن على تفوُّقٍ واضحٍ لدى مقارنته بعددٍ من أُطُر العمل المنافسة، حيث حقّق دقّةً وصلت إلى 99.2%، إلى جانب تفوّقه في مؤشّرات الأداء الأخرى. والجدير بالذكر أنّ إطار العمل المقترح أثبت أنّه ذو فاعلية كبيرة في الكشف عن برامج أندرويد الضّارّة، ويعمل على خفْض الحيّز الخاصّ بالسِّمات بنسبة 73%، الأمر الذي يدلّ على حُسن استغلاله للمصادر؛ فقد خفّض زمن الفحص بنسبة 44.4% والذّاكرة المستخدمة بنسبة 42.8%.