

Article

Cost-Aware Android Malware Detection Using an Early-Warning Behaviour Score

Ali Fenjan ^{1,2,*}, Mohammed Almulla ²  and Jalil Md. Desa ¹

¹ Asia Pacific University of Technology & Innovation (APU), Technology Park Malaysia, Bukit Jalil, Kuala Lumpur 57000, Malaysia; jalil.desa@apu.edu.my

² School of Engineering and Computing, American International University, Saad Al Abdullah, Jahra 91103, Kuwait; m.almulla@aiu.edu.kw

* Correspondence: a.fenjan@aiu.edu.kw

Abstract

Android malware detection systems commonly emphasize predictive accuracy while paying less attention to feature-acquisition cost, deployment efficiency, and early decision making. This paper presents a staged model-input budget framework for cost-aware Android malware screening and evaluates classification performance under progressively expanded static feature representations. The proposed framework uses a lightweight Behaviour Score as an early-warning model input derived from multiple static behavioural indicators, including permission risk, encryption evidence, network activity, and suspicious keyword evidence. Rather than treating the score as a cost-free feature, the framework distinguishes between the derived model input and the underlying static indicators required to construct it. Uncertain samples are progressively escalated from the early-warning stage to richer feature budgets using a confidence-based decision rule, while confident samples can be resolved before full-feature analysis. The experimental evaluation reports hyperparameter-tuned model performance, empirical inference-time profiling, confidence-based escalation behaviour, Matthews correlation coefficient, false positive rate analysis, low false-positive-rate operating points, cross-validation, statistical testing, and external proxy-budget validation using the Drebin benchmark. On the main Android application dataset, the Behaviour-Score stage achieved an F1-score of 0.8750. When low-cost static indicators were added, the framework achieved an F1-score of 0.9654 and a Matthews correlation coefficient of 0.9267. The full feature set achieved the highest F1-score of 0.9878 and Matthews correlation coefficient of 0.9741. The confidence-based escalation experiment showed that, at a predefined 0.95 confidence operating point, 90.32% of samples were resolved before full-feature analysis, reducing the average number of classifier model inputs used from 9 to 3.50 while maintaining an F1-score of 0.9785. These findings indicate that the proposed framework provides an incremental model-input budget approach for deployment-oriented Android malware screening, while preserving full analysis for uncertain samples.



Academic Editor: Paolo Bellavista

Received: 15 June 2026

Revised: 22 July 2026

Accepted: 28 July 2026

Published: 5 August 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: android malware detection; behavior score; budgeted feature acquisition; cost-aware classification; early-warning screening; false positive rate analysis; lightweight static analysis; malware classification; resource-constrained inference; staged detection

1. Introduction

Rapid growth of the Android ecosystem has significantly increased the exposure of mobile devices to security threats, making Android malware detection a critical research

challenge. Owing to its open-source nature and widespread adoption, Android has become a primary target for malicious applications that exploit system vulnerabilities and user trust. Recent studies have highlighted the increasing sophistication of Android malware, which frequently employs techniques such as code obfuscation, encryption, and dynamic payload execution to evade traditional detection mechanisms [1]. Machine-learning (ML)-based approaches have emerged as effective solutions for detecting both known and previously unseen malware by learning patterns from large-scale datasets [2].

Static analysis techniques that rely on features such as permissions, API calls, and application metadata have been widely adopted owing to their efficiency and scalability [3]. However, static approaches often suffer from reduced effectiveness when faced with advanced obfuscation techniques and feature redundancy [4]. In contrast, dynamic analysis provides deeper behavioral insights but incurs high computational overhead, limiting its applicability in real-time systems [5]. To address these limitations, recent research has explored hybrid and lightweight detection frameworks that aim to balance detection accuracy with computational efficiency [6]. Lightweight models are particularly important for deployment in resource-constrained environments such as mobile and edge devices, where low latency and reduced resource consumption are essential [7]. Despite these advancements, many existing solutions continue to rely on large and complex feature sets, resulting in increased processing times and reduced practicality for real-world deployment [8]. Another important research direction is the incorporation of explainability into malware-detection models. Explainable Artificial Intelligence (XAI) techniques, including SHAP and feature-importance methods, have been widely used to improve transparency and trust in model decisions [9]. However, most existing explainability approaches operate as post hoc analyses and do not directly contribute to reducing feature complexity or improving computational efficiency [10]. As a result, a gap remains between interpretability and deployability, where models are explainable but still computationally demanding.

In practical malware detection systems, feature extraction is not cost-free. Different features require varying levels of computational effort, and delaying detection to extract expensive features can negatively impact system performance, latency, and energy consumption, particularly during early-stage decision making. To address this limitation, this study proposes a cost-aware Android malware detection framework that evaluates detection performance under progressively increasing feature-acquisition budgets. Within this framework, a Behaviour Score (BS) is introduced as a lightweight behavioral aggregation that captures indicative application behavior using static indicators. Rather than serving as a standalone detection mechanism, the Behaviour Score functions as an early-stage signal within a broader cost-aware pipeline, enabling rapid initial screening before more computationally expensive features are considered. The proposed framework operates across multiple budget levels, beginning with minimal-cost detection using the Behaviour Score alone, followed by an intermediate configuration that incorporates low-cost static features, and finally a full-feature baseline. This design enables a systematic analysis of the trade-off between detection accuracy and computational cost, which is essential for real-time detection systems and large-scale deployment scenarios [11].

The originality of this study does not lie in proposing a fundamentally new malware classifier, nor in claiming that staged decision making itself is novel. Cascade-style screening, budgeted feature use, early-exit inference, and cost-sensitive feature selection are well-established concepts in machine learning and malware-detection research. Instead, the contribution of this work lies in adapting and empirically evaluating these principles within a compact static Android malware-screening framework centered on an interpretable Behaviour Score and progressively expanded model-input budgets. In this paper, cost-aware refers specifically to the staged use of classifier inputs and reduced reliance on the complete

static feature representation rather than to comprehensive measurements of APK feature-extraction energy, memory consumption, or physical-device resource usage. Consequently, the proposed framework should be interpreted as a staged model-input budget framework for cost-aware Android malware screening rather than as a new classifier architecture or a complete deployment-cost benchmark.

This distinction is important because the Behaviour Score is not treated as a cost-free raw feature. Instead, it is a derived model input constructed from multiple low-cost static behavioral indicators, including permission-risk evidence, encryption-related evidence, network-activity evidence, and suspicious-keyword evidence. The confidence-based escalation analysis therefore measures the number of model inputs used by the classifier at each stage while explicitly recognizing that derived inputs require multiple underlying static indicators for their construction. The paper evaluates whether such a compact derived input can effectively support early malware screening and whether uncertain samples benefit from escalation to progressively richer static feature budgets.

This positioning separates the established methodological foundation—namely staged and budgeted decision making—from the specific contributions of this study, including an interpretable Android static-analysis framework, the definition of the Behaviour Score, confidence-based escalation, low-false-positive evaluation, statistical robustness assessment, and external proxy-budget validation under a unified experimental protocol.

In the remainder of this paper, the Behaviour Score is denoted as BS, Matthew's correlation coefficient as MCC, false positive rate as FPR, receiver operating characteristic area as ROC-AUC, and precision–recall area as PR-AUC. Extensive experiments were conducted using multiple machine-learning models, including Logistic Regression, Random Forest, Extra Trees, and XGBoost. The evaluation framework includes hyperparameter-tuned classification, classifier-level inference-time profiling, confidence-based escalation analysis, low-false-positive-rate security evaluation, cross-validation, paired statistical testing, and external validation using the Drebin benchmark [12,13]. The main contributions of this study are summarized as follows:

1. A staged model-input budget framework is presented for Android malware screening by adapting established concepts from staged decision making, early-exit inference, and budgeted feature use to a compact static Android malware-detection setting.
2. An interpretable Behaviour Score is introduced as a derived model input that aggregates selected static behavioral indicators. The Behaviour Score is not presented as a standalone malware detector or a cost-free raw feature but rather as an early-stage abstraction within a broader staged screening pipeline.
3. A confidence-based escalation strategy is evaluated across progressively richer static feature budgets, reporting the percentage of samples resolved before full-feature analysis, the average number of acquired model inputs, model-input reduction, and security-oriented performance.
4. A comprehensive security-oriented evaluation is performed using F1-score, Matthew's correlation coefficient, false positive rate, low-false-positive operating points, cross-validation, statistical testing, and external proxy-budget validation using the Drebin benchmark.
5. A clear distinction is established between model-input budget reduction and complete feature-acquisition cost. Classifier inference time and model-input count are reported as operational indicators, whereas full APK feature-extraction latency, CPU usage, memory consumption, and energy profiling are identified as future deployment-level validation requirements.

6. The study strengthens reproducibility by documenting dataset construction, static feature-extraction logic, leakage-control procedures, duplicate handling, group-aware partitioning, random seeds, and the separation between training, validation, and final test evaluation.

2. Related Work

Recent advances in Android malware detection have explored static, dynamic, hybrid, explainable, lightweight, and deployment-aware approaches. Static analysis remains attractive because permissions, API calls, manifest attributes, and application metadata can be extracted without executing malware, making static models suitable for large-scale screening and resource-constrained settings [14,15]. However, static analysis can be weakened by obfuscation, repackaging, dynamic payload loading, and feature redundancy [16]. Dynamic analysis provides stronger runtime evidence, including system calls, network activity, and execution traces, but it requires controlled execution environments and introduces additional latency and infrastructure cost [17]. Hybrid approaches combine static and dynamic evidence and can improve accuracy, but they often increase feature-extraction complexity and may be less suitable for early-stage screening when rapid decisions are required [18,19].

A second line of research has focused on feature reduction, lightweight detection, and efficiency-oriented malware screening. Cost-sensitive learning and cost-effective feature selection demonstrate that classification performance should be considered together with feature utilization and computational constraints [20]. Android-specific feature-selection methods, such as FSDroid, show that reducing the feature space can preserve detection effectiveness while improving efficiency [21]. Other lightweight and fast multi-feature Android malware-detection frameworks likewise support the need for practical detection pipelines that reduce processing overhead [22,23]. These studies demonstrate that extracting the complete feature set is not always necessary. However, many of them evaluate a fixed reduced feature representation rather than a progressive staged process in which samples can be classified early or escalated only when prediction confidence is insufficient.

Explainable Artificial Intelligence (XAI) has also become increasingly important in Android malware detection because security analysts require transparent and interpretable model decisions. SHAP and other feature-importance techniques provide useful post hoc explanations for why an application is classified as malicious [24,25]. Behavioral modeling approaches similarly attempt to represent application activity using API sequences, system-call patterns, or probabilistic behavior profiles [26,27]. Although these methods improve interpretability and behavioral insight, they do not necessarily reduce the number of model inputs required during inference or explicitly support early-warning decision making. In contrast, the present study introduces the Behaviour Score as an interpretable derived model input within a staged screening framework. The Behaviour Score is not presented as a new classifier but as a compact behavioral abstraction that enables early-stage decisions before uncertain samples are escalated to richer static feature budgets [28].

Recent studies further confirm the growing emphasis on lightweight, scalable, and deployment-aware Android malware detection. Permission-level static indicators continue to play an important role in interpretable malware screening [29], while feature-influence-based approaches for resource-constrained devices reinforce the importance of reducing computational overhead without sacrificing detection performance [30]. Three-level feature-selection research likewise supports structured feature reduction as a means of improving deployment feasibility [31]. Runtime behavioral subsequence methods provide compact behavioral evidence but still require execution monitoring, resulting in additional

runtime overhead [32]. Dataset-focused studies such as Maloid-DS have contributed valuable benchmark datasets for contemporary Android malware research, while lightweight Markov-image representations, knowledge-distillation techniques, and large language model (LLM)-based approaches demonstrate the continuing diversification of Android malware detection methods and feature representations [33–36].

Collectively, these studies demonstrate significant progress in Android malware detection but also reveal an important research gap. Prior work has investigated static detection, dynamic analysis, feature selection, lightweight modeling, explainability, and dataset construction. However, comparatively few studies jointly evaluate an interpretable early-warning model input, staged model-input budgets, confidence-based escalation, low-false-positive operating behavior, statistical robustness, and external proxy-budget validation within a unified experimental framework. The present study addresses this gap by adapting established concepts of staged and budget-aware decision making to a compact static Android malware-screening framework. Its contribution is therefore not the introduction of cascade classification, early-exit inference, or budgeted learning, but rather the empirical evaluation of a Behaviour-Score-based Android malware detection framework that explicitly reports model-input budgets, confidence-guided escalation behavior, and security-oriented evaluation metrics.

As summarized in Table 1, the closest prior studies provide important but partial coverage of the problem addressed in this paper. Feature-selection studies show that reducing dimensionality can preserve malware-detection performance, but they generally stop at selecting a compact fixed feature subset rather than evaluating confidence-based escalation across multiple feature budgets [10,21,31]. Lightweight and explainable Android malware studies improve interpretability and reduce model complexity, but their explanation mechanisms are usually applied after classification and are not directly integrated into a staged early-warning pipeline [12,24,25]. Permission-based and behavioral approaches provide interpretable security evidence, but they do not fully examine how such evidence can be used as a derived early-stage model input before escalating uncertain samples to richer static features [29,32]. Dataset-construction and LLM-based studies broaden the available experimental resources and semantic modeling strategies, but they are not designed around model-input budget reduction, low-false-positive staged operation, or confidence-based early decision-making [33,35,36]. Therefore, the present study occupies a narrower but practically relevant position. It does not claim that staged decision-making, feature-budgeting, or early-exit logic is new. Instead, it evaluates how these ideas can be instantiated in a compact static Android malware-screening framework using a Behaviour Score, progressively richer model-input budgets, and confidence-based escalation. This analytical positioning clarifies both what is known from prior work and what is added by the present study.

Table 1. Analytical comparison of representative Android malware detection studies related to lightweight static analysis, explainable detection, feature selection, and efficiency-aware malware screening.

Ref.	Study Focus	Main Features/Representation	Method	Relevance to This Work	Remaining Gap
[10]	Feature selection for Android malware detection	Static Android features	Reinforcement-learning-based feature selection	Demonstrates that reducing the feature space can improve the efficiency of Android malware detection	Does not evaluate staged early-exit detection or confidence-based escalation

Table 1. Cont.

Ref.	Study Focus	Main Features/Representation	Method	Relevance to This Work	Remaining Gap
[12]	Lightweight and explainable Android malware detection	Static Android malware features supported by local explanations	Lightweight transformer with explainability	Supports the need for interpretable and lightweight Android malware detection models	Explainability is not directly linked to feature-budget escalation or staged decision-making
[23]	Fast multi-feature Android malware detection	Permissions and Dalvik opcode sequences	Fast multi-feature Android malware detection framework	Relevant to rapid detection and reduced feature complexity	Does not explicitly model progressive feature acquisition or confidence-based early decisions
[28]	Efficient and explainable Android malware detection	Drebin static features	Linear SVM with explainable feature representation	Provides a widely used public benchmark context for Android malware detection	Does not evaluate proxy feature budgets, staged escalation, or early-exit screening
[29]	Permission-based Android malware detection	Android permissions mapped to exploitation-related indicators	Scalable permission-to-exploitation mapping framework	Confirms the continued importance of permission-level indicators for lightweight Android malware screening	Focuses mainly on permission-oriented detection rather than adaptive staged feature-budget escalation
[30]	Efficient malware detection for resource-constrained devices	Feature-influence-based malware indicators	Machine learning with feature influence analysis	Supports the importance of reducing feature-processing overhead in constrained environments	Does not explicitly evaluate sample-level confidence-based escalation across multiple feature budgets
[31]	Feature selection for Android malware detection	Selected Android malware feature groups	Three-level feature selection framework	Supports the value of structured feature reduction in Android malware detection	Applies feature selection rather than adaptive staged inference with early stopping
[32]	Behavioural Android malware detection	Informative system-call subsequences	System-call subsequence-based detection	Provides compact behavioural evidence for Android malware detection	Relies on runtime-style behavioural evidence and does not address low-cost static staged screening
[33]	Android malware forensics dataset construction	Labeled Android malware dataset	Dataset development and forensic labeling	Provides a recent labeled Android malware dataset relevant to experimental validation	Focuses on dataset contribution and does not propose a cost-aware staged detector
[34]	Lightweight Android malware detection	Markov-image representation	Knowledge-distillation-based lightweight framework	Supports the recent trend toward lightweight Android malware detection	Uses transformed image-style representations rather than interpretable feature-budget stages

Table 1. Cont.

Ref.	Study Focus	Main Features/Representation	Method	Relevance to This Work	Remaining Gap
[35]	LLM-based Android malware detection	Android application representations processed through LLM-based analysis	Large-language-model-based malware detection	Shows the recent movement toward semantic Android malware detection	Not designed around low-cost staged feature acquisition, early-warning scoring, or feature-budget reduction
[36]	Context-driven Android malware detection and classification	Contextual Android code and application evidence	LLM-based context-driven detection and classification	Demonstrates recent work using contextual reasoning for Android malware classification	Does not focus on lightweight static feature budgets, low-FPR staged operation, or confidence-based escalation

3. Methodology

This section presents the proposed cost-aware Android malware detection framework, which formulates detection as a progressive multistage decision process. As demonstrated in Table 2, the framework integrates structured feature grouping, a behavior-level aggregation mechanism, and an adaptive escalation strategy to balance detection performance, computational efficiency, and interpretability.

Table 2. Main notation and acronyms used in the proposed cost-aware Android malware detection framework. The table defines the symbols, abbreviations, and evaluation metrics used in the formulation of feature-budget design, behaviour score construction, confidence-based escalation, and security-oriented performance analysis.

Symbol/Acronym	Meaning
X	Feature vector extracted from an Android application
Y	Class label; 0 = benign and 1 = malware
$f(x)$	Classification function
BS	Behaviour Score
W_j	Weight of the j-th behavioural indicator
F_j	Normalized value of the j-th behavioural indicator
P	Predicted probability of the malware class
C	Confidence score of the classifier decision
T	Confidence threshold used for escalation
Fr	Number of model inputs used for sample r
Ffull	Number of model inputs in the full static configuration
MCC	Matthews correlation coefficient
FPR	False positive rate
ROC-AUC	Area under the receiver operating characteristic curve

3.1. Problem Formulation

Let $x \in \mathbb{R}^d$ denote the feature representation of the Android application. The objective of this study is to determine the classification function.

$$f(x) = y, y \in \{0, 1\} \quad (1)$$

where x is the feature vector extracted from an Android application, $y = 0$ represents a benign application, and $y = 1$ represents a malicious one.

3.2. Model-Input Budget Grouping

The static feature representation is organized into three progressively expanded model-input budgets. The purpose of this grouping is to evaluate how detection performance changes when the classifier is allowed to use a compact derived input first and additional static inputs only when needed. Therefore, the grouping should be interpreted as a model-input budget design rather than as a direct measurement of raw APK feature-acquisition cost. Budget-1 uses the Behaviour Score as a single derived model input. This stage provides an early-warning representation constructed from multiple static behavioural indicators. Budget-2 extends the first budget by adding selected low-cost static indicators, including permissions, file size, network requests, and the network flag. Budget-3 uses the full static input representation and serves as the complete-feature baseline. For a sample r , the model-input usage is represented as F_r , while F_{full} denotes the number of inputs in the full static configuration. The model-input reduction is computed as:

$$\text{Model} - \text{input reduction} = \left(1 - \frac{F_r}{F_{full}}\right) \times 100 \quad (2)$$

This formulation supports evaluation of staged classifier input usage. It does not claim that the reported reduction is equal to raw feature-extraction latency, CPU utilization, memory usage, or physical-device energy saving. Those deployment-level cost measurements are treated as future work.

3.3. Behavior Score Construction

The Behaviour Score (BS) is an engineered static meta-feature designed to aggregate selected behaviour-aligned indicators into a compact and interpretable derived model input. BS is not a classifier, a malware label, or a standalone decision rule. Instead, it is used as one input within the staged machine-learning pipeline, and its predictive contribution is learned by the classifier during training. Conceptually, BS is defined as a weighted aggregation of static behavioural indicators:

$$BS = \sum_{j=1}^m (w_j \cdot f_{j(x)}) \quad (3)$$

where $f_{j(x)}$ denotes the j th behavioural indicator extracted from Android application x , w_j denotes the corresponding component weight, and m is the number of behavioural indicators included in the score. In the implemented static APK extractor, BS is operationalized as:

$$BS = P + 5E + 3N + 2SK \quad (4)$$

where P is the number of dangerous permissions requested by the APK, E is a binary encryption-evidence indicator, N is the binary suspicious network-intent flag, and SK is the count of distinct suspicious keyword matches. The permission component P is computed by counting the number of requested permissions that belong to a predefined set

of dangerous permissions. The encryption component E is set to 1 when encryption-related evidence is detected together with suspicious supporting evidence, and 0 otherwise. The network component N corresponds to the suspicious network-intent flag, which is set to 1 when suspicious infrastructure tokens, suspicious endpoints, or corroborated network evidence are detected (See Table 3). The suspicious keyword component SK counts distinct malware-relevant lexical indicators in the reduced static corpus, with capping applied to prevent repeated strings or library artefacts from dominating the score.

Table 3. Operational components of the Behaviour Score.

Component	Symbol	Definition	Weight in BS	Role
Dangerous permissions	P	Count of requested permissions belonging to the predefined dangerous-permission set	1	Manifest-level permission-risk evidence
Encryption evidence	E	Binary indicator of encryption-related evidence with suspicious supporting evidence	5	Possible payload-protection or covert-communication evidence
Suspicious network intent	N	Binary flag indicating suspicious infrastructure tokens, suspicious endpoints, or corroborated network evidence	3	Network-communication risk evidence
Suspicious keywords	SK	Count of distinct malware-relevant lexical indicators in the reduced static corpus, with capping to reduce repeated-string dominance	2	Supportive lexical risk evidence
Final Behaviour Score	BS	Weighted aggregation of the above static behavioural indicators	$BS = P + 5E + 3N + 2SK$	Compact derived behavioural model input

The coefficients in Equation (4) are prior-informed heuristic weights selected to preserve interpretability and forensic auditability. They encode conservative assumptions about the relative evidential strength of each component: dangerous permissions are informative but common in benign applications, encryption evidence may indicate payload protection or covert communication, suspicious network intent reflects communication risk, and suspicious keywords provide supportive lexical evidence. These weights are not optimized on the test set, are not learned independently for each classifier, and are not used as a decision threshold. Therefore, BS should be interpreted as a transparent derived feature rather than as a tuned classifier.

To examine whether the Behaviour Score depends strongly on the selected integer coefficients, an additional weight-sensitivity analysis was performed. The proposed heuristic weighting scheme was compared with learned logistic-regression weighting, mutual-information-based weighting, and a randomized upper-bound weighting search. The purpose of this analysis was not to prove that the heuristic coefficients are uniquely optimal, but to determine whether the behavioural aggregation concept remains discriminative

when the weighting strategy is changed. A Random Forest classifier trained using BS only achieved ROC-AUC values between 0.933 and 0.954 across the evaluated weighting schemes. The proposed heuristic formulation achieved a ROC-AUC of 0.936, while the learned logistic-regression weighting achieved the highest ROC-AUC of 0.954. The mutual-information and randomized alternatives achieved ROC-AUC values of 0.933 and 0.947, respectively (See Table 4). The results shown in Figure 1 indicate that BS remains discriminative across alternative weighting strategies, supporting the robustness of the aggregation concept rather than the uniqueness of the heuristic coefficients.

Table 4. ROC-AUC comparison across Behaviour Score weighting schemes using BS-only Random Forest classification. The comparison shows that the Behaviour Score remains discriminative across heuristic, learned, mutual-information-based, and randomized weighting strategies, supporting the robustness of the behavioural aggregation concept.

Weighting Scheme	ROC-AUC	Accuracy	F1-Score
Heuristic BS	0.9364	0.9101	0.9113
Learned logistic-regression weighting	0.9535	0.9268	0.928
Mutual-information weighting	0.9331	0.8995	0.9004
Best randomized upper-bound weighting	0.9473	0.918	0.9186

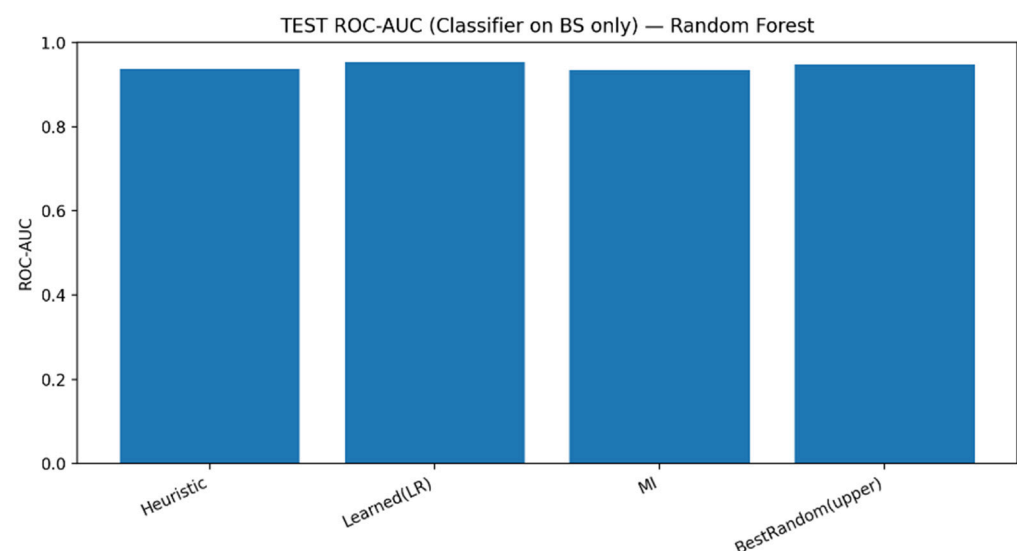


Figure 1. ROC-AUC comparison across different BS weighting schemes using Random Forest.

The fixed heuristic formulation is retained in the main staged framework because it is transparent, simple to reproduce, and semantically aligned with the intended feature-level abstraction. Although adaptive or dataset-specific weighting may improve predictive performance, it can reduce interpretability and may increase the risk of dataset-specific optimization. For this reason, learned and mutual-information-based alternatives are reported as sensitivity checks, while the interpretable heuristic BS definition is retained for the main framework. To reduce overfitting risk, BS is constructed using fixed static-extraction rules and is then treated as a regular input feature inside the training pipeline. Classifier parameters are learned only from the training data. No holdout-test labels are used to select BS weights, tune the BS formula, or choose the escalation threshold. This design separates feature construction from final model evaluation and prevents the Behaviour Score from being optimized directly on the final test set. Because BS is built from multiple underlying static indicators, it is counted as one derived model input in the staged classifier, not as one cost-free raw feature. Therefore, the escalation analysis reports model-

input reduction rather than complete raw feature-acquisition reduction. This distinction is important for interpreting the cost-aware scope of the paper: the study evaluates whether a compact derived behavioural input can support early screening, while full end-to-end APK feature-extraction cost remains a deployment-level evaluation requirement.

3.4. Multi-Stage Detection Framework

The proposed framework operates through a staged detection process in which model inputs are progressively expanded. Stage 1 performs early-warning classification using the Behaviour Score as one derived model input. If the classifier confidence is sufficient, the sample is resolved at this stage. Stage 2 is used when the Stage 1 confidence is insufficient and adds selected low-cost static inputs, including permissions, file size, network requests, and the network flag. Stage 3 is used for the remaining uncertain samples and applies the complete static input representation. This staged design allows confident samples to be classified using compact model-input budgets while preserving the full static representation for uncertain cases. Accordingly, the framework evaluates staged model-input usage rather than claiming that all underlying raw feature-acquisition operations are avoided.

3.5. Escalation Strategy

A confidence-based escalation mechanism governs the transition between stages. Let $P(y = 0 | x)$ and $P(y = 1 | x)$ denote the predicted class probabilities for the benign and malware classes, respectively. The confidence score $C(x)$ is defined as the maximum predicted class probability at the current stage. A sample is classified at the current stage when $C(x) \geq \tau$; otherwise, it is escalated to the next feature budget.

$$C(x) = \max(P(y = 0 | x), P(y = 1 | x)) \quad (5)$$

$$Escalate(x) = \begin{cases} 0, & C(x) \geq \tau \\ 1, & C(x) < \tau \end{cases} \quad (6)$$

where τ is the confidence threshold. $Escalate(x) = 0$ means that the current-stage prediction is accepted, whereas $Escalate(x) = 1$ means that the sample is passed to the next feature-budget stage.

In this study, the confidence thresholds used in the escalation analysis are treated as predefined operating points rather than thresholds optimized on the final test set. Specifically, τ values of 0.80, 0.85, 0.90, and 0.95 are used to examine how increasingly strict confidence requirements affect early-stage resolution, average model-input usage, model-input reduction, and security-oriented performance. Therefore, the escalation analysis should be interpreted as an operating-point sensitivity analysis, not as a claim that a globally optimal threshold was learned from the test data.

3.6. Model Training and Evaluation Protocol

The framework was evaluated using multiple machine learning models, including Logistic Regression, Random Forest, Extra Trees, and XGBoost, to ensure that the performance improvements were not model-specific. The model was trained and evaluated under predefined budget-wise feature configurations that emulated the staged feature acquisition during inference (See Figure 2). To prevent data leakage and ensure reliable evaluation, a group-based validation strategy was employed, in which samples from the same application were kept within the same partition. Model performance was assessed using accuracy, precision, recall, and F1-score, along with efficiency-related metrics such as training time and inference time under different feature-budget settings.

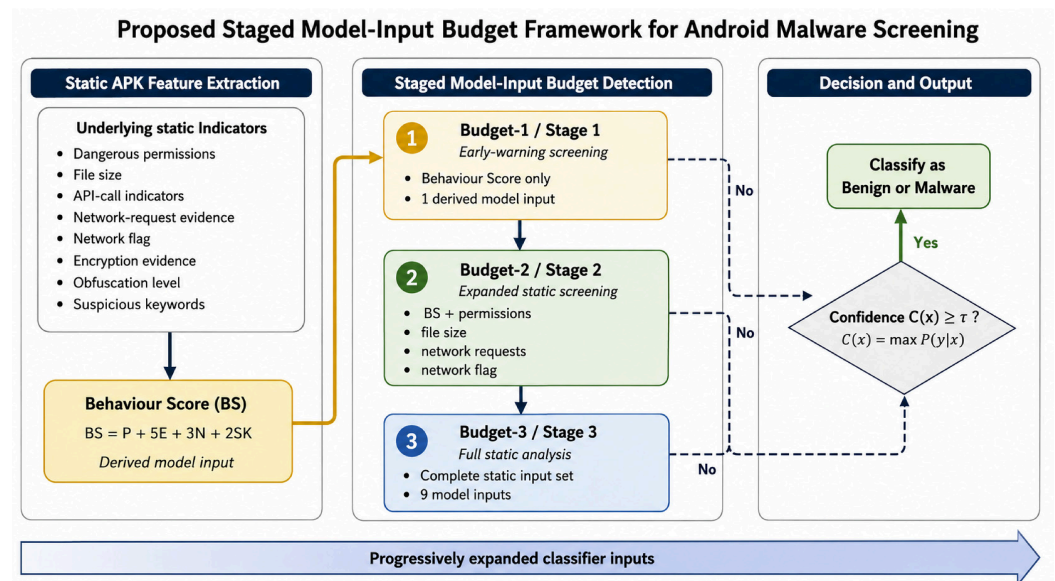


Figure 2. Proposed staged model-input budget framework for Android malware screening. The Behaviour Score (BS) is used as a derived early-warning input, followed by progressively richer static input budgets when confidence is insufficient. The figure represents classifier-input budgeting rather than full raw feature-extraction cost or device-level resource usage.

3.7. Classifiers and Hyperparameter Optimization

Four supervised learning models are evaluated to ensure that the observed staged performance trends are independent of a single classifier. Logistic Regression analysis was used as the linear baseline. Random Forest was used as a bagging-based ensemble that combined multiple decision trees trained on bootstrapped samples and random feature subsets. Extra Trees were included as a more randomized tree ensemble, in which split thresholds were selected randomly, often reducing variance and improving computational efficiency. XGBoost was evaluated as a gradient-boosting method that sequentially builds decision trees to minimize the regularized objective function. Hyperparameter optimization was performed using a randomized search with a cross-validation of the training split. The tuned parameters included the regularization strength for Logistic Regression, number of trees, maximum depth, minimum leaf size, feature sampling strategy for Random Forest and extra trees, number of estimators, maximum depth, learning rate, subsampling ratio, and column sampling ratio for XGBoost. The test set was not used during the hyperparameter tuning.

4. Results

This section presents an experimental evaluation of the proposed cost-aware Android malware-detection framework under progressively expanded feature budgets. The evaluation included dataset description, budget-wise tuned detection performance, empirical computational cost profiling, confidence-based escalation analysis, low-FPR security-oriented evaluation, cross-validation, statistical significance testing, and external validation using the Drebin benchmark. This multilevel evaluation assesses the detection performance, robustness, feature acquisition reduction, and practical deployment relevance.

4.1. Dataset Description and Experimental Setup

The primary evaluation was conducted using an author-constructed static Android APK dataset. The original corpus contained 4547 APK samples before duplicate-aware cleaning. Malware samples were collected from MalwareBazaar and the IEEE 2024 Maloid-DS labeled Android malware forensics dataset, while goodwill samples were collected

from the F-Droid open-source Android application repository using an automated downloader. These sources were selected to combine labeled malicious APKs with legitimate open-source Android applications. Malware APKs were handled inside an isolated virtualized environment and were not installed or executed. Goodware APKs were processed separately using the same static feature definitions, formulas, thresholds, and extraction logic to ensure consistency across classes. Duplicate-aware cleaning was applied before model training and evaluation. First, exact full-row duplicates were removed when all recorded fields were identical. Second, duplicate feature-label rows were removed using exact equality over the selected static feature vector and class label. This reduced the dataset from 4547 original samples to 3863 final samples. The cleaned corpus contained 2082 malware samples and 1781 goodware samples, corresponding to approximately 53.9% malware and 46.1% goodware. The dataset was therefore moderately imbalanced rather than severely imbalanced. No oversampling, undersampling, or class reweighting was applied, because the objective was to evaluate the extracted static-feature space under its natural post-cleaning distribution.

The cleaned dataset was divided using an 80/20 group-aware holdout strategy based on application identity. The development partition contained 3091 samples, including 1670 malware and 1421 goodware samples. The final holdout test partition contained 772 samples, including 412 malware and 360 goodware samples. The development partition was used for training, hyperparameter selection, cross-validation, and threshold-related development analysis, while the holdout test partition was reserved for final unseen evaluation. No holdout samples were used for preprocessing decisions, model selection, hyperparameter choice, feature-configuration selection, or Behaviour Score formula selection. This separation was used to reduce application-level leakage and to prevent final test information from influencing model development.

Each APK was processed using a custom Python-based static feature extraction pipeline built using Androguard version 4.1.4, which requires Python 3.4 up to Python 3.13. Before feature extraction, the APK files were checked for ZIP integrity and SHA-256 hashes were assigned for duplicate detection, traceability, and quality control. Successfully parsed APKs were processed to extract the requested permissions, sensitive API-call indicators, file size, encryption evidence, obfuscation level, network-request evidence, suspicious keywords, network flag, and derived behavior score. The extractor also generates a full debug output containing the parsing mode, ZIP status, extraction errors, suspicious endpoint counts, risky network-family indicators, method obfuscation statistics, INTERNET permission status, and SHA-256 values. APKs with corrupted ZIP structures, failed parsing, invalid labels, or incomplete critical fields were excluded before the model training. This process ensured that the final dataset was built from validated static APK artifacts with reproducible feature definitions and quality control evidence.

This dataset is author-constructed and is therefore not presented as a widely adopted public benchmark. Its role in this paper is to evaluate the proposed staged model-input budget framework under a controlled static-feature setting. To reduce overclaiming, the reported results are interpreted as evidence of performance on the cleaned and group-partitioned corpus rather than as proof of universal Android malware generalization. Broader validation on additional public datasets with matching feature schemas, complete malware-family labels, collection dates, and Android-version metadata remains an important direction for future work.

The model-ready dataset retains nine core static features and one label column. The selected features were permission requests, API calls, file size, encryption, obfuscation level, network requests, suspicious keywords, network flags, and behavior scores. MinMax-scaled versions of the numeric features were also generated for experimental consistency;

however, the original feature definitions were preserved to maintain interpretability. The full debug CSV was separately retained as an audit trail for the extraction quality, parser status, and feature provenance.

The main APK dataset was evaluated using a holdout test split, and five-fold cross-validation was performed to assess robustness across folds. Hyperparameter tuning was conducted only on the training data to avoid test-set leakage. Three budget configurations were used in the experimental design. Budget-1 used only the behavior score. Budget-2 uses a behavior score with low-cost static indicators including permissions, file size, network requests, and network flags. Budget-3 used a complete nine-feature representation. The models were trained using only the training partitions. The hyperparameters were optimized through cross-validation, and the final performance was reported for the holdout test partition. In addition to the primary dataset, the Drebin benchmark was used for the external validation. Because the primary dataset was newly constructed, additional care was taken to reduce leakage and improve reproducibility. Hash-based tracking was used during the extraction to identify repeated APK files, and parser/debug records were preserved to support the inspection of failed or incomplete samples. Holdout and cross-validation experiments were conducted only after cleaning the invalid and incomplete labels. This design reduces the likelihood of the reported performance being driven by extraction artifacts or invalid APK records. Because Drebin does not follow the same feature schema as the primary dataset, proxy feature budgets were created using mutual-information-selected feature subsets with 25 and 100 features, respectively, followed by the full 215-feature Drebin representation. Table 5 summarizes the feature-budget configurations used in the main APK dataset and the Drebin external-validation experiment. Because the Behaviour Score is a derived model input, the table distinguishes between the number of model inputs used by the classifier and the underlying static indicators required to construct those inputs. Therefore, Budget-1 is not interpreted as a cost-free single raw feature; rather, it represents one derived input constructed from multiple low-cost static behavioural indicators.

Table 5. Summary and provenance of the main Android application dataset used for the primary evaluation.

Item	Description
Dataset type	Author-constructed static Android APK dataset
Original APK samples before cleaning	4547
Final samples after duplicate-aware cleaning	3863
Final malware samples	2082
Final goodware samples	1781
Malware sources	MalwareBazaar and IEEE 2024 Maloid-DS labeled Android malware forensics dataset
Goodware source	F-Droid open-source Android application repository
Analysis type	Static APK analysis only
Malware execution	Not performed
Malware handling environment	Isolated virtualized environment
Core extracted features	Permissions Requested, API Calls, File Size, Encryption Used, Obfuscation Level, Network Requests, Suspicious Keywords, Network Flag, and Behaviour Score

Table 5. *Cont.*

Item	Description
Quality-control evidence	ZIP integrity status, SHA-256 hash, parsing mode, extraction errors, suspicious endpoint counts, risky network-family indicators, and method-obfuscation statistics
Duplicate cleaning	Full-row duplicate removal followed by exact feature-label duplicate removal
Final split strategy	80/20 group-aware holdout split based on application identity
Development partition	3091 samples: 1670 malware and 1421 goodware
Final holdout partition	772 samples: 412 malware and 360 goodware
Cross-validation	Five-fold cross-validation on the development partition only
Holdout leakage control	Holdout data were not used for preprocessing decisions, model selection, hyperparameter selection, feature-configuration selection, or Behaviour Score formula selection
Random seed policy	Fixed random seeds were used for data partitioning, model initialization, and repeated evaluation where applicable
Time-aware/family-aware limitation	Complete collection dates and malware-family labels were not uniformly available; therefore, time-aware and family-aware validation are treated as future work

4.2. Reproducibility and Leakage-Control Measures

Several measures were used to improve reproducibility and reduce leakage risk. First, APK-level SHA-256 hashes and parser/debug records were retained during feature extraction to support traceability, duplicate inspection, and quality control. Second, duplicate-aware cleaning was applied before any model training so that repeated records did not inflate downstream performance. Third, a group-aware 80/20 split based on application identity was used to reduce the chance that repeated instances of the same application appeared in both development and holdout partitions. Fourth, five-fold cross-validation was conducted only on the development partition, while the holdout test partition remained unseen until final evaluation. Fifth, random seeds were fixed for data splitting, model initialization, and repeated evaluation where applicable. The available metadata did not provide complete and uniform malware-family labels, collection timestamps, Android versions, or source-level temporal identifiers across all samples. Consequently, the present study does not claim family-aware or time-aware generalization. This limitation is explicitly acknowledged because family overlap, source bias, repeated application artifacts, and temporal drift can affect Android malware evaluation. The current evaluation should therefore be interpreted as duplicate-aware and group-aware, but not fully family-aware or time-aware. Future work will extend the framework using datasets with richer metadata to support family-based, source-aware, and time-aware splits. To support reproducibility, the feature matrix, split identifiers, random seeds, and non-executable metadata can be released where licensing and malware-handling restrictions allow. Raw malicious APK files are not redistributed directly because of safety, legal, and repository-use restrictions. Instead, reproducibility can be supported through SHA-256 hashes, static feature tables, configuration files, and documented extraction logic.

4.3. Budget-Wise Detection Performance

The primary experiment evaluated three model-input budget configurations. Budget-1 used one derived model input, the Behaviour Score, which was constructed from multiple low-cost static behavioural indicators. Budget-2 combined the Behaviour Score with

additional low-cost static indicators, including permissions, file size, network requests, and network flag. Budget-3 used the complete nine-input static feature representation. This design distinguishes between the number of inputs provided to the classifier and the underlying indicators required to construct derived inputs such as the Behaviour Score. For each configuration, Logistic Regression, Random Forest, Extra Trees, and XGBoost were independently evaluated after hyperparameter optimization of the training split. This setup provided a fair comparison across different learning paradigms while keeping the test set unseen during model selection. Table 6 lists the best hyperparameter-tuned models for each model-input budget configuration of the main APK dataset. The Behaviour-Score configuration achieved an F1-score of 0.8750, an MCC of 0.7488, and an ROC-AUC of 0.9386 using one derived model input. Because the Behaviour Score is constructed from multiple static behavioural indicators, this result is interpreted as the performance of a compact derived input rather than as the performance of one cost-free raw feature. When additional low-cost static inputs were added in Budget-2, the F1-score increased to 0.9654, the MCC increased to 0.9267, and the FPR decreased from 0.0932 to 0.0409. The full static input set achieved the strongest overall performance, with an F1-score of 0.9878, an MCC of 0.9741, an FPR of 0.0159, and an ROC-AUC of 0.9985. The results shown in Figure 3 indicate a clear staged performance progression while preserving a careful distinction between classifier input count and underlying feature-acquisition operations.

Table 6. Dataset and feature-budget configurations used in the main Android application and drebin external-validation experiments. The table describes the number of acquired features, feature composition, and purpose of each budget level used to evaluate staged malware detection under progressively expanded feature sets.

Dataset	Budget	Model Input Count	Underlying Acquired Indicators	Feature Configuration	Purpose
Main APK	Budget-1	1 derived input	Permission risk, encryption evidence, network activity, and suspicious keyword evidence	Behaviour Score	Early-warning screening
Main APK	Budget-2	5 model inputs	Behaviour Score components, permissions, file size, network requests, and network flag	Behaviour Score with low-cost static indicators	Low-cost static detection
Main APK	Budget-3	9 model inputs	Complete static feature set	Complete feature representation	Full-feature baseline
Drebin	Proxy Budget-1	25 model inputs	Mutual-information-selected Drebin features	Top 25 mutual-information features	External compact proxy
Drebin	Proxy Budget-2	100 model inputs	Mutual-information-selected Drebin features	Top 100 mutual-information features	External intermediate proxy
Drebin	Proxy Budget-3	215 model inputs	Full Drebin feature representation	Full Drebin feature set	External full proxy

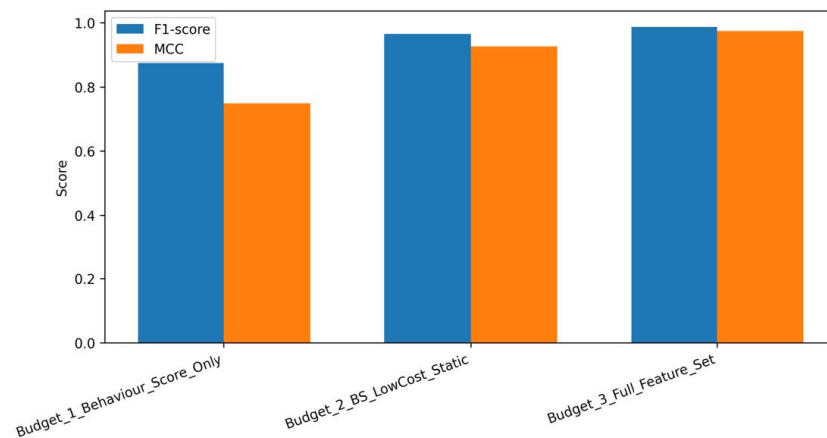


Figure 3. Detection performance across the three feature-budget configurations on the main Android application dataset. The figure compares the F1-score and Matthews correlation coefficient achieved by the best hyperparameter-tuned classifier under each budget, showing the improvement from behaviour-score-only screening to low-cost static features and the full-feature baseline.

Because the best-performing classifier can differ across model-input budgets, the results in Table 7 should be interpreted as budget-wise best-model performance rather than as a pure isolation of feature-budget effects under one fixed classifier. Although all candidate classifiers were evaluated within each budget during model selection, a dedicated same-classifier staged comparison is left for future work to further separate classifier-selection effects from model-input budget effects.

Table 7. Best hyperparameter-tuned classifier performance under each model-input budget configuration on the main Android application dataset. The table reports the model input count, selected classifier, F1-score, Matthews correlation coefficient, false positive rate, and receiver operating characteristic area to compare detection effectiveness and security-oriented behaviour across staged budgets.

Feature Budget	Model Input Count	Best Model	F1-Score	MCC	FPR	ROC-AUC
Budget-1: BS-derived input	1 derived input	Random Forest	0.875	0.7488	0.0932	0.9386
Budget-2: BS with low-cost static inputs	5 model inputs	Extra Trees	0.9654	0.9267	0.0409	0.9922
Budget-3: Full static input set	9 model inputs	Extra Trees	0.9878	0.9741	0.0159	0.9985

4.4. Model-Input Budget and Classifier Inference-Time Analysis

To provide operational evidence for the staged framework, training time and classifier inference time per sample were measured for the best-performing classifier under each model-input budget configuration. These measurements are reported as classifier-level timing indicators, not as complete end-to-end APK feature-extraction latency or physical-device deployment cost. This distinction is important because the time required to extract raw static features from APK files can differ from the time required for a trained model to produce predictions after the feature vector has already been constructed. Accordingly, the cost-aware interpretation in this study is based primarily on staged model-input budgets, confidence-based escalation, and the reduction in average acquired model inputs. Classifier inference time is reported as supporting operational evidence only. The study does not claim that the BS-only classifier necessarily has the lowest measured prediction latency in the implemented Python environment, nor does it claim complete feature-extraction energy, CPU, or memory savings on Android devices. Full end-to-end extraction-cost profiling is treated as a deployment-level validation requirement for future work.

Table 8 shows that all model-input budget configurations produced low classifier inference times in the evaluated Python implementation. However, inference time did not decrease monotonically with the number of model inputs. In particular, the BS-only stage required 0.0506 ms per sample, while the full static input set required 0.0367 ms per sample. This does not imply that full APK analysis is inherently cheaper than early-stage screening. Rather, it shows that classifier-level inference time is influenced by the selected model family, tree depth, ensemble structure, implementation overhead, and hardware-level timing variation. The BS-only configuration used Random Forest, whereas the full static input configuration used Extra Trees, so the timing values are not a pure measurement of feature-count cost. For this reason, the latency results are interpreted cautiously. They provide classifier-level timing information after feature extraction, but they do not by themselves prove end-to-end feature-acquisition savings. The main staged-framework evidence is instead the confidence-based reduction in average acquired model inputs and the percentage of samples resolved before full-feature analysis. A complete deployment-cost evaluation would require per-feature-group APK extraction timing, CPU and memory profiling, and energy measurements on representative hardware, which are identified as future work.

Table 8. Classifier-level timing and model-input budget summary for the best-performing classifier under each main Android application model-input budget. The table reports model-input count, selected classifier, training time, and inference time per sample after feature vectors have been constructed. These measurements should be interpreted as classifier-level timing indicators, not as complete APK feature-extraction latency or physical-device deployment cost.

Model-Input Budget	Best Model	Model Input Count	Training Time (s)	Inference Time (ms/Sample)
Budget-1: Behaviour Score-derived input	Random Forest	1 derived input	6.0725	0.0506
Budget-2: Behaviour Score with low-cost static inputs	Extra Trees	5 model inputs	2.4837	0.0505
Budget-3: Full static input set	Extra Trees	9 model inputs	2.7026	0.0367

4.5. Confidence-Based Escalation Analysis

The confidence-based escalation mechanism was evaluated using four predefined confidence operating points: 0.80, 0.85, 0.90, and 0.95. These thresholds were selected before the final analysis to examine the sensitivity of the staged framework to increasingly strict confidence requirements. Accordingly, they should be interpreted as predefined operating points rather than thresholds optimized on the final test set. Table 9 reports the percentage of samples resolved before full-feature analysis, the average number of acquired model inputs, model-input reduction, staged-runtime saving relative to an always-run-all-stages baseline, and the corresponding detection performance. At a confidence threshold of 0.95, 47.10% of the samples were classified at Stage 1, 43.23% at Stage 2, and only 9.68% required the full-feature stage. Consequently, 90.32% of the samples were resolved before full-feature analysis. Under this operating point, the framework achieved an F1-score of 0.9785, an MCC of 0.9548, an FPR of 0.0182, and a ROC-AUC of 0.9836. The average number of acquired model inputs decreased from 9 to 3.50, corresponding to a 61.08% reduction in model-input usage. Because the Behaviour Score is a derived model input constructed from multiple static behavioural indicators, the reported reduction should be interpreted as a reduction in classifier model inputs during staged inference rather than as a direct measurement of raw feature-acquisition cost, physical-device latency, CPU utilization, memory usage, or energy consumption. Therefore, the reported escalation results should

be interpreted within the scope of model-input budgeting rather than complete end-to-end deployment-cost evaluation. The confidence values used for escalation represent model confidence scores and are intended for operating-point analysis (See Figure 4). Although ensemble classifiers such as Random Forest, Extra Trees, and XGBoost generally provide useful confidence estimates for staged decision-making, their predicted probabilities are not assumed to be perfectly calibrated. Consequently, the thresholds reported in Table 8 should be interpreted as predefined confidence operating points for sensitivity analysis rather than universally optimal decision thresholds. Future work will investigate validation-based threshold selection, probability calibration, reliability diagrams, Brier score, expected calibration error, and calibrated confidence-based early-exit strategies to further strengthen the deployment methodology.

Table 9. Confidence-based escalation sensitivity analysis on the main Android application dataset under predefined confidence operating points. The table reports the percentage of samples resolved before full-feature analysis, average number of acquired model inputs, model-input reduction, staged-runtime saving relative to an always-run-all-stages baseline, and final detection metrics. The thresholds are reported as predefined operating points and are not presented as globally optimized thresholds.

Confidence Threshold	Early Resolved Before Full Analysis (%)	Average Acquired Model Inputs	Model-Input Reduction (%)	Runtime Saving (%)	F1-Score	MCC	FPR
0.8	97.96	2.2	75.51	52.45	0.9501	0.8974	0.0341
0.85	96.45	2.48	72.45	50.08	0.9637	0.9253	0.0205
0.9	95.05	2.75	69.49	47.77	0.9701	0.9377	0.0227
0.95	90.32	3.5	61.08	41.31	0.9785	0.9548	0.0182

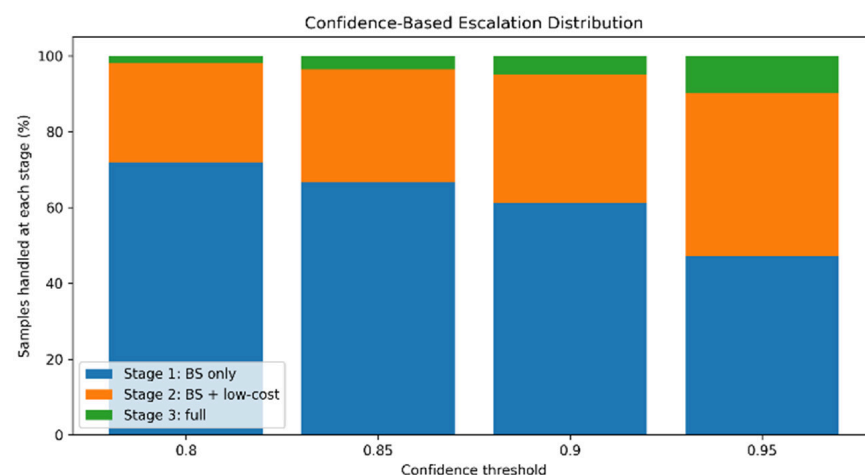


Figure 4. Confidence-based escalation distribution under different confidence thresholds on the main Android application dataset. The figure shows the proportion of samples resolved at the early behaviour-score stage, the low-cost static-feature stage, and the full-feature stage, illustrating how the proposed framework reduces unnecessary full-stage analysis while preserving detection performance.

The escalation results should therefore be interpreted within the model-input budget scope of this study. The framework reduces the frequency with which the full static input representation is required, but the current experiment does not directly measure per-feature APK extraction latency, CPU usage, memory consumption, or energy consumption on physical Android devices. This distinction is important because a derived input such as

BS can summarize multiple static indicators while still appearing as one model input to the classifier.

4.6. Low-FPR Security-Oriented Analysis

Because malware detection systems must control false alarms, the evaluation includes low FPR operating points. Table 10 presents the performance under target maximum FPR values of 0.01, 0.02, and 0.05. The Budget-2 model achieved an actual FPR of 0.0068 at a strict 0.01 target, with a precision of 0.9932, F1-score of 0.9409, and MCC of 0.8870. The full feature set achieved a stronger performance in the same low-FPR region, with an FPR of 0.0091, an F1-score of 0.9846, and an MCC of 0.9678. The results shown in Figure 5 demonstrate that the framework remains effective under security-oriented operating conditions where false positives must be minimized.

Table 10. Low false positive rate operating-point analysis for the main Android application dataset. the table reports classifier thresholds and security-oriented performance under strict target false positive rate levels, including precision, recall, F1-score, Matthews correlation coefficient, and the achieved false positive rate.

Feature Budget	Target FPR	Decision Threshold	Precision	Recall	F1-Score	MCC	Actual FPR
Budget-2	0.01	0.8	0.9932	0.8939	0.9409	0.887	0.0068
Budget-2	0.02	0.66	0.987	0.9306	0.958	0.9157	0.0136
Budget-2	0.05	0.39	0.9561	0.9776	0.9667	0.929	0.05
Full	0.01	0.66	0.9917	0.9776	0.9846	0.9678	0.0091
Full	0.02	0.4731	0.9858	0.9918	0.9888	0.9763	0.0159
Full	0.05	0.3831	0.9683	0.9959	0.9819	0.9616	0.0364

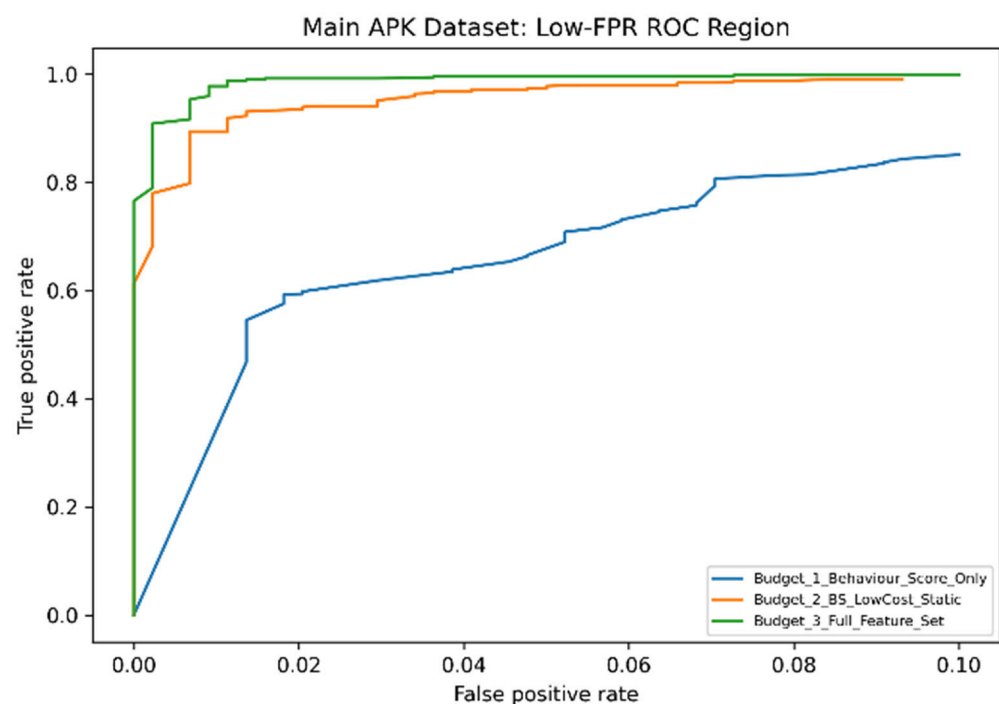


Figure 5. Low false positive rate receiver operating characteristic region for the main Android application dataset. The figure focuses on the practical malware-screening region where false alarms must be minimized, showing how the staged models behave under strict false positive rate constraints.

4.7. Cross-Validation and Statistical Significance

To verify that the observed improvements were not specific to a single holdout split, five-fold cross-validation was conducted using the best-performing model selected for each model-input budget. Table 11 shows that the BS-only stage achieved a mean F1-score of 0.8752 ± 0.0144 , Budget-2 achieved 0.9662 ± 0.0051 , and the full feature set achieved 0.9844 ± 0.0017 . Paired *t*-tests over fold-wise F1-scores showed that the improvement from Budget-1 to Budget-2 was statistically significant ($p = 0.000337$), as was the improvement from Budget-2 to the full feature set ($p = 0.002040$). These results support the robustness of the staged performance trend across cross-validation folds. However, because the analysis is based on five cross-validation folds and uses the best-performing classifier within each model-input budget, the reported *p*-values should be interpreted as supporting evidence rather than definitive proof of classifier-independent superiority. Future work will extend the statistical evaluation using repeated cross-validation, nested validation, confidence intervals for paired performance differences, and corrected comparison procedures.

Table 11. Five-fold cross-validation and paired statistical testing for the main Android application dataset. the table reports mean F1-score, standard deviation, mean improvement between feature budgets, and paired statistical significance values to verify that the staged performance trend is not dependent on a single holdout split.

Budget/Comparison	Model	Number of Features	Mean F1-Score	Std. F1-Score	Mean Difference	<i>p</i> -Value
Budget-1	Random Forest	1	0.8752	0.0144	—	—
Budget-2	Extra Trees	5	0.9662	0.0051	+0.0910 vs. Budget-1	0.000337
Budget-3 Full	Extra Trees	9	0.9844	0.0017	+0.0182 vs. Budget-2	0.002040

4.8. External Validation Using Drebin

The Drebin benchmark was used as an external proxy-budget validation experiment to examine whether the general performance trend observed in the main APK dataset also appears on a public Android malware benchmark. Because Drebin does not contain the same feature schema as the main APK dataset, this experiment does not directly validate the proposed Behaviour Score construction or the exact staged acquisition-cost model. Instead, three proxy model-input budgets were created using mutual-information-based feature selection: the top 25 features, the top 100 features, and the full 215-feature Drebin representation. Therefore, the Drebin experiment should be interpreted as external evidence for the general trend that richer feature subsets improve detection performance, not as an independent validation of the complete proposed cost-aware staging mechanism.

As shown in Table 12, the Drebin proxy-budget experiment followed the same general performance direction observed in the main APK dataset. The top-25 proxy budget achieved an F1-score of 0.9607, the top-100 proxy budget achieved an F1-score of 0.9826, and the full 215-input Drebin representation achieved an F1-score of 0.9870. However, these proxy budgets are based on mutual-information-selected Drebin features and are not equivalent to the Behaviour Score, low-cost static indicators, or full static input set used in the main APK framework (See Figure 6). Therefore, the Drebin results are used only to support the general observation that progressively richer feature representations tend to improve Android malware classification performance.

Table 12. External proxy-budget validation results on the Drebin benchmark. The table reports the number of selected Drebin model inputs, best-performing classifier, F1-score, Matthews correlation coefficient, false positive rate, and receiver operating characteristic area. Because Drebin does not share the same feature schema as the main APK dataset, these results support only the general richer-feature performance trend and are not presented as direct validation of the proposed behaviour score or exact acquisition-cost model.

Drebin Feature Budget	Model Input Count	Best Model	F1-Score	MCC	FPR	ROC-AUC
Top 25 MI features	25	Random Forest	0.9607	0.835	0.1091	0.9774
Top 100 MI features	100	Extra Trees	0.9826	0.9243	0.0737	0.9971
Full Drebin feature set	215	Random Forest	0.987	0.9437	0.0649	0.9977

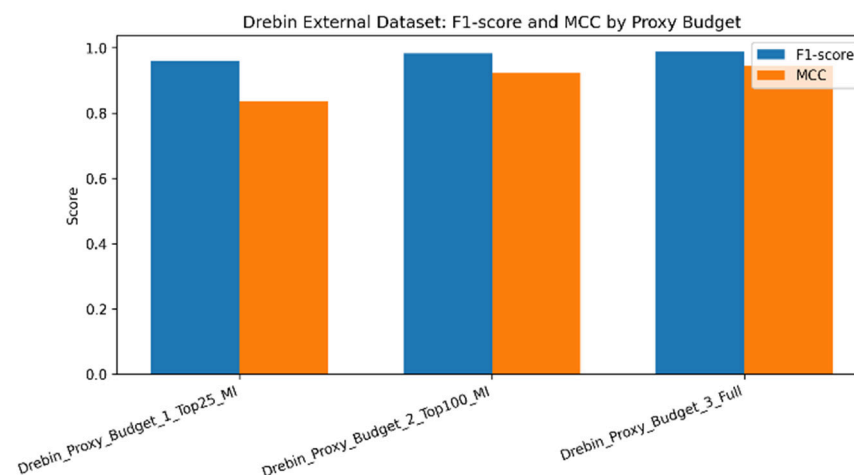


Figure 6. External Drebin validation using proxy feature-budget configurations. The figure shows that detection performance improves consistently when moving from compact mutual-information-selected features to larger proxy budgets and the full Drebin feature representation, supporting the general staged-performance trend beyond the main dataset.

5. Discussion

The experiments provided stronger evidence for the proposed staged cost-aware framework. The main result is not that the behavior score alone can replace full Android malware analysis but that it can operate as a useful early warning abstraction within a progressive detection pipeline. The BS-only stage achieved an F1-score of 0.8750, indicating that a compact behavioral signal can provide meaningful early discrimination. However, its FPR of 0.0932 also confirms that the BS alone is not sufficient for high-confidence deployment. This supports the intended role of the BS as an early-stage signal, rather than a standalone detector.

The addition of low-cost static features substantially improved performance. Using these five features, Budget-2 achieved an F1-score of 0.9654, an MCC of 0.9267, and an FPR of 0.0409. This demonstrates that a small feature set can capture a large portion of discriminative information required for Android malware screening. The full feature set achieved the strongest overall result, with an F1-score of 0.9878 and an MCC of 0.9741, confirming that richer features remain useful for difficult or uncertain samples.

The confidence-based escalation experiment is particularly important because it evaluates the staged mechanism directly rather than only comparing fixed-feature budgets. At a confidence threshold of 0.95, 90.32% of the samples were resolved before full-feature analysis, and only 9.68% required a full stage. The average number of acquired features decreased from 9 to 3.50 per sample, corresponding to a 61.08% feature acquisition reduction, while maintaining an F1-score of 0.9785, MCC of 0.9548, and FPR of 0.0182. This confirms

that the framework can reserve a full analysis of uncertain cases while preserving a strong detection performance. The empirical runtime measurements showed a very low inference latency across all budgets; however, the latency did not decrease monotonically with feature count. This was expected because the measured prediction time was influenced not only by the feature count but also by the classifier implementation, tree depth, model structure, and hardware-level timing variation. Therefore, the strongest cost-related evidence in this study was the reduction in acquired features and proportion of samples requiring full-stage analysis. Runtime profiling is included as supporting operational evidence, rather than as the sole basis for cost-aware claims.

A low-FPR analysis strengthened the security relevance of the proposed framework. Budget-2 achieved an actual FPR of 0.0068 under a 0.01 target, with a precision of 0.9932 and F1-score of 0.9409. The full feature set achieved an FPR of 0.0091, F1-score of 0.9846, and MCC of 0.9678 under the same low-FPR condition. These results are important because malware detection systems must reduce false alarms while maintaining a high detection capability.

The Drebin proxy-budget experiment further supported the general trend that richer feature subsets can improve Android malware classification performance on a public benchmark. However, because Drebin does not share the same feature schema as the main APK dataset, the Drebin experiment is not interpreted as direct validation of the proposed Behaviour Score construction or the exact staged acquisition-cost model. The cost-aware interpretation in this study is based on staged model-input budgets, confidence-based escalation, average acquired model-input reduction, and empirical inference-time profiling. However, the Behaviour Score is a derived model input constructed from multiple static indicators, including permission risk, encryption evidence, network activity, and suspicious keyword evidence. Therefore, it should not be interpreted as a cost-free raw feature. The reported reduction from 9 to 3.50 acquired inputs reflects a reduction in classifier model inputs during staged inference, not a complete accounting of all raw extraction operations required to construct each input. A full end-to-end deployment-cost analysis would require raw-feature extraction latency, CPU utilization, memory consumption, and energy measurements on physical Android devices, which are left for future work.

This study had several limitations. First, the primary dataset was newly constructed and is not yet a widely adopted public benchmark, thus limiting direct one-to-one comparisons with previous studies. To improve transparency, this study reports on the dataset provenance, class distribution, collection sources, static extraction workflow, processing environments, and quality control checks used during the dataset construction. The dataset contains 4594 APK samples, including 2412 malware samples from MalwareBazaar, the IEEE 2024 Maloid-DS-labeled Android malware forensic dataset, and 2182 goodware samples from the F-Droid open-source Android repository. Nevertheless, because the dataset was author-constructed, external validity limitations remain. This was partially mitigated through external validation of Drebin using proxy feature-budget configurations. However, future work should further evaluate the framework across additional public Android malware benchmarks and release reproducible extraction scripts whenever possible. Second, the behavior score is a weighted aggregation of existing static behavioral indicators and should be interpreted as an incremental engineering contribution rather than a fundamentally new malware representation. Third, the current framework focuses on static features such as dynamic behavior, adversarial evasion, concept drift, and zero-day adaptation, which require further investigation. Future work will extend the framework by measuring raw-feature acquisition latency, CPU utilization, memory usage, and energy consumption on physical Android devices, while also aligning each raw indicator with an explicit acquisition-cost model.

6. Conclusions

This paper presented a staged model-input budget framework for cost-aware Android malware screening using an early-warning Behaviour Score and progressively expanded static model-input budgets. The proposed framework was designed to investigate whether confident samples can be classified using compact derived and low-cost static model inputs while reserving the complete static feature representation for uncertain cases. Accordingly, the proposed approach should be interpreted as a model-input budget evaluation motivated by cost-aware malware screening rather than as a complete end-to-end deployment-cost benchmark.

The framework was comprehensively evaluated using hyperparameter-tuned classifiers, confidence-based escalation, low-false-positive-rate analysis, cross-validation, statistical significance testing, classifier-level inference-time profiling, and external proxy-budget validation using the Drebin benchmark. On the main Android application dataset, the Behaviour-Score stage achieved an F1-score of 0.8750. When additional low-cost static inputs were incorporated, the framework achieved an F1-score of 0.9654 and an MCC of 0.9267. The complete static input set produced the highest performance, with an F1-score of 0.9878 and an MCC of 0.9741. Furthermore, the confidence-based escalation experiment demonstrated that, at a confidence threshold of 0.95, 90.32% of samples were resolved before full-feature analysis while maintaining an F1-score of 0.9785. The average number of classifier model inputs decreased from 9 to 3.50, indicating that many applications could be screened effectively without requiring the complete static input representation. These findings suggest that staged model-input budgeting provides an effective strategy for improving the efficiency of Android malware screening within the evaluated static-analysis setting.

It is important to emphasize that the Behaviour Score is a derived model input constructed from multiple low-cost static behavioral indicators and should not be interpreted as a cost-free raw feature. Consequently, the reported reduction reflects a decrease in the number of classifier model inputs used during staged inference rather than a direct measurement of feature-extraction latency, CPU utilization, memory consumption, or energy consumption on physical devices. This distinction clarifies the scope of the proposed framework and the interpretation of its reported efficiency gains.

Several opportunities remain for future research. First, the framework should be extended to measure raw-feature acquisition latency, CPU utilization, memory usage, and energy consumption on physical Android devices to provide a comprehensive deployment-level evaluation. Second, additional studies should investigate adversarial robustness, temporal validation, near-duplicate control, family-aware evaluation, and broader validation across multiple public Android malware benchmarks to further assess the generalizability of the proposed framework.

Finally, the confidence thresholds used in this study were evaluated as predefined operating points rather than being selected through an independent validation-based probability calibration procedure. Future work will therefore investigate calibrated confidence estimation techniques, including reliability diagrams, Brier score, expected calibration error (ECE), and validation-based early-exit threshold selection. These extensions will provide a more rigorous assessment of confidence-guided staged inference and further strengthen the deployment readiness of cost-aware Android malware screening systems.

Author Contributions: Conceptualization, A.F.; methodology, A.F. and M.A.; software, A.F.; validation, A.F., M.A. and J.M.D.; formal analysis, A.F.; investigation, A.F.; resources, A.F. and M.A.; data curation, A.F.; writing—original draft preparation, A.F.; writing—review and editing, A.F., M.A. and J.M.D.; visualization, A.F.; supervision, J.M.D.; project administration, A.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data supporting the findings of this study are available from the corresponding author upon reasonable request. The study was conducted using static Android application feature data, and no personal or human-subject data were used.

Acknowledgments: The authors would like to acknowledge the institutional and academic support provided by their institutions during the preparation of this work.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kauser, S.K.; Anu, V.M. Hybrid deep learning model for accurate and efficient Android malware detection using DBN-GRU. *PLoS ONE* **2025**, *20*, e0310230. [\[CrossRef\]](#)
2. Ban, Y.; Ko, E.; Cho, H.; Yi, J. MAPAS: A practical deep learning-based Android malware detection system. *Int. J. Inf. Secur.* **2022**, *21*, 725–738. [\[CrossRef\]](#)
3. Lu, T.; Du, Y.; Li, O.; Chen, Q.; Wang, X. Android malware detection based on a hybrid deep learning model. *Secur. Commun. Netw.* **2020**, *2020*, 8863617. [\[CrossRef\]](#)
4. Aonzo, S.; Merlo, A.; Migliardi, M.; Oneto, L.; Palmieri, F. Low-resource footprint, data-driven malware detection on Android. *IEEE Trans. Sustain. Comput.* **2020**, *5*, 213–222. [\[CrossRef\]](#)
5. Feng, J.; Shen, L.; Chen, Z.; Wang, Y. A two-layer deep learning method for Android malware detection using network traffic. *IEEE Access* **2020**, *8*, 125786–125796. [\[CrossRef\]](#)
6. Salam, S.; Reddy, K.P. Android malware detection using machine learning techniques. *Int. Res. J. Innov. Eng. Technol. (IRJIET)* **2025**, *9*, 24–32. [\[CrossRef\]](#)
7. Ibrahim, M.; Issa, B.; Jasser, M. A method for automatic Android malware detection based on static analysis and deep learning. *IEEE Access* **2022**, *10*, 117334–117352. [\[CrossRef\]](#)
8. Ouk, P.; Pak, W. Unified detection of obfuscated and native Android malware. *Comput. Mater. Contin.* **2022**, *70*, 3099–3116. [\[CrossRef\]](#)
9. Yang, J.; Zhang, Z.; Zhang, H.; Fan, J. Android malware detection method based on highly distinguishable static features and DenseNet. *PLoS ONE* **2022**, *17*, e0276332. [\[CrossRef\]](#) [\[PubMed\]](#)
10. Wu, Y.; Li, M.; Zeng, Q.; Yang, T.; Wang, J.; Fang, Z.; Cheng, L. DroidRL: Feature selection for Android malware detection with reinforcement learning. *Comput. Secur.* **2023**, *128*, 103126. [\[CrossRef\]](#)
11. Tanveer, M.; Munir, K.; Alyamani, H.; Hassan, S.; Sheraz, M.; Chuah, T. Graph-augmented multi-modal learning framework for robust Android malware detection. *Sci. Rep.* **2025**, *15*, 38341. [\[CrossRef\]](#) [\[PubMed\]](#)
12. Bourebaa, F.; Benmohammed, M. Evaluating lightweight transformers with local explainability for Android malware detection. *IEEE Access* **2025**, *13*, 101005–101026. [\[CrossRef\]](#)
13. Singh, A.; Tanha, M.; Girdhar, Y.; Hunter, A. Interpretable Android malware detection based on dynamic analysis. In Proceedings of the 10th International Conference on Information Systems Security and Privacy (ICISSP 2024), Rome, Italy, 26–28 February 2024; pp. 195–202. [\[CrossRef\]](#)
14. Sutter, T.; Kehrer, T.; Rennhard, M.; Tellenbach, B.; Klein, J. Dynamic security analysis on Android: A systematic literature review. *IEEE Access* **2024**, *12*, 57261–57287. [\[CrossRef\]](#)
15. Kinkead, M.; Millar, S.; McLaughlin, N.; O’Kane, P. Towards explainable CNNs for Android malware detection. *Procedia Comput. Sci.* **2021**, *184*, 959–965. [\[CrossRef\]](#)
16. Lee, I.; Zhang, Q.; Yoon, S.; Won, D. A mixed integer linear programming support vector machine for cost-effective feature selection. *Knowl.-Based Syst.* **2020**, *203*, 106145. [\[CrossRef\]](#)
17. Kumar, S.; Mishra, D.; Panda, B.; Shukla, S. InviSeal: A stealthy dynamic analysis framework for Android systems. *Digit. Threat. Res. Pract.* **2022**, *4*, 1–31. [\[CrossRef\]](#)
18. Vanjire, S.; Lakshmi, M. A novel method of detecting malware on Android mobile devices with explainable artificial intelligence. *Bull. Electr. Eng. Inform.* **2024**, *13*, 2019–2026. [\[CrossRef\]](#)
19. Wu, B.; Chen, S.; Gao, C.; Fan, L.; Liu, Y.; Wen, W.; Lyu, M. Why an Android app is classified as malware. *ACM Trans. Softw. Eng. Methodol.* **2021**, *30*, 1–29. [\[CrossRef\]](#)
20. Araf, I.; Idri, A.; Chair, I. Cost-sensitive learning for imbalanced medical data: A review. *Artif. Intell. Rev.* **2024**, *57*, 80. [\[CrossRef\]](#)
21. Mahindru, A.; Sangal, A. FSDroid: A feature selection technique to detect malware from Android using machine learning techniques. *Multimed. Tools Appl.* **2021**, *80*, 13271–13323. [\[CrossRef\]](#) [\[PubMed\]](#)
22. Dayana, A. Feature extraction and classification in Android malware detection with the integration of autoencoders and transfer learning. *J. Inf. Syst. Eng. Manag.* **2025**, *10*. [\[CrossRef\]](#)

23. Bai, H.; Xie, N.; Di, X.; Ye, Q. FAMD: A fast multifeature Android malware detection framework, design, and implementation. *IEEE Access* **2020**, *8*, 194729–194740. [[CrossRef](#)]
24. Mahindru, A.; Arora, H.; Kumar, A.; Gupta, S.K.; Mahajan, S.; Kadry, S.; Kim, J. PermDroid: A framework developed using proposed feature selection approach and machine learning techniques for Android malware detection. *Sci. Rep.* **2024**, *14*, 10724. [[CrossRef](#)] [[PubMed](#)]
25. Begovic, K.; Al-Ali, A.; Malluhi, Q. Cryptographic ransomware encryption detection: Survey. *Comput. Secur.* **2023**, *132*, 103349. [[CrossRef](#)]
26. Ahmed, Y.A.; Koçer, B.; Al-rimy, B.A.S. Automated analysis approach for the detection of high survivable ransomware. *KSII Trans. Internet Inf. Syst.* **2020**, *14*, 2236–2257. [[CrossRef](#)]
27. Cai, X.; Zhang, H.; Ahmed, C.M.; Koide, H. Detecting advanced persistent threat exfiltration with ensemble deep learning tree models and novel detection metrics. *IEEE Access* **2025**, *13*, 81803–81822. [[CrossRef](#)]
28. Arp, D.; Spreitzenbarth, M.; Hübner, M.; Gascon, H.; Rieck, K. Drebin: Efficient and explainable detection of Android malware in your pocket. In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 23–26 February 2014.
29. Prasad, A.; Chandra, S.; Uddin, M.; Al-Shehari, T.; Alsadhan, N.A.; Ullah, S.S. PermGuard: A scalable framework for Android malware detection using permission-to-exploitation mapping. *IEEE Access* **2025**, *13*, 507–528. [[CrossRef](#)]
30. Panja, S.; Mondal, S.; Nag, A.; Singh, J.P.; Saikia, M.J.; Barman, A.K. An efficient malware detection approach based on machine learning feature influence techniques for resource-constrained devices. *IEEE Access* **2025**, *13*, 12647–12665. [[CrossRef](#)]
31. Kadir, A.; Peddoju, S.K. A three-level feature selection framework for Android malware detection. In *Proceedings of the 2025 IEEE 22nd International Conference on Mobile Ad-Hoc and Smart Systems (MASS)*; IEEE: New York, NY, USA, 2025; pp. 723–728. [[CrossRef](#)]
32. Surendran, R.; Uddin, M.M.; Thomas, T.; Pradeep, G. Android malware detection based on informative syscall subsequences. *IEEE Access* **2024**, *12*, 189180–189190. [[CrossRef](#)]
33. Almomani, I.; Almashat, T.; El-Shafai, W. Maloid-DS: Labeled dataset for Android malware forensics. *IEEE Access* **2024**, *12*, 73481–73546. [[CrossRef](#)]
34. Yao, L.; Xin, Y.; Liu, M.; Zhang, Q. A lightweight Android malware detection framework based on Markov images and knowledge distillation. *IEEE Internet Things J.* **2026**, *13*, 5224–5241. [[CrossRef](#)]
35. Feng, R.; Chen, H.; Wang, S.; Karim, M.M.; Jiang, Q. LLM-MalDetect: A large language model-based method for Android malware detection. *IEEE Access* **2025**, *13*, 81347–81364. [[CrossRef](#)]
36. Qian, X.; Zheng, X.; He, Y.; Yang, S.; Cavallaro, L. LAMD: Context-driven Android malware detection and classification with LLMs. In *Proceedings of the 2025 IEEE Security and Privacy Workshops (SPW)*; IEEE: New York, NY, USA, 2025; pp. 126–136. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.