IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155102

Detection of Mobile Malware (Android) using Machine Learning and Hybrid Analysis

Mrs. Uma S[1], Rahini R[2], Rudhramitra S[3]

Assistant Professor, Computer Science Engineering, Dhanalakshmi Srinivasan College Of Engineering and Technology, Chennai, Tamil Nadu - 603 104[1]

Student, Computer Science Engineering (Cyber Security), Dhanalakshmi Srinivasan College Of Engineering and Technology, Chennai, Tamil Nadu - 603 104[2,3]

Abstract: The exponential proliferation of mobile devices has catalyzed a parallel surge in Android malware, presenting critical security challenges in information technology. With Android commanding the lion's share of the global mobile operating system market, it has become the primary target for malicious actors employing sophisticated evasion techniques such as dynamic code loading, reflection, and automated repackaging (obfuscation). Detecting zero-day malware—attacks that exploit previously unknown vulnerabilities—has thus become a paramount objective for security researchers. Traditional detection paradigms, which predominantly leverage signature-based static analysis, are increasingly rendered ineffective by polymorphic malware. Conversely, dynamic analysis, while robust, incurs prohibitive computational overhead, rendering it unsuitable for real-time, on-device application.

In this paper, we propose a novel, intelligent, two-stage hybrid framework that synergizes the efficiency of Deep Learning with the forensic depth of Hybrid Analysis. The proposed model operates on a "Filter and Focus" principle. The first stage acts as a high-speed filter, employing a 1D Convolutional Neural Network (CNN) to analyze vectorized API call graphs extracted via FlowDroid. To address the "black box" nature of neural networks, we integrate Gradient-weighted Class Activation Mapping (Grad-CAM) to provide visual explainability of malicious triggers. Furthermore, a Jaccard Similarity module compares these features against known threat signatures. Only applications classified as 'Benign' or 'Uncertain' by this stage are forwarded to the second stage, which employs a rigorous hybrid engine combining Mobile Security Framework (MobSF) for static deepinspection and Quark-Engine for dynamic behavioral graphing. Experimental results on a diverse dataset of 13,298 applications (including obfuscated samples from PRAGuard) demonstrate that our hybrid model achieves an accuracy of 97.79%, significantly outperforming standalone Deep Belief Networks (DBN) and Gated Recurrent Units (GRU). The system drastically reduces false positives while maintaining a low average latency, making it a viable solution for scalable, real-world Android security.

Index Terms: Android Malware Detection, Network Security, Machine Learning, Static Analysis, Dynamic Analysis, CNN, Grad-CAM, Hybrid Analysis, Deep Belief Network (DBN), Gated Recurrent Unit (GRU), Cyber Threat Intelligence, Opcode Analysis, Adversarial Learning.

I. INTRODUCTION

A. Background

Mobile phones have evolved from simple communication devices into ubiquitous computing platforms indispensable to modern life. They facilitate critical activities ranging from mobile banking and instant messaging to healthcare monitoring and e-commerce. However, this ubiquity has expanded the attack surface for cybercriminals. As of 2024, the security landscape of mobile devices faces unprecedented threats. Attackers continuously develop sophisticated malware to exploit vulnerabilities in the Android ecosystem, which is particularly susceptible due to its open-source nature, fragmented update ecosystem, and the ease of sideloading applications from thirdparty stores.

Malicious behaviors in Android malware have diversified significantly. Common payloads include:

- Financial Fraud: Sending premium-rate SMS messages (SMS Trojans) or overlaying fake login screens on banking apps (Banking Trojans like *Cerberus* or *Joker*).
- Data Exfiltration: Stealing contacts, call logs, GPS location, and authentication tokens [10].
- Botnet Recruitment: Enslaving devices for DDoS attacks or crypto-mining.
- Ransomware: Encrypting user files and demanding payment for decryption keys.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155102

B. The Challenge of Evasion and Obfuscation

A critical challenge in modern malware detection is the widespread use of evasion techniques. Attackers employ Code Obfuscation—transformations that preserve the program's semantics while rendering the code unreadable to static analyzers. Techniques include:

- Identifier Renaming: Changing class and method names to random strings (e.g., 'a.b.c()').
- String Encryption: Encrypting hardcoded strings (URLs, API keys) and decrypting them only at runtime.
- Control Flow Flattening: Altering the structure of loops and conditionals to confuse decompilers.
- Repackaging: Disassembling legitimate apps, injecting malicious payloads, and reassembling them, often changing the hash signature entirely.

These techniques render traditional signature-based detection (hashing) ineffective, as even a single bit change results in a completely different file signature.

C. Need for Hybrid AI Models

To counter these advanced threats, the security paradigm must shift from reactive signature matching to proactive behavioral analysis. Machine Learning (ML) and Deep Learning (DL) have emerged as powerful tools in this domain.

1) Static ML: Analyzes the code structure (API calls, Permissions). It is fast but can be fooled by dynamic code loading.
2) Dynamic ML: Analyzes runtime behavior (Network traffic, System calls). It is accurate but computationally expensive and slow.
3) Hybrid Approaches: Combine both to balance tradeoffs.

Recent research, such as Lu et al. [10], suggests combining different neural architectures—Deep Belief Networks (DBN) for static features and Gated Recurrent Units (GRU) for dynamic sequences—to capture the full spectrum of malicious intent. Our work extends this by introducing a lightweight "Filter" stage to minimize the computational cost of the heavy "Hybrid" stage.

III. PRELIMINARIES AND THEORETICAL FOUNDATION

A. Feature Categories in Android Analysis

Understanding the types of features extractable from an Android Package Kit (APK) is crucial for model design.

1) Static Features: Static features are extracted without executing the application.

- Semantic Features: Derived from 'Android Manifest.xml'and'classes.dex'. These include requested Permissions (e.g., 'READ SMS'), Intent Filters, and API calls. While powerful, they are susceptible to obfuscation.
- Resource Features: Derived from non-code assets ('res/', 'assets/'). Inconsistencies here—such as mismatched file timestamps or hidden executables inside image files—are strong indicators of repackaged malware [10].

2) Dynamic Features: Dynamic features describe *what the app does*. Examples include:

- Cryptographic Operations: Frequency of 'Cipher.doFinal()' calls.
- Network Activity: Connections to known C&C servers.
- File I/O: Attempting to read '/data/data/' of other apps.

These features are inherently sequential. For instance, a sequence S = {GetLocation → OpenSocket → SendData} dictates a spyware behavior that unordered features might miss.

B. Opcode Analysis vs. API Analysis

A pivotal design choice in malware detection is the level of abstraction for feature extraction.

- Opcode Analysis: Involves disassembling the Dalvik Bytecode into opcodes (e.g., 'move', 'invoke-virtual', 'goto'). Opcodes are granular and offer a low-level view of execution logic. However, the sheer volume of opcodes (millions per app) creates high-dimensional sparse vectors that are computationally expensive to process. Furthermore, opcodes can be easily altered via compiler optimizations without changing logic.
- API Call Analysis: Focuses on System API calls (e.g., 'getDeviceId()', 'sendTextMessage()'). These are semantically richer and directly correlate with user understandable behaviors. By focusing on API calls, we reduce the feature space dimensionality while retaining the core behavioral intent, making the model faster and less prone to overfitting on compiler-specific artifacts.

C. Deep Learning Architectures

1) *Convolutional Neural Networks (CNN) for Code:* While traditionally used for image processing, CNNs are highly effective for code analysis. An APK's API call sequence can be treated as a 1D signal. The convolution operation extracts local patterns (n-grams of API calls). Mathematically, for an input sequence I and a kernel K of size m, the feature map S at position i is:

DOI: 10.17148/IJARCCE.2026.155102

$$S(i) = (I * K)(i) = \sum_{j=0}^{m-1} I(i+j) \cdot K(j) \quad (1)$$

This allows the model to detect local ”malicious motifs” (e.g., sequence of 3 specific API calls) regardless of their position in the file.

2) *Gradient-weighted Class Activation Mapping (GradCAM):* Grad-CAM provides explainability. It calculates the importance of each neuron in the last convolutional layer for the ”Malware” class. The weight α_k^c for feature map A^k and class c is computed via global average pooling of gradients:

$$\alpha_k^c = \frac{1}{Z} \sum_i \frac{\partial y^c}{\partial A_i^k} \quad (2)$$

where y^c is the logit for class c. The final heatmap $L^c_{Grad-CAM}$ is a weighted combination:

$$L^c_{Grad-CAM} = ReLU \left(\sum_k \alpha_k^c A^k \right) \quad (3)$$

This highlights the specific API calls that contributed most to the detection.

3) *Gated Recurrent Units (GRU):* For dynamic feature analysis, GRUs are preferred over standard RNNs due to their ability to mitigate the vanishing gradient problem. A GRU cell computes the hidden state h_t using update gate z_t and reset gate r_t:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \quad (4)$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \quad (5)$$

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t]) \quad (6)$$

$$ht = (1 - zt) * ht{-}1 + zt * \tilde{h}t \quad (7)$$

This structure allows the model to retain long-term dependencies in API execution traces [10].

IV. LITERATURE SURVEY

A. Evolution of Detection Techniques

The arms race between malware authors and defenders has driven significant innovation.

- Signature Based: Early antivirus relied on MD5 hashes. While extremely fast ($O(1)$ lookup), they fail against polymorphism.
- Permission Based: *APK Auditor* [10] analyzed permission combinations (e.g., ‘INTERNET‘ + ‘SEND SMS‘). However, many benign apps also request these, leading to high false positives. Permissions are coarse-grained; a flashlight app and a spyware app might naturally both request ‘CAMERA‘ permission, making distinction difficult based solely on this feature.
- Taint Analysis: *TaintDroid* [11] tracks the flow of sensitive data through the OS. It is highly accurate but introduces ≈ 14% CPU overhead, degrading the user experience. Moreover, recent Android versions have locked down system hooks that TaintDroid relies on, rendering it less effective on Android 10+.

B. Machine Learning Advancements

MaMaDroid [3] represented a leap forward by modeling API calls as Markov Chains. It achieved ≈ 99% accuracy on the Drebin dataset. However, constructing Markov models for thousands of apps is memory-intensive (requiring up to 34GB RAM during training), making it impractical for resourceconstrained environments. *MAPAS* [2] utilized deep learning (CNNs) to reduce feature engineering effort. It achieved 92% accuracy but often failed against zero-day attacks that used novel obfuscation not present in the training set.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155102

C. Adversarial Attacks and the 2024 Landscape

The year 2024 has seen a pivotal shift in the threat landscape with the rise of Adversarial Machine Learning. Sophisticated attackers are no longer just obfuscating code but actively crafting ”Adversarial Examples”—malware samples specifically perturbed to maximize the loss function of detection models while retaining malicious functionality. Key trends include:

1) Model Extraction Attacks: Attackers query the public API of a detection system (like our Stage 2 backend) to label their own dataset, eventually training a ”Surrogate Model” that mimics the target’s decision boundary. Once the surrogate is trained, white-box attacks can be generated offline [4]. This highlights the need for API rate limiting and non-deterministic response latencies.

2) Accessibility Service Abuse: Modern Trojans (e.g., *Xenomorph*) are increasingly abusing Android Accessibility Services to automate clicks, perform credential stuffing, and grant themselves permissions without user interaction.

3) Generative Attacks: The use of GANs (Generative Adversarial Networks) to automatically generate obfuscated variants that pass static checks.

These developments necessitate a move beyond simple ”accuracy” metrics towards ”robustness” and ”explainability”, motivating our integration of Grad-CAM and Jaccard Similarity.

V. PROPOSED METHODOLOGY

Our framework implements a Two-Stage Selective Execution architecture. This design is predicated on the observation that most apps (90%+) are benign or easily classifiable, and thus do not require expensive dynamic analysis.

A. Data Preprocessing and Balancing

Before training, the dataset undergoes rigorous preprocessing.

1) Decompilation: Uses *Androguard* to convert APKs to Dalvik Bytecode.

Stage 1: Static Filter (Screening) API Graph → Vectorization → CNN → Grad-CAM → Jaccard Check
Decision Gate If *Confidence* > θ, Finalize. Else, Forward.
Stage 2: Hybrid Engine (Deep Dive) → MobSF (Static) + Quark (Dynamic) → Weighted Risk Score

Fig. 1. The Proposed Two-Stage Architecture Flow

2) Filtering: Removes 3rd-party libraries (e.g., Google Ads, Facebook SDK) to prevent learning bias towards common ads.

3) Balancing: Malware datasets are often imbalanced (fewer malware than benign samples). We employ SMOTE (Synthetic Minority Over-sampling Technique) to generate synthetic minority class samples. For a feature vector x in the minority class, SMOTE selects a nearest neighbor x_{nn} and creates a new sample x_{new}:

$$x_{new} = x + \lambda \cdot (x_{nn} - x) \qquad (8)$$

where λ is a random number between 0 and 1. This ensures the model decision boundary is not biased towards the majority class.

B. Stage 1: Fast Static Screening

The objective of this stage is high-throughput filtration. We selected Static API Analysis over dynamic analysis for this stage because it offers a $O(N)$ complexity relative to code size, which is orders of magnitude faster than running an app in a sandbox (approx. 5 minutes per app).

1) Feature Engineering: Graph to Sequence: We use FlowDroid to generate the Control Flow Graph (CFG) of the application.

DOI: 10.17148/IJARCCE.2026.155102

- Step 1: CFG Construction: FlowDroid maps all possible execution paths.
- Step 2: API Extraction: We traverse these paths to collect a sequence of API calls. Unlike simple "Bag-ofWords" models that count frequency, preserving the *order* is critical. For example, 'setWifiEnabled(true)' followed by 'sendData()' is benign, but 'getGPS()' followed by 'sendData()' implies tracking.
- Step 3: Vectorization: We define a vocabulary V of the top $N = 10{,}000$ most frequent Android API calls. Each API call is mapped to a dense vector embedding $v \in \mathrm{R}^d (d = 64)$. This embedding is learned during training, allowing the model to understand that 'Log.d' and 'Log.i' are semantically similar.

2) Detailed CNN Architecture: The core classifier is a 1D Convolutional Neural Network. We chose CNN over RNN/L-STM for Stage 1 because CNNs are parallelizable (faster training/inference) and excellent at detecting local features (ngrams). The detailed architecture is as follows:

1) Input Layer: Takes a sequence of length $L = 500$ (padded/truncated). Input shape: (500,64).
2) Convolutional Block 1:
   - Filters: 128
   - Kernel Size: 3 (captures 3-gram patterns)
   - Activation: ReLU
3) MaxPooling 1: Pool Size 2. Reduces dimensionality by half to (250,128).
4) Convolutional Block 2:
   - Filters: 64
   - Kernel Size: 5 (captures 5-gram patterns)
   - Activation: ReLU
5) Global Average Pooling: Output shape (64). This layer is crucial for Grad-CAM as it preserves the spatial correspondence between feature maps and categories.
6) Dense Layer: 32 units, ReLU activation.
7) Dropout: Rate 0.5 to prevent overfitting.
8) Output Layer: 1 unit, Sigmoid activation (Probability of Maliciousness).

3) Explainability and Decision Logic: The network outputs a probability P_{mal}. Simultaneously, the Grad-CAM module computes the heatmap H. We verify this heatmap against known signature patterns using Jaccard Similarity:

$$J(H, S_{db}) = \frac{|H \cap S_{db}|}{|H \cup S_{db}|} = \frac{|H \cap S_{db}|}{|H| + |S_{db}| - |H \cap S_{db}|} \quad (9)$$

If $J > 0.43$, the app is flagged as "Known Malware Variant". The Selective Logic:

- If $P_{mal} > 0.9$ OR ($P_{mal} > 0.8$ AND $J > 0.43$): Flag Malicious.
- If $P_{mal} < 0.2$: Flag Benign.
- Otherwise: Forward to Stage 2 (Uncertain/Suspicious).

C. Stage 2: Deep Hybrid Analysis

Triggered only when $0.2 < P_{mal} < 0.8$ (Uncertain).

1) Static Analysis via MobSF: The Mobile Security Framework (MobSF) performs deep semantic analysis. It decompiles the APK to Smali/Java and checks for:

- ManifestIssues: 'android:debuggable=true', 'allowBackup=true'.
- Code Secrets: Regex matching for AWS keys, Google API keys, hardcoded passwords.
- Tracker Detection: Identifying known ad-trackers and analytics SDKs.

2) Dynamic Analysis via Quark-Engine: Quark-Engine operates on a "Crime-Based" rule system. It maps low-level API combinations to high-level malicious behaviors ("Crimes"). Algorithm for Quark Scoring:

VI. IMPLEMENTATION DETAILS

A. System Stack

The system is implemented as a microservices architecture:

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155102

TABLE I
QUARK-ENGINE CRIME RULE EXAMPLES

Crime Category	Rule Description / API Sequence
Credential Theft	1. Check for ‘AES‘ or ‘DES‘ encryption utilization. 2. Search for ‘getRunningTasks‘ to detect overlay attacks. 3. Monitor specific URL endpoints matchingbanking API patterns.
Keylogging	1. Registering ‘AccessibilityService‘. 2. Capturing ‘onAccessibilityEvent‘ with keytext data. 3. Writing data to external storage or socket.
SMS Fraud	1. ‘TelephonyManager.getLine1Number()‘ (Steal number). 2. ‘SmsManager.sendTextMessage()‘ (Send premium SMS). 3. Intercept ‘SMS RECEIVED‘ broadcast to hide notification.
Ransomware	1. Traverse file system recursively. 2. Read specific file extensions (.jpg, .pdf,.docx). 3. Encrypt file content and delete original.

Algorithm 1 Quark Risk Assessment

Require: Set of Rules R, APK Bytecode B

```
Score ← 0
for all rule ∈ R do
    Confidence ← check rule(rule,B)
    if Confidence > 0.8 then
        Score ← Score + rule.weight
    end if
end for
RiskLevel ← Normalize(Score)
return RiskLevel
```

B. Detailed Training Protocol

To ensure robust model performance and prevent overfitting, a rigorous training protocol was adopted.

1) Optimizer: We employed the Adam optimizer with an initial learning rate of 0.001. Adam was chosen for its adaptive learning rate properties, which suit the sparse nature of API data.

2) Loss Function: Binary Cross-Entropy (BCE) was used:

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \quad (10)$$

where y_i is the true label and $\hat{y}_i$ is the predicted probability.

3) Regularization: To mitigate overfitting, especially given the high dimensionality of features:

- Dropout layers with a rate of 0.5 were inserted after fully connected layers.
- Early Stopping was implemented with a patience of 10 epochs, monitoring validation loss.

C. Hardware Configuration

Experiments were conducted on a workstation equipped with:

• CPU: Intel Core i9-12900K (16 Cores)

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155102

TABLE II
IMPLEMENTATION TECHNOLOGY STACK

Component	Tool/Library	Role
Frontend	Android (Kotlin)	User Interface, File Upload
Backend API	FastAPI (Python)	Request Handling, Orchestration
ML Engine	TensorFlow 2.x	CNN Model Inference
Static Tool	MobSF (Docker)	Deep Static Analysis
Dynamic Tool	Quark-Engine	Behavioral Analysis
Database	PostgreSQL	Report Storage

Algorithm 2 Hybrid Model Training Procedure

Require: Dataset $D = \{(X_i, y_i)\}$, Epochs E, BatchSize B

Initialize CNN weights W with Xavier Initialization

Split D into D_{train} (70%), D_{val} (15%), D_{test} (15%)

Apply SMOTE on D_{train} to balance classes 4: for $e = 1$ to E do

    Shuffle D_{train}

    for all batch $b \in D_{train}$ do

        Forward Pass: $\hat{y} = \text{CNN}(b_X, W)$

        Compute Loss: $L = \text{BCE}(y, \hat{b}_y)$

        Backward Pass: $\nabla W = \frac{\partial L}{\partial W}$

        Update: $W \leftarrow W - \eta \cdot \nabla W$

    end for

    Calculate Validation Accuracy on D_{val}

    if $ValAcc$ not improved for 10 epochs then

        Break (Early Stopping)

    end if 16: end for

    return Trained Model W^*

- RAM: 64GB DDR5
- GPU: NVIDIA RTX 3090 (24GB VRAM) - for CNN training
- Storage: 2TB NVMe SSD - for Dataset storage

VII. EXPERIMENTAL RESULTS AND ANALYSIS

A. Dataset Composition

A robust dataset is critical for valid results. We aggregated samples from three primary sources to create a balanced dataset of 13,298 applications.

TABLE III
DATASET DISTRIBUTION

Class	Source	Count	Obfuscated?
Benign	Google Play	5,000	No
Benign	APKPure	2,000	No
Malicious	VirusShare	4,038	No
Malicious	PRAGuard	2,260	Yes
Total	-	13,298	-

The PRAGuard dataset is particularly significant as it contains malware samples obfuscated using seven different techniques (including reflection and class encryption), testing the model's resilience.

B. Comparative Performance

We compared our Proposed Hybrid Model against four baselines:

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 5, May 2026

DOI: 10.17148/IJARCCE.2026.155102

1) SVM: Classical Machine Learning with permission features.
2) KNN: k-Nearest Neighbors ($k = 5$).
3) DBN: Deep Belief Network (Static features only) [10]. 4) GRU: Gated Recurrent Unit (Dynamic features only)

TABLE IV

Model	Benign	Non-Obf. Malware	Obf. Malware	Overall
SVM	92.74	88.91	72.30	84.65
KNN	83.83	85.65	68.12	79.20
DBN (Static)	95.98	93.49	89.51	92.99
GRU (Dynamic)	93.78	91.74	88.20	91.24
Hybrid (Ours)	98.15	97.79	96.58	97.79

PERFORMANCE BENCHMARK (ACCURACY %)

C. Confusion Matrix Analysis

To gain deeper insights into misclassifications, we analyzed the Confusion Matrix.

- False Positives (FP): 1.2%. These were mostly ”Grayware” apps (e.g., Aggressive Adware) which, while not strictly malware, exhibit intrusive behavior. Our model’s sensitivity to ad libraries (despite initial filtering) caused these flags.
- False Negatives (FN): 1.0%. These were primarily ”Time-Bomb” malware which delay execution for weeks. Since our dynamic analysis (Quark) runs for a limited time window, these dormant payloads remained inactive during the scan. This highlights a limitation of sandboxbased dynamic analysis.

D. Ablation Study

We conducted an ablation study to quantify the contribution of each component.

1) Without Grad-CAM: If we remove the ”Explainability” branch, accuracy remains 97.79%, but trust decreases. Security analysts spent 3x more time verifying results without the heatmap.
2) Without Stage 2 (Filter Only): If we rely solely on the CNN (Stage 1), accuracy drops to 94.5%. The drop is most significant on the *PRAGuard* dataset (Obfuscated), confirming that the Hybrid (Quark/MobSF) stage is essential for catching complex obfuscation that bypasses the static CNN.

E. Latency Analysis

Despite the depth of analysis, the average processing time per APK was 12.4 seconds.

- Stage 1 (CNN): 0.8 seconds.
- Stage 2 (MobSF/Quark): 25-40 seconds.

Since Stage 1 filters out ≈ 85% of clear-cut cases, the *amortized* average time is low, validating the efficiency of the Two-Stage architecture.

VIII. ADVANCED DISCUSSION

A. Adversarial Robustness and Model Extraction

Deep learning models are susceptible to adversarial attacks, where subtle perturbations are introduced to deceive the classifier. In the context of Android malware, an attacker might inject ”nop” (No Operation) code or benign API calls (e.g., ‘Log.d()‘) to dilute the malicious sequence, effectively lowering the global risk score. Our framework counters this via Grad-CAM. By visualizing the activation areas, we observed that adversarial injections usually appear as ”cold spots” in the heatmap, while the actual malicious sequence remains a ”hot spot”. The Jaccard Similarity check (*Stage*1) specifically compares these hot spots against known malware signatures. If an app has a high resemblance to a known malware *core* despite having thousands of benign injections, the Jaccard index will remain high, triggering a detection.

Furthermore, regarding Model Extraction Attacks [4], where attackers query the backend to reverse-engineer the model:

- The complex Two-Stage Logic acts as a naturally ”noisy” oracle. The output seen by a potential attacker is not a simple probability from a single model, but a fused decision from a CNN, a Jaccard comparator, and a rulebased engine (Quark).
- This non-differentiable pipeline makes it mathematically difficult for attackers to train a surrogate model using gradient-based methods, significantly hardening the system against ”white-box” adversarial generation.

DOI: 10.17148/IJARCCE.2026.155102

B. Hardware-Aware Optimization

Running DL models on mobile devices exposes the ”Memory Wall” problem. Standard CNNs require substantial RAM, which causes battery drain. We explored Post-Training Quantization (PTQ) to convert 32-bit floating-point weights to 8-bit integers.

$$x_q = \text{round}\left(\frac{x}{s} + z\right) \qquad (11)$$

where s is the scale factor and z is the zero-point. This reduced the model size by 4× (from 12MB to 3MB) with a negligible accuracy drop of 0.4%, making the ”Stage 1” filter viable for on-device execution in future iterations.

IX. CONCLUSION AND FUTURE SCOPE

In this paper, we presented a comprehensive, hybrid Android malware detection framework designed to bridge the gap between fast static screening and deep dynamic analysis. By integrating a CNN-based initial filter with interpretable GradCAM visualization, and backing it with a rigorous MobSF/Quark hybrid engine, we achieved a detection accuracy of 97.79% across a diverse dataset of over 13,000 applications. Crucially, our system demonstrates exceptional resilience against code obfuscation, significantly outperforming traditional ML approaches.

Future work will focus on:

1) Adversarial Hardening: Training the CNN against adversarial samples generated by GANs.
2) On-Device Inference: Quantizing the model (TensorFlow Lite) to run Stage 1 directly on the Android device for privacy.
3) Federated Learning: Enabling collaborative model updates across distributed devices without sharing raw data, preserving user privacy while safeguarding the ecosystem.

REFERENCES

[1]. X. Wang, Y. Yang, and Y. Zeng, ”Accurate mobile malware detection and classification in the cloud,” *SpringerPlus*, vol. 4, no. 1, p. 583, 2015.

[2]. J. Kim, Y. Ban, E. Ko, H. Cho, and J. H. Yi, ”MAPAS: a practical deep learning-based android malware detection system,” *International Journal of Information Security*, vol. 21, pp. 725–738, 2022.

[3]. L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, ”MaMaDroid: Detecting android malware by building Markov chains of behavioral models,” *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pp. 1–34, 2019.

[4]. H. Hindy, R. Atkinson, C. Tachtatzis, J. Colin, E. Bayne, and X. Bellekens, ”Towards an Effective Zero-Day Attack Detection Using Outlier-Based Deep Learning Techniques,” *arXiv preprint arXiv:2006.15344*, 2020.

[5]. D. Arp et al., ”Drebin: Effective and explainable detection of android malware in your pocket,” in *Ndss*, vol. 14, pp. 23-26, 2014.

[6]. R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, ”Grad-CAM: Visual explanations from deep networks via gradient-based localization,” in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 618–626, 2017.

[7]. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, ”Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8]. G. LaMalva and S. Schmeelk, ”MobSF: Mobile Health Care Android Applications Through The Lens of Open Source Static Analysis,” in *2020 IEEE MIT Undergraduate Research Technology Conference (URTC)*, pp. 1-4, 2020.

[9]. Y. Dang, K. Chen, and S. Lu, ”Automatic Android Malware Detection Rule Generation based on Quark Engine,” *Communications of the CCISA*, vol. 28, no. 2, pp. 1-24, 2022.

[10]. T. Lu, Y. Du, L. Ouyang, Q. Chen, and X. Wang, ”Android Malware Detection Based on a Hybrid Deep Learning Model,” *Security and Communication Networks*, vol. 2020, Article ID 8863617, 11 pages, 2020. https://doi.org/10.1155/2020/8863617

[11]. W. Enck, P. Gilbert, S. Han, V. Tendulkar, L. P. Cox, A. R. Jung, P. McDaniel, and A. N. Sheth, ”TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones,” in *OSDI*, vol. 10, pp. 1-6, 2010.

[12]. S. Arzt et al., ”FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps,” *PLDI*, 2014.

