

A Hardware-Efficient Android Malware Detection Framework for Resource-Constrained Devices

ALOK KUMAR

`alokk.phd19.cs@nitp.ac.in`

National Institute of Technology Patna

Jyoti Prakash Singh

National Institute of Technology Patna

Prabhat Kumar

National Institute of Technology Patna

Research Article

Keywords: Android Malware Detection, TinyML, Static Analysis, IoT Security, signal processing-based features, GLCM

Posted Date: August 6th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10392103/v1>

License:  This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

A Hardware-Efficient Android Malware Detection Framework for Resource-Constrained Devices

Alok Kumar^{1,2*}, Jyoti Prakash Singh¹ and Prabhat Kumar^{1†}

^{1*}Computer Science and Engineering, National Institute of Technology Patna, Ashok Raj Path, Patna, 80006, Bihar, India.

²Computer Science and Engineering, Bakhtiyarpur College of Engineering, Bakhtiyarpur, Campapur, Dedaaur, Bakhtiyarpur, 803212, Bihar, India.

*Corresponding author(s). E-mail(s): alokk.phd19.cs@nitp.ac.in;

Contributing authors: jps@nitp.ac.in; prabhat@nitp.ac.in;

[†]These authors contributed equally to this work.

Abstract

Traditional Android malware detection frameworks rely heavily on computationally intensive deep learning architectures or exhaustive reverse-engineering paradigms, rendering them unviable for resource-constrained Internet of Things (IoT) edge devices. This paper proposes an ultra-lightweight, hardware-efficient malware detection framework optimized for execution directly at edge endpoints. Raw Android executables are transformed into discrete 1D opcode signals. From these signals, a 28-dimensional hybrid feature-vector is constructed, fusing 22 1D time-domain metrics—including statistical measures, entropy, and Linear Predictive Coding residual with 6 2D spatial feature descriptors derived via Gray-Level Co-occurrence Matrix. Four machine-learning algorithms along with a one-hidden layer Feed-Forward Neural Network (NN) were compared to classify 11,446 apk. Two best-suited models were then quantized for Hardware-level verification via cross-compilation targeting an ARM-Cortex-M4 microcontroller.

The Decision Tree and proposed NN model achieved a classification accuracy of 96.33% and 95.76%, respectively, and an F1-score of 96.16% and 95.68%, respectively. Cross-compilation of the DT model for the ARM-Cortex-M4 microcontroller compressed the model to a Flash storage size of 4.58 KB with an inference latency of **173** ns. Meanwhile, post-training 8-bit quantization compressed the NN into an ultra-lean hardware footprint, requiring merely 2.82 KB of Flash memory and a nominal 0.83 KB of peak active SRAM, with an inference latency of **9** ns.

The proposed work employs signal processing techniques to extract features, yielding a lightweight malware classification model that can be localized entirely within hardware registers. This establishes a high-fidelity, deterministic security solution for resource-constrained embedded systems.

Keywords: Android Malware Detection, TinyML, Static Analysis, IoT Security, signal processing-based features, GLCM

1 Introduction

The rapid proliferation of the Internet of Things (IoT) and edge computing has fundamentally transformed the modern digital infrastructure. By the end of 2025, the global population of connected devices exceeded 21 billion¹. Android and embedded Linux operating systems are among the most popular operating systems for smart home appliances, industrial sensors, and enterprise edge nodes [1]. The huge deployment of IoT and embedded systems and the open nature of operating systems have inadvertently created a massive and highly vulnerable attack surface. Most of these edge devices frequently ship with baseline firmware and operate autonomously in decentralized environments, exposing them to malware attacks. Recent threat intelligence reveals a staggering 124% year-over-year surge in IoT malware attacks heading into 2026, alongside a 151% increase in Android-specific malware variants²³.

Modern attackers no longer exploit compromised end devices solely to steal local data; instead, they conscript these resource-constrained devices into massive, coordinated botnets. For example, between late 2025 and early 2026, botnets such as *Aisuru-Kimwolf*⁴ and *JackSkid*⁵ hijacked millions of consumer-grade routers, smart TVs, and IP cameras. *Aisuru-Kimwolf* and *JackSkid* botnets compromised consumer Internet of Things (IoT) devices, DVRs, and smart TVs to launch record-breaking Distributed Denial-of-Service (DDoS) attacks exceeding 31 Terabits per second (Tbps), which paralyzed critical enterprise data centers and government infrastructures. Android banking trojans and mobile ransomware have also evolved drastically, with a year-over-year spike of 67% in fraudulent transactions. Recent banking trojans, such as *Rokarolla*,⁶ and mobile ransomware are no longer designed just to steal credentials for subsequent use; instead, they perform fraud directly on the device in real time, which makes detection significantly more difficult with traditional cloud-based infrastructure.

The recently discussed malware threats targeting resource-constrained devices present a significant challenge in developing security solutions that operate within the strict computational and storage constraints of edge nodes. While Android banking trojans like *Rokarolla* may evade traditional cloud-based infrastructure, a local

¹<https://iot-analytics.com>

²<https://sonicwall.com>

³<https://malwarebytes.com>

⁴<https://www.cloudflare.com/learning/ddos/glossary/aisuru-kimwolf-botnet/>

⁵<https://corporate.comcast.com/stories/evolution-of-jackskid-botnet-what-we-found-and-how-residential-proxies-amplify-its-reach>

⁶<https://www.malwarebytes.com/blog/mobile/2026/06/rokarolla-android-malware-can-take-over-your-phone-and-steal-banking-logi>

malware detector can identify these risks directly at the edge. Traditionally, malware detection is performed via cloud-based antivirus agents, signature matching, or dynamic analysis via sandboxing [2, 3]; however, these approaches are unsuitable for resource-constrained embedded environments.

Microcontrollers and edge nodes operate under strict limitations regarding computational capability, memory capacity, and energy availability [4–6]. For cloud-based antivirus solutions, continuous wireless transmission introduces unacceptable latency, rapidly depletes battery life, and introduces data privacy concerns. Furthermore, in the absence of network connectivity, cloud-dependent security fails completely, leaving the local endpoint entirely defenseless.

Consequently, to secure consumer IoT devices and other resource-constrained devices, an immediate shift of malware detection systems from the cloud to the hardware itself is required, which can operate strictly within the limited memory, power, and latency boundaries of the host device [7, 8].

To address this, a vast body of literature treats malware binaries as images to deploy massive Convolutional Neural Networks (CNNs) [9, 10] or executes heavy reverse-engineering workflows [11, 12]. Specifically, many existing frameworks rely on deep reverse-engineering pipelines to extract low-level structural artifacts like Smali code [12, 13], or they translate raw binary payloads into 2D structural images for deep convolutional processing [14–16]. While highly accurate, these heavy architectural approaches demand massive computational capacity, incur high energy penalties, and easily exhaust local memory pools.

To overcome the restrictive SRAM budgets, limited Flash memory, and low computational power inherent to edge devices, this paper introduces a highly efficient, resource-optimized malware identification framework tailored for strict embedded architectures. Unlike heavy deep learning networks, the proposed approach converts the target Android Package Kit (*apk*) payload into a one-dimensional discrete opcode signal. The pipeline extracts a lean 28-dimensional feature representation: 22 time-domain statistical metrics and 6 spatial texture features derived from a Gray-Level Co-occurrence Matrix (GLCM). We conduct a rigorous comparative analysis across four classical machine learning classifiers (i) Naïve Bayes (NB), (ii) Support Vector Machine (SVM), (iii) Decision Tree (DT), and (iv) K-Nearest Neighbors (KNN) and a lightweight Feed-Forward Neural Network (NN).

The proposed work identifies two models—i) a Decision Tree (DT) and ii) a single-layer Feed-Forward Neural Network (NN)—for final deployment validation to take care of the diverse and fluctuating hardware constraints of modern edge infrastructure. Each model addresses a distinct operational sweet spot on the edge-computing spectrum. We suggest the NN model to achieve small latency and real-time detection, as the compiled NN model delivers an extraordinary sub-microsecond inference latency of 9.00×10^{-6} ms while consuming just 2.82 KB of Flash memory and a nominal 0.83 KB of peak active SRAM. This extreme efficiency allows the model to run comfortably on legacy, ultra-low-power 8-bit or 16-bit microcontrollers. The DT model is favoured in situations where we have strict SRAM restrictions, as the model does not require any extra dynamic RAM and the Flash storage requirement is 4.58 KB. Our

empirical evaluation demonstrates that the proposed NN achieves a peak classification accuracy of 95.76% while the DT model achieves an accuracy of 96.33%.

The primary contributions of this work are summarized as follows:

1. We employed a preprocessing function to map raw hexadecimal opcode strings into 1D continuous numeric arrays to view them as a signal.
2. We implement a 28-dimensional feature architecture that combines 1D global statistics, information-theoretic entropies, and short-time Linear Predictive Coding (LPC) residuals with 2D GLCM spatial textures to capture both overt and obfuscated malicious behaviors.
3. We provide a comprehensive evaluation isolating the performance of 1D and 2D feature subsets across classical machine learning models along with a lightweight Neural Network.
4. We rigorously validate the model’s operational viability for edge devices. By profiling an emulated ARM Cortex-M4 microcontroller via Renode for DT and TFLight for NN model, we empirically confirm that the both the models operates within SRAM and Flash memory availability of ARM Cortex-M4 microcontroller with near-zero latency.
5. We establish the structural stability and immunity to over-fitting of the proposed architecture through extensive statistical validation across 20 independent execution runs.

The rest of this paper is structured as follows. Section 2 reviews relevant literature on edge security and mobile malware detection. Section 3 describes the proposed methodology, highlighting the feature extraction, classification, and quantization/cross-compilation processes. Section 4 presents the empirical results and performance evaluations across the emulated architectures. Section 5 highlights the major findings of the research, presents a comparative study, outlines limitations, and discusses the broader security implications and architectural trade-offs. Finally, Section 6 concludes the paper and points toward future research directions.

2 Related Work

The rapid proliferation of sophisticated malware variants has catalyzed an extensive body of literature dedicated to safeguarding resource-constrained devices against malicious attacks. Traditional Android security frameworks have heavily relied on static [17] or dynamic analysis [18, 19] to distinguish malicious *apk* files from benign ones. Static approaches scrutinize various components of an application package without executing it, checking files such as the *AndroidManifest.xml* for requested permissions and intents [20], parsing the compiled *classes.dex* bytecode [21, 22], or map control-flow graphs [23]. However, common obfuscation techniques, cryptographic packers, and dynamic class loading easily circumvent these static boundaries, effectively masking malicious behavior until runtime [24].

To counter this, dynamic analysis methods run target binaries inside isolated virtual sandboxes to track active runtime features, including system call sequences and network transmission spikes [19]. While highly resilient against code packing, dynamic

frameworks introduce prohibitive computational, memory, and energy overheads. These heavy resource dependencies make continuous execution-layer sandboxing structurally unviable for energy-limited microcontrollers and low-power edge endpoints operating on strict battery budgets.

To balance this trade-off, several recent approaches utilize deep learning architectures to extract features from binary executables automatically. A prominent strategy involves transforming raw application binaries into 2D grayscale images or structural visual matrices, reframing malware classification as a computer vision problem using deep Convolutional Neural Networks (CNNs) [25–27]. Similarly, driven by the success of Transformer architectures in vision fields, contemporary studies have applied Vision Transformers (ViTs) to these 2D representations of binary executables to analyze structural code anomalies across diverse malware families [28, 29].

Other popular detection frameworks employ extensive reverse-engineering pipelines to map raw binary sequences back into human-understandable formats. These approaches routinely decompile target payloads into intermediate representations, such as Smali code blocks for *apk* files [17]. These extracted Smali blocks are then processed through advanced Natural Language Processing (NLP) models, such as Bidirectional Encoder Representations from Transformers (BERT) [11], or specialized Transformer networks to capture long-range semantic code dependencies. Although these methods achieve high classification accuracy against multi-variant attacks, they rely on substantial computational resources and massive storage capacity. Consequently, these resource-intensive architectures cannot be deployed directly within stringent embedded layers or localized edge compute environments.

Because of these bottlenecks, a major subfield of static malware analysis focuses entirely on extracting operational code (opcode) sequences from disassembled binaries. Since these opcodes reflect the foundational machine instructions executed by the underlying hardware or virtual machine, they provide highly discriminative patterns for distinguishing malicious from benign applications. For instance, Subhameena et al. [30] used an Long Short-Term Memory (LSTM) model on opcode sequences to categorize diverse malware families, including Trojans, worms, and mobile ransomware. Saini et al. [31] similarly found that opcode-based static analysis can serve as a highly resilient pipeline for malware detection when combined with low-overhead feature engineering.

Researchers have experimented with several ways to structure this disassembled code to capture its underlying behavior. For instance, some approaches rely on basic opcode arrays [21] or compute complex n -gram frequency vectors [11, 32]. Other methods map the relationships between APIs and opcodes through relational clusters [33], or group instructions into raw sequential multisets [34]. Along these lines, Sharma et al. [34] showed that multiset encodings of Smali opcode sequences can make classifiers much more resilient to obfuscation, allowing them to track intent even when developers intentionally pack or mutate the underlying application binary.

The main drawback of these reverse-engineering frameworks is that they still require heavy processing for feature engineering, making them impractical for resource-constrained devices. To get around this limitation, a practical alternative is to map *apk* files directly into the digital signal processing domain. Early work in this area extracts

basic statistical features or geometric central moments straight from raw byte layouts to isolate malicious payloads. By viewing file structures through mathematical feature spaces like Gray-Level Co-occurrence Matrices (GLCM), lightweight classifiers can pick up on subtle spatial anomalies without needing heavy computational power [35].

The shift toward industrial edge computing has also brought this type of opcode analysis into the Internet of Things (IoT) domain. Since IoT malware usually targets specific processor architectures, a common strategy is to use hybrid feature selection models to filter out redundant instructions. This keeps the final feature sets small enough to run efficiently on low-power edge nodes [36, 37]. For classification, some recent methods use multi-view deep networks that look at both structural statistics and sequential instruction strings at the same time [38], while others rely on LSTM blocks to capture dependencies across long execution sequences [30, 39].

With the growth of edge computing and TinyML [5], verifying whether these machine learning models can actually run on target hardware has become a major priority. Researchers frequently turn to open-source emulators like Renode [40] for pre-silicon prototyping and cycle-accurate performance evaluation on microcontrollers. These simulation environments make it possible to track Static RAM (SRAM) usage, Flash memory footprint, and exact inference latency without needing the physical hardware on hand, ensuring that the proposed models stay safely within embedded hardware system limits.

To minimize this operational overhead, several static frameworks prioritize feature optimization over heavy deep learning networks. For instance, Yao [41] converted Dalvik opcode sequences into Markov-based grayscale images, relying on a lightweight CNN regularized by knowledge distillation to maintain a small footprint. Similarly, Şahin [42] bypassed deep architectures entirely by applying multiple linear regression and bagging methods to extracted Android permissions. However, a major limitation remains: even these explicitly lightweight static frameworks rarely provide empirical hardware profiling or emulation data to substantiate their claims of edge-deployment feasibility.

We address this validation gap by designing a lightweight framework that avoids deep learning models entirely, focusing instead on traditional machine learning classifiers. Drawing inspiration from biomedical time-series analysis, our pipeline treats extracted opcode sequences as 1D and 2D discrete signals to minimize preprocessing overhead. Crucially, unlike previous theoretical models, we evaluate our classifiers using the Renode simulation platform to profile their physical deployment constraints. This hardware-in-the-loop evaluation provides concrete empirical data on Static RAM (SRAM) footprint, Flash memory utilization, and exact inference latency, proving the framework can run reliably within strict embedded system budgets without losing classification accuracy.

3 Methodology

The complete workflow of the proposed research is shown in Figure 1. The proposed workflow consists of four main stages (i) Data Preparation, (ii) Data Preprocessing,

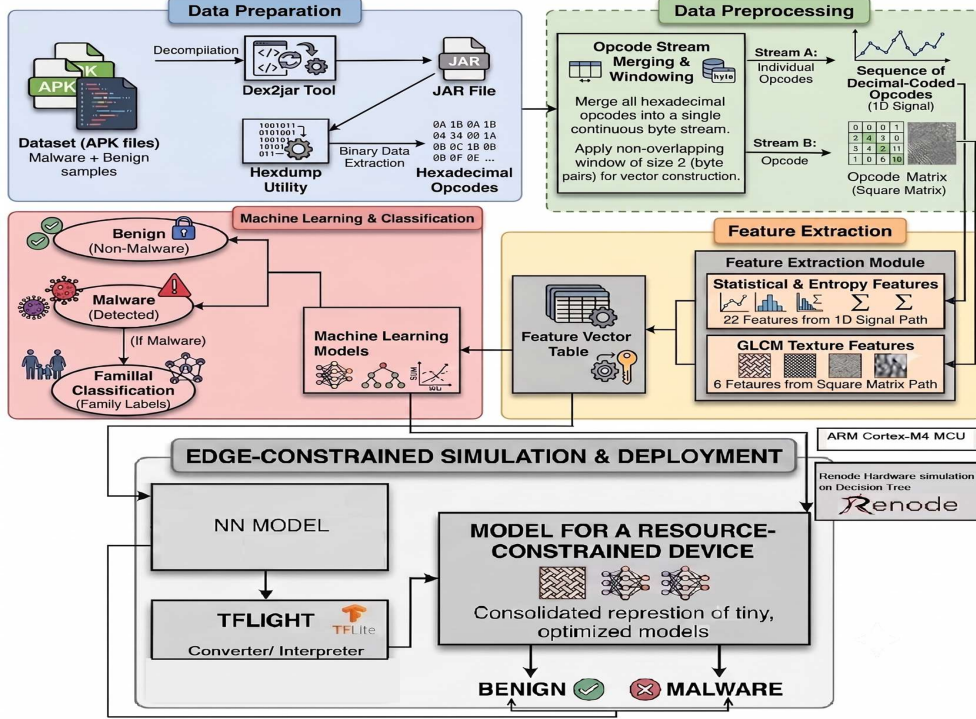


Fig. 1: Architecture of the proposed work

(iii) Feature Extraction, and (iv) Classification. The methodology involves extracting a range of statistical features, including Minimum, Maximum, Arithmetic Mean, Median, Mode, Sum, Standard Deviation, Variance, Skewness, Kurtosis, Shannon Energy, Log Energy, and 7 Central Moments. Additionally, Gray Level Co-occurrence Matrix (GLCM) features such as Contrast, Dissimilarity, Energy, Correlation, Homogeneity, and Angular Second Moment (ASM) are considered. In total, 28 statistical features are extracted from each *apk* file. These extracted features are used to train classifiers to classify an *apk* as benign or malware.

3.1 Data Preparation

We first convert raw *apk* files into the Java Archive (JAR) format using the Dex2Jar tool to unpack the compiled Dalvik Executable files. The resulting JAR files are then represented as hexadecimal instruction sequences by using the Hexdump utility. Further, these hexadecimal opcode strings are mapped into a 1D continuous numeric array for feature extraction.

3.2 Data Preprocessing

The 1D continuous numeric array representation of the hexadecimal opcodes of the JAR files a non-overlapping window of size two to partition the string into distinct,

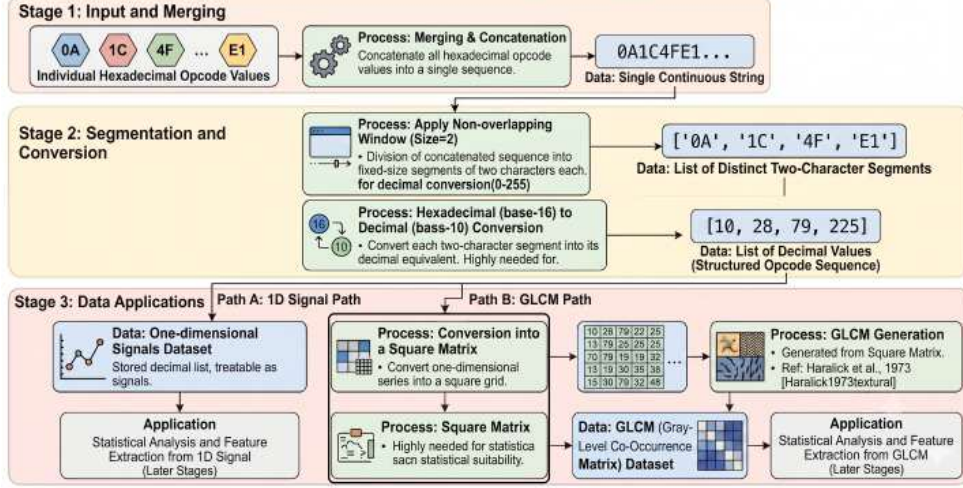


Fig. 2: Data Preprocessing and Transformation Pipeline

two-character segments. We convert each segment into its base-10 decimal equivalent and compile them into a structured list. Stages 1 and 2 of Figure 2 illustrate this preprocessing workflow. We treat the resulting decimal sequence as a 1D signal and extract 22 distinct features: Minimum, Maximum, Arithmetic Mean, Median, Mode, Sum, Standard Deviation, Variance, Skewness, Kurtosis, Shannon Energy, Log Energy, and Central Moments (1 to 7). We also reshape each 1D series into a square matrix to extract Grey-Level Co-occurrence Matrix (GLCM) features [35]. The GLCM-based features capture the spatial relationships and structural textures of the opcode sequence. The details of the GLCM-based feature extraction is shown in stage 3 of Figure 2.

3.3 1D opcode Feature Extraction

The preprocessed 1D discrete opcode signal are represented as $x[n]$, where $n = 1, 2, \dots, N$, where N represents the total number of samples in the dataset. Four different types of features based on (i) statistics of the opcodes, (ii) central moment, (iii) entropy and (iv) linear predictive analysis are extracted from this 1D discrete signal. The total count of the features are 22.

3.3.1 Statistical features

The time-domain statistical features extracted from the 1D signal is detailed below.

1. *Arithmetic Mean* ($\bar{x}$): is defined as in equation 1. The arithmetic mean represents the distinct baseline frequencies of core instructions. The malware containing packed binaries or cryptojacking modules deviate significantly from normal baselines values and helps differentiate malware from benign *apk*.

Algorithm 1: End-to-End Android Malware Detection Pipeline

Input : Raw *apk* dataset $D = \{apk_1, \dots, apk_M\}$, LPC prediction order $p = 8$, Sliding window length $W = 100$, Frame overlap $S = 50$, Train-test split ratio $\alpha = 0.80$

Output: Classification Performance Metrics Matrix $\mathbf{R}$

```
1 foreach apk_file  $\in D$  do
2   jar_file  $\leftarrow$  Dex2Jar(apk_file)
   raw_hex  $\leftarrow$  Hexdump(jar_file)
3    $X \leftarrow$  ExtractDecimalOpcodeSignal(raw_hex)

   ; // Multi-Domain Feature Extraction
4    $f_{stat} \leftarrow$ 
   [Mean( $X$ ), Median( $X$ ), Mode( $X$ ), Min( $X$ ), Max( $X$ ), Sum( $X$ ), Var( $X$ ), Std( $X$ ), Skew( $X$ ), Kurt( $X$ )]
    $f_{moment} \leftarrow$  ComputeCentralMoments( $X$ , order =  $n$ )
    $f_{entropy} \leftarrow$  [ShannonEnergyEntropy( $X$ ), LogEnergyEntropy( $X$ )]

   ; // Short-Time LPC Analysis
5    $e \leftarrow []$ 
   for  $n \leftarrow p + 1$  to Length( $X$ ) do
6      $\hat{x}[n] \leftarrow -\sum_{k=1}^p a_k X[n - k]$ 
      $e[n] \leftarrow X[n] - \hat{x}[n]$ 
7    $\mathbf{E} \leftarrow$  FrameResiduals( $e$ , length =  $W$ , overlap =  $S$ )
    $f_{lpc} \leftarrow$  [Mean( $\mathbf{E}$ ), Variance( $\mathbf{E}$ ), Sum( $\mathbf{E}$ )]

   ; // 2D Spatial Matrix & GLCM Extraction
8    $p\_dim \leftarrow \lfloor \sqrt{\text{Length}(X)} \rfloor$ 
    $\mathbf{I}_{2D} \leftarrow$  Reshape( $X[1 \dots p\_dim^2]$ ,  $p\_dim$ ,  $p\_dim$ )
    $\mathbf{P}_{glcm} \leftarrow$  GenerateGLCM( $\mathbf{I}_{2D}$ ,  $d = 2$ ,  $\theta = 0^\circ$ )
    $f_{glcm} \leftarrow$ 
   [ASM( $\mathbf{P}_{glcm}$ ), Energy( $\mathbf{P}_{glcm}$ ), Contrast( $\mathbf{P}_{glcm}$ ), Dissim( $\mathbf{P}_{glcm}$ ), Homog( $\mathbf{P}_{glcm}$ ), Corr( $\mathbf{P}_{glcm}$ )]

9    $\mathbf{v}_{apk} \leftarrow [f_{stat}, f_{moment}, f_{entropy}, f_{lpc}, f_{glcm}]$ ; // Feature Fusion (Dim: 25)
10  Append  $\mathbf{v}_{apk}$  to Global Feature Matrix  $\mathbf{M}$ 

   ; // Supervised Evaluation Framework
11 { $\mathbf{M}_{train}, \mathbf{M}_{test}$ }  $\leftarrow$  StratifiedSplit( $\mathbf{M}$ , ratio =  $\alpha$ )
    $\mathcal{C} \leftarrow$  {DT, KNN, Gaussian NB, , SVM}

12 foreach classifier  $\in \mathcal{C}$  do
13   Train(classifier,  $\mathbf{M}_{train}$ )
    $predictions \leftarrow$  Predict(classifier,  $\mathbf{M}_{test}$ )
    $\mathbf{R}[i]$   $\leftarrow$  EvaluateMetrics( $predictions, \mathbf{M}_{test}$ )

14 return  $\mathbf{R}$ 
```

$$\bar{x} = \frac{1}{N} \sum_{n=1}^N x[n] \quad (1)$$

2. *Median*: The median shows the exact geometric center of the signal's amplitude distribution. A comparison of the mean and the median tells whether the signature shift is systemic across the application logic, as is mostly the case in a benign file, or is asystematic and driven by localized blocks of dense malicious payload code. It is computed as:

$$\text{Median} = \begin{cases} x_{[\frac{N+1}{2}]}, & \text{for } N \text{ odd} \\ \frac{1}{2} (x_{[\frac{N}{2}]} + x_{[\frac{N}{2}+1]}), & \text{for } N \text{ even} \end{cases} \quad (2)$$

where the sequence $x[n]$ is assumed to be sorted in ascending order.

3. *Variance (s^2) and Standard Deviation (s)*: The variance represents the dispersion of the codes about the baseline. Benign *apk* files exhibit steady variance due to modular execution loops, whereas malware with injected code introduces sudden, volatile bursts of unique instructions, showing massive variance peaks. The variance is expressed as:

$$s^2 = \frac{1}{N-1} \sum_{n=1}^N (x[n] - \bar{x})^2 \quad (3)$$

The standard deviation $s = \sqrt{s^2}$ scales this dispersion linearly, establishing statistical boundaries for pattern detection.

4. *Skewness*: The skewness value is used to evaluate the structural asymmetry of the opcode distribution. Malware often compresses specialized routines, such as privilege escalation or obfuscation loops, into compact segments. This compression produces high positive or negative directional skewness that clearly deviates from standard benign *apk* file opcode distributions. It is formulated as:

$$\text{Skewness} = \frac{\frac{1}{N} \sum_{n=1}^N (x[n] - \bar{x})^3}{s^3} \quad (4)$$

5. *Excess Kurtosis*: We compute Excess Kurtosis to measure the impulsiveness or heavy-tailed behavior of the signal, determining whether variance spikes stem from systemic variations or infrequent, extreme amplitude deviations. For classification models, high kurtosis serves as a strong indicator of heavy-tailed code anomalies, such as localized reflection API calls or native library exploits embedded inside a normal application. It is defined as:

$$\text{Excess Kurtosis} = \frac{\frac{1}{N} \sum_{n=1}^N (x[n] - \bar{x})^4}{s^4} - 3 \quad (5)$$

3.3.2 n-th Central Moment-based features

The n -th central moment (m_n) maps higher-order geometric changes in the opcode distribution. For our discrete sequence, it is defined as:

$$m_n = \frac{1}{N} \sum_{i=1}^N (x_i - \mu)^n, \quad n \in \{1, 2, \dots, 7\} \quad (6)$$

Attacks frequently introduce dummy instructions, such as NOPs, to evade machine learning-based classification models. Central moments are invariant to these modifications because the moment around a distribution X is identical to that of $X + c$ for any scalar constant c . Consequently, while the insertion of NOP instructions by malware developers increases code size, it leaves the central moments unchanged, making them highly robust features for distinguishing malware from benign *apk* files.

3.3.3 Entropy-based features

While time-domain statistical moments extract features from the structural configurations of opcodes, code complexity and randomness-related properties are inherently captured by information-theoretic entropy measures. Attackers frequently employ packing, encryption, or string obfuscation techniques to bypass static scanning, a process that severely disrupts the structural layout of the resulting binary. To quantify these characteristics, we transform opcode amplitudes into normalized energy probabilities $p[n]$ (subject to $\sum_{n=1}^N p[n] = 1$) to derive two distinct entropy vectors.

1. *Shannon Energy Entropy*: Shannon energy entropy evaluates the global uniformity or spectral flatness of a signal's energy distribution. Benign *apk* files typically exhibit lower entropy owing to structured and predictable execution patterns. Conversely, obfuscated or packed malware maximizes signal disorder, yielding elevated Shannon entropy markers making it a highly effective feature for distinguishing malware from benign *apk* files. It is mathematically defined as:

$$H_{\text{Shannon}} = - \sum_{n=1}^N p[n] \log_e(p[n]) \quad (7)$$

2. *Log Energy Entropy*: Log energy entropy isolates subtle variations within low-power signal states. Whereas Shannon entropy is primarily driven by high-energy instruction blocks, log energy entropy accentuates micro-variations inside low-activity regions. This property enables the classification model to detect stealthy, low-frequency malicious behaviors, such as background Trojan threads, which would otherwise be masked by the dominant main-thread application logic. It is mathematically defined as:

$$H_{\text{Log}} = \sum_{n=1}^N \log_e(p[n]^2) \quad (8)$$

3.3.4 Linear Predictive Analysis (LPA)-based features

To extract features that characterize the underlying physical dynamics and deterministic transitions within the code sequence, we employ Linear Predictive Coding (LPC) analysis. While standard statistical moments treat the code stream as an independent distribution, LPC explicitly models the sequential, time-series dependencies of the execution flow. Specifically, the linear predictor formulates the current code sample as a linear combination of its p past instruction states, where p denotes the LPC prediction order. In this study, we utilize an empirical prediction order of $p = 8$ to capture the primary structural envelope and execution flow of the application without inducing overfitting or escalating computational overhead. The predicted sample $\hat{x}[n]$ is mathematically defined as:

$$\hat{x}[n] = - \sum_{k=1}^p a_k x[n-k] \quad (9)$$

where a_k represents the optimized LPC predictor coefficients computed via the Levinson-Durbin recursion algorithm.

We formally define the prediction mismatch as the LPC error or residual sequence $e[n]$. This sequence isolates the unmodeled execution source from the regular, predictable structural characteristics of the signal:

$$e[n] = x[n] - \hat{x}[n] = x[n] + \sum_{k=1}^p a_k x[n-k] \quad (10)$$

We use $e[n]$ as a highly sensitive anomaly detector. Because we optimize the LPC filter coefficients to match standard, highly repetitive benign instruction patterns, any sudden injection of malicious payload logic, packed loops, or shellcode severely violates these prediction rules, triggering prominent spikes in the residual energy.

To capture the time-varying nature of the signal under localized stationary conditions, we process the full sequence using a short-time framing approach with a localized sliding window framework. We utilize a fixed window length of $W = 100$ samples to guarantee statistical significance within each frame, paired with an overlapping frame shift step size of $S = 50$ samples. This 50% overlap configuration provides high temporal resolution while preserving structural continuity across code block boundaries.

Let $\mathbf{E} = [e_1, e_2, \dots, e_N]$ denote the resulting short-time LPC error feature vector containing localized metrics collected across N total frames. we extract three statistical descriptors from $\mathbf{E}$:

1. *LPC Error Mean (μ_e)*: The LPC Error Mean is utilized to quantify the overall systemic prediction bias and structural model-matching accuracy over time. For highly deterministic, benign applications, the optimized predictor exhibits an error mean approaching zero. Conversely, malware binaries that frequently disrupt execution patterns or incorporate heavily obfuscated modules introduce a high baseline prediction bias, thereby enabling the classification model to robustly differentiate between them. It is mathematically formulated as:

$$\mu_e = \frac{1}{N} \sum_{i=1}^N e_i \quad (11)$$

2. *LPC Error Variance (σ_e^2)*: The LPC Error Variance quantifies the temporal fluctuations and volatility of the residual energy across consecutive frames. Standard benign *apks* typically manifest a stable, uniform residual variance owing to their modular and tightly structured execution paths. Conversely, malware *apks* frequently integrate irregular API calling loops, permission escalations, or concealed payload drops inducing massive, erratic swings in the prediction error, resulting in pronounced variance peaks. It is mathematically expressed as:

$$\sigma_e^2 = \begin{cases} \frac{1}{N} \sum_{i=1}^N (e_i - \mu_e)^2, & \text{for } N > 1 \\ 0.0, & \text{for } N \leq 1 \end{cases} \quad (12)$$

3. *LPC Error Sum (S_e)*: The Cumulative Sum quantifies the total unmodeled structural energy retained across the entire duration of the execution sequence. Because this feature provides a global metric of execution anomaly density, it effectively thwarts adversarial attempts to bypass malware detectors. Even if an attacker distributes a malicious payload sparsely across a binary to evade localized detection, the aggregated deviation inevitably accumulates into a pronounced energy pool that exposes the malicious nature of the application. It is mathematically defined as:

$$S_e = \sum_{i=1}^N e_i = N \cdot \mu_e \quad (13)$$

3.4 2D opcode features via GLCM

While 1D statistical moments and linear predictive filters capture global sequence dynamics, they often fail to isolate localized structural layouts and instruction spacing. To overcome this limitation and expose complex instruction-to-instruction geometric patterns, we extract spatial texture features from a Gray-Level Co-occurrence Matrix (GLCM) [35]. We execute the transformation from a raw 1D opcode stream to highly discriminative 2D spatial descriptors. We extracted 6 features using GLCM.

- 1) *Angular Second Moment (ASM)*: The Angular Second Moment (ASM) is used to measure the global uniformity and orderliness of the opcode sequence. Packed, encrypted, or obfuscated malware *apks* show highly uniform, flat distributions across their byte streams, yielding very low ASM values. Conversely, standard benign *apks* feature repetitive, tightly organized modular loops that generate distinct probability clusters, producing sharp peaks and high ASM values. It is defined as:

$$ASM = \sum_{i=0}^{255} \sum_{j=0}^{255} p(i, j)^2 \quad (14)$$

- 2) *Energy*: The Energy feature is computed as the square root of the ASM to map structural uniformity back to a linear probability scale. This metric serves as a

reliable indicator of instruction predictability. Consequently, a chaotic or highly randomized payload attenuates the energy metric, whereas deterministic execution baselines maximize it. It is mathematically formulated as:

$$\text{Energy} = \sqrt{\text{ASM}} = \sqrt{\sum_{i=0}^{255} \sum_{j=0}^{255} p(i, j)^2} \quad (15)$$

- 3) *Contrast*: We compute Contrast to measure the local intensity variations present within the matrix, serving as an indicator of instruction transitions. High contrast indicates frequent, sharp jumps between wildly different opcode values, which typically flags polymorphic code behavior or packed block boundaries. Conversely, low contrast implies smooth, gradual transitions typical of a steady benign *apk* file execution sequence. It is given by:

$$\text{Contrast} = \sum_{i=0}^{255} \sum_{j=0}^{255} (i - j)^2 p(i, j) \quad (16)$$

- 4) *Dissimilarity*: The Dissimilarity metric quantifies local variation by scaling linearly, rather than quadratically, with respect to the distance from the main diagonal. This linear behavior enables the machine learning layer to detect localized, subtle variations in instruction pairs such as stealthy code injections within helper subroutines without being obscured by isolated, massive variance spikes. It is mathematically expressed as:

$$\text{Dissimilarity} = \sum_{i=0}^{255} \sum_{j=0}^{255} |i - j| p(i, j) \quad (17)$$

- 5) *Homogeneity*: Homogeneity measures the closeness of the distribution of GLCM elements to the main matrix diagonal. High homogeneity indicates that the instruction stream is dominated by pairs of highly similar or identical opcodes. This pattern is highly characteristic of repetitive malicious behaviors, such as NOP sled padding or heavy API calling loops. It is calculated as:

$$\text{Homogeneity} = \sum_{i=0}^{255} \sum_{j=0}^{255} \frac{p(i, j)}{1 + (i - j)^2} \quad (18)$$

- 6) *Correlation*: Correlation measures the linear dependencies between instruction pairs, mapping how strongly the occurrence of instruction i predicts the immediate arrival of instruction j . Benign software displays highly structured, predictable instruction pairs dictated by compiler rules, yielding high correlation scores. Conversely, malware payloads disrupt these standard patterns, causing a noticeable drop in correlation. It is defined as:

$$\text{Correlation} = \sum_{i=0}^{255} \sum_{j=0}^{255} \frac{(i - \mu_i)(j - \mu_j)p(i, j)}{\sigma_i \sigma_j} \quad (19)$$

where μ_i, μ_j and σ_i, σ_j represent the partial means and standard deviations of the row and column sums, respectively.

The features extracted from the 1D opcode sequence comprising statistical moments, entropy measures, and LPC error vectors yielded 22 dimensions, whereas the 2D spatial GLCM analysis contributed an additional 6 features. These features are subsequently concatenated to construct a unified feature matrix denoted as $\mathbf{M} \in \mathbb{R}^{M \times 28}$, where M represents the total number of analyzed *apk* files.

3.5 Classifiers

We evaluate our concatenated feature vector across five machine learning classifiers: Naïve Bayes (NB), Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Decision Tree (DT), and a 1 layer Feed-Forward Neural Network (NN).

The proposed neural network architecture comprises an input layer of 28 neurons, a single hidden layer containing 24 neurons regularized with a dropout rate of 0.4, and a singular output node. The hidden layer utilizes the Rectified Linear Unit (ReLU) activation function, while the output layer employs a Sigmoid activation function to compute the final binary classification probability. During the training phase, network optimization is driven by minimizing the binary cross-entropy loss function.

To deploy this NN classifier on resource-constrained edge devices, we quantize the network weights and activations to an 8-bit integer format. This optimization achieves an 90.93% model compression rate, drastically reducing the memory footprint to enable real-time inference.

To validate the bare-metal deployment performance of decision tree model, we simulate the compiled DT model directly on the Renode hardware emulation platform targeting an ARM Cortex-M4 microcontroller.

4 Results

To empirically validate our proposed lightweight malware detection framework, we executed an extensive evaluation pipeline using a combined dataset of 11,446 application packages, comprising 5,550 malicious samples from the Drebin benchmark and 5,896 benign samples from CICMalDroid2017 and CCCS-CIC-AndMal2020. We systematically evaluated the feature matrix to isolate the individual and combined predictive power of the 22 1D statistical features and 6 2D spatial features. Furthermore, we expanded our evaluation beyond structural metrics by tracking empirical computing constraints, recording both model size on disk and average inference time per sample to evaluate suitability for resource-constrained environments.

Our evaluation encompasses four classical Machine Learning (ML) classifiers—Gaussian Naïve Bayes (NB), Support Vector Machine with an RBF kernel (SVM), K-Nearest Neighbors (KNN), and a Decision Tree (DT) alongside our proposed lightweight Feed-Forward Neural Network (NN).

4.1 Performance Using 22 1D Features

Our first experiment evaluates classifier performance using only the 22 features extracted from the 1D signal. The upper portion of Table 1 presents these results. We observe that the KNN classifier achieves the highest baseline classification accuracy

at 96.3755%. However, it demands an unsustainable model footprint of 1646.44 KB, rendering it impractical for embedded edge endpoints. Because the KNN architecture must persistently retain all training reference data in system memory to calculate geometric distance constraints during live inference, expanding the training dataset will inevitably scale the physical size of the model.

The next best performers are the DT and NN architectures, where the DT achieves an accuracy of 95.5022% and the NN yields 94.5415%. While the DT model proves superior to the NN in both raw accuracy and processing speed per sample, the NN model establishes a clear advantage in physical storage. Our optimized NN requires an extremely lean footprint of just 33.17 KB, outperforming the uncompressed DT storage requirements by over 21%.

4.2 Performance Using 6 2D Spatial Features

Next, we evaluate the isolated contribution of the 6-dimensional Gray-Level Co-occurrence Matrix (GLCM) feature set. The middle portion of Table 1 presents these results. Because this spatial feature subset lacks global statistical context, we observe a predictable minor drop in overall performance metrics across most paradigms except NB. The KNN classifier remains impractical for edge nodes due to the memory retention limitations discussed earlier.

The next best performers are the DT and NN architectures. Although the raw performance of these models is inferior to the results achieved with the 1D features, the GLCM subset proves surprisingly robust. Even when operating with just six global spatial feature inputs, the GLCM features provide sufficient relational context to yield highly competitive diagnostic classifications.

4.3 Performance Using the Combined 28 Features

In our primary evaluation, we fused the 1D statistical baseline and 2D spatial feature descriptors into a unified 28-dimensional vector. This fusion pairs the global execution baseline of the application with localized structural refinements, allowing our models to capture both overt behavioral deviations and stealthy, obfuscated payload drops. We find that combining features maximizes the decision boundaries of our high-capacity models.

The KNN classifier reaches the absolute statistical upper bound with an accuracy of 97.3362% and an F1-score of 0.972682, but its memory requirement balloons to an unsustainable 2075.71 KB. The DT model achieves an accuracy of 96.3319% and an F1-score of 0.961609, maintaining a physical model size of 43.13 KB and an extraordinary average latency of just 0.00069 ms. Our proposed NN achieves an accuracy of 95.7642% and an F1-score of 0.956870, demanding an average latency of 0.09074 ms. Ultimately, the Decision Tree secures the fastest average inference speed of 0.00069 ms in our entire evaluation suite, followed by the proposed NN.

In Figure 3 and 4, we plot the confusion matrices with test data for DT and NN models. We observed that our proposed NN architecture restricts critical classification errors to a narrow margin, successfully separating benign *apks* from packed malicious binaries. While the DT model achieved slightly higher false negatives due to a lower

Table 1: Comprehensive evaluation tracking across independent and combined malware feature spaces

Model Paradigm	Acc.	Prec.	Rec.	F_1 -Score	Size(KB)	Lat. (ms)
Isolated 1D Statistical Features ($N = 22$)						
Naïve Bayes	0.860699	0.943860	0.748609	0.834972	1.88	0.00104
SVM (RBF)	0.949345	0.916090	0.982375	0.948075	255.89	0.08296
KNN	0.963755	0.939100	0.987013	0.962460	1646.44	0.07133
Decision Tree	0.955022	0.951807	0.952690	0.952248	42.42	0.00067
Proposed NN	0.945415	0.914708	0.974954	0.943871	33.17	0.09595
Isolated 2D GLCM Spatial feature Features ($N = 6$)						
Naïve Bayes	0.892576	0.926230	0.838590	0.880234	1.29	0.00067
SVM (RBF)	0.937555	0.893188	0.985158	0.936921	104.43	0.06863
KNN	0.952838	0.916667	0.989796	0.951829	1067.07	0.01270
Decision Tree	0.950655	0.938238	0.958256	0.948141	51.07	0.00060
Proposed NN	0.933188	0.896996	0.969388	0.931788	30.17	0.09703
Combined Fused Multi-Domain Feature Space ($N = 28$)						
Naïve Bayes	0.872052	0.956965	0.768182	0.852244	2.16	0.00120
SVM (RBF)	0.964192	0.944930	0.982727	0.963458	325.29	0.09252
KNN	0.973362	0.958517	0.987273	0.972682	2075.71	0.05818
Decision Tree	<i>0.963319</i>	<i>0.966912</i>	<i>0.956364</i>	<i>0.961609</i>	<i>43.13</i>	<i>0.00069</i>
Proposed NN	0.957642	0.936466	0.978182	0.956870	31.12	0.09074

Note: Acc.: Accuracy; Prec.: Precision; Rec.: Recall; Size: On computer RAM; Lat.: Average Inference Latency.

recall rate (0.95), it generated a larger number of false positives compared to our network.

Additionally, we plotted the Receiver Operating Characteristic (ROC) curves in Figure 5 and 6 to analyze the true-positive to false-positive trade-off across varying classification thresholds. Both our proposed NN and the DT achieved near-perfect Area Under the Curve (AUC) scores, significantly outperforming the baseline probabilistic models. This optimal AUC trajectory confirms that our fused multi-domain feature set provides highly robust, separable decision boundaries regardless of the operational threshold applied during inference.

4.4 Embedded Hardware Emulation

A core claim of this research is that our framework can run directly on resource-constrained edge and IoT devices. We compile and emulate both models on a 32-bit ARM Cortex-M4 microcontroller running at 168 MHz. Both frameworks fit comfortably within the tight physical limits of edge devices. The detailed comparison is provided in Table 2. Post-training INT8 quantization heavily reduces the size of the Neural Network to just 2.82 KB of Flash memory. The storage requirements of NN model beats the compiled Decision Tree binary, which requires 4.58 KB of static *.text* storage space. However, the two models handle volatile runtime memory very differently. The Decision Tree uses zero static SRAM because it leaves its *.data* and *.bss*

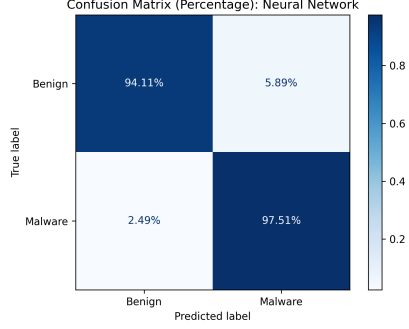


Fig. 3: Confusion Matrix of the Proposed NN utilizing the fused 28-dimensional space.

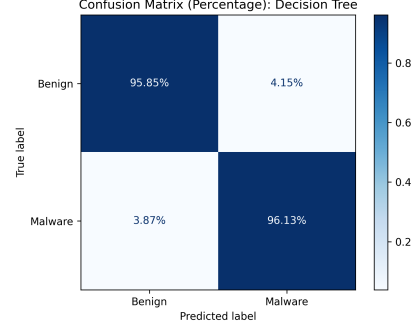


Fig. 4: Confusion Matrix of the DT classifier utilizing the fused 28-dimensional space.

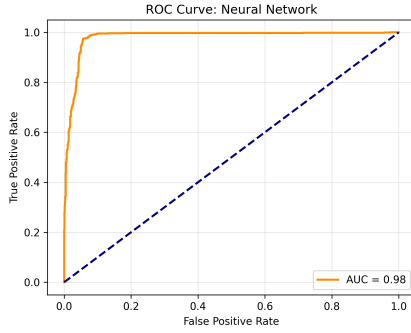


Fig. 5: ROC-AUC curve of the Proposed NN showcasing discriminative performance thresholds.

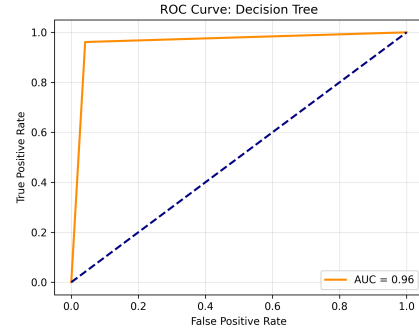


Fig. 6: ROC-AUC curve of the DT classifier showcasing discriminative performance thresholds.

segments unallocated. The quantized Neural Network requires a small peak workspace of 0.83 KB of SRAM during active inference.

Both frameworks achieve exceptional sub-microsecond execution speeds. The Decision Tree uses a highly optimized instruction to records an ultra-low latency of 0.000173 ms (173 nanoseconds). The quantized Neural Network leverages hardware-accelerated integer multiply-accumulate operations to give a latency value of 0.000009 ms (9 nanoseconds). This incredible throughput enables instant, hardware-level malware filtering at the edge.

4.5 Statistical Validation

We ran both models through 20 independent execution splits to test their reliability. The Decision Tree model achieves a slightly higher overall classification accuracy of $96.02\% \pm 0.41\%$. The Neural Network follows closely with an accuracy of $95.47\% \pm$

Table 2: Comprehensive Edge Benchmarking Profile on ARM Cortex-M4 (168 MHz)

Evaluation Dimension	Decision Tree (DT)	Proposed INT8 NN
Flash Storage Size (KB)	4.58	2.82
Peak Active SRAM (KB)	0.00	0.83
Inference Latency (ms)	0.000173	0.000009

0.45%. The Decision Tree also shows a superior precision score of 0.9578 compared to the Neural Network’s score of 0.9339. This means the Decision Tree generates fewer false-positive alerts. However, malware detection systems must prioritize catching every live threat. The proposed Neural Network shines in this area. It achieves a higher Recall score of 0.9754 ± 0.0057 , while the Decision Tree scores 0.9602 ± 0.0051 . This higher recall means the Neural Network catches a broader spectrum of malicious files. These differences balance out across the broader metrics. Both architectures deliver nearly identical F1-Scores of 0.9590 for the Decision Tree and 0.9541 for the Neural Network. This proves that both options offer excellent algorithmic reliability.

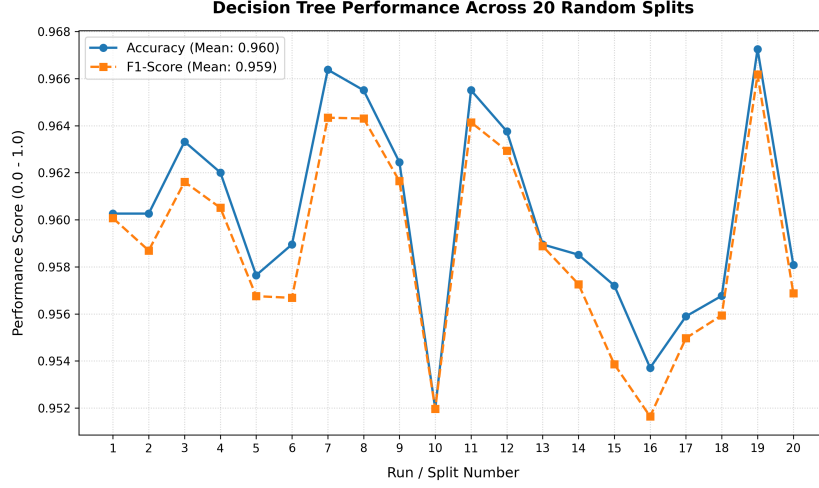
Table 3: Comparative Statistical Robustness Profile over 20 Independent Runs

Model	Metric	Mean (μ)	Standard Deviation (σ)
Proposed NN	Precision	0.9339	± 0.0092
	Recall	0.9754	± 0.0057
	F1-Score	0.9541	± 0.0043
	Accuracy (%)	95.47	$\pm$ 0.45
Decision Tree	Precision	0.9578	± 0.0063
	Recall	0.9602	± 0.0051
	F1-Score	0.9590	± 0.0041
	Accuracy (%)	96.02	$\pm$ 0.41

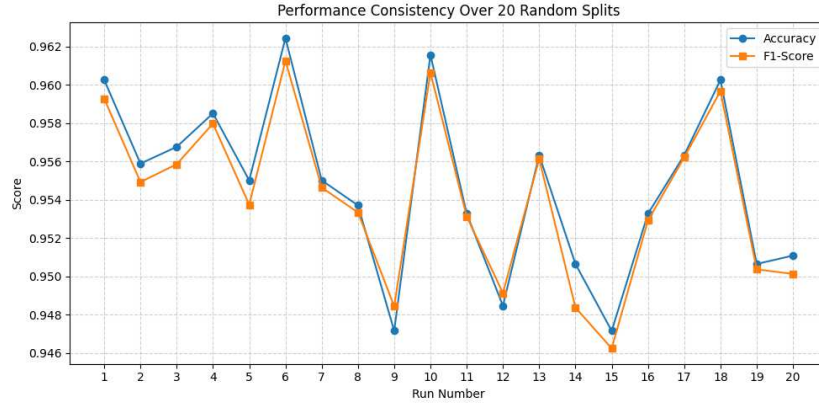
While Table 3 provides the aggregated statistical parameters, Figure 7 visually maps the trajectory of the Accuracy and F1-Score across the 20 independent execution runs. The visualization confirms the structural stability of the proposed framework. The narrow performance band—free from aggressive performance dips or erratic variance spikes—proves that the multi-domain feature extraction pipeline captures immutable features for classifying *apk* files. Consequently, the high classification accuracy is a result of consistent feature modeling rather than favorable data partitioning or over-fitting to a specific training subset.

5 Discussion

The major finding of this research is that abstracting raw Android bytecode into 1D and 2D opcode signals creates highly separable decision boundaries for malware and



(a) Decision Tree Baseline



(b) Proposed NN Model

Fig. 7: Comparative performance variance tracking Accuracy and F1-Score across 20 independent execution runs, validating the structural stability and generalization profiles of both models in a top-down configuration.

benign in Android *apks*. The proposed approach allows our models to isolate malicious behaviors without relying on heavy reverse-engineering workflows.

The second major finding of this research is that our 28-dimensional multi-domain feature set maximizes the classification boundaries of high-capacity models. Specifically, K-Nearest Neighbors (KNN) yields the highest absolute classification performance across our entire evaluation suite. In resourceful execution environments,

such as centralized servers or cloud-based analysis pipelines, KNN serves as the optimal choice to identify malware in the Android domain.

The third finding reveals that KNN is entirely unsuitable for resource-constrained edge devices. The massive storage requirements of the KNN model make it incompatible with low-storage hardware, as it must persistently retain large reference data matrices in the local system memory.

Conversely, our fourth finding identifies the Decision Tree (DT) as the best deployment model for edge devices where inference latency is the primary constraint. The hardcoded DT model can infer a security decision in just 0.00069 ms, completely bypassing heavy interpreter runtimes.

Finally, our evaluation shows that the Feed-Forward Neural Network (NN) serves as the most effective choice when physical storage space is the primary constraint. Once optimized via post-training 8-bit integer quantization, the NN model minimizes the read-only flash storage footprint to a bare minimum while maintaining robust diagnostic accuracy.

Our bare-metal **Renode hardware simulation** on an ARM Cortex-M4 based STM32F4 microcontroller running at 168 MHz empirically confirms this lightweight performance profile. Cross-compilation object parsing via `arm-none-eabi-size` revealed that our hardcoded Decision Tree consumes a minor 4580 Bytes of read-only Flash storage (*text* section) and a perfect score of 0 **Bytes** of initialized static RAM (*data* section). Crucially, the model also achieved 0 **Bytes** of uninitialized static storage (*bss* section). Because the tree executes using purely local processor registers and dynamic stack frames (measured under stack painting at less than 128 Bytes), it eliminates persistent global memory overhead. This hardware-level optimization proves that our framework can easily run on ultra-low-power, legacy 8-bit or 16-bit microcontrollers, establishing high feasibility for widespread decentralized IoT security.

Our feature-ablation experiments provide critical insight into the structural role of 1D and 2D feature fusion. When we completely removed the 2D GLCM spatial feature properties from the dataset, the classification accuracy for our selected DT dropped marginally from 96.3319% down to 95.5022% (a minor decrease of 0.8297%). This indicates that while baseline time-domain statistics and localized short-time LPC residuals bear the primary responsibility for drawing major class boundaries, the 2D GLCM parameters serve as an indispensable architectural refinement layer. By capturing higher-order geometric instruction spacing, these feature metrics provide the precise decision-boundary adjustments necessary to capture obfuscated or packed malicious payloads that might otherwise bypass global statistical aggregations.

Furthermore, our multi-class familial tracking experiment demonstrated that the proposed model handles granular classification effectively. When evaluating the Decision Tree model are detailed in Table 4, it categorized 15 distinct malware lineages with a global accuracy of 87.07%. Highly repetitive malware strains like *FakeInstaller* achieved an excellent class-specific F_1 -scores (0.91), whereas sparse, stealthy variants like *Adrd* scored significantly lower (0.21). In comparison, the Neural Network architecture yielded a lower global accuracy of 84.55%. This performance variance between models, alongside the low scores for stealthy strains, points to a data-density limitation rather than an architectural flaw in our feature mapping pipeline.

Table 4: Malware Familial Classification Profile under an Decision Tree Configuration

S. No.	Malware Family Strains	Precision	Recall	F_1 -Score	Support
1	Adrd	0.21	0.27	0.24	22
2	BaseBridge	0.82	0.78	0.80	104
3	DroidDream	0.45	0.61	0.52	28
4	DroidKungFu	0.67	0.66	0.67	196
5	ExploitLinuxLotoor	.25	0.33	0.29	21
6	FakeDoc	0.75	0.61	0.67	49
7	FakeInstaller	0.91	0.90	0.91	298
8	Geinimi	0.44	0.33	0.58	36
9	GinMaster	0.44	0.49	0.47	99
10	Glodream	0.44	0.46	0.45	24
11	Iconosys	0.67	0.70	0.68	46
12	Kmin	0.70	0.79	0.74	38
13	MobileTx	0.72	0.62	0.67	21
14	Opfake	0.90	0.96	0.93	182
15	Plankton	0.87	0.85	0.87	195
16	Benign	0.97	0.96	0.97	1729

To validate the academic and operational competitiveness of this methodology, Table 5 provides a direct comparison against prominent state-of-the-art models evaluated on the identical Drebin reference benchmark repository.

Table 5: Comparative performance evaluation metrics against state-of-the-art baseline models on the Drebin dataset

Research Work	Year	Extracted Primitives / Classifier Engine	Accuracy (%)	Target Environment
Jiang et al. [43]	2020	Opcode Sequence / Deep Transfer Learning	92.26	Server / Cloud High-Compute
Mahindru et al. [44]	2021	Decompiled API Features / Deep Neural Network	98.80	Server / Cloud High-Compute
Aldini et al. [45]	2024	classes.dex + Android Manifest / 2D CNN	97.11	High-RAM Edge GPU Nodes
Proposed Framework	2026	JAR Decompilation / Hard-coded Decision Tree	96.33	Bare-Metal MCU TinyML

Jiang et al. [43] tracked raw opcode sequences through long deep-learning networks, achieving an accuracy of 92.26%. Aldini et al. [45] on the other hand, achieved a strong accuracy of 97.11% by feeding `classes.dex` and manifest string mappings into a 2D Convolutional Neural Network (CNN). However, routing high-resolution 2D feature grids through deep convolutional layers requires substantial memory, preventing deployment on bare-metal systems. Conversely, Mahindru et al. [44] recorded a higher accuracy of 98.80% by evaluating deep permission layouts via a heavy neural

framework. However, their architecture relies on large, complex feature spaces that require high-power server environments, which limits its practical use on low-power edge gateways. By contrast, our framework maps instruction flows to 1D and 2D signals to achieve an accuracy of 96.3319% with a footprint of less than 3 KB, enabling on-chip inference directly at the edge hardware layer.

Despite the strong performance and low hardware overhead of our proposed framework, we acknowledge several critical limitations in the work. Our 1D opcode signal extraction relies on static analysis of Dalvik bytecode. Highly sophisticated malware that employs runtime reflection, dynamic class loading, or native code execution (`.so` files) can bypass our static extraction pipeline. The framework relies on the `Dex2Jar` and `Hexdump` utilities to unpack and decode `apk` files. If an attacker applies anti-disassembly techniques or corrupts the ZIP headers of the `apk` to break these tools, our preprocessing pipeline will fail to generate the 1D signal. By evaluating only static binary structures, the model cannot observe actual runtime behaviors, such as unauthorized network connections or privilege escalation. This makes it less effective against fileless malware or benign apps that download malicious payloads post-installation.

6 Conclusion

This work highlights the clear need for detecting malware directly on edge devices instead of relying entirely on cloud architectures. To address this, we developed a lightweight malware detection framework designed specifically for resource-constrained hardware like Android TVs and set-top boxes. Because the system relies on just 28 features extracted from opcode sequences using Java Archive (JAR) bytecode abstraction, it maintains a very low processing footprint. By converting each `apk` file into sequential hexadecimal opcodes and processing them with a non-overlapping sliding window, our framework turns software binaries into 1D and 2D signals to build a compact 28-dimensional feature matrix.

We tested this feature matrix by comparing traditional machine learning algorithms against a single-layer Feed-Forward Neural Network (NN). Our results show that the Decision Tree (DT) and NN models can accurately separate malware from benign applications, achieving an average accuracy of 96.02% and 95.77%, respectively, over 20 independent runs. The framework also performed well when classifying specific malware families.

To make sure the system could actually run on embedded hardware, we applied post-training quantization to the neural network, converting it to an 8-bit integer (INT8) format using TensorFlow Lite (TFLite) Micro. We then cross-compiled the model for the ARM Cortex-M4 architecture with the `arm-none-eabi-gcc` toolchain. Testing the network in a bare-metal virtual environment on the Renode emulation platform showed that the quantized model runs smoothly even under tight embedded memory and processing limits.

In the future, we plan to improve this pipeline by parallelizing the string concatenation methods and optimizing the hexadecimal-to-decimal conversion routines. This will cut down on the time it takes to generate signals, making the approach fast enough for large, enterprise-grade workloads. We also want to look into extracting features

from other parts of the *apk* files, such as native libraries (*.so* files) and asset resources. Finally, we plan to introduce adversarial samples into the training loop to test the model’s resilience and find ways to harden it against sophisticated evasion techniques.

Declarations

Funding

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Trial Registration

Not applicable. This study is not a clinical trial.

Ethics approval and consent to participate

Not applicable.

Consent to Publish

Not applicable.

References

- [1] YM Hashem, A., A Osman, F., AR Eltokhy, M., S Gab Allah, A.: Effective home automation using android and embedded system. *Engineering Research Journal* **166**, 1–14 (2020)
- [2] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of Android malware in your pocket. In: *Network and Distributed System Security Symposium*, vol. 14, pp. 23–26 (2014)
- [3] Mantoo, B.A., Khurana, S.S.: Static, dynamic and intrinsic features based Android malware detection using machine learning. In: *International Conference on Recent Innovations in Computing* (2019), pp. 31–45 (2020). Springer
- [4] Lin, J., Zhu, L., Chen, W.-M., Wang, W.-C., Gan, C., Han, S.: On-device training under 256kb memory. In: *Advances in Neural Information Processing Systems*, vol. 35, pp. 22941–22954. Curran Associates, Inc., Red Hook, NY (2022)
- [5] Dehrouyeh, F., Shaer, I., Nikan, S., Ajaei, F.B., Shami, A.: Tinymml-enabled resource-efficient framework for real-time network security in electric vehicle charging networks. *IEEE Transactions on Network Science and Engineering* **13**, 5092–5109 (2026)

- [6] Kumar, R., Kumar, P., Tripathi, R., Gupta, G.P., Garg, S., Hassan, M.M.: A distributed intrusion detection system to detect ddos attacks in blockchain-enabled IoT network. *Journal of Parallel and Distributed Computing* **164**, 55–68 (2022)
- [7] Villar-Rodriguez, E., Pérez, M.A., Torre-Bastida, A.I., Senderos, C.R., López-de-Armentia, J.: Edge intelligence secure frameworks: Current state and future challenges. *Comput. Secur.* **130**(C) (2023)
- [8] Smmarwar, S.K., Gupta, G.P., Kumar, S., Kumar, P.: An optimized and efficient Android malware detection framework for future sustainable computing. *Sustainable Energy Technologies and Assessments* **54**, 102852 (2022)
- [9] Bakour, K., Ünver, H.M.: Deepvisdroid: android malware detection by hybridizing image-based features with deep learning techniques. *Neural Computing and Applications* **33**(18), 11499–11516 (2021)
- [10] Kim, J., Ban, Y., Ko, E., Cho, H., Yi, J.H.: MAPAS: a practical deep learning-based Android malware detection system. *International Journal of Information Security* **21**(4), 725–738 (2022)
- [11] Banik, A., Singh, J.P.: A BERT and PSO framework for Android malware detection using real permissions and API calls. *Discover Computing* **29**(1), 256 (2026)
- [12] Gonçalves, M., Paiva, A.C.: Reverse engineering of Android applications: Reimpact. In: *International Conference on the Quality of Information and Communications Technology*, pp. 369–382 (2020). Springer
- [13] Anand, A., Singh, J.P.: Role of smali opcodes in identifying android malware obfuscation variants. In: *2024 4th International Conference on Computer, Communication, Control & Information Technology (C3IT)*, pp. 1–6 (2024)
- [14] Ren, Z., Wu, H., Ning, Q., Hussain, I., Chen, B.: End-to-end malware detection for Android IoT devices using deep learning. *Ad Hoc Networks* **101**, 102098 (2020)
- [15] Almomani, I., Alkhayer, A., El-Shafai, W.: An automated vision-based deep learning model for efficient detection of Android malware attacks. *IEEE Access* **10**, 2700–2720 (2022)
- [16] Singh, J., Thakur, D., Gera, T., Shah, B., Abuhmed, T., Ali, F.: Classification and analysis of Android malware images using feature fusion technique. *IEEE Access* **9**, 90102–90117 (2021)
- [17] Anand, A., Singh, J.P., Singh, A.K.: Smali code-based deep learning model for Android malware detection. *The Journal of Supercomputing* **81**(4), 1–31 (2025)
- [18] Suo, D., Xue, L., Huang, W., Tan, R., Sun, G.: Assessing the capability of Android

dynamic analysis tools to combat anti-runtime analysis techniques. *Journal of Systems and Software*, 112747 (2025)

- [19] Zhang, S., Su, H., Liu, H., Yang, W.: MPDroid: A multimodal pre-training Android malware detection method with static and dynamic features. *Computers & Security* **150**, 104262 (2025)
- [20] Banik, A., Singh, J.P.: Android malware detection by correlated real permission couples using fp growth algorithm and neural networks. *IEEE Access* **11**, 124996–125010 (2023)
- [21] Anand, A., Singh, J.P.: Smali opcode-based Android malware detection and obfuscation identification. *Proceedings of the National Academy of Sciences, India Section A: Physical Sciences* (2026)
- [22] Mudunuru, K., Rani, M.U.: A hybrid deep-handcrafted feature fusion framework for image based Android malware detection. *Future Technology* **5**(2), 129–137 (2026)
- [23] Ullah, F., Ullah, S., Srivastava, G., Lin, J.C.-W.: Droid-mcfg: Android malware detection system using manifest and control flow traces with multi-head temporal convolutional network. *Physical Communication* **57**, 101975 (2023)
- [24] Ma, B., Zhou, L., Liao, C., Zhou, Y., Li, J., Ma, J.: Ransomsentry: runtime detection of Android ransomware with compiler-based instrumentation. *IEEE Transactions on Dependable and Secure Computing* **22**(4), 3354–3370 (2025)
- [25] Jaju, C., Agrawal, D., Poddar, R., Badjate, S., Anand, S., Mitra, B., Dey, S.: Analyzing image based strategies for Android malware detection and classification: An empirical exploration. In: *SECRYPT*, pp. 863–869 (2023)
- [26] Udeze, S., Rafiq, H., Jeremiah, D., Ta, V.-T., Usman, M.: Image-based Android malware detection using deep learning. In: *International Conference on Global Security, Safety, and Sustainability*, pp. 213–225 (2023). Springer
- [27] Tang, J., Xu, W., Peng, T., Zhou, S., Pi, Q., He, R., Hu, X.: Android malware detection based on a novel mixed bytecode image combined with attention mechanism. *Journal of Information Security and Applications* **82**, 103721 (2024)
- [28] Syed, T.A., Nauman, M., Khan, S., Jan, S., Zuhairi, M.F.: ViTDroid: Vision transformers for efficient, explainable attention to malicious behavior in Android binaries. *Sensors* **24**(20), 6690 (2024)
- [29] Yang, D., Ding, Y., Zhang, H., Li, Y.: PVitNet: An effective approach for Android malware detection using pyramid feature processing and vision transformer. In: *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2440–2444 (2024). IEEE

- [30] Subhameena, M., Rajesh, A.: LSTM-based malware classification using bytecode and opcode features for IoT security. In: Proceedings of the 8th International Conference on Computing, Control and Communication Systems (IC4S), pp. 45–51 (2025). IEEE
- [31] Saini, V., Gupta, R., Soni, N.: Opcode-based malware classification using machine learning and deep learning techniques. arXiv preprint [arXiv:2504.13408](https://arxiv.org/abs/2504.13408) (2025) [arXiv:2504.13408](https://arxiv.org/abs/2504.13408) [cs.CR]
- [32] Li, B., Yu, S., Zhao, Y., Sun, Y., Zhou, H., Zhang, Y.: Intelligent detection and forensics method for malware based on memory opcode genes. *Journal of King Saud University - Computer and Information Sciences* **38**(3), 109 (2026)
- [33] Arslan, R.S.: JDroid: Android malware detection using hybrid opcode feature vector. *PeerJ Computer Science* **11**, 3051 (2025)
- [34] Sharma, Y.K., Tomar, D.S., Pateriya, R.K., Bhandari, S.: MOSDroid: Obfuscation-resilient Android malware detection using multisets of encoded opcode sequences. *Computers & Security* **152**, 104379 (2025)
- [35] Priya, V., Sofia, A.S.: An efficient deep learning framework for malware image classification using gray-level co-occurrence matrix and sparse convolution. *Iranian Journal of Science and Technology, Transactions of Electrical Engineering* **49**(1), 65–88 (2025)
- [36] Ahmed, B.T., Jameel, N.G.M., Saeed, B.I.: A novel hybrid opcode feature selection framework for efficient and effective IoT malware detection. *IoT* **7**(1), 24 (2026)
- [37] Anand, A., Singh, J.P.: Generating hash value at edge device using sram-puf and autoencoder in iot network. *Proceedings of the Indian National Science Academy* **91**(3), 498–511 (2025)
- [38] Ambedkar, J.B., Sreenivasu, M., Gold, A.I., Uppalapu, V.K., Kalidindi, S., Krishna, T.V.S.: A multi-view deep learning framework for malware detection using opcode analysis. In: 2026 9th International Conference on Intelligent Computing and Control Systems (ICICCS), pp. 1702–1707 (2026). IEEE
- [39] Upadhyay, K., Dayal, N.: Opcode-based malware detection using lstm networks. In: *Synergies in Data Analytics and Cyber Security*, pp. 141–153. Springer, Singapore (2026)
- [40] Gielda, M.: Renode: A flexible, open-source simulation framework for building scalable, well-tested RISC-V systems. In: 7th RISC-V Workshop Proceedings (2017). Antmicro
- [41] Yao, L., Xin, Y., Liu, M., Zhang, Q.: A lightweight Android malware detection

framework based on markov images and knowledge distillation. *IEEE Internet of Things Journal* **13**(3), 5224–5241 (2026)

- [42] Şahin, D.Ö., Akleylek, S., Kiliç, E.: LinRegDroid: Detection of Android malware using multiple linear regression models-based classifiers. *IEEE Access* **10**, 14246–14259 (2022)
- [43] Jiang, Y., Li, R., Tang, J., Davanian, A., Yin, H.: Aomdroid: detecting obfuscation variants of Android malware using transfer learning. In: *Security and Privacy in Communication Networks: 16th EAI International Conference, SecureComm 2020, Washington, DC, USA, October 21-23, 2020, Proceedings, Part II* 16, pp. 242–253 (2020). Springer
- [44] Mahindru, A., Sangal, A.L.: MLDroid—framework for Android malware detection using machine learning techniques. *Neural Computing and Applications* **33**(10), 5183–5240 (2021)
- [45] Aldini, A., Petrelli, T.: Image-based detection and classification of Android malware through CNN models. In: *Proceedings of the 19th International Conference on Availability, Reliability and Security*, pp. 1–11 (2024)