2025 International Conference on Intelligent Computing
July 26-29, Ningbo, China
https://www.ic-icc.cn/2025/index.php

GCT-Net: A malicious Android application detection method based on multimodal feature fusion

Yuheng Huang[1], Weihao Huang[1], Chunhong Jiang[1], Song Xie[1], and Hongsong Wang[1]✉

School of Artificial Intelligence, South China Normal University, Foshan, China
whs2020@126.com

Abstract. Despite the persistent threat of malicious Android applications, many existing detection methods struggle to effectively integrate and analyze the heterogeneous threat indicators embedded within APKs. This fragmented analysis often fails to capture the complex interplay between different threat vectors. To address this challenge, we propose GCT-Net(GNN-CNN-Tree-LSTM-Net), a novel deep learning framework that synergistically fuses Graph, Convolutional, and Tree-structured features for unified malware detection. We first disassemble APKs via reverse engineering to derive three critical modalities: API call sequences modeled as directed graphs and processed by a Graph Neural Network (GNN) to capture semantic dependencies; binary code converted into greyscale images and analyzed via a two layers Convolutional Neural Network (2D-CNN) to detect spatial malware patterns; and URL strings parsed into syntax trees and encoded using a hierarchical Tree LSTM network to learn structural embeddings. These modality-specific features are adaptively integrated through three dense layers. Evaluated on two datasets, GCT-Net achieves state-of-the-art performance with 96.48%/93.75% accuracy, 97.62%/96.45% precision, 97.05%/95.40% re- call and 97.33%/95.42% F1-score, outperforming other models. Ablation studies confirm the critical contributions of all three modalities and validate the fusion efficacy, establishing a new method for multimodal malware analysis.

Keywords: Android Malware Detection · Multimodal Learning · Graph Neural Network · Tree-Structured LSTM

1 Introduction

With the rapid advancement of mobile internet technologies, the proliferation of malicious applications and URLs—especially those involved in fraud, gambling, and pornography—has emerged as a significant threat to the digital information society. These malicious entities often disguise themselves as legitimate applications or links, aiming to steal users' private data, defraud victims of funds, and disseminate illegal content. Such behavior not only severely undermines information security and financial stability but also disrupts social ethics and the rule of law[1]. Their diverse forms, sophisticated evasion strategies, and high concealment introduce considerable challenges to traditional detection methods. Therefore, timely and accurate identification of malicious apps is essential not only for protecting individual rights and public safety but also for fostering a trustworthy and secure cyberspace, thereby advancing the broader goals of cybersecurity research.

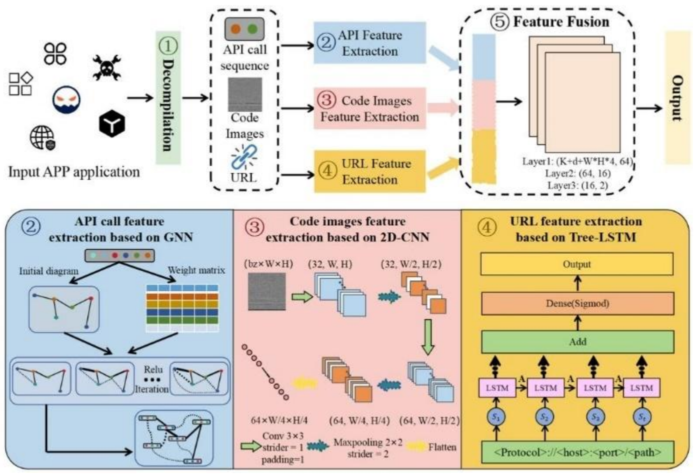

Fig. 1. Overview of our method (a)APK Decompilation: Uses apktool to extract and restructure Android application components. (b) API Call Graph Analysis: Implements GNN with dynamic edge weighting to detect suspicious call patterns. (c) Binary Code Visualization: Transforms APK bytecode into grayscale images processed by 2D-CNN. (d) URL Syntax Parsing: Employs attention-based Tree-LSTM to analyze URL structure and semantics. (e) Multimodal Fusion: Integrates all features through DNN-based alignment and cross-modal learning. This architecture synergistically combines static code analysis, behavior monitoring, and network traffic inspection for robust malware identification.

The complexity of modern malicious software necessitates a detection strategy that goes beyond single-perspective analysis. However, most existing approaches still rely on unimodal detection techniques, which are inherently limited by their dependence on isolated feature dimensions—such as API call traces[2], static code signatures[3], or network traffic patterns[4]—making them less effective when confronting increasingly cross-dimensional and evasive malicious behaviors. To overcome these limitations, we propose a multidimensional detection framework that integrates three complementary analytical perspectives: (1) **API Feature Extraction**, which dynamically captures runtime behaviors and uncovers sensitive or abnormal operations; (2) **Code Images Feature Extaction**, which reverse-engineers APK files to expose obfuscation tactics and hidden malicious logic; and (3) **URL Feature Extraction**, which inspects embedded communication flows to detect potential interactions with malicious command-and-control servers. Together, these three modalities encompass program logic, file-level structure, and network behavior, enabling a holistic and more accurate understanding of malicious activity.

In recent years, malware detection techniques have largely fallen into two categories: static and dynamic analysis. Static analysis extracts code-level features via reverse engineering [5, 6], while dynamic analysis captures runtime behaviors such as network activity and permission usage [7, 8]. The adoption of AI-based methods, particularly those using machine learning and deep learning, has improved detection accuracy and scalability [9], yet key challenges remain in feature representation, model robustness, and multimodal fusion.

Many approaches still rely on single-feature inputs or shallow fusion. For example, Sharfah et al. [10] used a permission-based Bayesian model (91.1% accuracy), but its assumption of feature independence limits adaptability. Song Kyoo Kim et al. [11] applied XGBoost and LightGBM for feature optimization, though performance was hindered by high-dimensional inputs. Similarly, Rafapa's ensemble model [12] achieved 94.2% accuracy for ransomware, but performance dropped >20% on variants; Khan's DNA-sequence method [13] reached only 87.9%, showing poor generalization. Bagui et al. [14] used recursive feature elimination, yet achieved just 79.5% due to limited semantic learning.

Deep learning offers stronger feature abstraction, but unimodal approaches still fall short. Kim et al.'s MAPAS [15] used CNNs on API call graphs and achieved 91.27% on unknown malware, but dropped to 64.71% under obfuscation. This has prompted a shift toward multimodal fusion. Johny et al. [16] fused grayscale images with SimHash and GANs, but linear fusion failed to capture cross-modal dependencies. Lisa et al. [17] integrated images and tabular data with XAI, improving interpretability but lacking deep intermodal synergy.

In summary, current research faces three major limitations: (1) unimodal analysis (e.g., permissions, APIs, or images) fails to capture the full behavioral–code–network attack chain; (2) existing multimodal fusion methods lack the depth to model complex intermodal correlations and remain vulnerable to interference; and (3) shallow architectures limit classification accuracy due to insufficient modeling of sophisticated malware patterns. To address these challenges, this study proposes GCT-Net (GNN-CNN-Tree-LSTM-Net), a multimodal fusion-based model for malicious APP identification, with the following core contributions:

- We propose a multimodal deep fusion framework that integrates API call sequences, APK code images, and URL features. GCT-Net improves detection accuracy and robustness, significantly outperforming unimodal baselines across all metrics.
- We design a unified representational mapping strategy that transforms heterogeneous features into a shared space, enabling deep semantic alignment through DNN-based cross-modal correlation learning.
- GCT-Net achieves state-of-the-art performance, with 96.48%/93.75% accuracy, 97.62%/96.45% precision, 97.05%/95.40% recall, and 97.33%/95.42% F1-score on two datasets, demonstrating strong generalization and real-world applicability in malicious app detection

2 Methods

The GCT-Net proposed in this study adopts an innovative three-channel feature fusion architecture, constructing an intelligent detection system through the integration of various behavioral characteristics of mobile applications. As illustrated in Fig. 1, the framework consists of a decompilation preprocessing module and four core analytical modules.

2.1 Decompilation of APK Files

In Android malware detection, decompiling APK files serves as the primary step in data preprocessing[18]. This process enables the acquisition of application source code, resource files, and the manifest file (AndroidManifest.xml), which form the foundation for extracting API call sequences and conducting further behavioral analysis. To accomplish this step, our research employs APKTool[19], an open-source Android decompilation tool capable of unpacking APK files and reconstructing the application's project structure while extracting relevant files and code. When executing the corresponding commands, APKTool unpacks the APK file and generates the corresponding file structure, including smali code, resource files, and AndroidManifest.xml.

2.2 API Call Feature Extraction Based on GNN

In static analysis methods, the detection method based on API call sequences is an effective strategy. By analyzing the API call interaction between Android applications and the operating system, the behavioral characteristics of the applications can be extracted, and these characteristics can reflect the potential malicious behaviors of the applications. API call sequences can not only be processed as one-dimensional time series data, but also be constructed as graph-structured data for modeling. The relationships and dependencies among API calls can be better represented through a graph structure, thereby capturing more complex features. The advantage of this method lies in that the graph structure can effectively represent the interactions, dependencies among API calls and the topological structure of the call sequence. In this section, we propose a method based on graph structure to represent the sequence of API calls. The relationships between API calls can be captured through a graph structure[20], where each API call is represented as a node in the graph, and the edges between the nodes represent the relationships or dependencies between API calls. By transforming the API call sequence into a graph structure, we can utilize the GNN model to mine the potential patterns in the API call sequence and effectively detect malicious applications.

Construction of API Call Sequences. The decompiled smali files serve as the core data source for analyzing malicious application behaviors in this study. Smali files represent the low-level code of decompiled applications, containing the core logic and API calls of the application. By analyzing these files, we can extract the API call sequences of the application. In smali files, API calls are typically represented using instructions such as invoke−virtual, invoke−static, and invoke − direct. Through parsing tools and scripts, these instructions can be analyzed to extract API calls. Each API call corresponds to a unique ID, which is converted into a numeric ID based on a mapping table. Leveraging full static analysis, all possible call path combinations (including unexecuted logical branches) are extracted from the smali code and deduplicated. These are then arranged in the order of invocation to construct the API call sequence. A call sequence aci consisting of n API calls can be expressed as:

$$ac^i = [A^1, A^2, \cdots, A^n\ ] \tag{1}$$

Furthermore, each ac must be assigned a label y indicating whether the application is malicious. These API call sequences and their corresponding labels constitute the dataset. The API call sequence of each application serves as the input feature, while the label acts as the output, which is fed into the GNN for feature extraction:

$$ac^i = [A^1, A^2, \cdots, A^n, y^i] \tag{2}$$

For each node, a 128-dimensional vector is generated using the Node2Vec algorithm[21].

Graph Structure Construction. Graph Structure Construction. Each API call is represented as a node in the graph, corresponding to an element in the API call sequence. The node feature is the embedding vector of the API call, with a dimension of d, i.e., $h\epsilon R^d$, where i denotes the $i - th$ API call. Edges in the graph represent relationships between API calls. This study considers the following types of relationships for edge construction:

Sequential Relationship: If API call A_i appears after A_{i-1}(i.e., temporally dependent on A_{i-1}), a directed edge is established from A_i to A_{i-1}. This edge indicates a sequential dependency, with a fixed weight $W_{i-1} = 1$.

*Input-Output Dependency:*This relationship captures cases where the output of an API call (e.g., return values, modified global states, or generated files) influences the input of a subsequent API call (e.g., parameters, file paths). If the output of Ai affects the input of A_i, a directed edge is created from A_i to A_{i-1}. The weight for such edges is determined based on the dependency strength:

$$W_{i-1,i} = \alpha \cdot MP + \beta \cdot COS \tag{3}$$

Among them, α and β are the weighting coefficients. COS refers to whether the return value and the input parameter names match. denotes the cosine similarity between the input and output parameters, which measures whether the input and output parameters of two API calls align. A value of 1 indicates that they belong to the same functional category. Let hy_out_{i-1} be the output parameter of the call A_{i-1} and hy_in be the input parameter of A . Then, MP is defined as follows:

$$MP_{A_{i-1},A_i} = IsCompared(hy\_out_{i-1}, hy\_in_i) \tag{4}$$

*Functional Correlation Relationship:*Functional correlation refers to the scenario where different API calls may be closely related in functionality and are frequently invoked together in practical applications. For instance, file operation APIs and network request APIs are often called within the same application. Establishing edge relationships between such functionally related API calls helps capture behavioral patterns in applications. For API calls that may have functional correlations, bidirectional directed edges are created between them. The weights of such edges are set as follows:

$$w_{i,j} = \gamma \cdot COS + \delta \cdot CF \tag{5}$$

Among them, α and β are weighting coefficients. CF represents the cooccurrence frequency of A_i and A_j(i.e., the frequency at which they appear together in the same sequence). Given a total of N invocations, if A_i and A_j co-occur n times, CF can be defined as $N_{i,j}/N$.

Based on the above rules, we can construct a graph $G = (V, E)$, where V represents the set of nodes in the graph, with each node v_i corresponding to an API call A_i. The node features are the embedding vectors of the respective API calls. E represents the set of edges in the graph, where each edge $e_{i,j}$ denotes the relationship between API calls $A_i$and A_j.

Construction of the GNN Model. GNN is a deep learning model capable of processing graph-structured data. Its core idea is to update the representation of each node through message passing between nodes, thereby capturing dependencies and relationships among them. We employ a GNN to model the API call graph, enabling the detection of potential malicious behavior patterns within API call sequences. The GNN updates node representations iteratively. The representation of a node v_i after the $k + 1$ iteration can be computed using the following formula:

$$h_i^{(k+1)} = G(W \cdot aggregate(\{h_j^K | v_j \in N(v_i)\}) + b) \tag{6}$$

Here, $h_i^{(k+1)}$represents the updated representation of node v_i after the $k + 1$ iteration, W is a trainable weight matrix with dimensions of input dimension multiple of output dimension for node v_i after aggregation, and b denotes the bias term. $aggregate(\cdot)$ is an aggregation function that computes the feature representation of the current node based on the representations of its neighboring nodes. In our study, mean aggregation is adopted as the aggregation function. The mean aggregation method calculates the representation of the current node as the average of its neighboring nodes' features. By averaging, the influence of varying neighbor counts on the results is mitigated, leading to smoother aggregated representations. Let h_i denote the set of features of neighboring nodes of node v_i, with $N(v_i)$ being the number of neighbors, then the aggregated representation of node v_i is given by:

$$h_i^{(aggregated)} = \frac{1}{|N(v_i)|} \sum_{v_j \in V} h_j \tag{7}$$

To enable whole-graph learning, we utilize Global Average Pooling (GAP) to aggregate all node representations in the graph into a fixed-length global representation. Assuming that each node is represented as a vector of dimensions d, the grouping operation produces a global representation of dimensions fixed h_G, denoted as:

$$h_G = AVGPooling(h_i^{(k)}) \tag{8}$$

2.3 Binary Code Image Feature Extraction Based on 2D-CNN

This paper adopts a method based on malicious code image generation, in which code files are transformed into grayscale images and deep learning models are employed for malicious application feature extraction. By processing the application's code to generate corresponding image representations and leveraging CNN for training and classification, the detection accuracy of malicious apps is effectively improved.

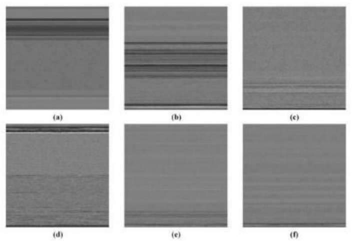

Fig. 2. Overview of a subset of the grayscale code images (a)–(d) correspond to malicious apps, and (e) and (f) correspond to benign (compliant) apps.

Code Image Generation. First, the .dex file of each application is read. Each file consists of a sequence of bytes (8-bit), with each byte having a value ranging from 0 to 255. These byte values can be interpreted as pixel intensities of an image. A fixed image width of 64 is defined, and the height is calculated based on the file size and the fixed width, allowing the entire file to be represented as a grayscale image with a consistent width. Each byte value (0–255) is then mapped directly to a grayscale intensity and filled into a two-dimensional matrix, which represents the pixel values of the image. As shown in Fig. 2, the code of each application is transformed into a grayscale image with a fixed size. The image generation process can be formally defined as follows:

$$I = f(M), M \in R^{m \times n}, I \in R^{h \times w} \tag{9}$$

Here, $m \times n$ denotes the size of the original code file, and $h \times w$ represents the dimensions of the generated grayscale image.

Since the sizes of malicious code samples vary, and subsequent tasks such as feature extraction require consistent input formats, it is necessary to perform operations such as image resizing, standardization, and normalization. These preprocessing steps are meaningful only when all images share the same dimensions. After resizing, standardization and normalization can be effectively applied, ensuring the consistency and validity of downstream tasks such as feature extraction and serving as input to the network. The image standardization processing algorithm is defined as follows:

$$I_{std} = \frac{x-\mu}{adjust\_std} \tag{10}$$

Here, μ denotes the mean value of the image pixels, x represents the image matrix, and the value of $adjust\_std$ is:

$$adjust_{std} = \max\left(\sigma, \frac{1.0}{\sqrt{N}}\right) \tag{11}$$

Here, σ denotes the standard deviation, and N represents the total number of pixels in the image.

2D-CNN Architecture. The grayscale images generated by the steps described above do not contain color features. In this study, we design a 2D-CNN architecture to extract features from such data. By converting the malicious application's code into grayscale images, the 2D-CNN is capable of automatically learning local visual features (e.g., edges, textures) and extracting more abstract, high-level representations through a two-layer convolutional network, without the need for manual feature engineering, thereby improving the performance of feature extraction.

The input data consists of preprocessed grayscale images represented as two-dimensional matrices of size $W \times H$, where W is the image width and H is the image height. Since the images are grayscale, they contain only one channel. Each image corresponds to the code features of a malicious application. The first convolutional layer consists of 32 filters, each of size 3 × 3 with a stride of 1 and padding of 1. After the convolution, the output maintains the same spatial dimensions as the input, but the number of channels becomes 32. A max pooling layer follows the first convolutional layer to perform downsampling, reducing the spatial resolution and extracting dominant features. We apply MaxPooling with a pooling window size of 2×2 and a stride of 2, resulting in an output of reduced spatial dimensions. The second convolutional layer contains 64 filters, also of size 3×3, with a stride of 1 and padding of 1. Its output is again downsampled using the same max pooling operation as in the first pooling layer, resulting in further reduced feature map dimensions. Next, a flattening operation is performed to convert the final pooled feature maps into a one-dimensional feature vector. The resulting flattened vector is of size $(W, H, 4)$, which can then be used as input to the subsequent fully connected layers or classification module.

2.4 URL feature extraction based on Tree-LSTM

For each APK file, a set U is defined, representing the collection of URLs extracted from its corresponding decompiled files. Each $u_i (u_i \in U)$ is preprocessed and then fed into a Tree-LSTM model enhanced with an attention mechanism for feature extraction. The resulting feature representation has a shape of $(N_u, 64)$, where N_u denotes the number of URLs identified as malicious from the given APK file, and 64 is the number of hidden units in the Tree-LSTM model. Tree- LSTM is capable of effectively capturing the hierarchical structure inherent in URLs. When combined with an attention mechanism, the model further improves the accuracy of URL classification[22].

Representation of URL Sequences. In this study, the dataset is represented as a collection of pairs u_i , y_i , where u_i denotes the i-th URL and y_i is the corresponding label. A URL typically consists of multiple structured components, such as the protocol, domain name, path, and query parameters. The label y_i is a binary indicator specifying whether the URL is malicious. Specifically, $y_{\ i} = 0$ represents a benign URL, while $y_i = 1$ denotes a malicious one. Essentially, each URL u_i is a sequence of characters or tokens, typically delimited by special characters. In this work, we randomly initialize an embedding matrix and learn its features during end-to-end training. An instance consisting of a sequence of L components (tokens) can be represented as:

$$x = x_1 \oplus x_2 \oplus \cdots \oplus x_L \tag{12}$$

The symbol $\oplus$ denotes the concatenation operator. Since the lengths of URLs in the dataset are not uniform, it is necessary to standardize all URL sequences to a fixed length to ensure consistency of input. For URLs shorter than the predefined length L, padding is applied; for URLs longer than L, truncation is performed.

Character-Level Embedding. A standard URL consists of multiple components. To adapt URL data to the input requirements of deep learning models, we employ a character-level embedding approach to represent URLs. This involves transforming the URL strings into low-dimensional vector representations. Specifically, we use a pre-trained character-level embedding model, Char2Vec, to map each character into a fixed-dimensional vector space, thereby effectively capturing the character-level features of URLs. The detailed steps are as follows: First, we define a character vocabulary that includes all possible characters that may appear in a URL. The character set consists of letters, digits, symbols, and special characters (such as ":", "/", "?", "&", "#", etc.). Suppose the total number of unique characters in the vocabulary is C_{num}. Each character ci is assigned a unique integer index for lookup and embedding idx_{c_i}.

Next, we utilize an embedding matrix $R \in R^{C_{num} \times k}$, where k denotes the embedding dimension. Each row in the embedding matrix corresponds to the embedding vector of a specific character. In this way, every character is mapped to a vector in a k dimensional embedding space. Assuming each URL consists of U_{num} characters, a URL can be represented as a sequence of characters:

$$u^i = c_1 \oplus c_2 \oplus \cdots \oplus c_{U_{num}} \tag{13}$$

To ensure consistency in input data, we perform length normalization on each URL, standardizing all character sequences to a fixed length L. Specifically, we apply padding and truncation strategies to normalize the length of each URL. After this step, every URL ui is transformed into a character sequence of length L.

For each normalized URL u_i, we use the embedding matrix E to map each character $E_{c_i} \in R^k$ to its corresponding embedding vector. Here, c_i represents the i-th character in the sequence. As a result, each URL is represented as an embedding matrix $x_i \in R^{l \times k}$, where each row corresponds to the embedding vector of a character:

$$x_i = E_{c_1} \oplus E_{c_2} \oplus \cdots \oplus E_{c_i} \tag{14}$$

Here, L denotes the normalized URL length, and k is the dimensionality of the character-level embedding vectors. The matrix thus serves as the final feature representation of the URL. At this point, the character-level features of all URLs have been transformed into fixed-dimensional embedding matrices, making them suitable for further model processing. This character-level embedding approach facilitates the model in capturing both semantic and structural information of each character in a URL, thereby providing a strong feature representation for malicious URL detection.

Tree-LSTM Model with Integrated Attention Mechanism. The core structure of the proposed algorithm is a Tree-LSTM integrated with an attention mechanism. Tree-LSTM is well-suited for capturing both syntactic and semantic structures in textual data, and is particularly effective in modeling the hierarchical nature of URL structures. By incorporating an attention mechanism, the model further enhances its accuracy in URL classification tasks. In the context of URL analysis, a URL's character sequence can be viewed as a tree structure, where each character corresponds to a node in the tree. The hierarchical layout of the tree reflects different components of the URL (e.g., protocol, domain name, path, query parameters, etc.). The Tree-LSTM processes this tree-structured data by recursively updating the state of each node. The fundamental idea is that each node (i.e., character) carries a state vector, which depends not only on the embedding vector of the current character but also on the state vectors of its child nodes. The feature embedding matrix is fed as input to the Tree-LSTM. Attention mechanisms are applied across the nodes in the tree to dynamically assign different levels of importance to different components. Through its recursive structure, the Tree-LSTM computes the hidden state of each node as follows:

$$h_n = f(h_{parent}, h_{child1}, h_{child2}, \cdots) \tag{15}$$

Here, $f(\cdot)$ denotes the LSTM update function, which integrates features from multiple child nodes along with attention-weighted information. Compared with a standard LSTM, the Tree-LSTM module proposed in this study is capable of recursively processing input features while better capturing the underlying hierarchical structure. Moreover, by processing information in parallel across nodes, it reduces computational time and resource consumption, thereby improving training efficiency and real-time inference performance. To further enhance the model's effectiveness, we integrate an Attention Mechanism into the Tree-LSTM framework. The purpose of the attention mechanism is to assign different weights to each character (i.e., node), enabling the model to focus more on the parts of the URL that are most influential in identifying malicious behavior. In the out-put of the Tree-LSTM, an attention layer is applied to compute the importance of each node. Specifically, given the output hidden states from the Tree-LSTM, we compute an attention weight for each node using a trainable weight matrix and a bias term. The attention weight α_i for the node h_i is defined as:

$$\alpha_i = softmax(W_a h_i + b_i) \tag{16}$$

Here, W_a is a trainable weight matrix, where h denotes the dimensionality of each hidden state. b_a is the bias term. The $softmax$ function ensures that the attention weights sum to 1, meaning they form a valid probability distribution.

By computing a weighted sum over the outputs of all nodes, we obtain the final representation:

$$v = \sum(\alpha_i h_i) \tag{17}$$

Here, v represents the final input representation of the model, which incorporates information from all nodes while assigning different levels of importance to each node based on the attention mechanism.

2.5 Multimodal Feature Fusion

This study adopts a representation-level fusion strategy to integrate features from three different modalities. A Deep Neural Network (DNN) is then employed to learn the fused feature representation, which is used for the final prediction in the malicious application detection task. For each APK file sample, we consider three distinct types of feature representations:

API Call Features. These features are extracted using a Graph Neural Net-work (GNN), representing node embeddings in the API call graph. Since each APK file contains a different number of API calls, the API feature tensor has a shape of (m, d), where m is the number of API calls and d is the dimensionality of the feature vector for each API.

Binary Code Image Features. These features are obtained by converting the binary code into a grayscale image and extracting its features. Assuming the image size is (H, W), the extracted image feature for each APK is flattened into a one-dimensional vector of shape $(H \times W \times 4)$.

URL Features. Each APK file may contain multiple potentially malicious URLs, each represented by a feature vector. Thus, the URL feature matrix for a sample has a shape of $(n, 64)$, where n is the number of malicious URLs and 64 is the number of hidden units in the Tree-LSTM model. To make the features from all modalities compatible, each feature type is mapped into a common dimensional space, after which the vectors are concatenated into a unified representation for downstream classification. For the URL and API call features, average pooling is applied to aggregate the sequences into fixed-length vectors. Specifically, the n URL vectors are averaged into a 64-dimensional vector, and the m API call vectors are aggregated into a d_{new} vector. After processing, the final unified feature vector is formed by concatenating the pooled URL feature, the pooled API feature, and the binary code image feature. Assuming the post-pooling dimensionalities are k for the URL feature, d for the API feature, and $(H \times W \times 4)$ for the image feature, the resulting fused feature vector has the dimensionality:

$$d_f = k + d + W \times H \times 4 \tag{18}$$

After feature fusion, the resulting representation is fed into a three-layer fully connected network to further learn high-level interactions among the fused features. The first two layers use the ReLU activation function to introduce non-linear representational capacity. The final layer outputs a binary classification result. The fused feature vector is first passed through the first hidden layer, which is a fully connected layer. This layer has an input dimension of d_f, and an output dimension of d_{h1}:

$$h_1 = Relu(W_1 \cdot d_f + b_1 \tag{19}$$

Here, $W_1$is the weight matrix, and b_1 is the bias vector.

The second hidden layer is also a fully connected layer, with an input dimension of d_{h1} and an output dimension of d_{h2}:

$$h_1 = Relu(W_2 \cdot d_{h1} + b_2) \tag{20}$$

Here, $W_2$is the weight matrix, and b_2 is the bias vector.

The final layer is the output layer, responsible for mapping the learned features to a binary classification result. It has an input dimension of d_{h2} , and outputs a single scalar value. A $Softmax$ function is applied to produce a probability of a malicious or a benign app:

$$y_{pred} = Softmax(Relu(W_3 \cdot d_{h2} + b_3)) \tag{21}$$

Here, W_3 is the weight matrix, and b_3 is the bias term. Since this task is a binary classification problem (malicious vs. benign applications), we adopt the Binary Cross-Entropy (BCE) Loss function to measure the discrepancy between the model's prediction and the ground truth label. Binary Cross-Entropy is a widely used loss function for binary classification tasks, as it effectively quantifies the distance between the predicted probability and the actual label:

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i long(\hat{y}_i) + (1 - y_i)(\log(1 - \hat{y}_i)] \tag{22}$$

Here, L denotes the loss value, N is the number of samples, y_i represents the ground truth label, and $\hat{y}_i$ is the predicted probability that the sample belongs to the positive class.

3 EXPERIMENTAL RESULTS

3.1 Dataset Description

DataSet1. This is a publicly available dataset provided by China Mobile on the Jiutian· Bisheng Cloud Platform. After filtering out APK files that could not be properly parsed, a total of 474 APK files were retained for analysis, including 223 malicious and 251 benign applications.

DataSet2.[23] This is a balanced dataset for verifying the overall performance of our method in Android malware detection. We collected the API keys from the AndrooZoo website and downloaded the apk files. Then, the virustotal tool was used to test whether these APKs were benign or malicious software, and 250 benign applications and 250 malicious applications were randomly selected from them to form our dataset.

3.2 Experimental Settings

In this experiment, we propose and implement a multimodal feature fusion method that integrates API call sequences, code image features, and malicious URL features to perform classification of non-compliant applications. The experiment includes a complete pipeline involving data preprocessing, model training, and performance evaluation to comprehensively assess the effectiveness of the proposed model. The details of preprocessing are described in Section 2.1.

We use accuracy, precision, and recall as metrics, defined as follows:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{23}$$

$$Precision = \frac{TP}{TP+FP} \tag{24}$$

$$Recall = \frac{TP}{TP+FN} \tag{25}$$

$$F1\,Score = \frac{2TP}{2TP+FP+FN} \tag{26}$$

Where True Positive (TP) refers to malicious samples correctly identified as malicious applications, while True Negative (TN) denotes benign samples correctly classified as benign applications. Conversely, False Positive (FP) represents malicious samples mistakenly identified as benign applications, and False Negative (FN) refers to benign samples inaccurately recognized as benign applications. The raw data, after being preprocessed, are fed into the GCT-Net model for training. The training process is conducted over 150 epochs with a learning rate of 0.0001, a batch size of 32, and the Adam optimizer, with beta1 and beta2 set to 0.9 and 0.999, respectively.

3.3 Experimental Results and Analysis

Table 1. Comparison of classification performance of various models (%).

Dataset	Method	Evaluation Metrics			
		Accuracy	Precision	Recall	F1 Score
Dataset1	MserNetDroid[24]	91.61	89.82	88.31	89.06
	LeNet[25]	92.58	93.48	95.50	94.48
	KNN P[26]	90.33	96.00	89.19	92.47
	SigPID[27]	91.74	93.12	94.52	93.81
	Drebin[28]	94.71	94.55	**97.65**	96.08
	XMAL[29]	92.57	93.57	95.39	94.47
	GCT-Net(Ours)	**96.48**	**97.62**	97.05	**97.33**
Dataset2	MserNetDroid[24]	89.42	87.35	85.63	86,48
	LeNet[25]	90.17	91.20	93.05	92.11
	KNN P[26]	88.25	93.15	86.74	89.85
	SigPID[27]	89.83	90.67	92.31	91.48
	Drebin[28]	92.60	92.18	95.39	93.76
	XMAL[29]	90.52	91.43	93.25	92.33
	GCT-Net(Ours)	**93.75**	**96.45**	**95.40**	**95.92**

Our model also performs exceptionally well, achieving the highest classification accuracy, precision, recall, and F1 Score on both datasets in the test set. As shown in Table. 1, on Dataset1, GCT-Net achieved an accuracy of 96.48%, precision of 97.62%, recall of 97.05%, and F1 Score of 97.33%. On Dataset2, GCT-Net achieved an accuracy of 93.75%, precision of 96.45%, recall of 95.40%, and F1 Score of 95.92%. These results clearly demonstrate the strong performance of the model, with both high precision and recall, indicating its ability to accurately classify both positive and negative samples while minimizing errors. Moreover, the consistently balanced F1 score further reinforce the effectiveness of GCT-Net in malicious application detection tasks across diverse datasets.

Building on these promising outcomes, a closer examination of Table. 1 reveals that GCT-Net consistently and significantly outperforms existing baseline methods in all four evaluation metrics. For instance, in Dataset1, while traditional approaches such as Drebin and XMAL deliver relatively strong recall and F1 performance, with Drebin reaching a recall of 97.65% and XMAL reaching a F1 score of 94.47%—these methods typically exhibit trade-offs between precision and recall, which compromise overall classification robustness. In contrast, GCT-Net achieves the best performance across the board, highlighting its ability to simultaneously maximize detection sensitivity and specificity. Similarly, in Dataset2, although some methods such as SigPID and Drebin attain high precision or recall individually, they fall short in achieving a balanced and comprehensive performance. GCT-Net, on the other hand, not only secures the highest precision and recall but also delivers a top-tier F1 score of 95.92%, underscoring its generalization capability and resilience across varying data distributions. The model's high precision indicates its effectiveness in minimizing false positives—a key requirement in real-world deployments to avoid wrongly flagging benign applications—while its high recall demonstrates strong competence in capturing the full spectrum of malicious behaviors. Taken together, the superior and stable results across both datasets confirm that GCT-Net offers a robust, reliable, and deployable solution for Android malware detection, with clear advantages over existing methods in both academic benchmarks and practical scenarios.

3.4 Ablation Study

Based on the ablation results shown in Table. 2, it is evident that each of the three modules—API call sequence analysis, grayscale image analysis, and URL traffic analysis—contributes meaningfully to the overall performance of GCT-Net. The combination of all three modalities consistently yields the highest performance across all evaluation metrics on both two dataset. When examining the contribution of individual modules, it is clear that the API call sequence (API) modality tends to offer the strongest standalone performance compared to Img or URL, particularly on DataSet1, where it achieves 85.40% accuracy and 81.47% F1 score. This suggests that visual patterns extracted from reverse-engineered APK code images provide rich semantic information for malware classification. However, the API call module also demonstrates substantial effectiveness, especially in recall, indicating its advantage in capturing behavioral traits of malicious applications. The URL module, while relatively weaker on its own, shows

Table 2. Experimental results showing the contributions of different modules to model performance(%).

Dataset	**Modules**			**Metrics**			
	API	**Img**	**URL**	**Accuracy**	**Precision**	**Recall**	**F1 Score**
Dataset1	√			85.40	82.71	80.62	81.47
		√		82.10	79.51	77.81	78.65
			√	78.30	75.20	72.94	74.05
	√	√		90.82	85.24	84.71	84.97
	√		√	88.74	86.34	85.63	85.98
		√	√	87.98	87.13	86.87	87.00
	√	√	√	**96.48**	**97.62**	**97.05**	**97.33**
Dataset2	√			83.15	80.26	78.92	79.58
		√		80.42	77.83	75.64	76.72
			√	76.50	73.15	70.88	72.00
	√	√		88.37	83.15	82.60	82.87
	√		√	86.20	84.01	83.25	83.63
		√	√	85.63	84.92	84.10	84.51
	√	√	√	**93.75**	**96.45**	**95.40**	**95.92**

its value when combined with other modalities, particularly in enhancing precision by identifying external communication behaviors. Notably, multimodal combinations consistently outperform unimodal configurations, highlighting the importance of complementary information across feature domains. For instance, the Img+URL configuration boosts the F1 score to 87.00% on DataSet1, significantly higher than any single module. When any one modality is removed from the full model, performance degrades across all metrics, confirming the necessity of each component in building a robust detection system. Overall, the ablation study validates the design rationale of GCT-Net's modality-model alignment framework. By integrating diverse and complementary information sources—each targeting different behavioral and structural aspects of malicious apps—the model achieves a more comprehensive understanding of threats, resulting in superior classification performance and generalization capability across datasets.

4 Conclusion

To overcome the limitations of unimodal approaches in malicious app detection, we propose GCT-Net, a multimodal fusion model that integrates API call graphs, grayscale code images, and URL traffic analysis. By combining the strengths of GNNs for behavioral logic, 2D-CNN for structural feature extraction, and Tree- LSTM for network pattern modeling, GCT-Net achieves robust cross-modal semantic fusion. Experimental results demonstrate that our model significantly improves detection accuracy, precision, recall and F1 Score. In future work, we will explore lightweight deployment, dynamic incremental learning, and the construction of a malicious behavior knowledge graph to enhance interpretability and support intelligent, layered mobile security systems.

References

1. Kenedi Binowo and Rondo VSA Morihito. Prevention and strategies for avoiding social media fraud: A case of fraud prevention in indonesia. In Proceeding of The International Conference on Natural Sciences, Mathematics, Applications, Research, and Technology, volume 3, pages 9–13, 2023.
2. Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring api embedding for api usages and applications. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 438–449. IEEE, 2017.
3. Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Automated static code analysis for classifying android applications using machine learning. In 2010 international conference on computational intelligence and security, pages 329–333. IEEE, 2010.
4. Min-Sheng Lin, Chien-Yi Chiu, Yuh-Jye Lee, and Hsing-Kuo Pao. Malicious url filtering—a big data application. In 2013 IEEE international conference on big data, pages 589–596. IEEE, 2013.
5. Jabunnesa Jahan Sara and Shohrab Hossain. Static analysis based malware detection for zero-day attacks in android applications. In 2023 International Conference on Information

and Communication Technology for Sustainable Development (ICICT4SD), pages 169–173. IEEE, 2023.
6. Mülhem İbrahim, Bayan Issa, and Muhammed Basheer Jasser. A method for automatic android malware detection based on static analysis and deep learning. IEEE Access, 10:117334–117352, 2022.
7. Asma Razgallah, Raphaël Khoury, Kobra Khanmohammadi, and Christophe Pere. Comparing the effectiveness of static, dynamic and hybrid malware detection on a common dataset. In 2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pages 2452–2458. IEEE, 2023.
8. Arunab Singh, Maryam Tanha, Yashsvi Girdhar, and Aaron Hunter. Interpretable android malware detection based on dynamic analysis. In ICISSP, pages 195–202, 2024.
9. Amir Djenna, Ahmed Bouridane, Saddaf Rubab, and Ibrahim Moussa Marou. Artificial intelligence-based malware detection, analysis, and mitigation. Symmetry, 15(3):677, 2023.
10. Sharfah Ratibah Tuan Mat, Mohd Faizal Ab Razak, Mohd Nizam Mohmad Kahar, Juliza Mohamad Arif, and Ahmad Firdaus. A bayesian probability model for android malware detection. ICT Express, 8(3):424–431, 2022.
11. Song-Kyoo Kim, Feng Xiaomei, Hussam Al Hamadi, Ernesto Damiani, Chan Yeob Yeun, and Sivaprasad Nandyala. Advanced machine learning based malware detection systems. IEEE Access, 2024.
12. Julian Rafapa and Arthur Konokix. Ransomware detection using aggregated random forest technique with recent variants. 2024.
13. Firoz Khan, Cornelius Ncube, Lakshmana Kumar Ramasamy, Seifedine Kadry, and Yunyoung Nam. A digital dna sequencing engine for ransomware detection using machine learning. IEEE Access, 8:119710–119719, 2020.
14. Sikha Bagui and Hunter Brock. Machine learning for android scareware detection. Journal of Information Technology Research (JITR), 15(1):1–15, 2022.
15. Jinsung Kim, Younghoon Ban, Eunbyeol Ko, Haehyun Cho, and Jeong Hyun Yi. Mapas: a practical deep learning-based android malware detection system. International Journal of Information Security, 21(4):725–738, 2022.
16. Jahez Abraham Johny, KA Asmitha, P Vinod, G Radhamani, KA Rafidha Rehiman, and Mauro Conti. Deep learning fusion for effective malware detection: leveraging visual features. Cluster Computing, 28(2):135, 2025.
17. Fahmida Tasnim Lisa, Sheikh Rabiul Islam, and Neha Mohan Kumar. Multimodal machine learning model for interpretable malware classification. In World Conference on Explainable Artificial Intelligence, pages 334–349. Springer, 2024.
18. Ulf Kargén, Noah Mauthe, and Nahid Shahmehri. Android decompiler performance on benign and malicious apps: an empirical study. Empirical Software Engineering, 28(2):48, 2023.
19. Heena Rawal and Chandresh Parekh. Android internal analysis of apk by droid_safe & apk tool. International Journal of Advanced Research in Computer Science, 8(5), 2017.
20. Yuchen Zhou, Yanmin Shang, Yanan Cao, Qian Li, Chuan Zhou, and Guandong Xu. Api-gnn: attribute preserving oriented interactive graph neural network. World Wide Web, 25(1):239–258, 2022.
21. Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, pages 855–864, 2016.
22. Mahtab Ahmed, Muhammad Rifayat Samee, and Robert E Mercer. Improving tree-lstm with tree attention. In 2019 IEEE 13th international conference on semantic computing (ICSC), pages 247–254. IEEE, 2019.

23. Marco Alecci, Pedro Jesús Ruiz Jiménez, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. Androzoo: A retrospective with a glimpse into the future. In Proceedings of the 21st International Conference on Mining Software Repositories, pages 389–393, 2024.
24. Hui-juan Zhu, Wei Gu, Liang-min Wang, Zhi-cheng Xu, and Victor S Sheng. Android malware detection based on multi-head squeeze-and-excitation residual network. Expert Systems with Applications, 212:118705, 2023.
25. Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, 2002.
26. Juan Hu, Hong Peng, Jun Wang, and Wenping Yu. knn-p: A knn classifier optimized by p systems. Theoretical Computer Science, 817:55–65, 2020.
27. Lichao Sun, Zhiqiang Li, Qiben Yan, Witawas Srisa-an, and Yu Pan. Sigpid: significant permission identification for android malware detection. In 2016 11th international conference on malicious and unwanted software (MALWARE), pages1–8. IEEE, 2016.
28. Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In Ndss, volume 14, pages 23–26, 2014.
29. Mohammed M Alani, Atefeh Mashatan, and Ali Miri. Xmal: A lightweight memory-based explainable obfuscated-malware detector. Computers & Security, 133:103409, 2023.