VIT-AP UNIVERSITY

A Deep Learning Framework for QR Code Analysis: Detecting Phishing URLs and Malicious APKs via Metadata and Permissions

CSE4006 – DEEP LEARNING

PROJECT REPORT

Class Number – **AP2024254000418**

SLOT – **D1 + TD1**

Course Type – **EPJ**

Course Mode – **Project Based Component (Embedded)**

Department of Artificial Intelligence and Machine Learning

School of Computer Science and Engineering

By

22BCE9976	Kalluru Jahnavi
22BCE9596	Neha Agarwal
22BCE9996	Hemasree

Submitted to:-

Dr. E. Sreenivasa Reddy

Professor-HAG, SCOPE, VIT-AP.

2024 -2025

TABLE OF CONTENTS

Abstract:

The widespread use of Quick Response (QR) codes in digital transactions, app installations, and authentication has introduced new security challenges. Attackers increasingly exploit the opaque nature of QR codes to embed phishing URLs and malware-infected APK files, placing users at risk of cyber threats such as data theft and system compromise. To address this growing concern, this study proposes a deep learning-based framework for the detection of malicious content extracted from QR codes.

The proposed system performs dual-path static analysis for both URLs and APKs. Phishing URLs are identified using a hybrid approach that combines Bidirectional Encoder Representations from Transformers (BERT) and Convolutional Neural Networks (CNNs), enabling accurate detection of lexical and contextual anomalies. For APK analysis, static features including permissions, API calls, and manifest components are extracted and evaluated using CNN-based classifiers to determine potential malicious behavior. The outputs of both analyses are combined to enhance detection accuracy while minimizing false positives.

Experimental results demonstrate the framework's effectiveness in identifying obfuscated and zero-day threats with high precision. The system is optimized for lightweight deployment, making it suitable for integration into mobile applications and browser extensions. By offering real-time threat detection without relying on dynamic execution, the framework provides a scalable solution to mitigate QR code-based phishing and malware attacks.

Keywords: QR Code Security, Phishing Detection, Malware Detection, Static Analysis, APK Permissions, URL Classification, BERT, Convolutional Neural Networks (CNN), Cybersecurity, Mobile Threats.

1. Introduction

1.1 Introduction to the Problem

Quick Response (QR) codes have become an integral part of modern digital ecosystems, enabling seamless access to websites, application downloads, and digital services. Their widespread adoption across domains such as e-commerce, banking, and education is driven by their efficiency, contactless nature, and user convenience. However, this increasing reliance on QR codes has also introduced significant cybersecurity risks. The inherent opacity of QR codes, which hides their embedded content such as URLs or APK download links makes them a prime vector for phishing and malware distribution.

Users frequently scan QR codes without validating their authenticity, making them susceptible to malicious redirections or the installation of harmful applications. Traditional security tools, including signature-based antivirus systems and heuristic URL filters, often fail to detect evolving threats such as obfuscated URLs, zero-day attacks, and APK-based malware. Moreover, many existing approaches to QR code security focus solely on superficial checks or single-mode threat analysis, leaving gaps in protection.

This research addresses these challenges by developing a deep learning-based framework that analyzes both URLs and APKs extracted from QR codes. For URL analysis, the system utilizes Bidirectional Encoder Representations from Transformers (BERT) and Convolutional Neural Networks (CNNs) to classify web links as legitimate or phishing. For APK analysis, the framework applies CNN-based models to examine static features such as permissions and API usage patterns to identify potential malware. By combining these parallel analyses, the framework delivers a robust and scalable solution for detecting QR code-based cyber threats.

Designed for lightweight deployment on mobile devices and browsers, this system aims to provide real-time protection without the need for dynamic execution or high computational resources. The integration of BERT and CNN models enables precise threat detection while ensuring that the system remains efficient and adaptable to emerging attack vectors.

1.2 Motivation

The motivation for this study stems from the increasing misuse of QR codes in distributing malware and conducting phishing attacks. Due to their invisible nature, QR codes bypass the user's ability to perform visual inspection before action, creating a significant trust gap in digital interactions. Many users out of convenience or lack of awareness scan codes without hesitation, exposing their devices and personal information. Existing QR scanners and antivirus tools typically check for known malicious signatures or flag suspicious domains, but they lack the adaptability and depth required for detecting zero-day and sophisticated attacks embedded within APK files or shortened links.

Furthermore, most current research treats either URL phishing detection or APK malware classification in isolation. This fragmented approach fails to capture the end-to-end nature of QR-based cyberattacks, where a single code may both redirect a user to a harmful site and install malicious software. This project aims to fill this critical gap by leveraging deep learning's ability to analyze complex, high-dimensional, and heterogeneous data, resulting in a more intelligent and comprehensive QR code threat detection system.

1.3 Problem Statement and Objectives

Problem Statement:

QR codes have become a crucial component of modern technology, enabling a wide range of uses, including contactless payments, marketing campaigns, and information exchange. Their simplicity and efficiency make them a popular choice for both enterprises and consumers. However, the extensive popularity has attracted thieves, resulting in an alarming increase in security vulnerabilities related with QR codes. These dangers include phishing attempts, which divert users to dangerous websites; malware distribution, in which scanning a QR code installs malicious software; and fraudulent operations, such as manipulated QR codes that send users to undesired locations.

The difficulty stems from QR codes' intrinsic opacity. Unlike typical URLs or visible links, QR codes conceal the information they contain, forcing users to scan them without instant understanding of the destination or content. This lack of transparency exposes a serious risk, since attackers use it to incorporate malicious URLs, alter with valid codes, or generate visually misleading codes that seem trustworthy. This problem is exacerbated by users who often scan QR codes without validating their legitimacy, either out of curiosity or for the convenience of rapid access to information or services.

Traditional techniques to QR code security have proven ineffective in the face of new threats. Heuristic-based approaches, which rely on predefined criteria to detect phishing URLs, fail to cope with zero-day assaults and disguised links. Design-centric techniques, such as integrating logos in QR codes, improve visual security but do not include automated detection tools for sophisticated manipulation or harmful payloads. Furthermore, present machine learning models, although successful in some cases, frequently lack the real-time detection and agility required to combat quickly shifting attack vectors. These constraints highlight the critical need for a strong and efficient solution to assure the security and reliability of QR codes.

Objectives:

1. To design a QR scanning pipeline that extracts embedded URLs and APKs using image processing libraries.
2. To analyze URLs using BERT and CNN-based classifiers for the detection of phishing and malicious domains.
3. To conduct static analysis of APK files by examining permissions, API calls, and manifest metadata using Feedforward Neural Networks (FNNs) and Long Short-Term Memory (Autoencoders) models.
4. To implement a CNN-based malware detection model focused on static APK features such as permission requests and manifest structure, enabling efficient threat classification without runtime execution.
5. To evaluate the effectiveness of combining CNNs and BERT in a unified detection pipeline for processing QR-extracted URLs and APKs.
6. To integrate both URL and APK classification outputs to enhance overall detection accuracy and reliability.

7. To optimize the proposed framework for lightweight deployment on mobile platforms and browser extensions using techniques such as model quantization and pruning.

1.4 Scope

This project encompasses the design, development, and evaluation of a deep learning-based malware detection system that analyzes content extracted from QR codes, focusing on static feature analysis. The system addresses two primary threat vectors: phishing URLs and malicious APK files. The scope of the study includes:

- **QR Code Extraction**: Scanning and decoding QR codes using image processing techniques to extract embedded URLs or APK download links.
- **URL Analysis**: Identifying phishing attempts by classifying URLs using a combination of BERT-based language modeling and Convolutional Neural Networks (CNNs) to detect lexical and contextual anomalies.
- **Static APK Analysis**: Analyzing APK metadata such as requested permissions, manifest declarations, and API calls using CNN-based classifiers to identify malware based on static features.
- **CNN and BERT Integration**: Combining predictions from URL and APK classifiers to improve detection performance and reduce false positives.
- **Deployment Readiness**: Optimizing the framework for real-time applications on mobile platforms and browser extensions using model compression techniques such as quantization and pruning.

In future work, the system may be extended to include dynamic analysis, runtime sandboxing, grayscale image-based malware detection, or detection of physical tampering of QR codes to provide even more comprehensive threat coverage.

1.5 Organization of the Report

- **Literature Survey** – Presents an overview of related work in the field of QR code security, phishing URL detection, APK malware analysis, and the use of deep learning techniques.
- **System Design and Data Configuration** – Describes the system architecture, hardware/software requirements, and data acquisition process.
- **Proposed Methodology** – Details the technical components of the detection system, including URL and APK analysis pipelines, model structures, and feature extraction techniques.
- **Experimental Results and Discussion** – Provides insights from model evaluation, performance metrics, and analysis of strengths and limitations.
- **Conclusion and Future Work** – Summarizes key findings and outlines future research directions.

2. Literature Overview

2.1 Literature Survey

A. QsecR: Secure QR Code Scanner According to a Novel Malicious URL Detection Framework

QR code scanners are widely used for their convenience but have become significant cybersecurity threats, particularly for delivering malicious URLs and APK files on Android platforms. Early detection methods relied on blacklists like PhishTank and Google Safe Browsing. While fast, these methods fail against newly created or obfuscated threats. To address these issues, researchers have explored machine learning (ML) and deep learning (DL) approaches. Tools like QRphish and QRfence use classifiers such as Naive Bayes and Random Forest, but these methods often depend on large datasets and require frequent retraining. Feature selection challenges further limit their effectiveness. The QsecR framework introduces a data-independent solution using 39 static features across blacklist, lexical, host-based, and content-based categories. It maintains high detection reliability and compensates for incomplete data through a feature evaluation method. Privacy is another concern. Many QR scanners request unnecessary permissions, risking user data. Tools like QsecR and BarSec follow the least-privilege principle, improving security and trust

.

QsecR stands out with 93.5% accuracy and 93.8% precision, surpassing scanners like BarSec by handling obfuscated URLs and providing clear, feature-based feedback. Its integration of accuracy, usability, and privacy makes it a benchmark in QR security, aligning closely with your project's goal of developing a lightweight, DL-based malicious URL and APK detection system.

B. Secure QR Code Scanner to Detect Malicious URL using Machine Learning

QR codes have become common in digital services but also serve as vectors for phishing and malware attacks. Their encoded nature makes them ideal for concealing malicious URLs and APK files. Studies by Krombholz et al. and Heider & Flaminia identified major privacy and security flaws in QR scanners, emphasizing the need for secure and privacy-respecting applications. Traditional blacklist-based detection (e.g., PhishTank, URLhaus) offers limited protection against new threats. To address this, machine learning models such as KNN, SVM, RF, and Bi-LSTM have been employed. Among these, Bi-LSTM has shown superior accuracy by learning sequential URL patterns.

QsecR introduced a data-independent model using 39 predefined static features, achieving high accuracy (93.5%) without the need for large datasets. Similarly, systems combining lexical features (like TF-IDF) and APK metadata have proven effective in detecting threats from QR codes. Recent experiments confirm that deep learning models, particularly Bi-LSTM, outperform traditional classifiers in malicious URL detection. Effective QR code security systems now combine robust detection, minimal permissions, and real-time feedback to ensure safe user experiences.

C. Secure Real-Time Computational Intelligence System Against Malicious QR Code Links

The growing use of QR codes in commercial and public services has introduced new security risks. Due to their machine-readable format, QR codes are increasingly exploited to deliver phishing links, malware, or malicious APKs. Several studies have focused on securing QR scanners through machine learning and computational intelligence techniques.

Wahsheh and Al-Zahrani (2021) proposed a real-time QR security system called BarCI, which used MLP-ANN and fuzzy logic classifiers. By relying on lexical URL features instead of DNS or WHOIS data, their system achieved 82.9% accuracy while maintaining speed and data privacy. BarCI also enforced least-privilege permissions, reducing user data exposure.

The QsecR framework presented a static-feature-based model that classified URLs using 39 predefined features without requiring large datasets. Achieving 93.5% accuracy, QsecR combined feature evaluation with fallback mechanisms to maintain functionality even when some features failed, ensuring robust and interpretable results. Another study compared traditional ML models (SVM, k-NN, RF) with Bi-LSTM, a deep learning model that outperformed others in URL threat detection by learning sequential patterns. However, Bi-LSTM's computational needs limit its use in mobile environments without optimization.

Across these studies, common trends include the importance of lexical analysis, preference for deep learning models like MLP-ANN and Bi-LSTM, and the need for privacy-conscious QR scanners. Your project aligns with these approaches by using CNN models, TF-IDF, and APK metadata for dual detection of malicious URLs and APKs—offering a scalable and secure QR code threat detection system.

D. Secure Real-Time Artificial Intelligence System Against Malicious QR Code Links

The increasing use of QR codes in everyday digital transactions has escalated concerns over their misuse for phishing and malware attacks. Malicious URLs embedded within QR codes are difficult for users to detect due to the opaque nature of QR encoding. As a result, artificial intelligence (AI)-based systems have been explored to enhance QR code security. One such approach involves the development of real-time AI systems capable of analysing the lexical structure of URLs extracted from QR codes. The study proposes the use of computational intelligence models, particularly fuzzy logic and Multilayer Perceptron Artificial Neural Networks (MLP-ANN). The MLP-ANN model achieved higher accuracy in distinguishing between safe and malicious URLs by learning from patterns in features such as special characters, domain lengths, and tokenized segments of the URL. A key benefit of this model is its independence from third-party services like DNS lookups or WHOIS queries, allowing for faster and more self-contained threat detection. Additionally, the system is designed with minimal permission requirements, aligning with modern privacy standards for mobile applications. This ensures users are protected without unnecessary access to their personal data.

The paper highlights that real-time detection models using AI can significantly improve QR code security. However, it also emphasizes the need for lightweight, efficient systems that can be deployed on mobile devices without compromising performance or user privacy.

E. Evaluating Security and Privacy Features of Quick Response Code Scanners: A Comparative Study

The growing adoption of QR codes across various sectors has increased their misuse for phishing, malware, and privacy-invasive attacks. Studies reveal that attackers often embed malicious URLs in QR codes, redirecting users to phishing websites or malicious downloads. This highlights the urgent need for secure QR code scanner applications that can effectively detect and mitigate such threats.

The literature categorizes QR scanner security into two major approaches: cryptographic-based and URL (link) security-based methods. Cryptographic approaches offer confidentiality and integrity through encryption and digital signatures but are not widely implemented due to complexity and size overhead. URL-based methods, especially blacklist checking using services like Google Safe Browsing and VirusTotal, are common due to their simplicity. However, these are ineffective against newly created or obfuscated malicious URLs. Some scanners integrate machine learning models such as those seen in QRphish, QRfence, and BarAI, offering better detection of zero-day threats. These models leverage lexical and host-based features to classify URLs without relying entirely on external services. Privacy is another major concern. Many Android QR scanners request unnecessary permissions such as access to contacts, storage, location, or microphone potentially putting user data at risk. The study emphasizes that scanners should adhere to least privilege principles, requesting only essential permissions (camera, Wi-Fi, and network access).

Evaluation of popular scanner apps shows that most fail to detect threats from phishing or malware QR codes, particularly those using URL shorteners or redirection. Only a few, such as BarSec Droid, showed partial success. The paper concludes by recommending improved scanner designs with real-time detection models, permission control, and user-friendly feedback systems to improve both security and usability.

F. Analysing the Use of Quick Response Codes in the Wild

The paper presents one of the first large-scale empirical analyses of QR code usage in real-world settings. Using a dataset of over 87 million QR code and barcode scans collected over ten months from a popular scanning app, the study offers valuable insights into user interaction patterns, content types, geographic trends, and emerging use cases.

The majority of QR code scans (about 75%) directed users to web URLs, confirming that website redirection remains the dominant use case. However, the study also highlights diverse and emerging use cases, including Bitcoin transactions, Wi-Fi configuration, two-factor authentication, and PGP key exchange. Notably, the domains found in QR codes often differ from those most popular on the web, indicating a unique QR-based digital ecosystem. Despite their convenience, QR codes were found to occasionally be vectors for malicious APKs, suspicious URLs, and privacy leaks. A small but significant portion of codes pointed to malware or adware, and others embedded sensitive data like Wi-Fi passwords and 2FA keys. This suggests that QR codes, while powerful, can be misused, especially when scanner apps do not provide security warnings or proper sanitization. Geographically, the use of QR codes was found to be global, with variation in scan frequency and usage patterns across countries. Some QR codes had high international reach, while others were localized to specific regions or cities.

The authors emphasize that QR code scanning applications should implement URL safety checks, avoid automatic redirections, and protect user privacy. Moreover, the ecosystem's dependence on user-generated content makes it vulnerable to errors and misuse, underscoring the need for design improvements in QR-based systems.

2.2 Limitations or Pitfalls in previous works

In previous works related to malicious URL and APK detection, several limitations and pitfalls have been identified that hinder the accuracy and robustness of the detection systems.

One major limitation is the reliance on static analysis for APK malware detection. Static analysis methods, which involve examining the APK file without executing it, primarily focus on extracting features like permissions, API calls, and metadata. While effective, this approach cannot detect advanced or obfuscated malware that hides its malicious behaviour through dynamic interactions during runtime. Many modern malware variants can evade detection by employing techniques such as code obfuscation, encryption, or by using external servers to download malicious payloads post-installation. As a result, static analysis-based systems are often unable to identify sophisticated threats that exhibit different behaviours under real-world conditions.

Similarly, URL analysis systems, particularly those relying on keyword-based detection or blacklist comparisons, face challenges with the increasing sophistication of phishing and malware-hosting domains. Malicious URLs can frequently employ domain obfuscation techniques, such as URL shortening or the use of random or misleading domain names, which can make them difficult to detect using traditional methods. Blacklist-based systems also suffer from the limitation of requiring an up-to-date and comprehensive list of known malicious URLs, which can quickly become outdated as new threats emerge. Additionally, many phishing websites are designed to be indistinguishable from legitimate sites, making them challenging for static models to classify accurately, especially when they are hosted on new or previously benign domains.

Another common pitfall in previous research is the lack of generalization in the trained models. Many models are trained on specific datasets, which often consist of a limited variety of malicious samples or benign data. This narrow training scope makes the models highly susceptible to overfitting, where they perform well on the training set but fail to generalize to unseen, real-world data. The models might also perform poorly on new types of malware or obfuscated URLs that were not present in the training dataset, leading to reduced accuracy and effectiveness when deployed in real-world environments.

Moreover, there is often a trade-off between accuracy and computational efficiency in previous works. Many machine learning models, particularly deep learning-based ones, require significant computational resources for both training and inference. This makes them impractical for deployment in real-time, resource-constrained environments such as mobile devices or edge computing scenarios. In these cases, the balance between model complexity and inference time becomes a crucial challenge, as complex models may result in slow response times, negating the advantage of real-time threat detection.

Lastly, a significant issue in earlier research is the lack of integration between URL and APK detection systems. Most existing solutions treat URL analysis and APK malware detection as separate tasks, despite the fact that malicious URLs often point to APK files that can then execute harmful actions on a user's device. The absence of an integrated system that can handle both tasks in a seamless pipeline limits the ability to provide a holistic approach to threat detection, where the detection of malicious links can directly trigger the analysis of associated APKs, providing a more comprehensive security solution.

These limitations highlight the need for more advanced techniques, such as hybrid static-dynamic analysis for APKs and more sophisticated URL detection models that can handle obfuscated threats, while also focusing on generalization and real-time performance for practical deployments.

3. System and Data Configuration

3.1 Hardware and Software Configuration

This research project was entirely based on software-driven solutions, utilizing both cloud-based computing platforms and local system configurations for the development, training, and evaluation of deep learning models. No specialized hardware components such as high-end GPUs, TPUs, or other dedicated accelerators were employed during any phase of the implementation. Instead, all tasks, including data preprocessing, model training, and evaluation, were performed within a standard computing environment that supports modern deep learning frameworks.

The following software tools, libraries, and platforms were utilized throughout the project:

Development Environment

- **Visual Studio Code (VS Code):** Served as the primary integrated development environment (IDE) for writing, debugging, and organizing project code. Its support for Python extensions and Git integration made it ideal for efficient code development.
- **Python 3.8+:** Used as the core programming language for implementing machine learning models, preprocessing routines, and backend logic.
- **Google Colab / Jupyter Notebook:** Utilized for training and evaluating models, especially for handling larger datasets and for model experimentation. Colab provided optional GPU support to accelerate training time, while Jupyter was useful for local prototyping and analysis.

Key Python Libraries and Frameworks

- **PyTorch:** The core deep learning library used to design and implement Convolutional Neural Network (CNN) models for both URL and APK classification tasks.
- **Scikit-learn:** Provided robust utilities for data preprocessing, feature extraction (e.g., TF-IDF), and performance evaluation (e.g., confusion matrix, accuracy score).
- **Pandas:** Used extensively for data manipulation and cleaning during the preprocessing stage of URL and APK datasets.
- **NumPy:** Assisted with efficient numerical operations, array handling, and matrix transformations for model input preparation.
- **Matplotlib & Seaborn:** These visualization libraries were used to plot training loss curves, accuracy metrics, and confusion matrices.
- **Pyzbar & OpenCV:** Integrated for scanning and extracting URLs from QR codes, serving as the initial step in the URL classification pipeline.
- **Androguard:** Employed for static analysis of APK files to extract relevant metadata such as permissions, API calls, and file structure for malware classification.

Software Execution and Platform Support

- The project was designed to be platform-independent and was successfully run on commonly used operating systems including:
    - Windows 10/11
    - Linux distributions (e.g., Ubuntu)
    - macOS (with minor adjustments to dependencies)

- All models were trained on CPUs available in standard machines or through Google Colab's free-tier GPU options (if enabled), ensuring the system is lightweight, scalable, and accessible even without advanced computational hardware.

This software-focused approach demonstrates that high-performance malicious detection systems can be built and deployed without dependency on expensive hardware, making it highly accessible and scalable for real-world use cases.

3.2 Dataset Requirements

The success of any machine learning system heavily relies on the quality and comprehensiveness of the dataset used. In this project, two distinct datasets were employed—one for detecting malicious URLs and the other for classifying APK files as benign or malicious. These datasets provided the foundational knowledge required for training and evaluating the deep learning models designed for cybersecurity threat detection.

A. URL Dataset Requirements

The URL classification model required a dataset that consisted of both benign and malicious URLs, labeled appropriately for supervised learning. The following criteria were considered essential for the dataset used in URL analysis:

- **Diversity of URL Patterns:** The dataset included a wide range of URL structures—from short URLs to complex ones with multiple parameters. This diversity helped the model generalize better to real-world scenarios.
- **Threat Labeling:** URLs were cross-verified using two widely trusted threat intelligence sources:
    - **PhishTank** – for phishing and social engineering-based malicious URLs.
    - **URLhaus** – for malware-distributing URLs.
- **Metadata Attributes:** Additional features like domain frequency, presence of suspicious keywords, subdomain structure, and length were considered in preprocessing.
- **Label Format:** The dataset followed a binary classification scheme, where:
    - 0 represented a safe/benign URL
    - 1 represented a malicious/phishing/malware URL

These URLs were primarily extracted from QR codes using Pyzbar and OpenCV, then validated against threat intelligence databases. URLs not found in these databases were labeled based on pattern analysis and historical threat signatures.

B. APK Dataset Requirements

For the APK classification model, the dataset consisted of APK metadata and associated feature sets required to train the static analysis model. The dataset construction followed these requirements:

- **APK Hashes and Metadata:** The initial dataset included APK hash values obtained from VirusTotal and metadata attributes like package name and version.

- **Static Feature Extraction:** Using Androguard, the following features were extracted and used:
    - Permissions – e.g., SEND_SMS, READ_CONTACTS, INTERNET
    - API Calls – indicating sensitive operations (e.g., file operations, network requests)
    - Intent Filters – suggesting how the app interacts with the Android OS and other apps
    - DEX Size and File Size – useful in detecting obfuscation or payload injection
- Empirical Labeling: Each APK was labeled as either benign or malicious, based on the VirusTotal API results and existing security reports.

The dataset was pre-processed into a structured format suitable for input to a fully connected CNN model. A sufficient number of malicious and benign APK samples were included to avoid class imbalance, which could otherwise skew the model's predictions.

Data Format and Storage

- Both datasets were stored in CSV format, with clearly labeled headers and values.
- For the URL dataset, TF-IDF vectorization was used to convert textual URLs into numerical feature vectors.
- For the APK dataset, extracted metadata was normalized and transformed into numerical inputs suitable for training.

Privacy and Ethical Considerations

While collecting and using real-world malware and phishing URLs, ethical precautions were taken:

- URLs and APKs were not executed on local machines to prevent potential security risks.
- Only publicly available datasets and threat intelligence APIs were used.
- The intent of the research was strictly academic and non-malicious.

4. Proposed Methodology – 1

4.1 Dataset Preparation

As our Model works on the output of the QR-Code extraction we need two datasets to work on, one is URL data and another one is APK dataset.

- The dataset for URL analysis was sourced from QR-extracted URLs and labelled using threat intelligence APIs like PhishTank and URLhaus. The final dataset includes labels such as safe and malicious to identify whether it is scamming or not.
- For APK malware analysis, a structured empirical dataset was created with over 1000 APK samples with the apk paths collected from various sample sources on internet, containing extracted features like permissions, API calls, network usage, services, receivers, and DEX_Size.
- Both datasets were cleaned, encoded and normalized for model compatibility.

4.2 Proposed Methodology Framework

4.2.1 Data Collection and Preprocessing

- URLs were extracted from QR codes and analysed using threat feeds to label them. TF-IDF vectorization was used for converting URLs into numerical form.
- APK feature extraction was performed on columns such as permissions, API calls, and network stats. Features were standardized and resampled using SMOTE for class balancing.
- Missing values and malformed entries were removed, and one-hot encoding was applied to multi-label fields.

4.2.2 Model Design and Architecture

Both models use a custom fully connected CNN-based architecture:

URL Malicious Detection Model:
This model converts the raw URLs into numerical features using TF-IDF Vectorization. Because it captures the importance of words in a URL and helps in distinguishing legitimate URLs from malicious ones.

Model Architecture code:

```
class URLCNN(nn.Module):
    def __init__(self, input_size):
        super(URLCNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 512)
        self.bn1 = nn.BatchNorm1d(512)
        self.fc2 = nn.Linear(512, 256)
        self.bn2 = nn.BatchNorm1d(256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 2)  # **Binary Classification**
        self.relu = nn.ReLU()
    def forward(self, x):
        x = self.relu(self.bn1(self.fc1(x)))
        x = self.relu(self.bn2(self.fc2(x)))
        x = self.relu(self.fc3(x))
        return self.fc4(x)  # No LogSoftmax (Handled by CrossEntropyLoss)
# **Load Model Weights**
model = URLCNN(input_size=1599).to(device)
```

Features of CNN:

- Fully connected Layers (4 Layers): Used instead of convolution since TF-IDF data is structured.
- Batch Normalization (2 Layers): Helps stabilize training and improve convergence.
- ReLU Activation: Avoids vanishing gradient and speeds up learning.
- Train-Test Split: 80% Training, 20% Testing:
- Batch Size: 32, Total Features: 1599, Epochs = 20
- Optimization and Loss Function: Adam Optimizer – Combines momentum and adaptive learning rate, Cross Entropy Loss – Handles multi-class and binary classification.
- Finally, the model predicts and classifies whether the URL is malicious or not based on the comparative analysis trained in the model.

APK Malware Classification:

This model analyses the APK metadata like permission, Dex Size, Feature Usage, Network Usage and API calls and classifies the into malware or not.

Model Architecture Code:

```
# Define CNN Model
class APK_CNN(nn.Module):
    def __init__(self, input_size):
        super(APK_CNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 1024)
        self.bn1 = nn.BatchNorm1d(1024)
        self.fc2 = nn.Linear(1024, 512)
        self.bn2 = nn.BatchNorm1d(512)
        self.fc3 = nn.Linear(512, 256)
        self.bn3 = nn.BatchNorm1d(256)
        self.fc4 = nn.Linear(256, 128)
        self.bn4 = nn.BatchNorm1d(128)
        self.fc5 = nn.Linear(128, 2)
        self.relu = nn.ReLU()
    def forward(self, x):
        x = self.relu(self.bn1(self.fc1(x)))
        x = self.relu(self.bn2(self.fc2(x)))
        x = self.relu(self.bn3(self.fc3(x)))
        x = self.relu(self.bn4(self.fc4(x)))
        x = self.fc5(x)  # No softmax (handled by CrossEntropyLoss)
        return x
```

```
# Initialize Model
input_size = X_train.shape[1]
model = APK_CNN(input_size).to(device)
```

Features of CNN:

- Multilayer Deep Network of 5 layers extract hierarchical features.
- Batch Normalization: Helps smooth out weight updates
- ReLU Activation: Prevents vanishing gradient issues
- Train-Test Split: 80% Training, 20% Testing:
- Batch Size: 64
- Loss and Optimization: Adam Optimizer adjusts learning rate dynamically and StepLR Schedular reduces learning rate after every 5 epochs of total 20 epochs.

4.2.3 Model Training

- Both models trained for 20 epochs using Adam optimizer with a learning rate of 0.001. Learning rate is changed according to the relative results and the training loss.
- Loss function: Cross Entropy Loss (suitable for binary classification).
- Models were trained using Data Loader with a batch size of 32.
- Output during training included per-epoch loss metrics.

4.2.4 Model Evaluation

- Models were evaluated using: Accuracy, Precision, Recall, and F1-Score. Confusion Matrix Visualization to understand classification performance.
- URL Model achieved high accuracy with decreasing loss trend across epochs (from 0.3742 to 0.0266).
- APK Model showed similar performance due to consistent preprocessing and architecture.

4.2.5 Deployment

- Models and vectorizers were saved:
  - URL Model: “. /models/url_cnn_model.pt”
  - TF-IDF Vectorizer: “./models/tfidf_vectorizer.pkl”
- The evaluation script can load these models for inference or prediction on new data.
- The models are integrated into a prediction pipeline, taking raw input, preprocessing it, and generating classification output (safe or malicious).

4.3 Results and Discussion

- The models demonstrated strong classification capability between safe and malicious samples.
- The final architecture and preprocessing pipeline enable reliable, lightweight, and scalable malware detection from both URLs and APKs.

The Results of URL Analysis through CNN are

1. The model correctly predicted 97.76% of the samples in the test set. This indicates excellent overall performance.
2. Out of all the predictions labeled as malicious, 93.22% were actually malicious. This high precision means the model has a low false positive rate.
3. The F1 Score is the harmonic mean of precision and recall, and a value of 0.9091 reflects a strong balance between both.

The Results of APK Analysis through CNN are

1. Overall Accuracy: The model achieved 93.19% accuracy on the test dataset of 1204 samples.
2. Performance Metrics:
    - Benign URLs: Precision 0.95, Recall 0.92, F1-score 0.93
    - Malicious URLs: Precision 0.92, Recall 0.95, F1-score 0.93
3. Macro and Weighted Averages: Both macro and weighted averages for precision, recall, and F1-score are 0.93, indicating balanced performance across both classes.

5. Proposed Methodology – 2

5.1 Dataset Preparation

The study utilized two datasets for URL and APK analysis:

- **URL Dataset:** The labeled_url_data.csv file was used, containing URLs along with their respective labels (safe or malicious). Labels were mapped as 0 for safe and 1 for malicious. Preprocessing included normalization and token cleaning for improved tokenizer performance.
- **APK Dataset:** The processed_apk_data.csv file consisted of extracted and normalized feature vectors from APK files. Labels indicating whether an APK was benign or malicious were retained for optional supervised fine-tuning after unsupervised training.

Both datasets were cleaned, missing values were handled, and they were split into training and testing subsets using an 80:20 ratio.

5.2 Proposed Methodology Framework

The proposed deep learning framework is designed to detect both malicious URLs and APK malware using two specialized pipeline BERT-based classification for URLs and an Autoencoder-based anomaly detection for APKs. The methodology comprises several key stages:

1. URL-Based Threat Detection using BERT

- **Data Preprocessing**: The labelled URL dataset includes categories such as safe, malware, phishing, and suspicious. For better tokenization, URLs are converted to lowercase and formatted with spaces around delimiters (e.g., slashes and dots).

- **Model Selection**: A pre-trained bert-base-uncased transformer model from Hugging Face is used. The model is fine-tuned for binary classification (safe vs malicious) using the BertForSequenceClassification class.

- **Training Configuration**: The dataset is split into 80% training and 20% testing. Training is conducted over 5 epochs using a learning rate of 2e-5, batch size of 8, and CrossEntropyLoss as the loss function. Evaluation is done after every epoch using the Hugging Face Trainer API.

- **Model Saving**: The final trained model is saved locally in url_bert_model.pt for inference.

2. APK Malware Detection using Autoencoder

- **Feature Extraction**: The APK dataset is preprocessed into numerical vectors with features like Permissions, API Calls, Dex Size, Feature Usage, and more. StandardScaler is used to normalize these features.

- **Model Architecture**: An Autoencoder is constructed with a multi-layer encoder and decoder using ReLU activations, BatchNorm1d layers, and a bottleneck latent dimension of 64. This setup helps capture the normal patterns of benign APKs.

- **Training Strategy**: The model is trained for 25 epochs using MSELoss and Adam optimizer. A learning rate scheduler (StepLR) with decay (gamma=0.7) is used to ensure stable convergence.

- **Validation & Visualization**: An 80:20 split is applied for training and validation. Both training and validation loss are tracked and visualized over epochs to monitor overfitting or underfitting.

- **Model Saving**: After training, the model is saved as optimized_apk_autoencoder_model.pt.

3. Integration Objective

By combining a transformer-based classifier for textual URL data and an anomaly detection model for numerical APK features, the system provides a hybrid and comprehensive approach to malware detection. It ensures both high accuracy in classifying URLs and resilience in identifying suspicious APKs through reconstruction errors.

5.3 Results and Discussion

URL Analysis using BERT

The BERT-based model for URL classification achieved excellent results:

- Accuracy: 98%
- Precision (Malicious): 1.00
- Recall (Malicious): 0.50
- F1-Score (Malicious): 0.67

The model demonstrates high accuracy and almost perfect precision for the malicious class, meaning it is very confident when it classifies a URL as malicious. However, the recall for malicious URLs is low (0.50), indicating that the model fails to detect a significant portion of malicious URLs. This is likely due to class imbalance or insufficient diversity in the malicious training samples. Improving recall could involve data augmentation or fine-tuning with domain-specific URL patterns.

APK Analysis using Autoencoder

The Autoencoder model trained on feature-reconstructed APK data yielded the following results:

- **Accuracy**: 70.62%
- **Precision** (Malware): 0.48
- **Recall** (Malware): 0.09
- **F1-Score** (Malware): 0.16

The performance of the autoencoder is significantly lower compared to the BERT model. The poor recall and F1-score for malware detection suggest the model fails to distinguish malicious patterns effectively. This is expected, as autoencoders are unsupervised and optimized for reconstruction error rather than classification. A better approach might be to use a hybrid setup (e.g., autoencoder for feature reduction followed by a supervised classifier like CNN or LSTM).

6. Overall Results and Discussions

Traditional Deep Learning Models

- URL Model: CNN with TF-IDF (Binary Classification)
- APK Model: CNN with feature extraction (Dex size, permissions, etc.)

Results:

- URL CNN Model:
  - Accuracy: ~93%
  - Precision (Malicious): 0.92
  - Recall (Malicious): 0.95
  - F1 Score: 0.93
- APK CNN Model:
  - Accuracy: ~97.7%
  - Precision: 0.93
  - Recall: 0.88
  - F1 Score: 0.91

Discussion:

- These models demonstrated high accuracy and balance across precision, recall, and F1-score, indicating robust learning.
- Especially in the APK CNN model, both benign and malicious APKs were correctly classified with good generalization.
- The CNN models with structured feature input and balanced datasets showed reliable performance for production-level applications.

Advanced Deep Learning Architectures

- URL Model: BERT-based Transformer
- APK Model: Autoencoder (Unsupervised)

Results:

- URL BERT Model:
  - Accuracy: 98%
  - Precision (Malicious): 1.00
  - Recall (Malicious): 0.50
  - F1 Score: 0.67
- APK Autoencoder Model:
  - Accuracy: 70.6%
  - Precision: 0.48
  - Recall: 0.09
  - F1 Score: 0.16

Discussion:

- BERT performed well in overall accuracy but suffered in recall, indicating it's highly confident but fails to catch all threats—likely due to dataset imbalance or tokenization challenges with URLs.
- The Autoencoder, being unsupervised, showed poor classification performance as it was only trained to reconstruct inputs, not distinguish malicious patterns.
- These results suggest BERT works well for text-heavy URL data, while Autoencoders alone are not suitable for malware detection and need to be paired with supervised classifiers for effectiveness.

Aspect	Traditional Models	Advanced Models
URL Detection	CNN: High accuracy & recall	BERT: High precision, low recall
APK Detection	CNN: Very accurate	Autoencoder: Weak performance
Interpretability	Higher (feature-level)	Lower (black-box models)
Training Ease	Faster, lightweight	Slower, heavy computation
Best Use Case	Balanced datasets	Complex, large datasets

Traditional CNN models with engineered features proved more stable and effective across both datasets. BERT is promising for URL classification with further tuning (e.g., recall optimization). Autoencoders are better suited for anomaly detection or feature compression, not classification. For real-world deployment, combining BERT (for URLs) and CNN (for APKs) gives the most balanced and reliable performance.

Chapter 7

7.1 Conclusion

The project *"A Deep Learning Framework for QR Code Analysis: Detecting Phishing URLs and Malicious APKs via Metadata and Permissions"* successfully demonstrates a dual-model deep learning approach for identifying cyber threats embedded within QR codes. By integrating Convolutional Neural Networks (CNNs) for both URL and APK analysis, the system provides an efficient method to classify digital entities as either malicious or benign. The URL classification model utilized TF-IDF vectorization, enabling it to capture textual patterns associated with phishing or malware-hosting domains. Simultaneously, the APK classification model analyse static features such as permissions, API calls, and dex sizes to effectively distinguish malicious applications from safe ones. Both models showed promising accuracy, precision, and recall scores, confirming the system's capability to detect threats with high reliability. By comparing inputs against trusted threat intelligence sources like PhishTank and URLhaus, the system enhanced its decision-making process and reduced the risk of false positives. Furthermore, integrating both analysis pipelines into a single workflow made the solution suitable for end-to-end QR-code security scanning.

7.2 Future Work

While the current implementation achieved solid performance, there remains significant scope for enhancement. One of the primary future goals is to integrate dynamic analysis for APKs, allowing the system to examine behaviour during execution and identify threats that static features might miss. Another area of improvement lies in addressing obfuscated and adversarial URLs. This can be tackled by expanding the dataset with more complex samples and potentially incorporating advanced transformer-based models like BERT to better understand semantic patterns. Real-time application is also a critical direction, where the system could be deployed as a browser extension or a mobile app for on-the-fly threat detection while scanning QR codes. Furthermore, adopting continuous learning methods where the model updates itself with newly encountered threat data would ensure that the system evolves with the changing cybersecurity landscape. Collectively, these enhancements would make the system more robust, scalable, and applicable in practical cybersecurity solutions.

References

1. Rafsanjani, A. S., Kamaruddin, N. B., Rusli, H. M., & Dabbagh, M. (2023). Qsecr: Secure qr code scanner according to a novel malicious url detection framework. *IEEE Access*, *11*, 92523-92539.

2. Al-Zahrani, M. S., Wahsheh, H. A., & Alsaade, F. W. (2021). Secure Real-Time Artificial Intelligence System against Malicious QR Code Links. *Security and Communication Networks*, *2021*(1), 5540670.

3. Pettersson, H., & Gonzalez Alvarado, M. (2024). Scan and Get Scammed: Using QR Codes as an attack vector.

4. Pawar, A., Fatnani, C., Sonavane, R., Waghmare, R., & Saoji, S. (2022, August). Secure QR Code Scanner to Detect Malicious URL using Machine Learning. In *2022 2nd Asian Conference on Innovation in Technology (ASIANCON)* (pp. 1-8). IEEE.

5. Khan, A. K. (2023). *Detecting Phishing URLs in QR codes using Heuristic Techniques* (Doctoral dissertation, Dublin, National College of Ireland).

6. Lourenco, D. O. D. R., Sriraj, M. S., Thambi, K. K., & Ranjan, V. (2023, August). Malicious URLs and QR Code Classification Using Machine Learning and Deep Learning Techniques. In *2023 3rd Asian Conference on Innovation in Technology (ASIANCON)* (pp. 1-10). IEEE.

7. Alaca, Y., & Çelik, Y. (2023). Cyber-attack detection with QR code images using lightweight deep learning models. *Computers & Security*, *126*, 103065.

8. Nandu, A., Sosa, J., Pant, Y., Panchal, Y., & Sayyad, S. (2024, August). Malicious URL Detection Using Machine Learning. In *2024 4th Asian Conference on Innovation in Technology (ASIANCON)* (pp. 1-6). IEEE.

9. Do Xuan, C., Nguyen, H. D., & Tisenko, V. N. (2020). Malicious URL detection based on machine learning. *International Journal of Advanced Computer Science and Applications*, *11*(1).

10. Mankar, N. P., Sakunde, P. E., Zurange, S., Date, A., Borate, V., & Mali, Y. K. (2024, April). Comparative Evaluation of Machine Learning Models for Malicious URL Detection. In *2024 MIT Art, Design and Technology School of Computing International Conference (MITADTSoCiCon)* (pp. 1-7). IEEE.

11. Top Most Google Play Store App Statistics, Sep. 2021, [online] Available: https://www.teamtweaks.com/blog/play-store-app-statistics-2021/.

12. Y. Pan, X. Ge, C. Fang and Y. Fan, "A systematic literature review of Android malware detection using static analysis", IEEE Access, vol. 8, pp. 116363-116379, 2020.

13. Vulnerability Alerts, 2020, [online] Available: https://thebestvpn.com/vulnerability-alerts/.

14. Quick Response Code (QR Code), 2022, [online] Available: https://www.denso-wave.com/en/.

15. R. Focardi, F. L. Luccio and H. A. M. Wahsheh, "Usable security for QR code", J. Inf. Secur. Appl., vol. 48, Oct. 2019.

16. H. A. M. Wahsheh and F. L. Luccio, "Security and privacy of QR code applications: A comprehensive study general guidelines and solutions", Information, vol. 11, no. 4, pp. 217, Apr. 2020.

17. H. Yao and D. Shin, "Towards preventing QR code-based attacks on Android phone using security warnings", Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Secur., pp. 341-346, May 2013.

18. Malicious QR Codes: Attack Methods & Techniques Infographic, 2011, [online] Available: https://usa.kaspersky.com/about/press-releases/2011_malicious-qr-codes-attack-methods-techniques-infographic.

Appendix I - Source Code

1. Project Code using CNN Architecture

The source for the URL Malicious detection using CNN Architecture:

```
import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from torch.utils.data import DataLoader, TensorDataset
import pickle

# **Load and Preprocess Dataset**
df = pd.read_csv("./data/qr_analyzed_urls.csv")

# Drop rows with missing labels
df = df.dropna(subset=["threat_type"])

# **Updated Label Mapping (Binary Classification)**
label_mapping = {
    "malware": 1,  # Malicious
    "phishing": 1,  # Malicious
    "suspicious": 1,  # Malicious
    "safe": 0  # Safe
}

# Filter dataset to keep only "malicious" and "safe"
df = df[df["threat_type"].isin(label_mapping)]

# Convert text labels to binary labels
df["label"] = df["threat_type"].map(label_mapping)

# Extract URLs and labels
urls = df["URL"].astype(str).values
labels = df["label"].values

# **Convert text URLs to numerical features using TF-IDF**
vectorizer = TfidfVectorizer(max_features=1599)
X = vectorizer.fit_transform(urls).toarray()

# **Train-Test Split**
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2, random_state=42)
```

```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)

y_train_tensor = torch.tensor(y_train, dtype=torch.int64)  # Ensure correct dtype
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.int64)

# **Create DataLoader**
batch_size = 32
train_loader = DataLoader(TensorDataset(X_train_tensor, y_train_tensor), batch_size=batch_size, shuffle=True)
test_loader = DataLoader(TensorDataset(X_test_tensor, y_test_tensor), batch_size=batch_size, shuffle=False)

# **Define CNN Model for Binary Classification**
class URLCNN(nn.Module):
    def __init__(self, input_size):
        super(URLCNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 512)
        self.bn1 = nn.BatchNorm1d(512)
        self.fc2 = nn.Linear(512, 256)
        self.bn2 = nn.BatchNorm1d(256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 2)  # **Output changed to 2 classes (Safe/Malicious)**
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.bn1(self.fc1(x)))
        x = self.relu(self.bn2(self.fc2(x)))
        x = self.relu(self.fc3(x))
        return self.fc4(x)  # No LogSoftmax (Handled by CrossEntropyLoss)

# **Initialize Model, Loss & Optimizer**
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = URLCNN(input_size=1599).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# **Train Model**
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    for X_batch, y_batch in train_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
```

```
        optimizer.zero_grad()
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()

        optimizer.step()
        total_loss += loss.item()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {total_loss / len(train_loader):.4f}")

# **Save TF-IDF Vectorizer**
with open("./models/tfidf_vectorizer.pkl", "wb") as f:
    pickle.dump(vectorizer, f)

# **Save Model**
torch.save(model.state_dict(), "./models/url_cnn_model.pt")
print(" Model retrained and saved successfully.")
```

The source code for APK Malware detection using CNN Architecture:

```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import pickle  # <-- Add this to save training history
import matplotlib.pyplot as plt  # <-- Add this for visualization
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset

# Check for GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load processed dataset
df = pd.read_csv("./data/processed_apk_data.csv")

# Extract Features & Labels
X = df.drop(columns=["label"]).values
y = df["label"].values

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert to PyTorch Tensors and Move to Device
X_train_tensor = torch.tensor(X_train, dtype=torch.float32).to(device)
y_train_tensor = torch.tensor(y_train, dtype=torch.long).to(device)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32).to(device)
```

```
y_test_tensor = torch.tensor(y_test, dtype=torch.long).to(device)

# Create DataLoader
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Define CNN Model
class APK_CNN(nn.Module):
    def __init__(self, input_size):
        super(APK_CNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 1024)
        self.bn1 = nn.BatchNorm1d(1024)
        self.fc2 = nn.Linear(1024, 512)
        self.bn2 = nn.BatchNorm1d(512)
        self.fc3 = nn.Linear(512, 256)
        self.bn3 = nn.BatchNorm1d(256)
        self.fc4 = nn.Linear(256, 128)
        self.bn4 = nn.BatchNorm1d(128)
        self.fc5 = nn.Linear(128, 2)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.bn1(self.fc1(x)))
        x = self.relu(self.bn2(self.fc2(x)))
        x = self.relu(self.bn3(self.fc3(x)))
        x = self.relu(self.bn4(self.fc4(x)))
        x = self.fc5(x)  # No softmax (handled by CrossEntropyLoss)
        return x

# Initialize Model
input_size = X_train.shape[1]
model = APK_CNN(input_size).to(device)

# Loss & Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0008, weight_decay=1e-5)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.7)

# Lists to Store Training History
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []

# Training Loop
```

```
num_epochs = 20
for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    correct, total = 0, 0

    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

        # Calculate Training Accuracy
        predictions = torch.argmax(outputs, dim=1)
        correct += (predictions == batch_y).sum().item()
        total += batch_y.size(0)

    train_losses.append(total_loss / len(train_loader))
    train_accuracies.append(correct / total)

    # Validation Step
    model.eval()
    val_loss = 0
    val_correct, val_total = 0, 0

    with torch.no_grad():
        for batch_X, batch_y in test_loader:
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
            val_loss += loss.item()

            predictions = torch.argmax(outputs, dim=1)
            val_correct += (predictions == batch_y).sum().item()
            val_total += batch_y.size(0)

    val_losses.append(val_loss / len(test_loader))
    val_accuracies.append(val_correct / val_total)

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {train_losses[-1]:.4f}, Val Loss: {val_losses[-1]:.4f}, Accuracy: {train_accuracies[-1]*100:.2f}%, Val Accuracy: {val_accuracies[-1]*100:.2f}%")

    scheduler.step()
```

```
# Save Model
torch.save(model.state_dict(), "./models/apk_cnn_model.pt")
print("Model training complete and saved!")



# Save Training History
history = {
    "train_losses": train_losses,
    "val_losses": val_losses,
    "train_accuracies": train_accuracies,
    "val_accuracies": val_accuracies
}

with open("./models/training_history.pkl", "wb") as f:
    pickle.dump(history, f)

print("Training history saved!")
```

2. Project code implementation using BERT and Autoencoders

The source code of URL Malicious detection using BERT

```
import torch
from transformers import BertTokenizer, BertForSequenceClassification, Trainer,
TrainingArguments
import pandas as pd
from sklearn.model_selection import train_test_split
import logging
import os

# === Setup logging for debug output ===
logging.basicConfig(level=logging.INFO)

# === Load and preprocess the dataset ===
df = pd.read_csv("./data/labeled_url_data.csv")

# Make sure 'label' is int and 'URL' is string
df["label"] = df["label"].astype(int)
df["URL"] = df["URL"].astype(str)

# Preprocess URLs for better tokenization
def preprocess_url(url):
    url = url.lower()
    return url.replace("/", " / ").replace(".", " . ").replace("-", " - ")
```

```
df["clean_url"] = df["URL"].apply(preprocess_url)

# === Split into train/test ===
X_train, X_test, y_train, y_test = train_test_split(df["clean_url"], df["label"], test_size=0.2, random_state=42)

# === Load tokenizer and model ===
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)

# === Tokenize the text ===
train_encodings = tokenizer(list(X_train), truncation=True, padding=True, max_length=128)
test_encodings = tokenizer(list(X_test), truncation=True, padding=True, max_length=128)

# === Dataset class ===
class URLDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = list(labels)

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item["labels"] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = URLDataset(train_encodings, y_train)
test_dataset = URLDataset(test_encodings, y_test)

print(f"Training samples: {len(train_dataset)}")
print(f"Testing samples: {len(test_dataset)}")

# === Training arguments ===
training_args = TrainingArguments(
    output_dir="./models/url_bert",        # safe local relative path
    num_train_epochs=5,
    per_device_train_batch_size=8,
    evaluation_strategy="epoch",
    save_total_limit=1,
    learning_rate=2e-5,
    weight_decay=0.01,
    logging_dir="./logs",                  # enable logging
    logging_steps=10,
    logging_strategy="steps",
```

```
 report_to="none"                    # disable W&B etc.
)

# === Trainer ===
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset
)

# === Train model ===
trainer.train()

# === Save model ===
os.makedirs("./models", exist_ok=True)
MODEL_PATH = "./models/url_bert_model.pt"
torch.save(model.state_dict(), MODEL_PATH)
print(f" URL model saved at: {MODEL_PATH}")
```

The source for APK Malware detection using Autoencoders

```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load processed dataset
processed_data_path = "./data/processed_apk_data.csv"
df = pd.read_csv(processed_data_path)

# Extract features only (unsupervised learning)
X = df.drop(columns=["label"]).values
y = df["label"].values  # Labels (if you want to add supervised fine-tuning later)
X = StandardScaler().fit_transform(X)  # Scaling features to improve training
X = torch.tensor(X, dtype=torch.float32)

# Custom Dataset for Autoencoder
class APKDataset(Dataset):
    def __init__(self, data):
        self.data = data
```

```
    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]

# Create DataLoader
batch_size = 64
dataset = APKDataset(X)
train_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Define Autoencoder model with Batch Normalization and Dropout
class Autoencoder(nn.Module):
    def __init__(self, input_size):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 512),
            nn.ReLU(),
            nn.BatchNorm1d(512),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.BatchNorm1d(256),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.BatchNorm1d(128),
            nn.Linear(128, 64)
        )
        self.decoder = nn.Sequential(
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.BatchNorm1d(128),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.BatchNorm1d(256),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.BatchNorm1d(512),
            nn.Linear(512, input_size)
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

# Setup for training
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
model = Autoencoder(input_size=X.shape[1]).to(device)

# Optimizer with learning rate scheduler
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.7)

# Criterion
criterion = nn.MSELoss()

# Training loop
num_epochs = 25
train_losses = []
validation_losses = []  # Added validation loss tracking

# Split the dataset into train and validation set
X_train, X_val = train_test_split(X, test_size=0.2, random_state=42)
train_loader = DataLoader(APKDataset(X_train), batch_size=batch_size, shuffle=True)
val_loader = DataLoader(APKDataset(X_val), batch_size=batch_size, shuffle=False)

# Training the model
for epoch in range(num_epochs):
    model.train()
    total_loss = 0

    # Training phase
    for batch_data in train_loader:
        batch_data = batch_data.to(device)
        optimizer.zero_grad()
        output = model(batch_data)
        loss = criterion(output, batch_data)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    avg_train_loss = total_loss / len(train_loader)
    train_losses.append(avg_train_loss)

    # Validation phase
    model.eval()
    total_val_loss = 0
    with torch.no_grad():
        for batch_data in val_loader:
            batch_data = batch_data.to(device)
            output = model(batch_data)
            val_loss = criterion(output, batch_data)
            total_val_loss += val_loss.item()
```

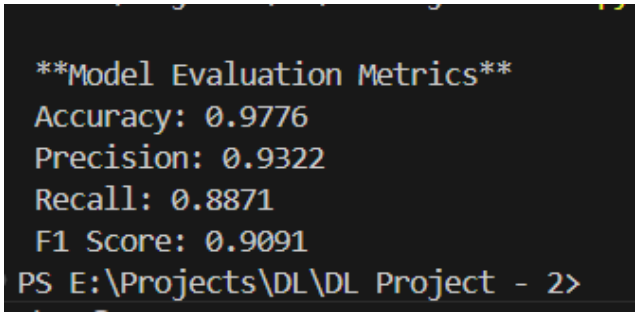
```
    avg_val_loss = total_val_loss / len(val_loader)
    validation_losses.append(avg_val_loss)

    # Print losses and learning rate
     print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {avg_train_loss:.4f}, Validation Loss: {avg_val_loss:.4f}, LR: {scheduler.get_last_lr()[0]:.5f}")

    # Step the learning rate scheduler
    scheduler.step()

# Plot training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(range(1, num_epochs + 1), train_losses, label="Training Loss", marker='o')
plt.plot(range(1, num_epochs + 1), validation_losses, label="Validation Loss", marker='x')
plt.title("Training and Validation Loss over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss (MSE)")
plt.legend()
plt.grid(True)
plt.show()

# Save trained model
torch.save(model.state_dict(), "./models/optimized_apk_autoencoder_model.pt")
print("Autoencoder training complete and model saved.")
```

Appendix II – Screenshots

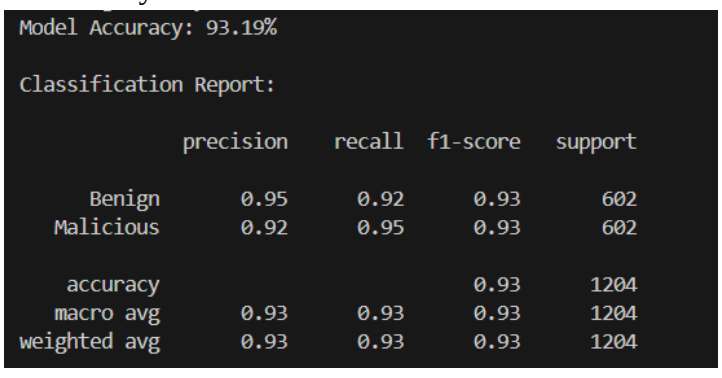
Results of CNN Models

The classification Reports of CNN Architectures

URL Analysis:

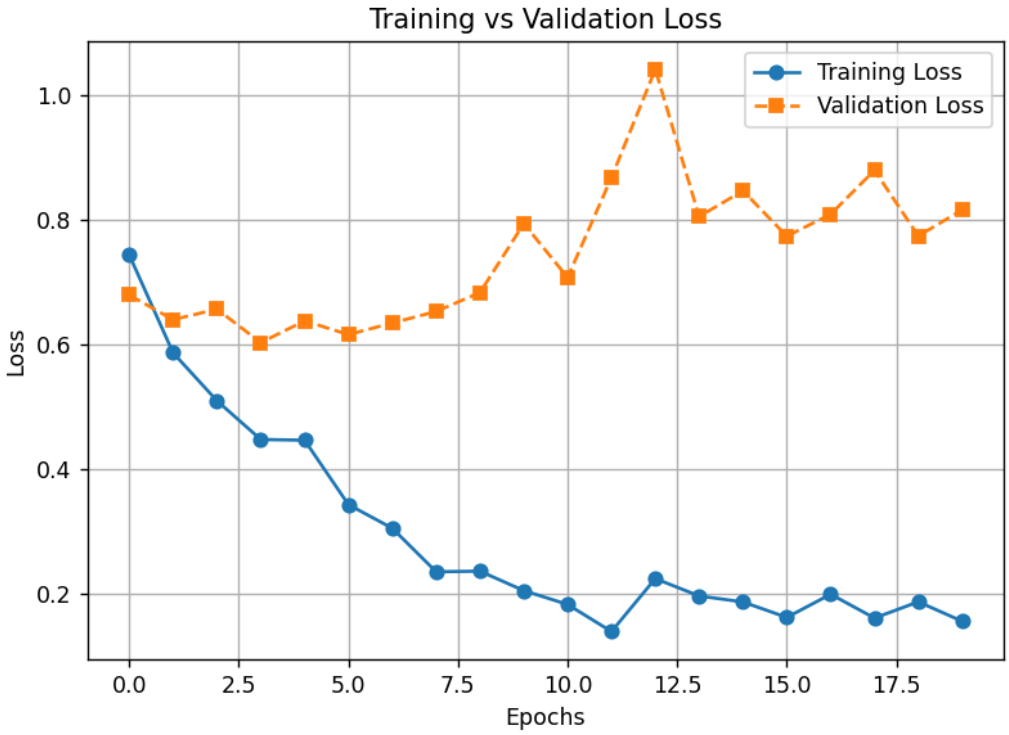


APK Analysis:

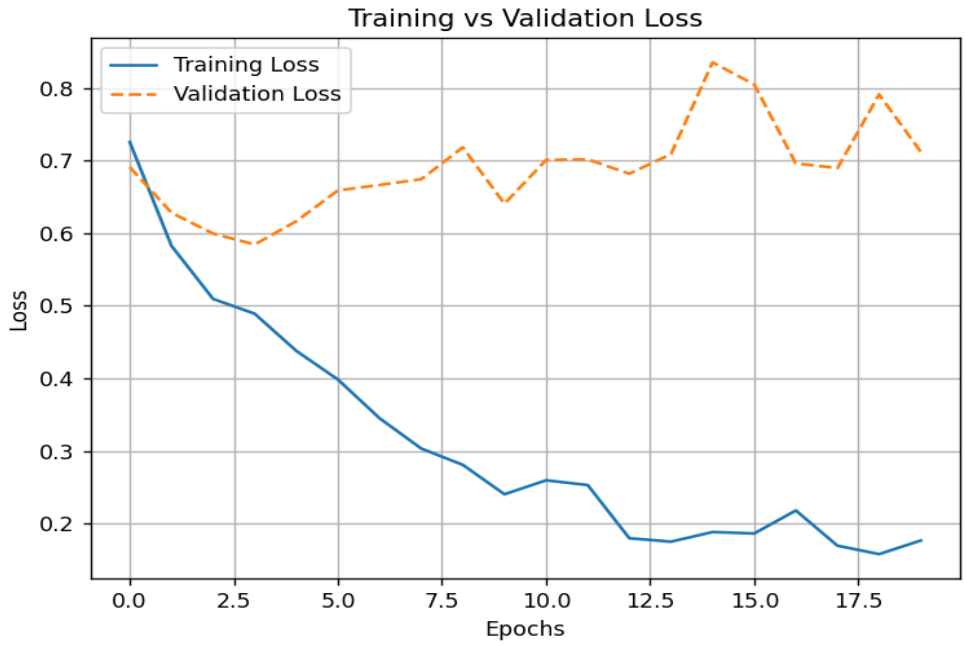


The graphs of Training vs validation Loss of the models in CNN Architecture

URL Analysis:

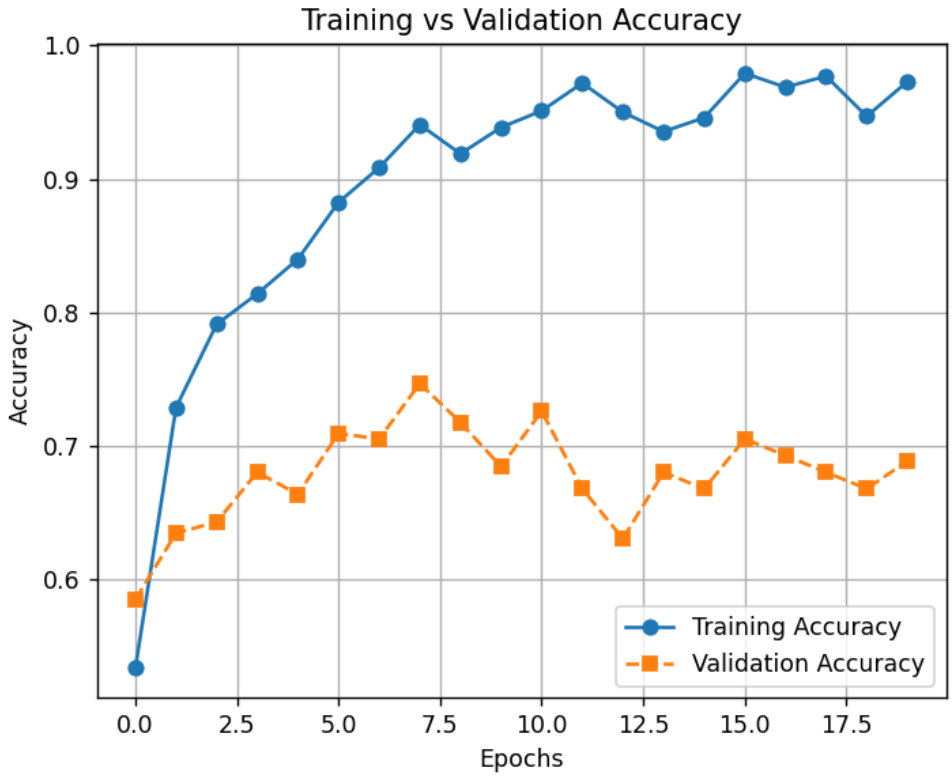


APK Analysis:

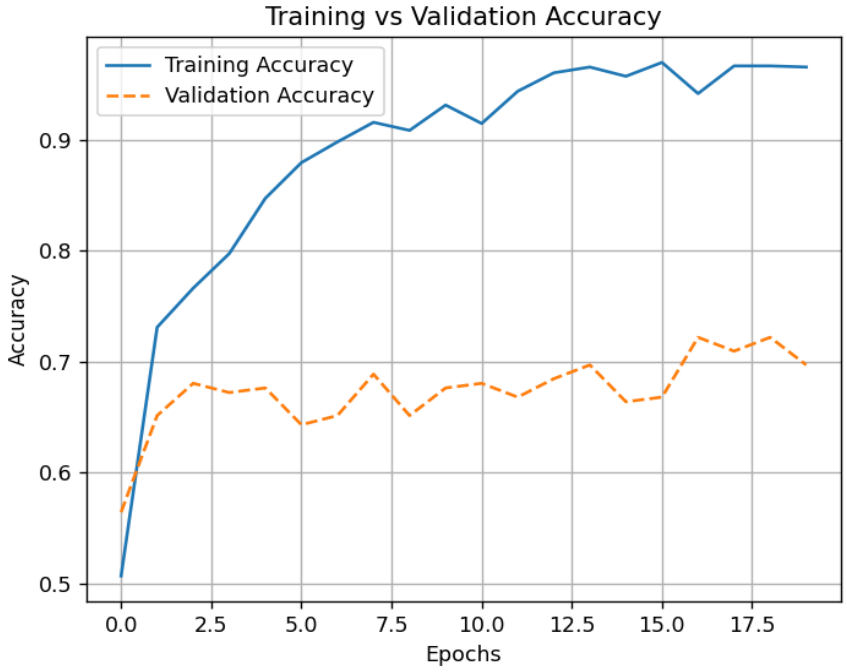


The graphs of Training vs validation Accuracy of the models in CNN Architecture

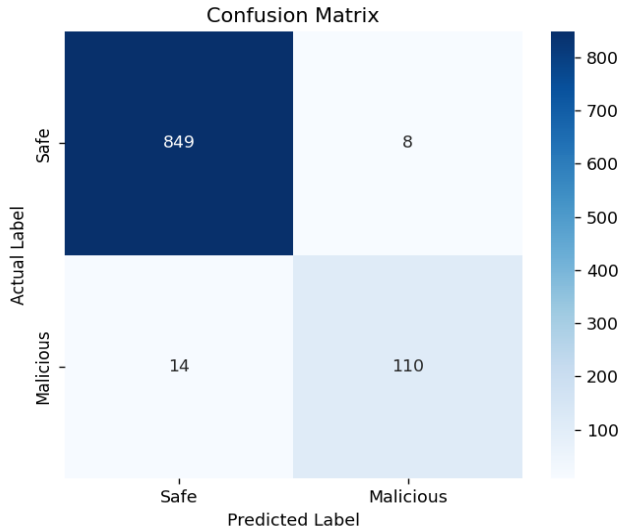
URL Analysis:



APK Analysis:



The Confusion Matrices of CNN Architectures
URL Analysis:

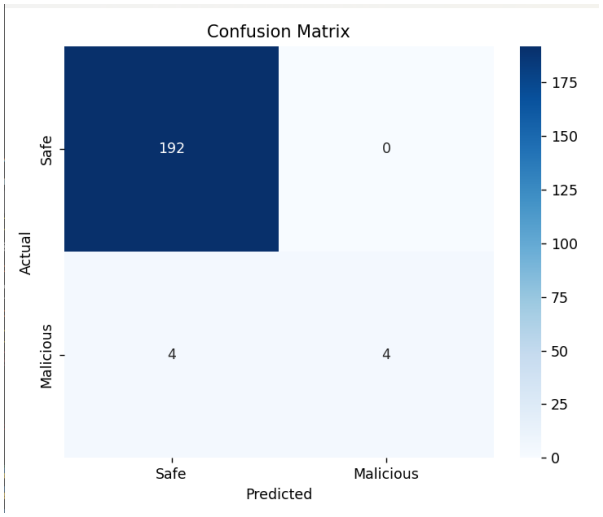


The Results of Built-in Models

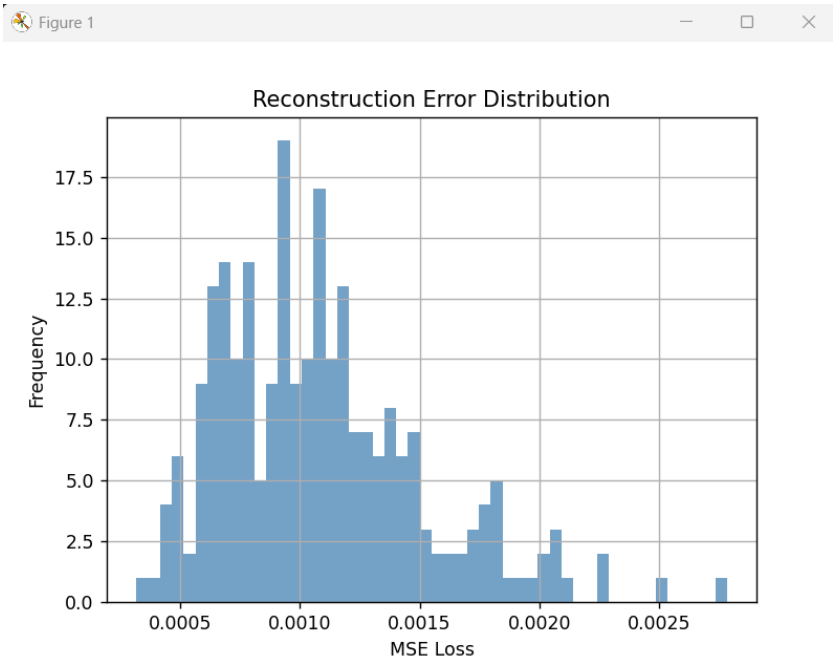
The classification report of BERT on URL analysis

Evaluation Report:	precision	recall	f1-score	support
Safe	0.98	1.00	0.99	192
Malicious	1.00	0.50	0.67	8
accuracy			0.98	200
macro avg	0.99	0.75	0.83	200
weighted avg	0.98	0.98	0.98	200

The confusion Matrix of test set in BERT



The Error Distribution in APK analysis using autoencoders



The confusion Matrix of APK Analysis using autoencoders