UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Design and Implementation of Device Continuous Behavioral Authentication Mechanisms for Improved Risk Management and Preventing Fraud in Communication Systems

Diploma Thesis

Lantzos Stergios

Supervisor: Athanasios Korakis

Volos, September 2024

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Design and Implementation of Device Continuous Behavioral Authentication Mechanisms for Improved Risk Management and Preventing Fraud in Communication Systems

Diploma Thesis

Lantzos Stergios

Supervisor: Athanasios Korakis

Volos, September 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδιασμός και Υλοποίηση Μηχανισμών Συνεχούς Αυθεντικοποίησης Συσκευών βάση της Συμπεριφοράς Λειτουργίας τους για βελτιωμένη Διαχείριση Ρίσκου και Αποφυγή Απάτης σε Συστήματα Επικοινωνιών

Διπλωματική Εργασία

Λάντζος Στέργιος

Επιβλέπων: Αθανάσιος Κοράκης

Βόλος, Σεπτέμβριος 2024

Approved by the Examination Committee:

Supervisor **Athanasios Korakis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Parisis Flegkas**
Assistant Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Antonios Argyriou**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

I would like to sincerely thank Professor Athanasios Korakis for giving me the opportunity to work on my diploma thesis and for allowing me to become part of the Network Implementation Testbed Laboratory (NITlab) team. This experience granted me the opportunity to collaborate with distinguished individuals and gain knowledge that far exceeded my expectations. I would also like to extend my gratitude to Assistant Professor Parisis Flegkas and Associate Professor Antonios Argyriou for serving on the examination committee and approving this thesis.

I extend my deepest gratitude to Senior Researcher and Postdoctoral Apostolos Apostolaras for his invaluable support throughout this thesis. His guidance and insight consistently steered me in the right direction, and his advice on organizational skills greatly enhanced my work and research. I would also like to extend my gratitude to Ilias Sirigos, PhD Candidate in the Department of Electrical and Computer Engineering at the University of Thessaly, for generously sharing his expertise and patiently answering all my questions and concerns throughout my thesis journey.

Last but most importantly, I would like to thank my Mom and Dad for their continuous encouragement, which supported me in achieving my goals throughout this challenging yet emotionally fulfilling 6-year journey. Their belief in me has been my greatest source of strength, and without them, this dream would have never come to life. Every step I've taken is a reflection of their endless dedication, and for that, I am forever grateful.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Lantzos Stergios

Diploma Thesis

Design and Implementation of Device Continuous Behavioral Authentication Mechanisms for Improved Risk Management and Preventing Fraud in Communication Systems

Lantzos Stergios

Abstract

In today's era of globalization and digitalization, the prevalence of electronic formats leads to an increase in fraud incidents. Current authentication methods primarily rely on user-held elements such as PINs, biometrics and two-factor authentication. While these methods are typically reliable, they remain vulnerable to compromise by fraudsters due to data breaches, phishing attacks, or other physical threats, such as the theft of a device. Once compromised, these credentials can be misused to gain unauthorized access to sensitive information, financial accounts, and personal data, leading to significant losses and privacy violations. Furthermore, attackers can exploit vulnerabilities in these authentication methods, bypassing security measures that users believe to be secure.

To address this issue, we propose the concept of *Continuous Authentication*. The primary goal is to continuously authenticate users based on their device's behavioral patterns. By collecting logs generated by the Android OS and applying machine learning techniques to analyze them, we can identify typical behavior patterns of the device. This approach enables the detection of anomalous activities that deviate from the device's usual operational patterns, allowing for early intervention in cases of theft or misuse. Continuous Authentication not only enhances security by providing ongoing verification but also reduces reliance on static credentials, which are prone to compromise. This method provides a higher level of protection against attacks and unauthorized access, as it continuously monitors the device's behavior rather than relying solely on credentials that can be compromised.

Keywords:

Fraud incidents, Android System Logs, Machine Learning, Anomalous Behaviors, Device Theft, Continuous Authentication, Authentication Mechanisms

Διπλωματική Εργασία

Σχεδιασμός και Υλοποίηση Μηχανισμών Συνεχούς Αυθεντικοποίησης Συσκευών βάση της Συμπεριφοράς Λειτουργίας τους για βελτιωμένη Διαχείριση Ρίσκου και Αποφυγή Απάτης σε Συστήματα Επικοινωνιών

Λάντζος Στέργιος

Περίληψη

Στην εποχή της παγκοσμιοποίησης και του ψηφιακού μετασχηματισμού, η επικράτηση ηλεκτρονικών μορφών οδηγεί ολοένα σε αύξηση των περιπτώσεων απάτης. Οι τρέχουσες μέθοδοι αυθεντικοποίησης βασίζονται κυρίως σε στοιχεία που κρατούν οι χρήστες, όπως οι προσωπικοί αριθμοί ταυτοποίησης (PINs), τα βιομετρικά δεδομένα και ο έλεγχος ταυτότητας δύο παραγόντων. Αν και αυτές οι μέθοδοι είναι συνήθως αξιόπιστες, παραμένουν ευάλωτες σε παραβιάσεις από απατεώνες λόγω επιθέσεων ηλεκτρονικού ψαρέματος ή άλλων φυσικών απειλών, όπως η κλοπή μιας συσκευής. Μόλις παραβιαστούν, αυτά τα στοιχεία μπορούν να χρησιμοποιηθούν για την απόκτηση μη εξουσιοδοτημένης πρόσβασης σε ευαίσθητες πληροφορίες, χρηματοοικονομικούς λογαριασμούς και προσωπικά δεδομένα, οδηγώντας σε σημαντικές απώλειες δεδομένων. Επιπλέον, οι επιτιθέμενοι μπορούν να εκμεταλλευτούν τις ευπάθειες σε αυτές τις μεθόδους αυθεντικοποίησης, παρακάμπτοντας τα μέτρα ασφαλείας που πιστεύουν οι χρήστες ότι είναι ασφαλή.

Για να αντιμετωπίσουμε αυτό το ζήτημα, προτείνουμε την έννοια της *Συνεχούς Αυθεντικοποίησης*. Ο κύριος στόχος είναι η συνεχής αυθεντικοποίηση των χρηστών βάσει της συμπεριφοράς της συσκευής τους. Συλλέγοντας τα αρχεία καταγραφής που παράγονται από το λειτουργικό σύστημα Android και εφαρμόζοντας τεχνικές μηχανικής μάθησης για την ανάλυσή τους, μπορούμε να προσδιορίσουμε τα τυπικά πρότυπα συμπεριφοράς της συσκευής. Αυτή η προσέγγιση επιτρέπει την ανίχνευση ανώμαλων δραστηριοτήτων που αποκλίνουν από τα συνήθη λειτουργικά πρότυπα της συσκευής, επιτρέποντας πρώιμη παρέμβαση σε περιπτώσεις κλοπής ή κακής χρήσης. Η Συνεχής Αυθεντικοποίηση λοιπόν, όχι μόνο ενισχύει την ασφάλεια παρέχοντας συνεχή επαλήθευση, αλλά μειώνει επίσης την εξάρτηση από στατικά στοιχεία, τα οποία είναι επιρρεπή σε παραβιάσεις. Παρέχει υψηλότερο επίπεδο προστασίας κατά των πολύπλοκων επιθέσεων, καθώς παρακολουθεί συνεχώς τη συμπεριφορά της συσκευής αντί να βασίζεται αποκλειστικά σε στοιχεία που μπορούν να παραβιαστούν.

Λέξεις-κλειδιά:

Περιστατικά Απάτης, Αρχεία Καταγραφής Συστήματος Android, Μηχανική Εκμάθηση, Ανώμαλη Συμπεριφορά, Κλοπή Συσκευής, Συνεχής Έλεγχος Ταυτότητας, Μηχανισμοί Αυθεντικοποίησης

Table of contents

Acknowledgements **ix**

Abstract **xii**

Περίληψη **xiii**

Table of contents **xv**

List of figures **xix**

List of tables **xxi**

Abbreviations **xxiii**

1 Introduction **1**

1.1 Thesis Goals 1

1.1.1 Contributions 2

1.2 Outline 2

2 Authentication Mechanisms **5**

2.1 Challenges and Limitations of Current Authentication Mechanisms 5

2.2 Continuous Authentication: Its Meaning and Importance in Modern Security 6

3 Android System Logs **7**

3.1 Overview of Android System Logs 7

3.1.1 Types of Logs 7

3.1.2 Structure of Logs 8

3.2 Android Logging System 9

3.3 Logging Options for Android . . . 10
3.4 Log Access and Restrictions . . . 11
3.4.1 Log Access . . . 11
3.4.2 Rooting Android Devices and Restrictions . . . 13

4 System Architecture 15
4.1 Authenticator: A Continuous Authentication App For Android Devices . . . 16
4.1.1 UI . . . 16
4.1.2 Functionalities . . . 24
4.2 Server . . . 26
4.2.1 Endpoints . . . 26

5 Data Pre Processing 31
5.1 Parsing Android Logs . . . 31
5.1.1 Unstructured Android Logs into a Structured Format . . . 32
5.1.2 Structured Android Logs into Log Templates . . . 32
5.1.3 Log Templates to Event IDs . . . 33
5.2 Sequences Formation . . . 35
5.2.1 Sliding Window Technique . . . 36

6 Machine Learning Model 39
6.1 Machine Learning . . . 39
6.2 Transformers . . . 40
6.3 Overview of the BERT Model . . . 43
6.3.1 BERT Model Architecture . . . 44
6.4 Machine Learning Model Architecture . . . 46
6.4.1 Pre-Training . . . 46
6.4.2 Fine-Tuning . . . 49
6.4.3 Comprehensive Model Architecture . . . 51
6.5 Evaluation . . . 53
6.5.1 Evaluation on Normal Training Dataset . . . 53
6.5.2 Evaluation with 5% of Dataset Randomly Shuffled . . . 55
6.5.3 Evaluation with 10% of Dataset Randomly Shuffled . . . 56
6.5.4 Evaluation with 20% of Dataset Randomly Shuffled . . . 57

6.5.5 Average Model's Accuracy 58

7 Conclusions **59**

7.1 Conclusion and Results 59

7.2 Future Work 59

Bibliography **61**

List of figures

3.1 Executing Logcat Command 12
3.2 Executing Logcat Command with Log Filtering 12

4.1 System Architecture 15
4.2 Authenticator Application Logo 16
4.3 Superuser Access Request for the Authenticator 16
4.4 Instructions for Granting Superuser Rights 17
4.5 User Login Interface 18
4.6 User Register Interface 19
4.7 User Register Interface Example 20
4.8 User Login Interface Example 21
4.9 User Logged In Interface Example 22
4.10 Wi-Fi Disabled by User During Log Capture Process 23
4.11 User Login Session Expired Notification 24
4.12 Registration Pipeline Workflow 27
4.13 Authentication Pipeline Workflow 28
4.14 Log Transmission Pipeline Workflow 29
4.15 Logout Pipeline Workflow 30

5.1 Unstructured Android Log into Structured 32
5.2 Structured Android Log into Log Template 33
5.3 Log Template into Event ID 33
5.4 Total Distinct Hexadecimal Event IDs: 264 34
5.5 Overview of the Data Pre-Processing Pipeline for Android Logs 35

6.1 Transformers Architecture 41

6.2 BERT uses encoders, GPT uses decoders 43
6.3 BERT Base and Large Models Parameters 44
6.4 BERT Base and Large Models Architecture 44
6.5 Custom Encoder 48
6.6 Pre-training Phase Pipeline 48
6.7 Pre-training Loss 49
6.8 Fine-Tuning Phase Pipeline 50
6.9 Fine-Tuning Loss 51
6.10 Comprehensive Model Architecture - Pipeline 52
6.11 Evaluation on Normal Training Dataset 54
6.12 Evaluation with 5% of Dataset Randomly Shuffled 55
6.13 Evaluation with 5% of Dataset Randomly Shuffled 56
6.14 Evaluation with 10% of Dataset Randomly Shuffled 57
6.15 Evaluation with 20% of Dataset Randomly Shuffled 58

List of tables

3.1 Android Log Structure 9

4.1 Filtered Android Logs 25
4.2 ADB Command for Filtering and Capturing Logs 25

5.1 Android Log Fields 32
5.2 Sliding Window Example 38

6.1 Hardware Components of the PC Tower 53

Abbreviations

CA	Continuous Authentication
UI	User Interface
ML	Machine Learning
BERT	Bidirectional Encoder Representations from Transformers
GPT	Generative Pre-trained Transformer
PID	Process ID
TID	Thread ID
OS	Operating System
REGEX	Regular Expressions
ID	Identifier
JWT	JSON Web Token
JSON	JavaScript Object Notation
GELU	Gaussian Error Linear Unit
FFNN	Feed-Forward Neural Network
RNN	Recurrent Neural Network
CNN	Convolutional Neural Network
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Units
2FA	Two-Factor Authentication
ADB	Android Debug Bridge
APK	Android Package Kit

Chapter 1

Introduction

Authentication is a critical security process designed to verify the identity of a user, device, or system. It ensures that only authorized individuals or entities can access sensitive information or perform certain actions. The primary goal of authentication is to establish trust by confirming that the entity requesting access is indeed who or what it claims to be.

Although state-of-the-art authentication methods are generally effective, they are not without vulnerabilities. Because these methods rely on user-held elements, they are inherently susceptible to security threats and theft. When such credentials are compromised, unauthorized access to sensitive information can occur, leading to substantial financial losses and privacy breaches. Furthermore, if a device is stolen, the thief may obtain the authentication credentials, reducing the effectiveness of these methods, as the stolen device combined with compromised credentials can facilitate unauthorized access. This scenario highlights a significant limitation: once a device is stolen, traditional authentication methods may fall short, underscoring the urgent need for more comprehensive and adaptive security solutions capable of detecting abnormalities in real-time and addressing these challenges effectively.

1.1 Thesis Goals

To address the challenges outlined, this thesis introduces a solution based on **Continuous Authentication**. This mechanism involves the deployment of an Android application specifically designed for rooted devices. Upon user login, the application filters and captures Android System logs, focusing on relevant interfaces such as Networking, Battery, and Bluetooth. These logs are then transmitted remotely to a server. The server utilizes advanced

Machine Learning technologies to process and analyze these sequences of logs, enabling it to learn and understand the device's typical behavioral patterns. Once the model is adequately trained, it can continuously monitor and detect deviations from the normal behavior. This approach enhances security by identifying unusual activities and reduces the risks associated with stolen or compromised credentials. It provides a more dynamic solution to the limitations of traditional authentication methods, as it continuously monitors user device behavior rather than relying solely on simple credentials.

1.1.1 Contributions

This thesis provides the following contributions:

1. Development of an Android Application exclusive for rooted devices that efficiently collects, filters, and transmits Android System logs to a Server.
2. Development of a Server responsible for conducting in-depth analysis and preprocessing of Android System logs to extract valuable patterns from device behavior.
3. Demonstration of how Machine Learning technologies can be employed to enable Continuous Authentication, ensuring safety and protection for users.
4. Evaluation of the proposed System Architecture for continuous user authentication based on device behavioral patterns.

1.2 Outline

The thesis is structured as follows: In Section 2, the state-of-the-art Authentication Mechanisms are explained in depth, along with a detailed discussion of the significance of Continuous Authentication mechanisms in modern Security Systems. In Section 3, a comprehensive introduction to Android System logs is provided for better understanding, including the types of logs, their structure, and the importance of rooting an Android device to retrieve these logs and obtain the necessary permissions. In Section 4, the System Architecture is analyzed, covering both the Android Application and the Server. This section includes details about the user interface (UI), the main functionalities of the Application, and the key endpoints of the Server. In Section 5, the preprocessing of Android System logs is discussed, along with data

preparation to serve as inputs for the Machine Learning model. In Section 6, an introduction to Machine Learning technology is provided. The architecture of the Machine Learning model is presented, detailing the two main phases—pre-training and fine-tuning—and their respective pipelines. Finally, the evaluation results of the model are included. Last, in Section 7, the thesis concludes with a discussion of potential future work and improvements.

Chapter 2

Authentication Mechanisms

2.1 Challenges and Limitations of Current Authentication Mechanisms

In today's digital world, protecting user data is more essential than ever. The most common Authentication Mechanisms [1] today, as mentioned before, include PINs, biometrics and two-factor authentication. While these mechanisms generally work well and provide safety for user data, they face serious issues because these credentials are in the user's possession and can be accessed in certain scenarios. For instance, if the user is under threat from thieves or if the device is stolen, the user may be coerced into providing these credentials. Beyond physical threats, these mechanisms also suffer from other vulnerabilities:

- **PINs**: Users often choose simple or easily guessable PINs, making them susceptible to attacks. Additionally, PINs can be observed by others through a method known as shoulder surfing, where an attacker watches the user enter their PIN. Without additional security measures, PINs are also vulnerable to brute force attacks, where an attacker tries multiple combinations until the correct one is found.

- **Biometrics**: Biometric authentication methods, such as facial recognition, voice recognition, and fingerprint scanning, offer a convenient and often more secure alternative to PINs. However, they are not without their own set of issues. Biometric data can be spoofed using high-quality replicas, such as silicone fingerprints or detailed photos for facial recognition. Furthermore, once biometric data is compromised, it cannot be changed, unlike passwords. This permanent nature of biometric data poses a significant

risk.

- **Two-Factor Authentication (2FA)**: 2FA adds an extra layer of security by requiring a second form of verification, typically something the user possesses (like a device) in addition to something they know (like a password). Despite its enhanced security, 2FA is not foolproof. SIM swapping attacks, where an attacker hijacks the victim's phone number, can enable the attacker to receive 2FA codes. This vulnerability can undermine the additional security that 2FA is supposed to provide.

2.2 Continuous Authentication: Its Meaning and Importance in Modern Security

Continuous Authentication (CA) [2],[3] is a security mechanism that continuously verifies a user's identity throughout their session, rather than only at the initial login. This approach utilizes various behavioral and contextual data to authenticate the users in real-time. In this Thesis, the proposed method involves monitoring Android System logs to analyze and capture the device's behavioral patterns during normal usage, reflecting the specific user's interaction with the device. By continuously analyzing these logs, Continuous Authentication ensures that the individual using the device remains the authorized user. This method enhances security by significantly reducing the risk of unauthorized access, even if the initial authentication credentials are compromised. For instance, if these credentials are compromised or the device is stolen, recovery is often impossible. However, if unusual activity is detected in the device's behavior, the system can lock the device or prompt for the device's PIN, thereby preventing potential leakage of sensitive data.

Chapter 3

Android System Logs

3.1 Overview of Android System Logs

Android System Logs are crucial records created by the Android OS to track various activities and events occurring on a device. These logs are instrumental for monitoring system performance, troubleshooting issues, debugging applications, and ensuring the security and stability of the device. They provide a real-time snapshot of system behavior, helping developers and system administrators to diagnose problems and understand device usage patterns. Analyzing these logs can reveal insights into device's behavior, enabling the development of efficient applications and enhancing overall user experience.

3.1.1 Types of Logs

Android generates several types of logs [4], each serving a specific purpose:

- **Application Logs**: These logs capture information specific to the application's operation, including runtime messages, errors, and debugging details. They help developers track the application's behavior, identify issues, and understand user interactions within the app.
- **System Logs**: These logs record low-level system messages, including kernel events and system-wide errors. They are crucial for diagnosing and debugging issues related to the operating system, drivers, and system components.
- **Event Logs**: These logs document significant system events such as user actions, system alerts, and state changes. They provide insight into the sequence of events and

system behavior, which is useful for tracking system usage patterns and diagnosing issues.

- **Radio Logs**: These logs contain information related to radio communications, including cellular network activity and phone-related events. They are essential for troubleshooting issues related to mobile network connectivity and phone operations.

3.1.2 Structure of Logs

Android log entries [4] follow a specific structure that helps in systematically recording and retrieving log information. Each log entry typically contains the following components:

- **Timestamp**: The timestamp records the exact date and time of the log entry creation. It follows the format *MM-DD HH:MM:SS.sss*, where:
  - *MM*: Month (two-digit format)
  - *DD*: Day of the month (two-digit format)
  - *HH*: Hour of the day in 24-hour format (00-23)
  - *MM*: Minutes (00-59)
  - *SS*: Seconds (00-59)
  - *sss*: Milliseconds (000-999)
- **Process ID (PID)**: Identifies the process that generated the log entry. It is a unique identifier assigned by the Android OS to each running process.
- **Thread ID (TID)**: Identifies the specific thread within the process that generated the log entry, useful for debugging multi-threaded applications.
- **Log Level**: Represents the severity or priority of the log message. Common log levels, along with their importance, are as follows:
  - **V (Verbose)**: Detailed information typically of interest only when diagnosing problems. *Importance: Low*. Used for in-depth troubleshooting.
  - **D (Debug)**: Debugging information, helpful for tracking code execution and system behavior during development. *Importance: Low to Medium*. Mainly used by developers.

  - **I (Info)**: General informational messages about normal system operations. *Importance: Medium*. Useful for understanding overall system health.
  - **W (Warning)**: Indicates potential issues or non-critical problems that could lead to more serious errors. *Importance: High*. Requires attention but not immediate action.
  - **E (Error)**: Signifies critical errors that affect system functionality and need immediate resolution. *Importance: Very High*. Urgent problems requiring immediate attention.

- **Tag**: A string that indicates the source of the log message. It typically represents the class or module within the application or system that generated the log entry.
- **Log Message**: The log message contains the actual information or event being logged. It provides context and details about what is happening at the moment the log entry is created.

Table 3.1 shows a typical Android Log. The first row presents the log in its raw format, while the following row breaks it down into the respective fields discussed above. As observed, the Log Message field provides details about the device's currently connected network.

Table 3.1: Android Log Structure

07-08 15:27:40.090 818 928 D WifiNetworkSelector: Current connected network: "nitlab" , ID: 1					
Timestamp	**Process ID (PID)**	**Thread ID (TID)**	**Log Level**	**Tag**	**Log Message**
07-08 15:27:40.090	818	928	D	WifiNetworkSelector	Current connected network: "nitlab" , ID: 1

3.2 Android Logging System

The Android Logging System [4] is an essential component designed to capture, store, and manage the logs detailed in Section 3.1. By leveraging this system, users can effectively monitor device performance, diagnose issues, and track system behavior. Its primary components include:

- Kernel Driver and Kernel Buffers: The Kernel Driver responsible for logging in Android is known as the 'logger'. This driver manages the interaction between the operating system and hardware for logging purposes. It facilitates the temporary storage of

log messages in Kernel Buffers, which are specialized memory areas within the kernel. These buffers hold log messages temporarily before they are accessed or processed, ensuring efficient handling and retrieval of system logs.

- Logging Classes: Logging Classes in Android are implemented in both low-level and high-level languages. C and C++ Classes offer low-level logging capabilities and facilitate interaction with system components, allowing for detailed control over logging processes. In contrast, Java Classes provide high-level APIs designed for creating, managing, and retrieving log entries within Android applications. These high-level APIs simplify the logging process for developers by offering user-friendly methods and functions to handle log data effectively.

- Logcat: A command-line tool used for viewing and analyzing log messages. Displays log messages in real-time as they are generated. Also, allows developers to filter log messages based on various criteria such as tag, priority, and process ID.

- DDMS (Dalvik Debug Monitor Server): a tool used for interacting with Android devices from a development environment. It provides a graphical interface that allows developers to view and filter log messages collected by Logcat. Additionally, DDMS aids in debugging by offering access to device logs and various other debugging tools, making it an essential utility for developers working on Android applications.

3.3 Logging Options for Android

The Android framework has it own set of logging options. There is a special **Log** [5] class which provides the needed functionality for printing a log message. There are also several different types of log messages, which can be used in different situations:

- **Log**.e: Utilized for logging messages of type ERROR. This method is typically employed within catch statements to indicate that an error has occurred.

- **Log**.w: Used for logging messages of type WARN (warning). This method is appropriate when an unusual condition arises that does not necessarily indicate an error.

- **Log**.i: Employed for logging messages of type INFO. This method is ideal for logging useful information, such as establishing a connection to a server or successfully

completing a process.

- **Log**.d: Represents DEBUG messages. This method is suitable for logging debug-level information during development.

- **Log**.v: Stands for VERBOSE. This method is used for general logging and is appropriate for outputting detailed information about events that occur within the application.

3.4 Log Access and Restrictions

3.4.1 Log Access

To access these logs, several tools and steps can be utilized. Specifically, the following methods are recommended:

- **Android Debug Bridge (ADB) Command-Line Interface** [6]: Developers and administrators can access the logs using the ADB tool. By executing the command:

  ```
  adb logcat
  ```
  (3.1)

  ,the developers can view real-time logs from a connected device, as illustrated in Figure 3.1 . Additionally, the adb logcat command offers various options to filter and format the log output. For example:

  ```
  adb logcat -s <TAG>
  ```
  (3.2)

  ,filters logs by the specified tag as depicted in Figure 3.2.

- **Logcat** [7] in Android Studio: Android Studio provides an integrated Logcat viewer, also mentioned in Section 3.2, that displays log messages from connected devices. Developers can use this tool to monitor and debug application behavior. The Logcat window in Android Studio allows filtering by log level (e.g., VERBOSE, DEBUG, INFO, WARN, ERROR, ASSERT) and searching through logs using keywords.

- **Direct Access on the Device**: On some devices, users and administrators can access log data directly through the settings or developer options. This access is usually limited to less detailed logs compared to what can be obtained via ADB.

- **Third-Party Applications**: Various third-party applications are available on the Google Play Store that can capture, view, and analyze logs on Android devices. These apps often provide additional features such as log export and advanced filtering.

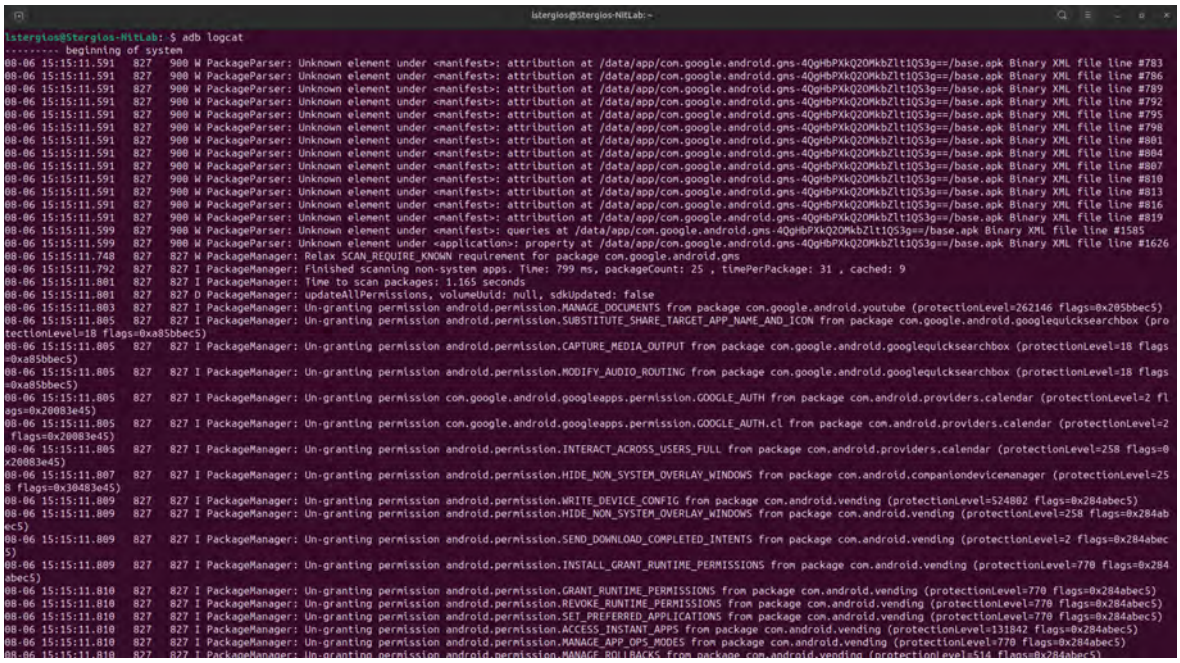

Figure 3.1: Executing Logcat Command

Figure 3.2: Executing Logcat Command with Log Filtering

3.4.2 Rooting Android Devices and Restrictions

However, there are several restrictions that must be addressed. The Android OS only allows the retrieval of system logs when the device is connected to a PC via USB. In other words, if someone attempts to access these logs through an application, the Android OS will only allow access to the logs generated by that specific application. To access system logs **remotely** from a device—an objective pursued in this thesis—the device must be rooted.

Rooting an Android device [8] is a challenging and complex process that involves bypassing the device's built-in security measures. This process requires technical expertise and careful execution, as it can void warranties, expose the device to security vulnerabilities, and potentially render the device inoperable if not performed correctly. During the rooting process for the Android 10.0 device (LENOVO TB-8505F), multiple instances of software damage were encountered before ultimately achieving success. The steps to root an Android device are as follows:

1. Connect the device with a USB cable to the PC. Navigate in the device's settings:

   - Find Developer Options → USB Debugging → ON
   - Find Developer Options → OEM unlocking → ON

2. Download the Magisk Manager application Android Package Kit (APK) file from the web onto the PC, and install it on the device by executing the command:

   ```
   adb install Magisk-v26.4.apk
   ```
   (3.3)

   This application allows users and developers to root Android devices.

3. At this stage, the stock ROM of the device must be located online and installed. The folder size is approximately several gigabytes and contains *.img* files corresponding to the partitions of the specific device, which are used for flashing during the fastboot process. Among these files, *boot.img* and *vbmeta.img* are of particular interest.

4. With the *boot.img* file, corresponding to the stock ROM, stored on the device, the file is launched via the Magisk Manager application. The option to *Patch a file* is selected, allowing the *boot.img* file to be chosen. This process generates another *.img* file named

magisk_patched.img, which can be transferred back to the PC by executing the command:

```
adb pull /filepath/to/magisk_patched.img
```
(3.4)

Subsequently, this file is moved into the folder containing the previously downloaded stock ROM for the device.

5. At this stage, the rooting process is nearly complete. On the PC, execute the following commands:

   - Wait for the device to enter fastboot mode to unlock the bootloader:

     ```
     adb reboot fastboot
     ```
     (3.5)

   - In order to unclock the bootloader now, execute:

     ```
     fastboot flashing unlock
     ```
     (3.6)

     The device will prompt for confirmation of this action.

   - After confirmation, execute the following command:

     ```
     fastboot flash vbmeta-disable-verification ./vbmeta.img
     ```
     (3.7)

     This command prevents the device from entering a boot loop, which is an infinite loop. This step is crucial.

   - Finally:

     ```
     fastboot flash boot ./magisk_patched.img
     ```
     (3.8)

     This command overwrites the boot partition by flashing the custom patched boot image generated in the Magisk Manager application. Once this process successfully completes, execute:

     ```
     fastboot reboot
     ```
     (3.9)

     and wait until the device reboots.

6. The device should now be **rooted**.

Chapter 4

System Architecture

The System Architecture consists of an Android Application and a Server. The main purpose is for the user to log in to the application by providing credentials (username, password). As long as the user remains logged in, the application transmits the system logs to the Server for further processing. This allows continuous monitoring of the device's behavior to detect any deviations or abnormalities that may occur. The main functionality pipeline is depicted in Figure 4.1.

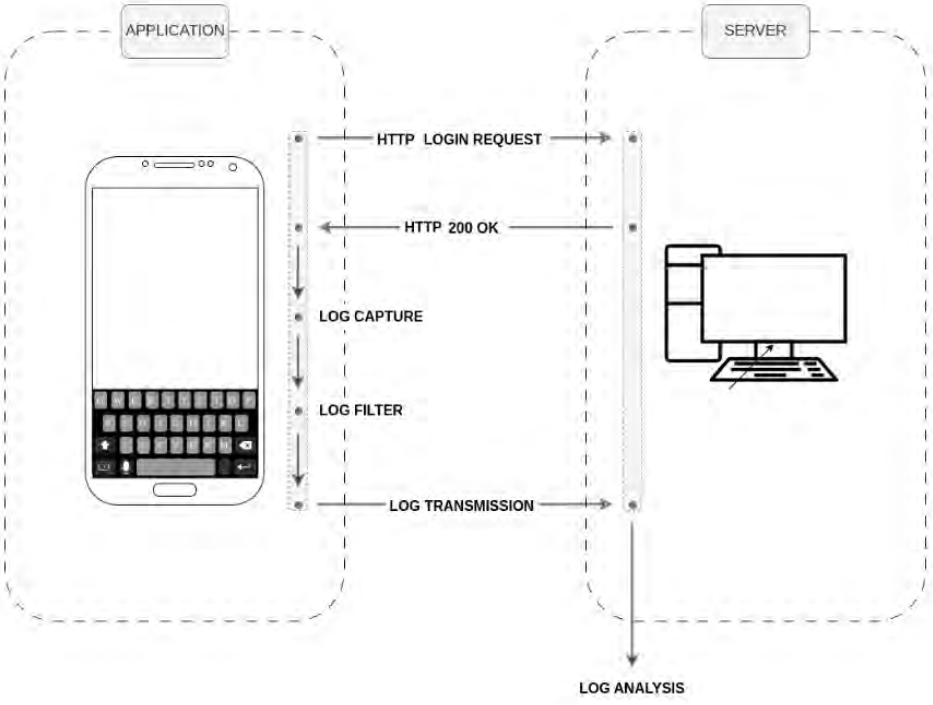

Figure 4.1: System Architecture

4.1 Authenticator: A Continuous Authentication App For Android Devices

The application was developed in Android Studio using Java, targeting devices running Android 10.0 and higher. Approximately twenty different classes were implemented, communicating with each other and resulting in multiple activities within the application.

4.1.1 UI

The Android Application Logo is illustrated in Figure 4.2.

Figure 4.2: Authenticator Application Logo

When the user opens the application for the first time, they are prompted to grant superuser (root) access. This permission is necessary to enable the application to retrieve Android system logs effectively. The corresponding dialog is displayed as follows:

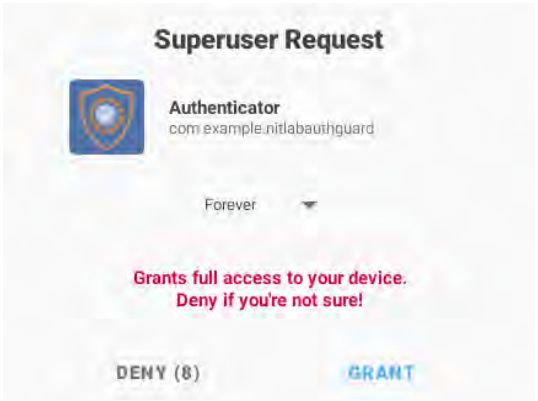

Figure 4.3: Superuser Access Request for the Authenticator

If the user denies this request by selecting *DENY*, a new message will appear, informing the user that the application requires superuser rights to function. The user will be guided

to install a third-party application that facilitates granting superuser rights to the Authenticator. If the user is unsure about which app to use, they are prompted to install the *Magisk* application, as shown in Figure 4.4:

Figure 4.4: Instructions for Granting Superuser Rights

In the scenario where the user selects *GRANT*, the setup process is completed, allowing them to interact with the Authenticator app without further interruptions. The superuser access prompt will not appear again unless the user manually revokes superuser rights through a third-party app like *Magisk* or modifies the device's settings. The application consists of three main user interfaces. First is the Login Page, where users enter their credentials to access their accounts. Second is the Register Page, allowing new users to create an account by entering a username and password. Finally, the Logged-In Page is displayed after successful authentication, initiating the Android log capture and transmission process to the server. The primary interface that users interact with is the Login Page, as shown in Figure 4.5.

Figure 4.5: User Login Interface

Users can enter their credentials in the designated fields and tap the sign-in button to log in. If a user is new and wishes to register, they can select the *Don't have an account? Sign Up* option. This action will navigate them to the registration page, as the Figure 4.6 illustrates.

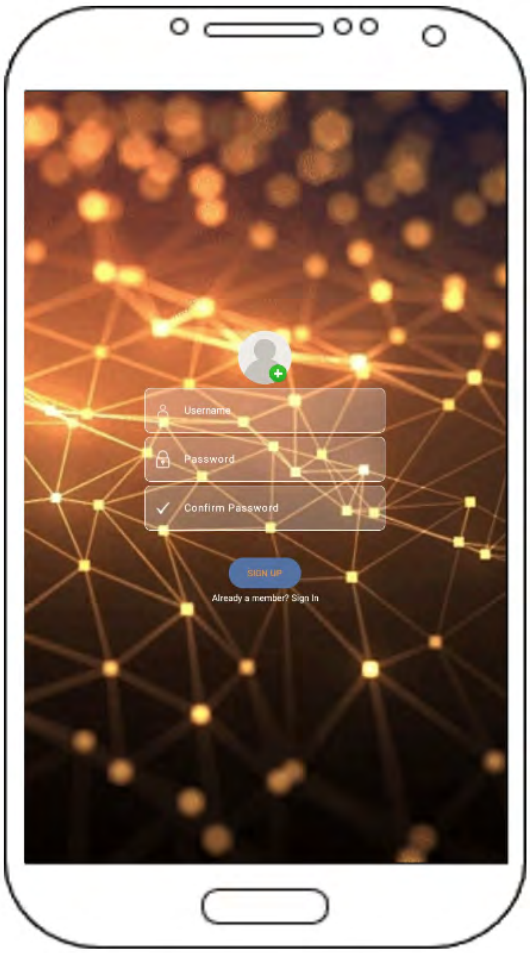

Figure 4.6: User Register Interface

Upon successful registration, the application redirects the user back to the Login Page. After logging in, the main interface, referred to as the Logged-In Page, is displayed. This page confirms the user's successful login and displays relevant details about the user's account. To better illustrate this process, consider a scenario where a user registers and logs into the application. First, the user must create an account by entering their chosen credentials:

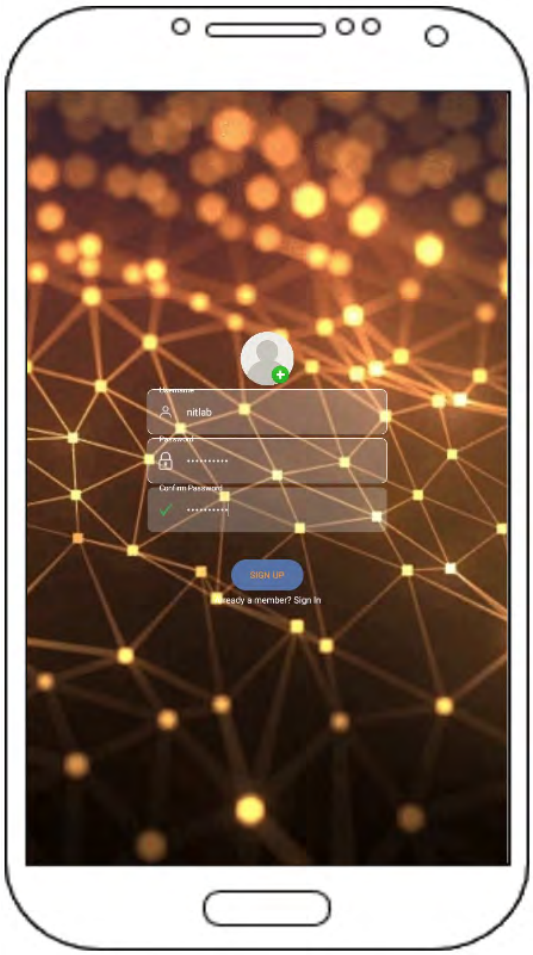

Figure 4.7: User Register Interface Example

The passwords must match for the checkmark to turn green; otherwise, it will turn red, and the user will be prevented from signing up. Additionally, the password must be at least 6 characters long, with the application notifying the user if the entered password is shorter. Upon successful registration, the application notifies the user of the successful registration and redirects them back to the login page to log in with the newly created credentials. When logging in, the user has the option to view or hide their password during the process, as the Figure 4.8 depicts.

Figure 4.8: User Login Interface Example

Upon successful login, the user will see their username displayed, along with a notification indicating that their Android system logs are being captured and transmitted to the server associated with their account, as illustrated in the Figure 4.9.

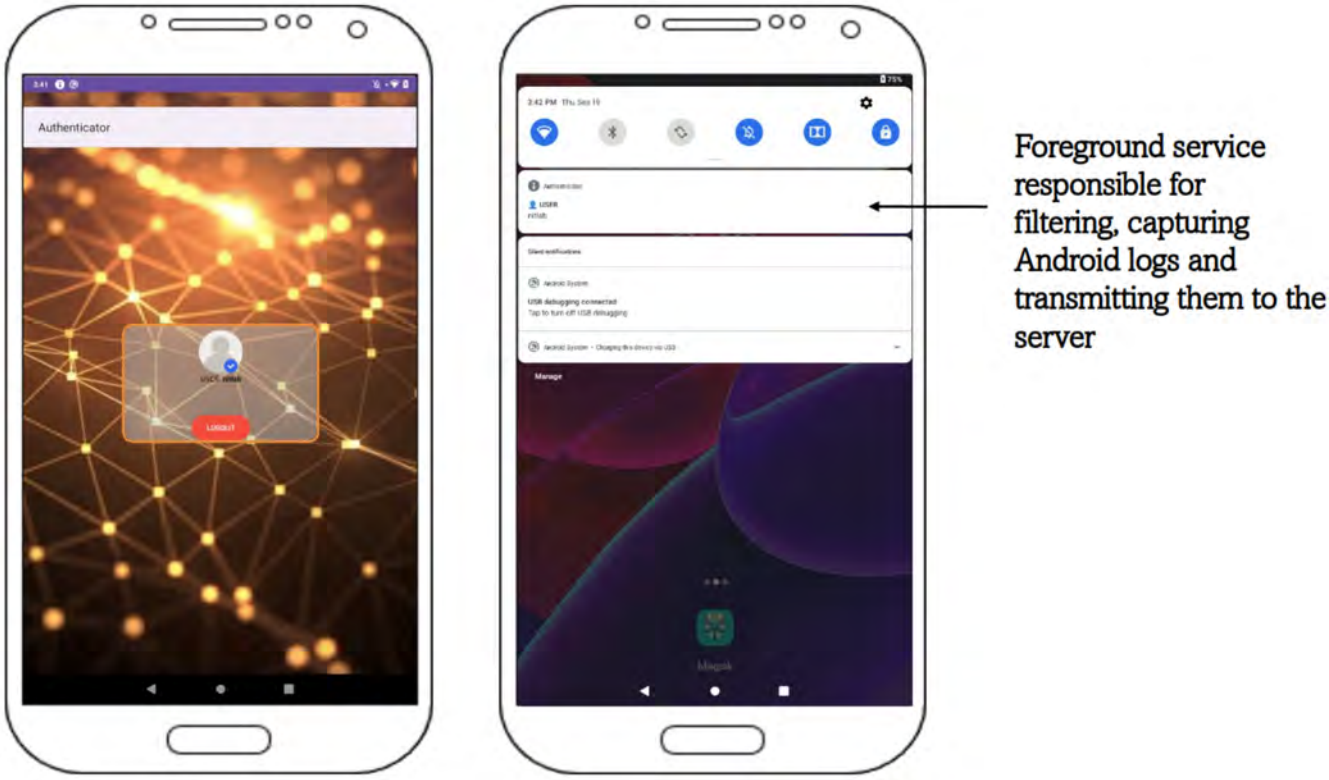

Figure 4.9: User Logged In Interface Example

From this point forward, the user can close the application and continue interacting with their device as usual. The foreground service will continue running in the background, ensuring that Android system logs are captured and transmitted to the server without interruption. At this point, it is crucial to prepare for potential scenarios. For example, what happens if the user disables Wi-Fi while the Android Log Capture is running? In this scenario, the Android Log Capture continues to record and save the logs in a file stored in the app's internal storage, which was created during user registration, associated with the user. The application then notifies the user that an internet connection is required for transmitting the logs and for proper functionality. This notification appears at the top of the screen, and if tapped, directs the user to a page providing details about the connectivity issue, as shown in the Figure 4.10.

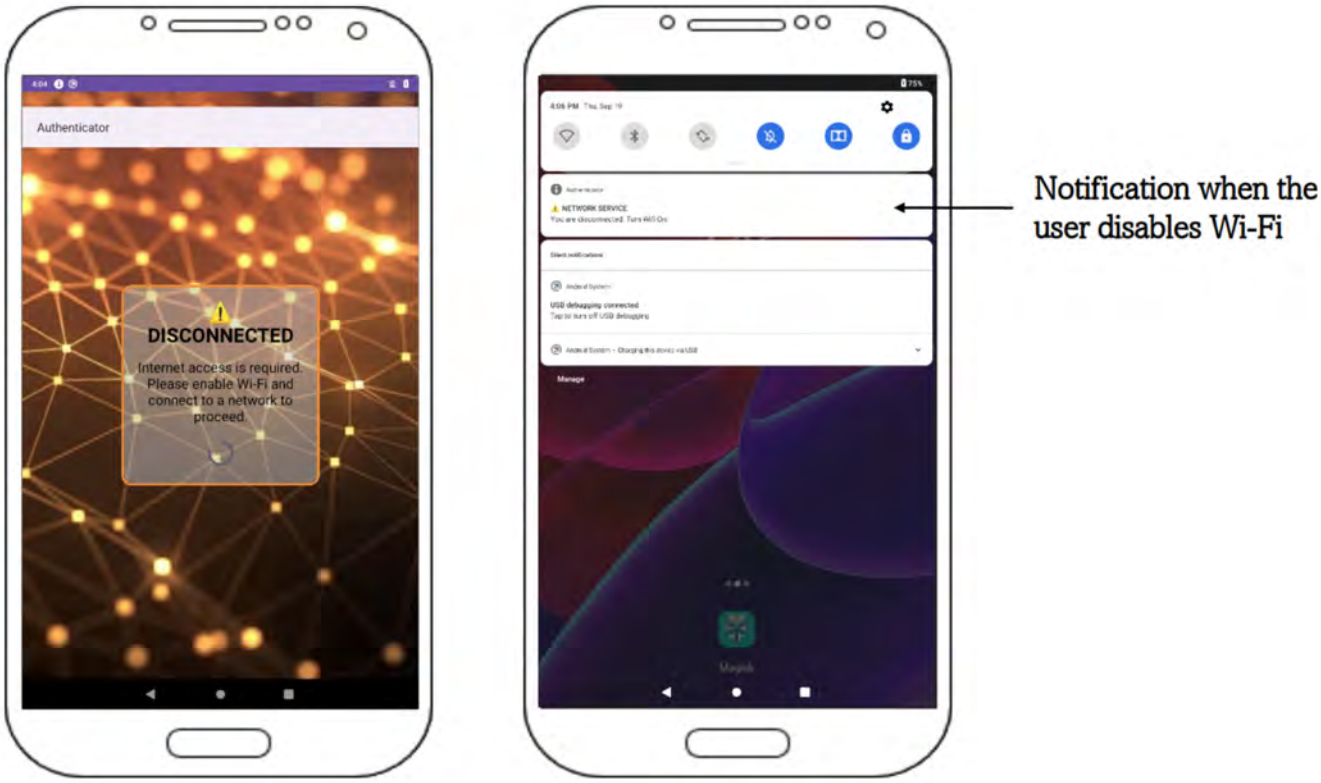

Figure 4.10: Wi-Fi Disabled by User During Log Capture Process

The page shown in the figure above, along with the same notification, will also appear if the user disables the Wi-Fi while on any other page of the application. This ensures the user is always informed about the need for internet connectivity to properly transmit the logs. Lastly, it is possible that during log capture and transmission, the user's login session may expire due to the JSON Web Token (JWT) token timeout. This token is generated by the server during the login session(as detailed in subsection 4.2.1), with an expiration date and time to keep users logged in and the foreground service active, thereby eliminating the need for re-authentication. If log capture is active and the JWT expires, the Authenticator notifies the user with an appropriate message, and the foreground service is terminated. Tapping the notification redirects the user to the login page for re-authentication, as depicted in Figure 4.11.

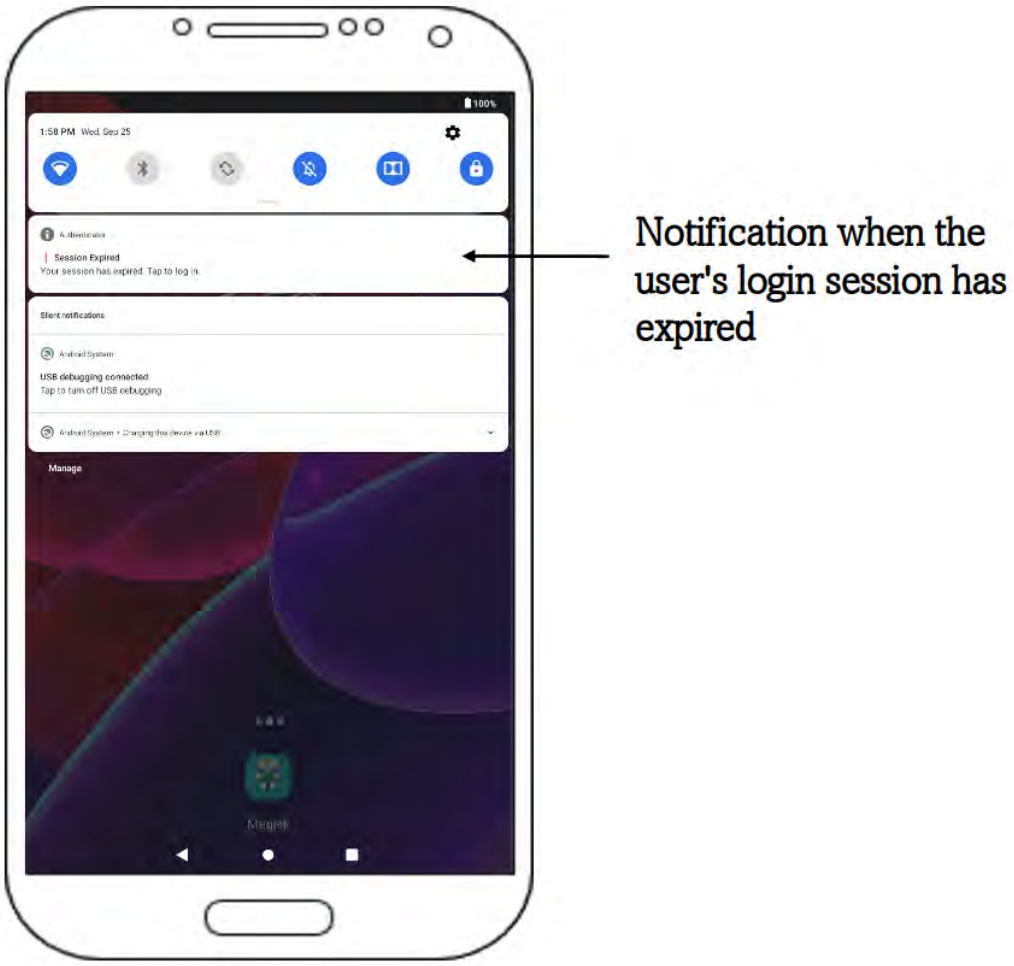

Figure 4.11: User Login Session Expired Notification

4.1.2 Functionalities

The primary function of the application is to authenticate users based on their device behavior by transmitting the logs from the device to the Server endpoint for analysis. Given the substantial number of logs generated daily by the Android OS—ranging from approximately 5,000 to 10,000 logs per day depending on the user's interaction with the device—our focus is solely on logs pertaining to three key interfaces: **Network**, **Bluetooth**, **Battery**. After conducting a thorough analysis to identify all relevant logs associated with these interfaces, it was found that the Android OS does not provide any logs related to location. The Foreground Service, as shown in Figure 4.9, is responsible for filtering logs from the General System logs based on these interfaces due to the enormous volume of logs and then transmitting them to the Server. These logs, along with their use cases are illustrated in Table 4.1

Table 4.1: Filtered Android Logs

ANDROID LOGS			
TAG	**INTERFACE**	**GENERATED BY**	**USE CASE**
WifiDisplayController	NETWORK	ANDROID OS	DEVICE BEHAVIOR
NetworkinfoNotification	NETWORK	ANDROID OS	DEVICE BEHAVIOR
WifiService	NETWORK	ANDROID OS	DEVICE BEHAVIOR
WifiConfigManager	NETWORK	ANDROID OS	DEVICE BEHAVIOR
netd	NETWORK	ANDROID OS	DEVICE BEHAVIOR
WifiScanningService	NETWORK	ANDROID OS	DEVICE BEHAVIOR
WifiConnectivityManager	NETWORK	ANDROID OS	DEVICE BEHAVIOR
ConnectivityPacketTracker.wlan0	NETWORK	ANDROID OS	DEVICE BEHAVIOR
WifiNetworkSelector	NETWORK	ANDROID OS	DEVICE BEHAVIOR
DhcpClient	NETWORK	ANDROID OS	DEVICE BEHAVIOR
MtkConnectivityService	NETWORK	ANDROID OS	DEVICE BEHAVIOR
WifiClientModeImpl	NETWORK	ANDROID OS	DEVICE BEHAVIOR
SettingsState	NETWORK - BLUETOOTH	ANDROID OS	DEVICE BEHAVIOR
AdapterState	BLUETOOTH	ANDROID OS	DEVICE BEHAVIOR
CachedBluetoothDevice	BLUETOOTH	ANDROID OS	DEVICE BEHAVIOR
BluetoothDatabase	BLUETOOTH	ANDROID OS	DEVICE BEHAVIOR
BluetoothAdapterService	BLUETOOTH	ANDROID OS	DEVICE BEHAVIOR
StatusBar	BATTERY	ANDROID OS	DEVICE BEHAVIOR
LPPeService	BATTERY	ANDROID OS	DEVICE BEHAVIOR

The service executes the *ADB command* outlined in Table 4.2. The *su* [9] option is used to run the command with superuser privileges. If this option is omitted, the command will only capture logs generated within the Authenticator application. Accessing Android OS logs requires superuser rights; without these privileges, the command can only capture logs generated within the application itself, excluding lower-level system logs.

Table 4.2: ADB Command for Filtering and Capturing Logs

COMMAND
`su -c logcat -s WifiDisplayController NetworkInfoNotification WifiService WifiConfigManager netd WifiScanningService WifiConnectivityManager ConnectivityPacketTracker.wlan0 SettingsState WifiNetworkSelector DhcpClient MtkConnectivityService WifiClientModeImpl AdapterState LPPeService CachedBluetoothDevice BluetoothDatabase BluetoothAdapterService StatusBar`

4.2 Server

The server is implemented using Node.js with the Express framework, designed to manage user authentication and data processing efficiently. Operating over HTTP due to the absence of a public IP, it can be easily configured for HTTPS if a public IP is provided. The server handles concurrent requests with asynchronous processing and mutexes, ensuring smooth and conflict-free operation. It integrates key libraries such as *jsonwebtoken* for secure token-based authentication and *bcrypt* for password hashing. User data is managed using file-based storage, while Python scripts are employed for feature extraction and data preprocessing. The server's core functionalities include user registration, login, and logout, as well as the storage and processing of Android system logs. This capability allows for the analysis of patterns and the detection of suspicious or abnormal behavior on users' devices, thereby enhancing overall security and performance.

4.2.1 Endpoints

The server provides a set of endpoints designed to facilitate various functionalities, including user authentication, data retrieval, and system management. Each endpoint is tailored to handle specific requests, ensuring efficient communication between the client and server while maintaining security and performance. The most crucial of them are the followings, categorized due to their specific operationals for the better understanding:

- **REGISTER**: The endpoints related to the register operation, are two:
  - **/users/register/request**: When the user initiates a POST HTTP request to the registration endpoint, they provide their username and password. The server responds by generating a random registration challenge to prevent replay attacks and ensure the response is from the intended party. This challenge ensures the request or response is current and not a replay of an old one. The server sends the challenge, username, and additional relying party information back to the client, and temporarily stores the registration request with the hashed password, utilizing the *bcrypt* library. The client must then provide a signed challenge using their private key, the corresponding public key for signature verification, and the relying party information to complete the registration.

– **/users/register/request/upload/credential**: Once the client (Authenticator App) generates a public-private key pair, the private key signs the registration challenge and remains on the client device. The client then initiates a POST HTTP request to upload the username, public key, signed registration challenge, a URI to the log file stored solely on the client device, and the relying party information received earlier. The URI use case is to temporarily store logs when the device is without internet access or connection. The server verifies the relying party information and uses the public key to validate the signature. If the signature is valid, registration succeeds; otherwise, it fails. Upon successful registration, the server stores the user's credentials in the database, creates a folder for the Android logs, and deletes the temporary registration request from the first phase.

The Registration Pipeline is depicted in the Figure 4.12.

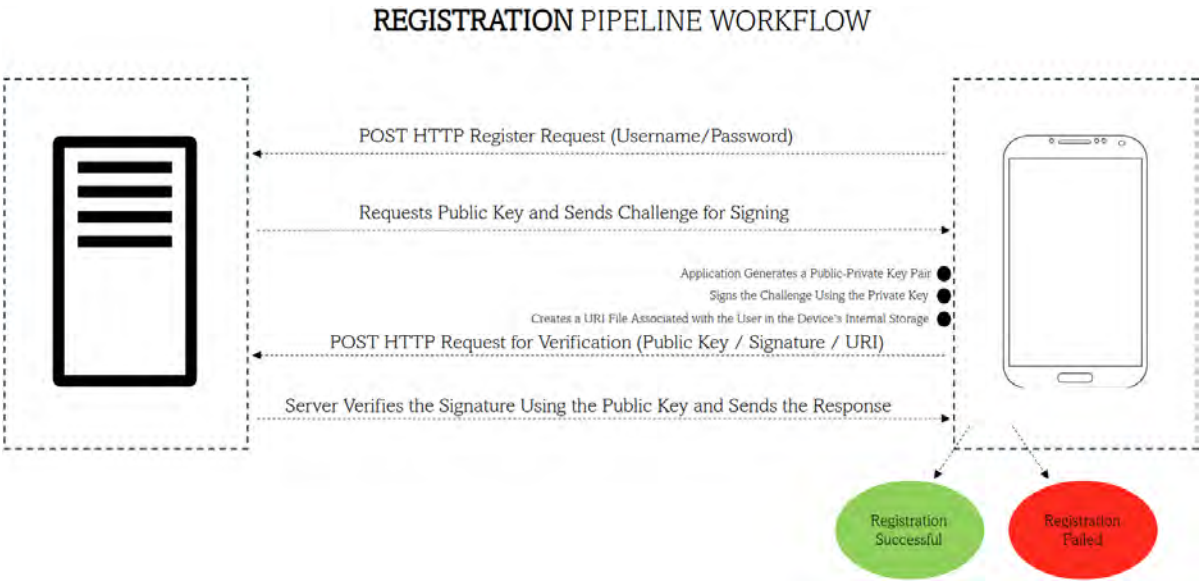

Figure 4.12: Registration Pipeline Workflow

- **LOGIN**: The endpoints related to the login operation, are also two:

  – **/users/login/request**: When a user attempts to log in, they make a POST HTTP request to this endpoint, providing their username and password. The server searches the database for the username; if it does not exist, the request is rejected. If the username is found, the server compares the stored hashed password with the provided password. If they match, the server generates a random login challenge,

similar to the registration process, and sends it back along with the relying party information and the username for verification.

– **/users/login/request/upload/signature**: Upon receiving the login challenge from the server, the client retrieves the private key stored securely on the device and uses it to sign the challenge. This signed challenge is then sent back to the server for verification. The server first verifies the relying party information, then retrieves the corresponding public key associated with the user from the database. Using the public key, the server decodes the signed challenge. When the decoded challenge matches the original login challenge, it confirms that the user has been successfully authenticated. At this point, a JWT is generated to maintain the user's session. This token is then sent back to the client application, which stores it internally. The token is associated with the authenticated user and can be used in future requests to the server, ensuring the user remains logged in without needing to re-enter credentials.

The Authentication Pipeline is depicted in the Figure 4.13.

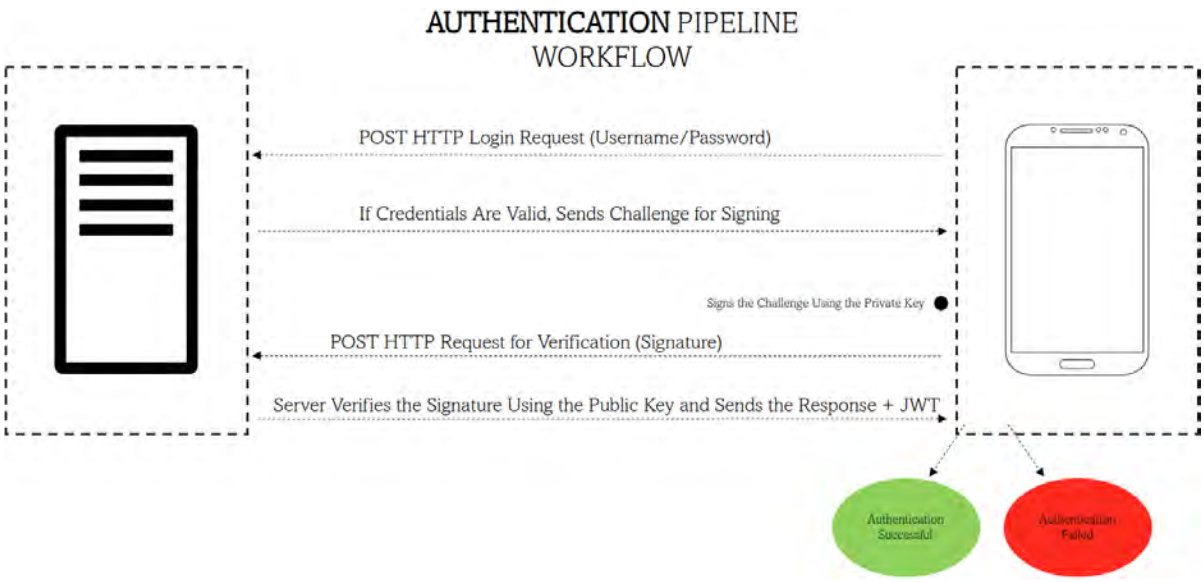

Figure 4.13: Authentication Pipeline Workflow

• **ANDROID LOG TRANSMISSION**: The endpoint related to the log capture and transmission operation, is:

– **/users/android/log/capture**: When an HTTP POST request is made to this endpoint with a Username, a log entry and a JWT, the request is queued for sequential

processing to handle one request at a time. Each request is processed individually, verifying the presence of log data and managing it accordingly. If the JWT is still active, the log data is stored in a user-specific directory, which is created if it does not already exist. The logs are appended to a file, and the processing task related to anomaly detection is executed. After processing, a status code is sent back to the client. If any errors occur, an appropriate error message is provided. If the JWT is expired, the server responds with an appropriate message. The notification created in this scenario is depicted in Figure 4.11.

The Log Transmission Pipeline is depicted in the Figure 4.14.

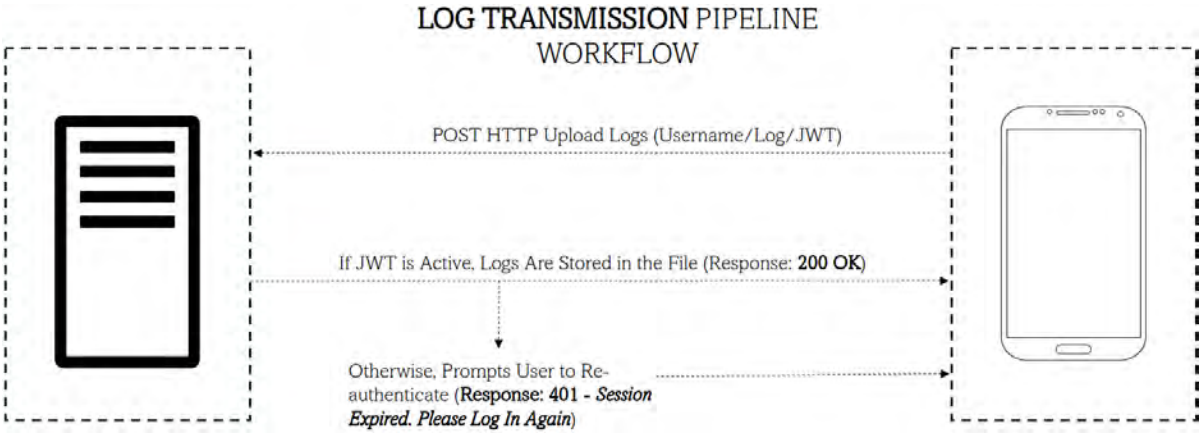

Figure 4.14: Log Transmission Pipeline Workflow

- **LOGOUT**: The endpoint related to the logout session, is:

  - **/users/logout**: Handles the user logout process. When a POST request is made with a Username and JWT at this endpoint, the server verifies the JWT. Once verified, the token is removed from the in-memory storage, effectively revoking the user's session. The server then responds to the client, confirming the user has been logged out successfully.

The Logout Pipeline is depicted in the Figure 4.15.

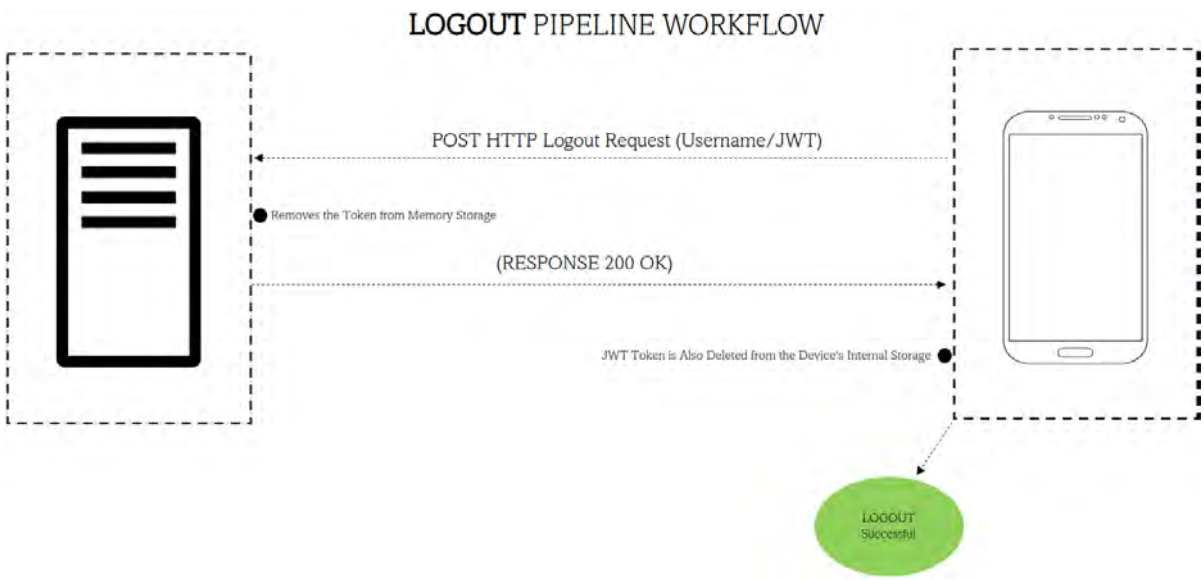

Figure 4.15: Logout Pipeline Workflow

Chapter 5

Data Pre Processing

Data pre-processing is a fundamental step in machine learning that significantly influences the success of model training. It involves transforming raw data into a clean and structured format, making it easier for the machine learning model to interpret and extract meaningful patterns. This process includes tasks like removing noise, handling missing or inconsistent data, normalizing or scaling features, and encoding categorical variables. Effective pre-processing ensures that the model can focus on learning relevant relationships within the data, rather than being distracted by irrelevant or misleading information. Without proper pre-processing, even sophisticated models can struggle to achieve accurate and reliable predictions, as unprocessed or poorly structured data can lead to overfitting, biased results, or poor generalization to new data. Hence, data pre-processing plays a critical role in ensuring that the machine learning algorithm can learn efficiently and perform optimally.

5.1 Parsing Android Logs

To detect abnormalities in Android System Logs, it is essential to process the logs, as they are initially in a raw text format. The first step in this process involves using a parser to transform the raw logs into structured Log Templates, which serve as a simplified representation of the system events. Once these Log Templates are generated, they are mapped to unique event IDs. The parser employed in this thesis to transform Android logs into log templates, and subsequently convert those templates into event IDs, is based on the implementation found in [10]. This parser has been modified to suit the specific structure of Android logs by applying a set of regular expressions to remove the dynamic variable portions from the

log messages. The final step is to organize these events into meaningful sequences, which is achieved through the Sliding Window Mechanism. This method allows the system to capture temporal patterns by analyzing overlapping segments of the log data, ultimately enabling more effective detection of anomalies in the system behavior.

5.1.1 Unstructured Android Logs into a Structured Format

The first step in log pre-processing involves parsing the logs to transform them from an unstructured format into a structured format. To achieve this, each log must be divided into its corresponding fields, as shown in Table 5.1.

Table 5.1: Android Log Fields

Log Format						
Date	Time	PID	TID	Log Level	Tag	Log Message

By specifying the log format fields and providing the corresponding regular expression as a parameter to the parser, the log message can be divided into separate fields. This process results in a structured Android log message, as illustrated in Figure 5.1.

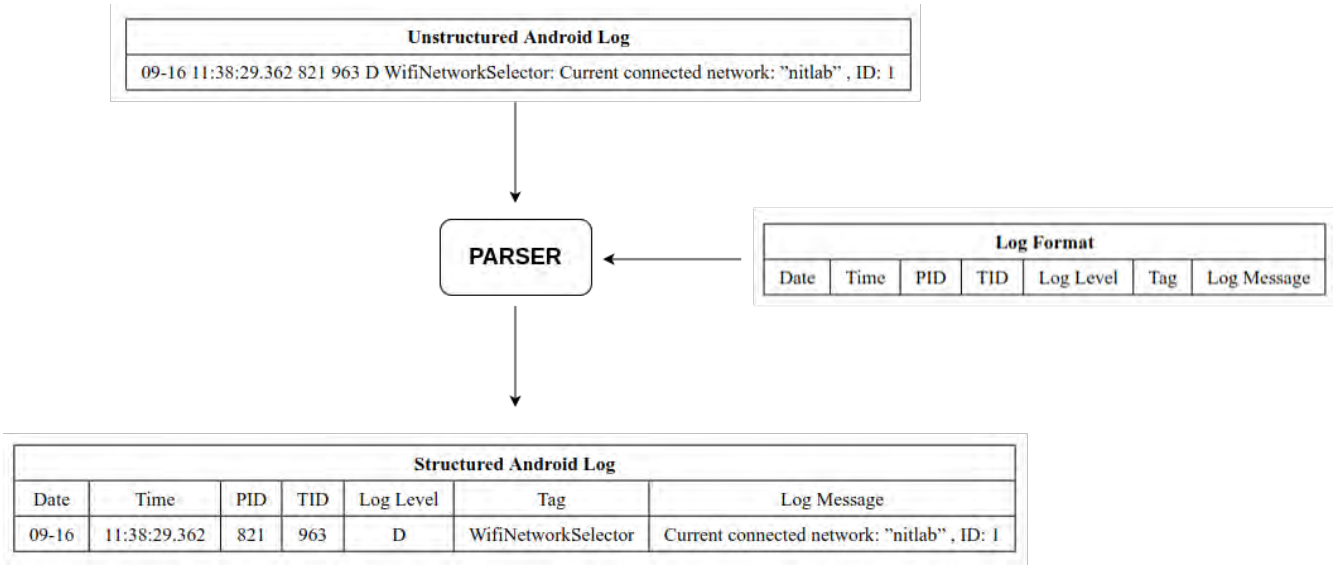

Figure 5.1: Unstructured Android Log into Structured

5.1.2 Structured Android Logs into Log Templates

The next step involves extracting the dynamic parts from the structured logs. Specifically, the log message field of each structured Android log must be processed to identify and remove

these dynamic variables - elements. Following extensive research and analysis of the logs to be examined, a set of regular expressions is developed to effectively isolate the dynamic portions of each log message. These dynamic segments are then replaced with placeholders, denoted as "<*>", as illustrated in the Figure 5.2.

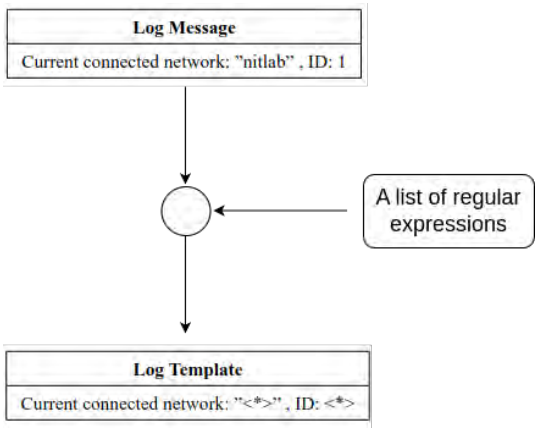

Figure 5.2: Structured Android Log into Log Template

5.1.3 Log Templates to Event IDs

The process of transforming Log Templates into Event IDs involves generating a unique ID for each template using a hash function. Specifically, the *hashlib* function is applied to the encoded string representation of the Log Template, which produces a fixed-length hash value, converted into a hexadecimal string, as the Figure 5.3 depicts.

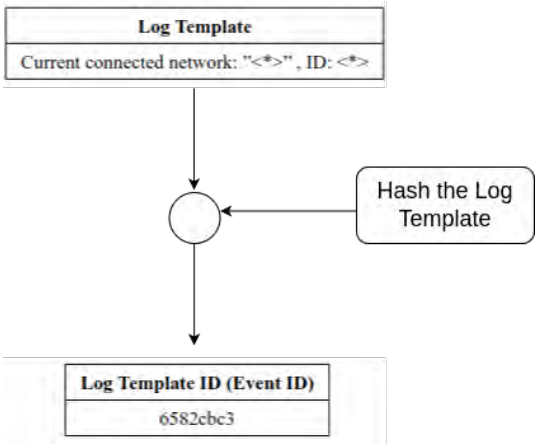

Figure 5.3: Log Template into Event ID

In our scenario, we have a total of 264 distinct hexadecimal event IDs, as illustrated in the Figure 5.4.

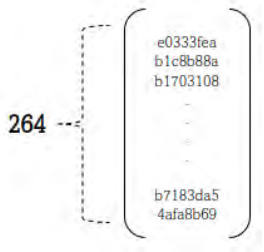

Figure 5.4: Total Distinct Hexadecimal Event IDs: 264

For clarity, the entire process of transforming unstructured Android Logs into Event IDs is illustrated in Figure 5.5.

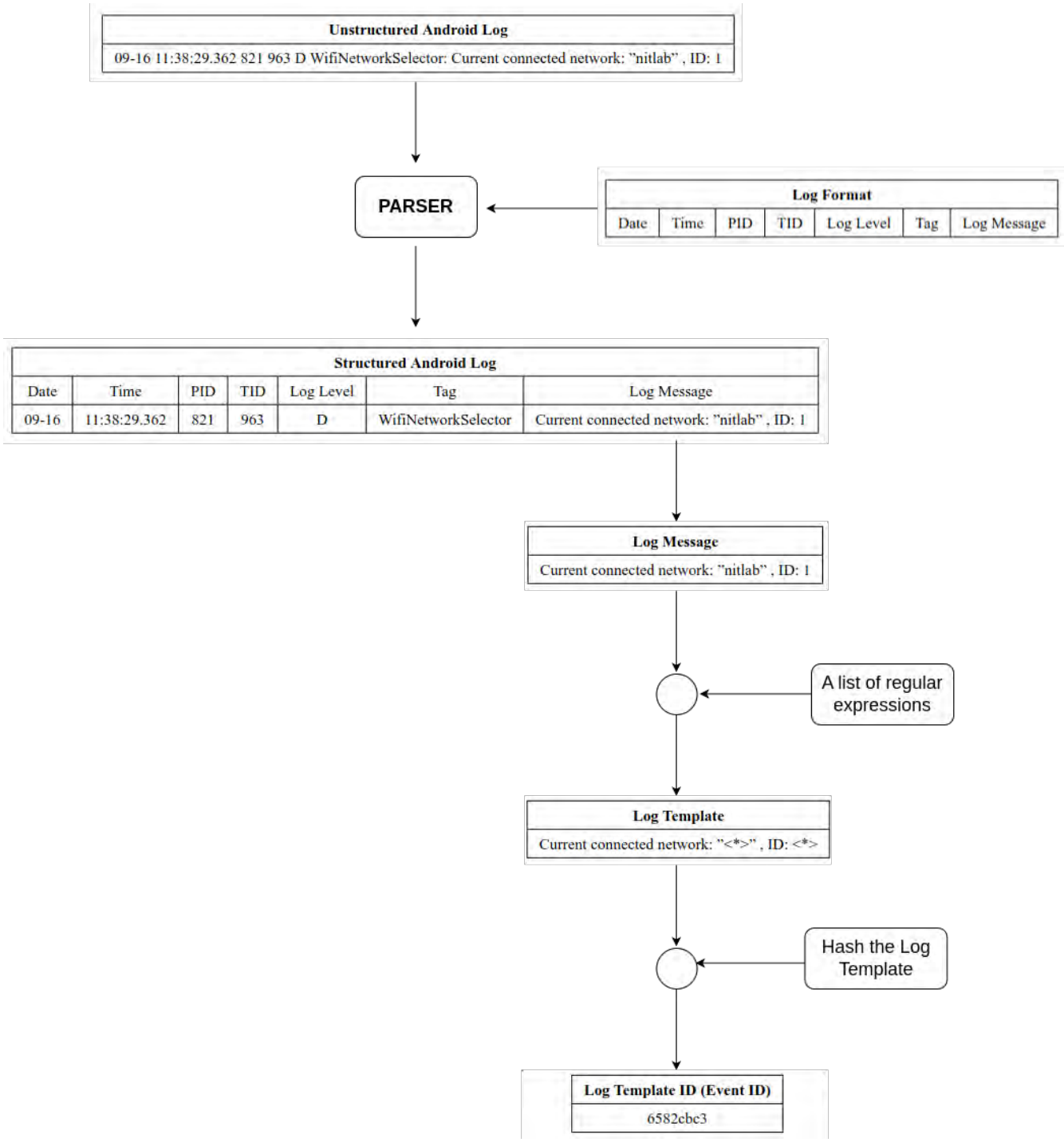

Figure 5.5: Overview of the Data Pre-Processing Pipeline for Android Logs

5.2 Sequences Formation

The final step of data pre-processing involves the formation of sequences, a crucial aspect for the effective operation of ML models. Since the thesis model is designed for *Next Event Prediction*, it is necessary to create sequences that enable the prediction of the up-

coming event based on these sequences. In ML applications, especially in anomaly detection and predictive modeling, the ability to analyze sequential data allows for the identification of patterns and trends over time. By structuring the data into sequences of Event IDs, where each ID corresponds to a specific Android log, the model can learn contextual relationships and detect deviations that might indicate abnormalities. This sequence-based approach not only enhances the model's understanding of temporal dependencies but also enables it to generalize better across varying scenarios. Given the dynamic nature of system logs, creating well-defined sequences facilitates the extraction of meaningful insights and enhances the overall performance of the machine learning framework, which is analyzed in depth in Chapter 6.

5.2.1 Sliding Window Technique

The mechanism used to form these sequences of events is the Sliding Window technique. The Sliding Window Technique is a powerful method used in data processing and analysis, particularly in time-series data or sequential data contexts. This technique involves maintaining a fixed-size window that moves over a dataset to extract subsets of data for analysis. In the context of machine learning, particularly for anomaly detection or pattern recognition, the sliding window allows models to examine overlapping segments of data sequentially. This approach enables the capture of temporal dependencies and relationships within the data, which are crucial for understanding trends and detecting anomalies. By analyzing these segments, the model can identify patterns, transitions, and deviations over time, leading to more accurate predictions and insights. The sliding window technique is particularly advantageous in environments where data is continuously generated, allowing for real-time analysis and can be categorized into several types based on its implementation and application. Here are the main categories:

- **Fixed-Size Sliding Window** [11]: This is the most common type, where the window moves over the dataset by a defined step size (e.g., one data point at a time) while maintaining a constant size. Each window contains the same number of data points, enabling consistent analysis.

- **Variable-Size Sliding Window** [11],[12]: In this approach, the window size can change based on certain conditions or data characteristics. This flexibility allows the analysis

to adapt to varying data distributions or patterns, capturing more significant features when needed.

- **Non-Overlapping Sliding Window** [13]: This category involves moving the window without any overlap between consecutive windows. Each window is analyzed independently, which may simplify computations but can miss nuances captured in overlapping segments.

- **Overlapping Sliding Window** [13]: Here, the window overlaps with previous windows, which allows for capturing transitions and gradual changes in the data. This is particularly useful in time-series analysis where continuity is essential.

- **Dynamic Sliding Window** [14]: This approach adjusts the window size and position based on the analysis results or the detection of specific events. It is useful in scenarios where the data exhibits sudden changes or irregular patterns.

The approach we follow is the **Fixed-Size Sliding Window**. By setting a constant *window size*, the event IDs are grouped into sequences of a fixed length. In our case, the window size is set to 20, meaning that each sequence consists of 20 event IDs, which correspond to 20 Android Logs. The algorithm for the Fixed-Length Sliding Window is detailed in Algorithm 1.

Algorithm 1 Sliding Window Algorithm

Input: $Dataset- > D, WindowSize- > W$
Output: List of sequences
Initialize empty list `sequences`
for $i \leftarrow 0$ **to** length$(D) - W$ **do**
    Initialize empty sequence W_i
    for $j \leftarrow 0$ **to** $W - 1$ **do**
        Add $D[i + j]$ to W_i
    end for
    Add W_i to `sequences`
end for

To provide better clarity, a simple example illustrating the application of the Sliding Window Algorithm to a dataset is shown in Table 5.2.

Table 5.2: Sliding Window Example

Dataset = [25,3,1,6,10,11]	
Window Size = 3	
Set of All Possible Events = [1,…,29]	
ID	**Sliding Windows**
1	[**25**,**3**,**1**,6,10,11] = [**25 3 1**]
2	[25,**3**,**1**,**6**,10,11] = [**3 1 6**]
3	[25,3,**1**,**6**,**10**,11] = [**1 6 10**]
4	[25,3,1,**6**,**10**,**11**] = [**6 10 11**]

Notice how the sliding window mechanism efficiently captures all elements while overlapping by one element at each step. This ensures that each log is thoroughly processed and no data is overlooked. By using this overlapping approach, we maximize the coverage of logs and create sequences that maintain continuity, which is critical for detecting patterns and abnormalities in sequential data. This method ensures that every event is incorporated into multiple windows, enhancing the model's ability to learn subtle relationships within the data.

Chapter 6

Machine Learning Model

6.1 Machine Learning

Machine Learning (ML) [15],[16] is a transformative field of artificial intelligence that focuses on enabling computers to learn from data and make decisions without explicit programming. At its core, ML involves developing algorithms that can identify patterns, make predictions, and improve their performance over time as they are exposed to more data. This capability is increasingly crucial in today's world, where vast amounts of data are generated daily across various domains, from healthcare and finance to entertainment and transportation. The ability to analyze and derive insights from this data drives innovations, optimizes processes, and enhances decision-making.

The importance of ML is underscored by its applications in diverse areas, such as personalized recommendations, predictive analytics, and autonomous systems. Key to ML models are their foundational layers, which include various types of neural networks and algorithms. Among the most common layers [17], [18] are:

- Dense (Fully Connected) Layers: These layers connect every neuron in one layer to every neuron in the next layer, enabling complex pattern recognition and feature extraction.
- Convolutional Layers: Often used in image processing, these layers apply convolutional operations to extract spatial hierarchies and features from data, making them essential for tasks like object detection and image classification.
- Recurrent Layers (RNN, LSTM, GRU): These layers are designed for sequence data,

allowing the model to maintain and utilize information over time. LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Units) are specialized recurrent layers that handle long-term dependencies effectively, crucial for tasks like language modeling and time-series prediction.

- Regularization Layers (Dropout, L2 Regularization): Regularization layers help prevent overfitting by adding noise or constraints during training. Dropout randomly deactivates neurons, while L2 regularization penalizes large weights, encouraging the model to generalize better and avoid learning spurious patterns.

- Attention Layers: Attention layers enable the model to focus on specific parts of the input sequence, dynamically weighting the importance of each element. This mechanism is highly effective in tasks such as translation, text generation, and sequence-to-sequence models, as it allows the model to capture long-range dependencies and prioritize key information.

- Pooling Layers: These layers reduce the dimensionality of feature maps while retaining important information, aiding in computational efficiency and model robustness.

- Normalization Layers: These layers stabilize and accelerate the training process by normalizing the inputs of each layer.

By leveraging these layers, machine learning models can tackle a wide range of problems and deliver valuable insights and solutions in various fields.

6.2 Transformers

Transformers [19],[20],[21] are a foundational architecture in modern deep learning, developed by researchers at Google. They are designed to handle sequential data by utilizing a mechanism called self-attention, which allows the model to weigh the importance of each word (or token) in a sequence relative to every other word, regardless of its position. Unlike traditional recurrent networks, which process data sequentially, transformers can process entire sequences in parallel, leading to faster and more efficient computations. This architecture is widely used in natural language processing (NLP) tasks, powering models like

BERT(Bidirectional Encoder Representations from Transformers) and GPT(Generative Pre-trained Transformer). Transformers consist of two main components: the **encoder** and the **decoder**.

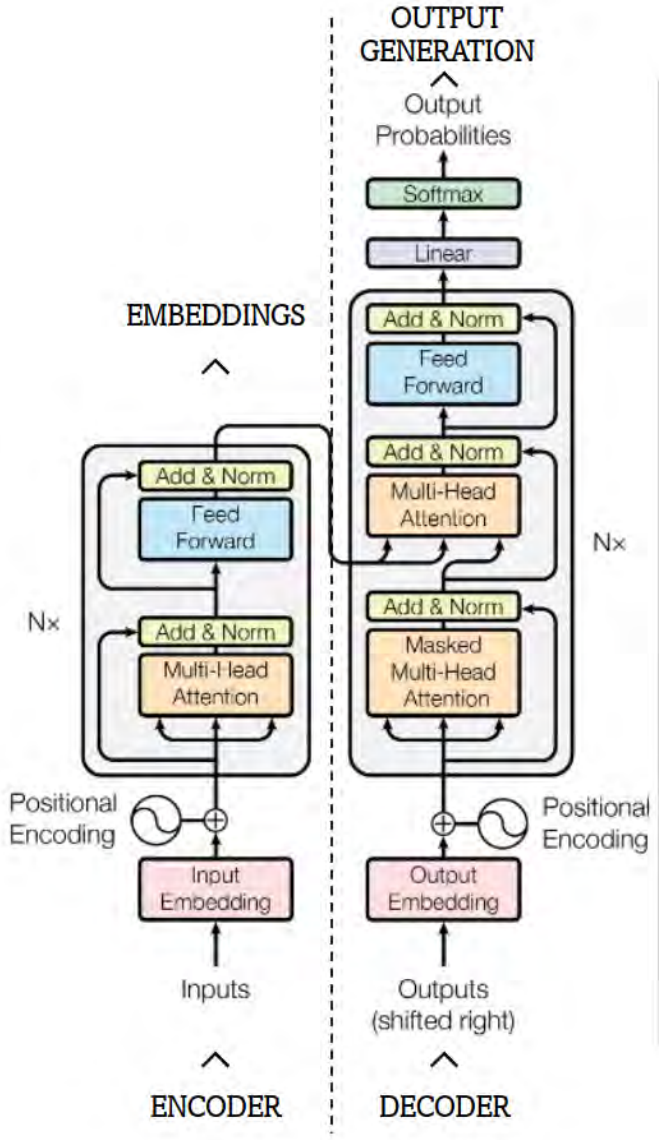

Figure 6.1: Transformers Architecture

As depicted in the Figure 6.1, it is observed that:

- **Encoder**: The encoder processes the input sequence and is responsible for extracting relevant features and understanding the context. It is composed of multiple layers, each containing two main sub-layers:
  - Self-Attention Mechanism: This layer allows each position in the input sequence to attend to all other positions, capturing dependencies and relationships between

tokens.

  - Feed-Forward Neural Network (FFNN): After the self-attention layer, the encoder has a feed-forward network applied to each position independently.

- **Decoder**: The decoder is responsible for generating the output sequence, often used for tasks like translation or text generation. It also consists of multiple layers, and each layer has:

  - Self-Attention Mechanism: Similar to the encoder, but with an additional masking mechanism to prevent future positions from being attended to during training.

  - Encoder-Decoder Attention Mechanism: This layer allows the decoder to focus on relevant parts of the encoded input sequence (output from the encoder).

  - Feed-Forward Neural Network (FFNN): Like in the encoder, this applies a feed-forward network to the attended outputs.

Real-world applications of Transformer models are exemplified by GPT and BERT. GPT [22],[23], developed by OpenAI, is a state-of-the-art language model designed for natural language generation, producing human-like text based on a given prompt. This highlights the strength of Transformers in text generation tasks, as GPT relies solely on the decoder component of the architecture to generate understandable sequences.

On the other hand, BERT [24],[25],[26] developed by Google, utilizes bidirectional attention mechanisms within the encoder component to understand the full context of words in a sentence by looking at both preceding and following words. This bidirectional approach significantly enhances performance on various natural language understanding tasks, such as question answering, sentiment analysis, and text classification.

In practical terms, stacking 12 encoder layers forms the BERT Base model (or 24 layers for BERT Large), while stacking only decoders results in models like GPT, which excel in tasks requiring sequence generation, as Figure 6.2 shows.

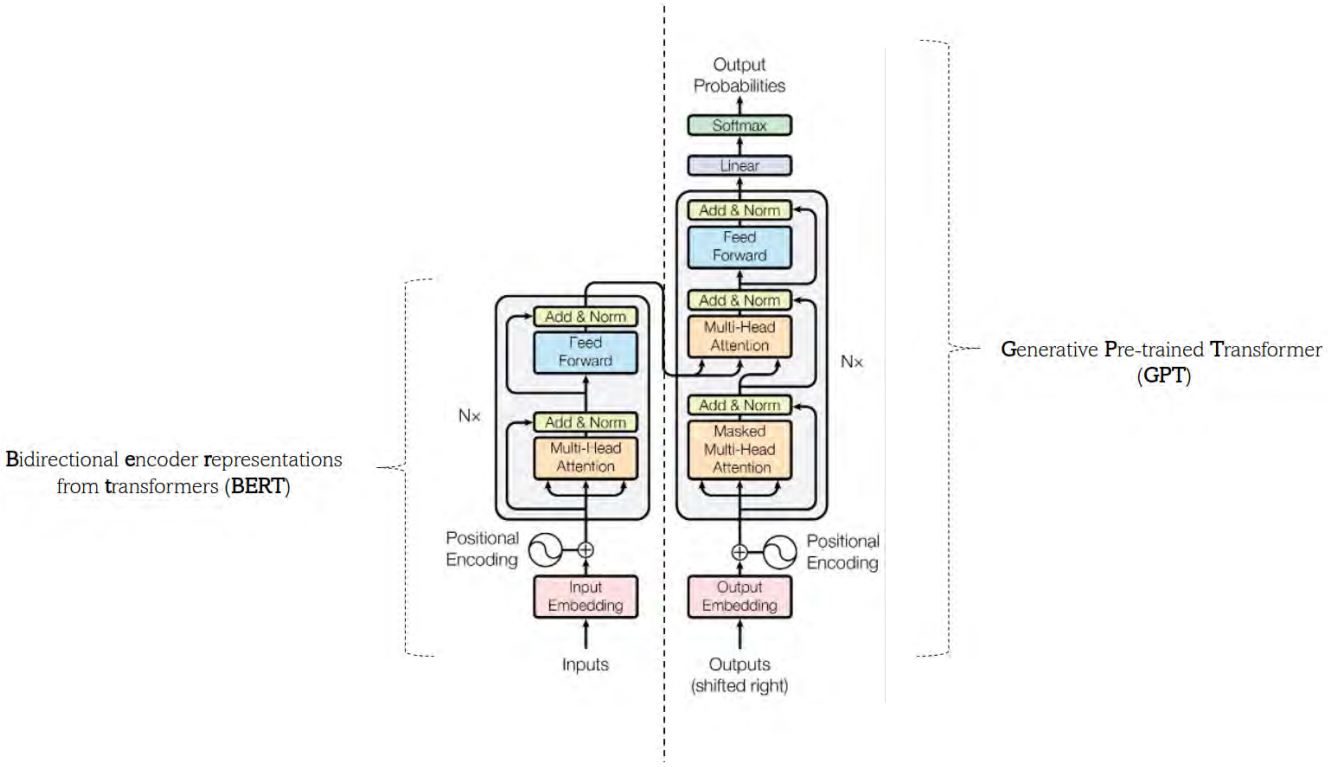

Figure 6.2: BERT uses encoders, GPT uses decoders

6.3 Overview of the BERT Model

The BERT model revolutionized natural language processing by pre-training a deep bidirectional representation, which allows it to understand the context of a word based on both its preceding and following words. The *BERT base model* consists of 12 layers (Transformer encoder blocks), with 12 self-attention heads and a hidden size of 768 dimensions. It has approximately 110 million parameters. In comparison, the *BERT large model* is much deeper, with 24 layers, 16 self-attention heads, a hidden size of 1024 dimensions, and around 340 million parameters.

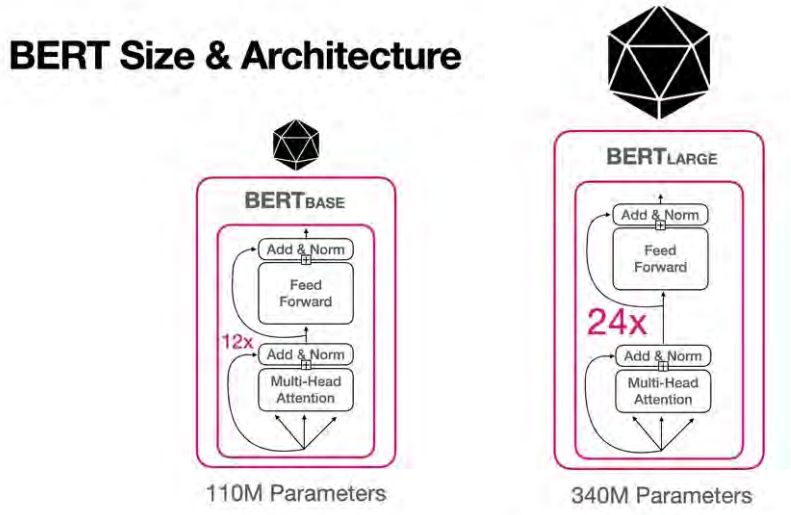

Figure 6.3: BERT Base and Large Models Parameters

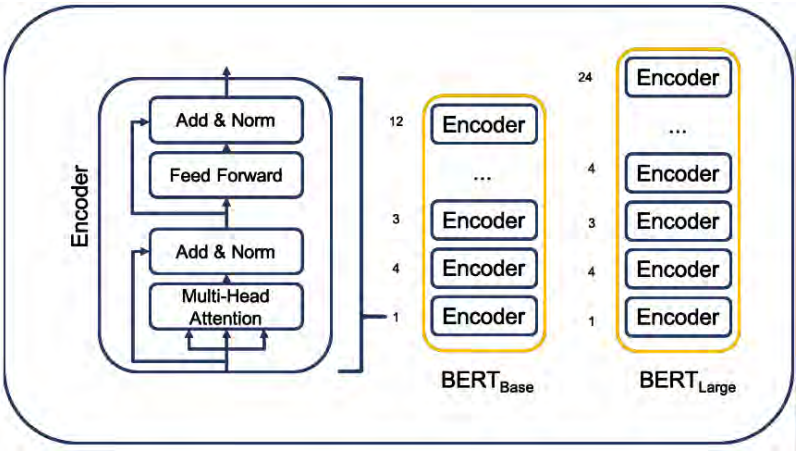

Figure 6.4: BERT Base and Large Models Architecture

Unlike traditional language models that read text in a unidirectional manner (left to right or right to left), BERT reads the entire sequence of words bidirectionally during its pre-training phase. This bidirectionality enables BERT to better capture the context of words in a sentence, improving its performance on various NLP tasks, such as question answering, text classification, and named entity recognition.

6.3.1 BERT Model Architecture

As mentioned before, BERT uses a stack of Transformer encoder layers, each of which performs a series of operations to process the input. Here's a breakdown of key components within a BERT encoder layer:

- **Multi-Head Self-Attention**
  - Self-Attention: This mechanism allows the model to weigh the importance of each word in relation to every other word in the sequence. It captures relationships between words at different positions in the input sequence.
  - Multi-Head Attention: Instead of computing a single self-attention score, BERT uses multiple attention heads (12 in BERT base, 16 in BERT large). Each head learns different aspects of the relationship between words. These attention heads run in parallel, and their outputs are combined to capture richer relationships.
- **Feed-Forward Neural Network (FFNN)**
  - After the attention layer, each token's representation is passed through a fully connected feed-forward neural network. This consists of two linear transformations with a Gaussian Error Linear Unit(GELU) activation in between.
  - The FFNN is applied to each position (token) in the sequence independently.
  - This layer enables non-linear transformations and captures higher-level features.
  - For BERT base, the hidden size of the feed-forward network is 3072.
- **Layer Normalization and Residual Connections**
  - Residual Connections: To help with gradient flow and improve training stability, each encoder layer employs residual (skip) connections. This means that the input to a sub-layer (attention or FFNN) is added to the output of that sub-layer, allowing the model to learn both from the raw input and the transformation.
  - Layer Normalization: After both the self-attention and feed-forward layers, layer normalization is applied. This helps stabilize and speed up the training process by ensuring that the activations are standardized.
- **Position-wise Encoding**
  - BERT is not inherently aware of word positions because Transformers, unlike RNNs or CNNs(Convolutional Neural Network), do not have a built-in sense of order. To solve this, BERT adds positional encodings to the word embeddings at the input level, which provide information about the position of each token in the sequence.

- **Token Embeddings**
  - The inputs to BERT consist of three embeddings: token embeddings (word-piece tokens), segment embeddings (which sentence the token belongs to), and positional embeddings (the position of each token).
  - These embeddings are summed and passed through the encoder layers.

6.4 Machine Learning Model Architecture

Our model leverages the various utilities and advantages offered by **BERT**6.3. Given our objective of developing a model that can learn all the patterns present in the dataset, the BERT model is the most suitable choice for this purpose. For better understanding, we will proceed step by step, and the entire model architecture will be analyzed in subsection 6.4.3. The model's approach and architecture are inspired by the work in [27]; however, instead of building a Transformers model from scratch, we leveraged BERT's capabilities, as done in [28] and [29]. The primary objective is to pre-train and fine-tune the BERT model for the task of next event prediction.

6.4.1 Pre-Training

The initial phase focuses on learning the patterns present in the sequences by capturing the embedding representations of the events in the dataset. To achieve this, we utilize the *BertForMaskedLM* from the BERT-Base Model, which retains the architecture of BERT Base while adding an additional output layer. This output layer generates prediction probabilities for the masked tokens, with the '`[MASK]`' token assigned the ID 103. In this phase, 15% of the events in each sequence are randomly masked during every epoch. According to researchers, this masking ratio significantly aids BERT in effectively capturing patterns within the data. As previously mentioned in the Figure 5.4, there are a total of 264 distinct events, represented in hexadecimal format. To prepare these events for input into the BERT model, they must be properly encoded. This process is critical, as BERT is designed with specific ID encodings for its special tokens, which include:

- **'[PAD]': 0**

  Used to pad sequences to ensure uniform input lengths in batch processing. Padding allows models to handle variable-length sequences efficiently.

- **'[UNK]': 100**

  Represents unknown tokens that are not present in the model's vocabulary. It is used to handle out-of-vocabulary words or tokens during tokenization.

- **'[CLS]': 101**

  The classification token added to the beginning of sequences for tasks such as classification or regression. It aggregates information from the entire sequence for final predictions.

- **'[SEP]': 102**

  Serves as a separator token between different segments in a sequence, particularly in tasks involving multiple sentences or text pairs, such as question answering and sentence similarity.

- **'[MASK]': 103**

  Used during training for masked language modeling tasks. It replaces tokens in the input, allowing the model to predict the masked tokens based on their context.

Since these IDs are already assigned to special tokens and cannot be modified, a **Custom Encoder** was developed to prevent conflicts. For instance, if an event ID is assigned the encoded ID 103, the BERT model will recognize this ID as the mask token and make predictions for it during each epoch. Therefore, these five special IDs remain unchanged, while the remaining hexadecimal events are mapped to distinct ID numbers, as illustrated in Figure 6.5.

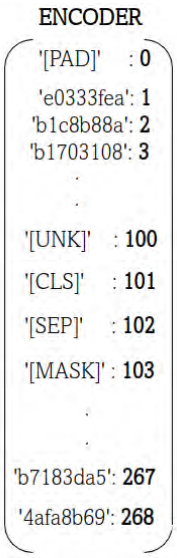

Figure 6.5: Custom Encoder

The BERT vocabulary size is then modified to recognize only these values, ranging from 0 to 268, resulting in a total vocabulary size of 269. The model is pre-trained to accurately predict the masked events. Once the model successfully makes predictions, it generates precise embedding representations for each event in the sequence. The loss function utilized is Cross Entropy Loss combined with Softmax Activation, since its ideal for this multi-class classification task, where the model predicts which event should replace the mask tokens. The entire pre-training phase process is illustrated in Figure 6.6.

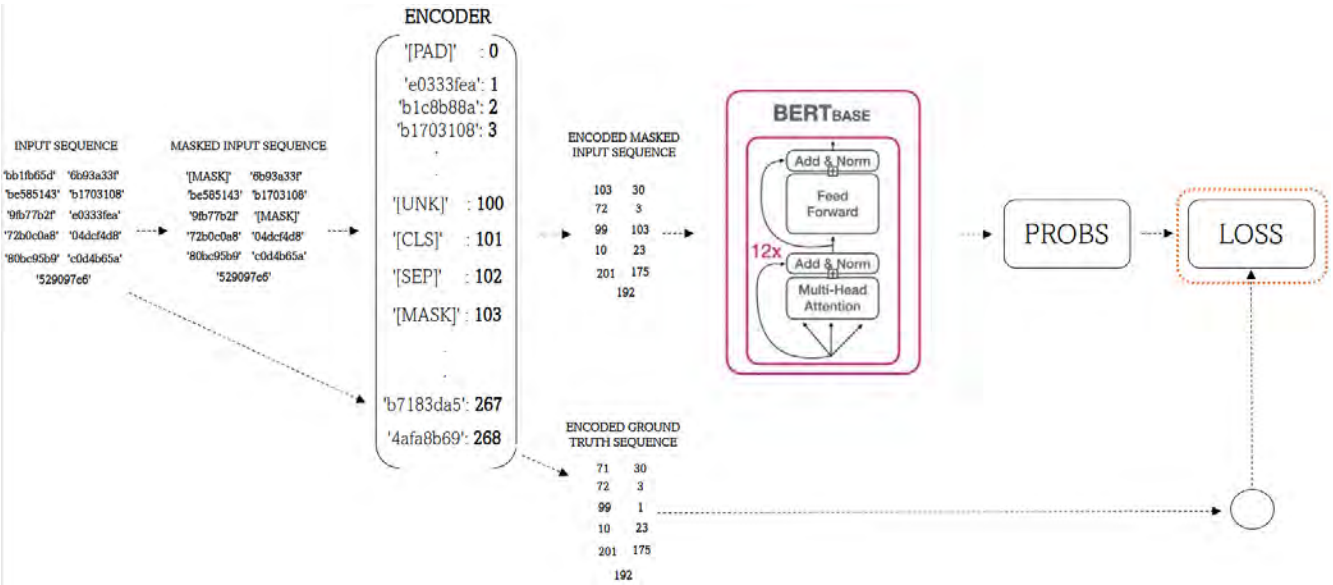

Figure 6.6: Pre-training Phase Pipeline

The pre-training loss is illustrated in Figure 6.7.

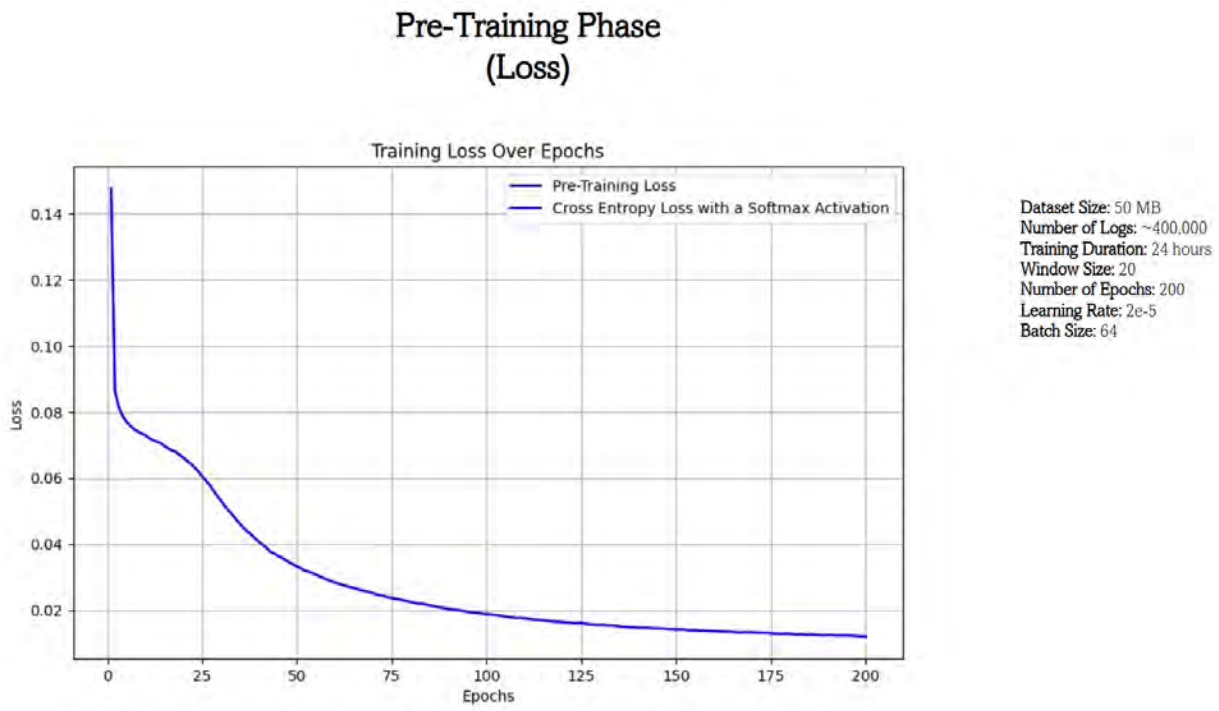

Figure 6.7: Pre-training Loss

As observed, the minimum loss achieved after 200 epochs is approximately 0.01 for a custom 50MB Android 10.0 dataset, which included nearly 400,000 logs. The window size was set to 20, resulting in sequences with a length of 20 events. The batch size was set to 64, and the learning rate was configured to 2e-5. This low error loss indicates that the BERT model effectively pre-trained on the sequences, successfully capturing event embeddings and patterns.

6.4.2 Fine-Tuning

The next phase focuses on fine-tuning the pre-trained BERT model, as described in subsection 6.4.1. Specifically, as mentioned earlier, the architecture is adapted for next-event prediction. The output layer originally used for masked token prediction is removed, and all parameters of the model's Transformer encoders are frozen, as these layers have already captured the embeddings of the events in the sequences. Then, a fully connected LSTM layer is attached to the modified model to predict the probabilities of the next event in the sequence. The expectation is that, given the pre-trained model's successful pattern capture and its ability to produce embeddings, the LSTM will generate accurate predictions for the last event in each sequence. Thus, this new architecture takes input sequences of length *window size - 1*

and uses the last event of each sequence as the target. Additionally, the LSTM takes timestamps of the logs as an extra input. These timestamps are converted into a cyclical format (using sine and cosine transformations) to allow the model to capture the recurring nature of time-based patterns. This conversion ensures that the model recognizes the periodicity of time, such as hours in a day or days in a week, rather than interpreting timestamps as simple linear values. By using cyclical timestamps, the model better understands relationships between logs that occur at similar times, improving its ability to predict events influenced also by time. Since the sequences are formed using the sliding window technique described in Algorithm 1, every log in the dataset is utilized for prediction. This means that all logs present in the dataset are thoroughly examined. The entire fine-tuning phase process is illustrated in Figure 6.8.

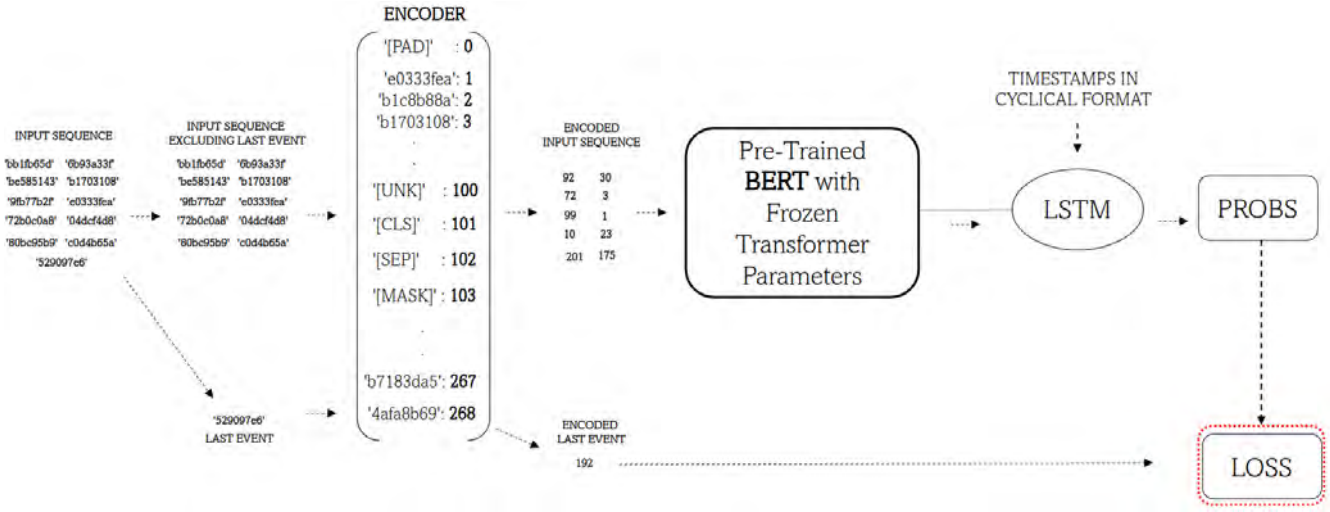

Figure 6.8: Fine-Tuning Phase Pipeline

The fine-tuning loss is illustrated in Figure 6.9.

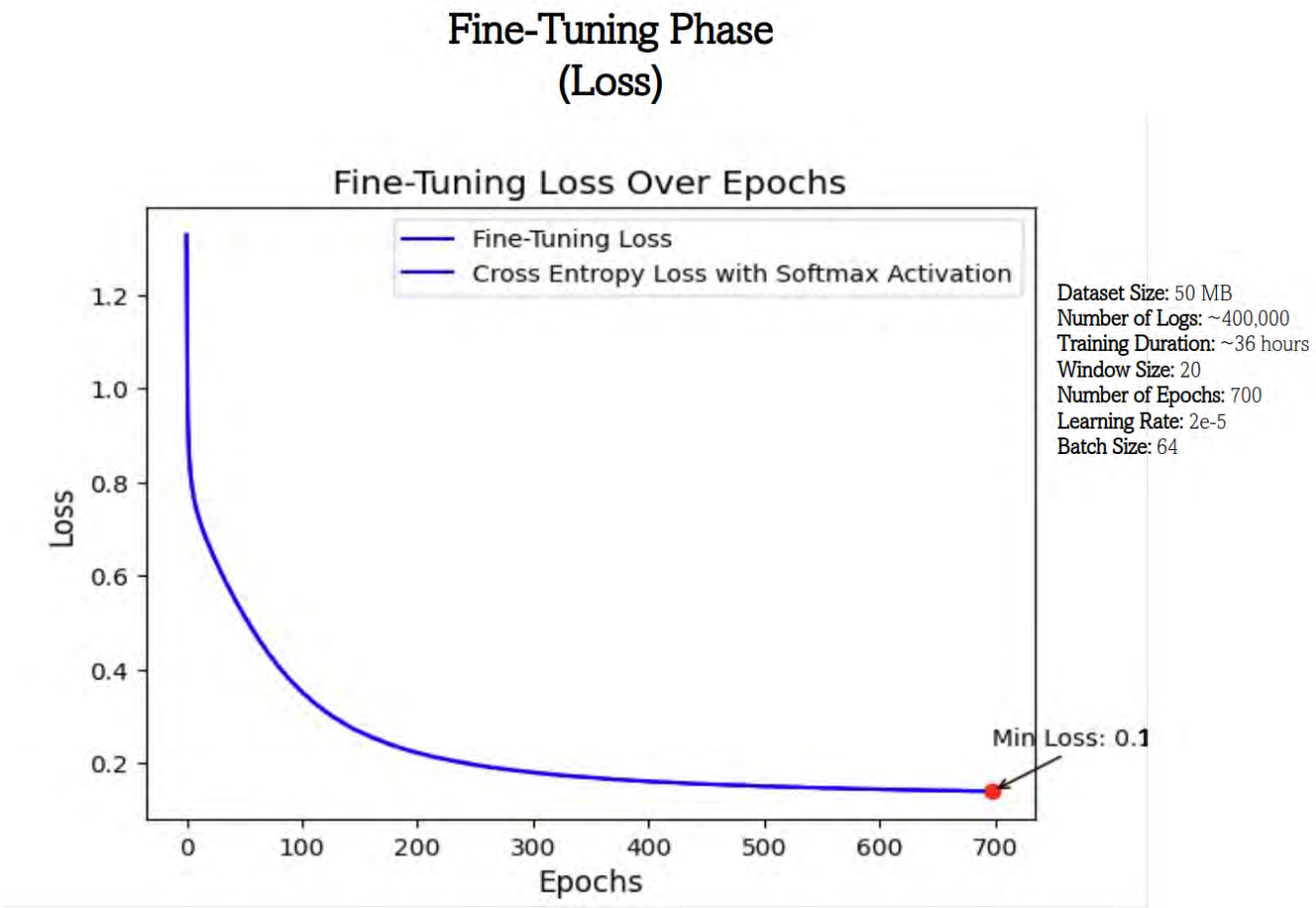

Figure 6.9: Fine-Tuning Loss

As observed, the minimum loss achieved after 700 epochs is approximately 0.1 for the same dataset used in the pre-training phase, which included nearly 400,000 logs. The window size was set to 20, creating sequences of 20 events. With a batch size of 64 and a learning rate of 2e-5, this low error loss indicates that the fine-tuned BERT model makes nearly successful predictions of the last event for each sequence of length *window size - 1* in the dataset.

6.4.3 Comprehensive Model Architecture

The final stage of the model involves introducing two additional custom layers:

- **SlicingLayer**: This layer processes each sequence by separating it from its final event. Specifically, for every event intended for evaluation, this layer takes a sequence of events with a length equal to the *window size* and extracts the first *window size - 1* events, isolating the last event. The core idea is to assess whether the event to be examined (which is the last event) could have occurred, based on the patterns and behaviors

learned by the model during the pre-training and fine-tuning phases, given the previous *window size - 1* events.

- **SequenceAbnormalityClassifierLayer**: This layer serves as a classifier that processes the outputs from the Slicing Layer (input sequence excluding the last event and the last event) along with the probabilities generated by the fine-tuned model. It evaluates whether the last event is included among the top-g classes with the highest predicted probabilities. If the last event is present in the top-g predicted classes, the layer outputs "**NORMAL**"; if not, it outputs "**ABNORMAL**". The value of g represents a threshold that determines the model's strictness. Currently, g is set to 10. If the last event appears among the top-10 predictions made by the model, the log is classified as normal based on the preceding *window size - 1* events; otherwise, it is classified as abnormal.

The final model architecture and pipeline are illustrated in Figure 6.10. For clarity, the fields highlighted in light orange represent the comprehensive architecture of the model, while the overall diagram depicts the primary flow of the pipeline's architecture.

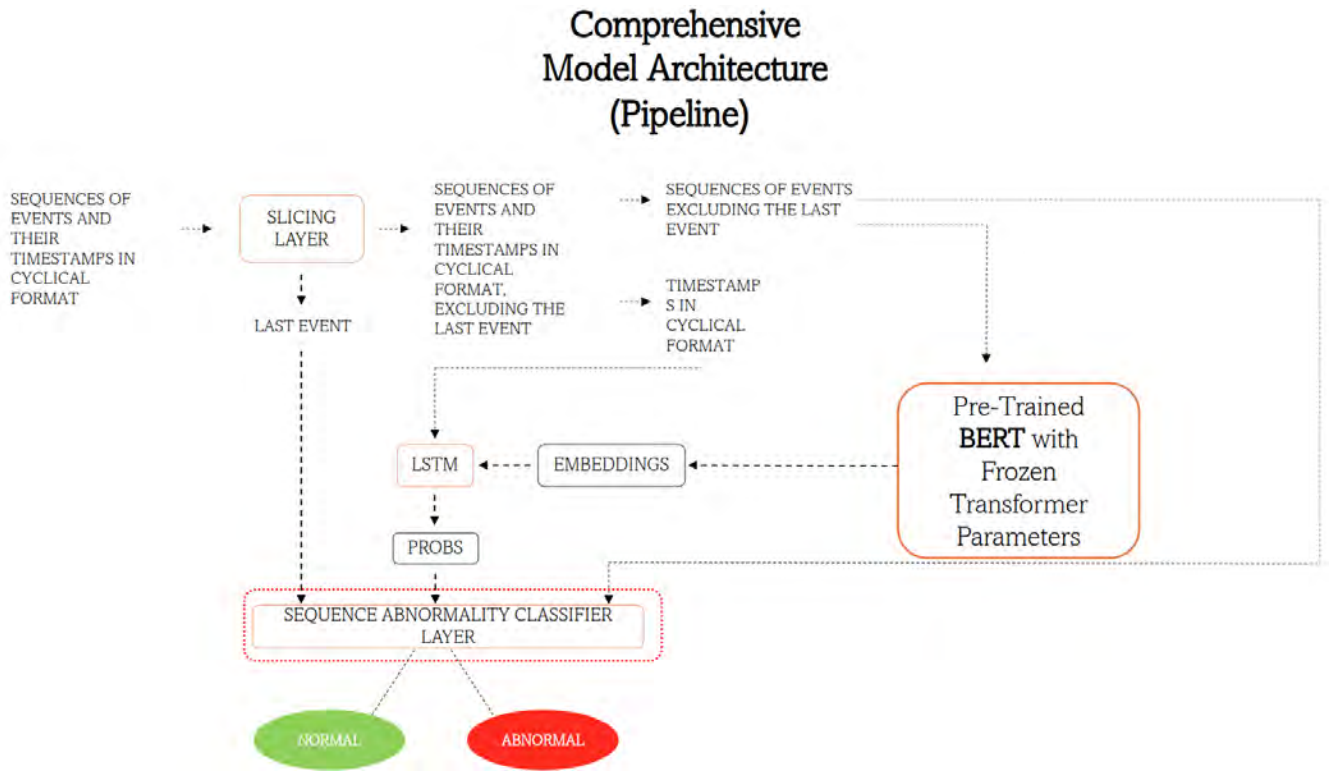

Figure 6.10: Comprehensive Model Architecture - Pipeline

6.5 Evaluation

The architecture of the model, as depicted in Figure 6.10, was specifically designed to be trained on data consisting solely of normal behavioral patterns. In other words, the dataset will exclusively capture the default behaviors of user devices under typical conditions. The goal is that, after training, the model will effectively learn the sequence patterns inherent in the normal dataset. As a result, it should be capable of detecting deviations from these learned patterns, allowing it to identify anomalous behavior that diverges from the baseline dataset. We first evaluated the model on the normal dataset, followed by evaluations where the dataset was randomly shuffled according to varying shuffle ratios. This approach was used to assess the model's accuracy in detecting deviations from normal patterns, since labeled data is unavailable and is necessary to create custom scenarios to evaluate the model's accuracy. As mentioned in subsection 6.4.3, given that the model is designed to predict the next event in a sequence, we defined abnormality based on the top-g likelihood events, where g is set to 10. It is worth noting that the pre-training, fine-tuning, and evaluation phases are conducted on a PC tower, the hardware components of which are detailed in Table 6.1.

Table 6.1: Hardware Components of the PC Tower

Component	**Specifications**
Processor	13th Gen Intel Core i9-13900KF x 8
Memory	128.0 GB DDR5
Graphics	NVIDIA GeForce RTX 4070
Disk Capacity	5.0 TB

6.5.1 Evaluation on Normal Training Dataset

The first evaluation involves testing the model on the dataset it was trained on. The goal here is for the model to classify all logs as normals, meaning it should make accurate predictions for the upcoming events based on the learned patterns in the sequences. While this may seem unusual, after experimenting with various model architectures and implementations, this particular approach was the only one that successfully flagged the entire training dataset as normal. This indicates that the model effectively captured all the behavioral patterns present in the dataset, as illustrated in Figure 6.11.

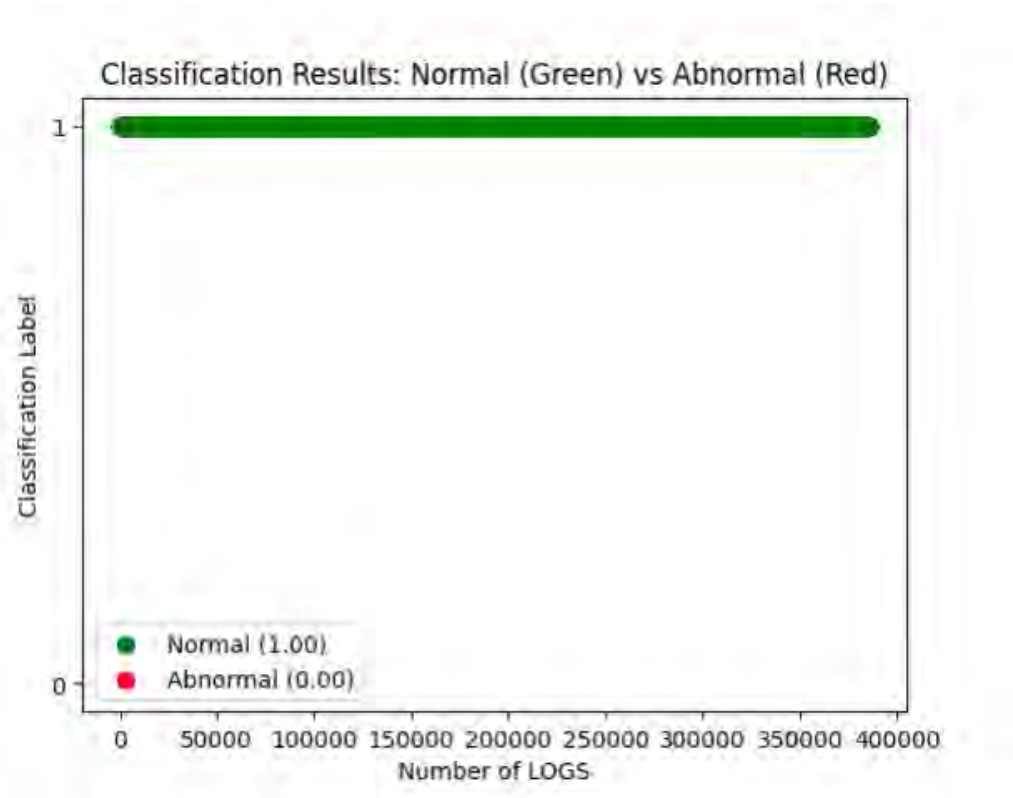

Figure 6.11: Evaluation on Normal Training Dataset

As observed, all the logs are marked as normals, as the model has effectively learned to predict the next event based on the identified patterns.

6.5.2 Evaluation with 5% of Dataset Randomly Shuffled

The second evaluation involves randomly shuffling 5% of the logs in the training dataset. As a result, we expect an average abnormality rate of approximately 5%, since shuffling this portion of the logs alters 5% of the sequences as well.

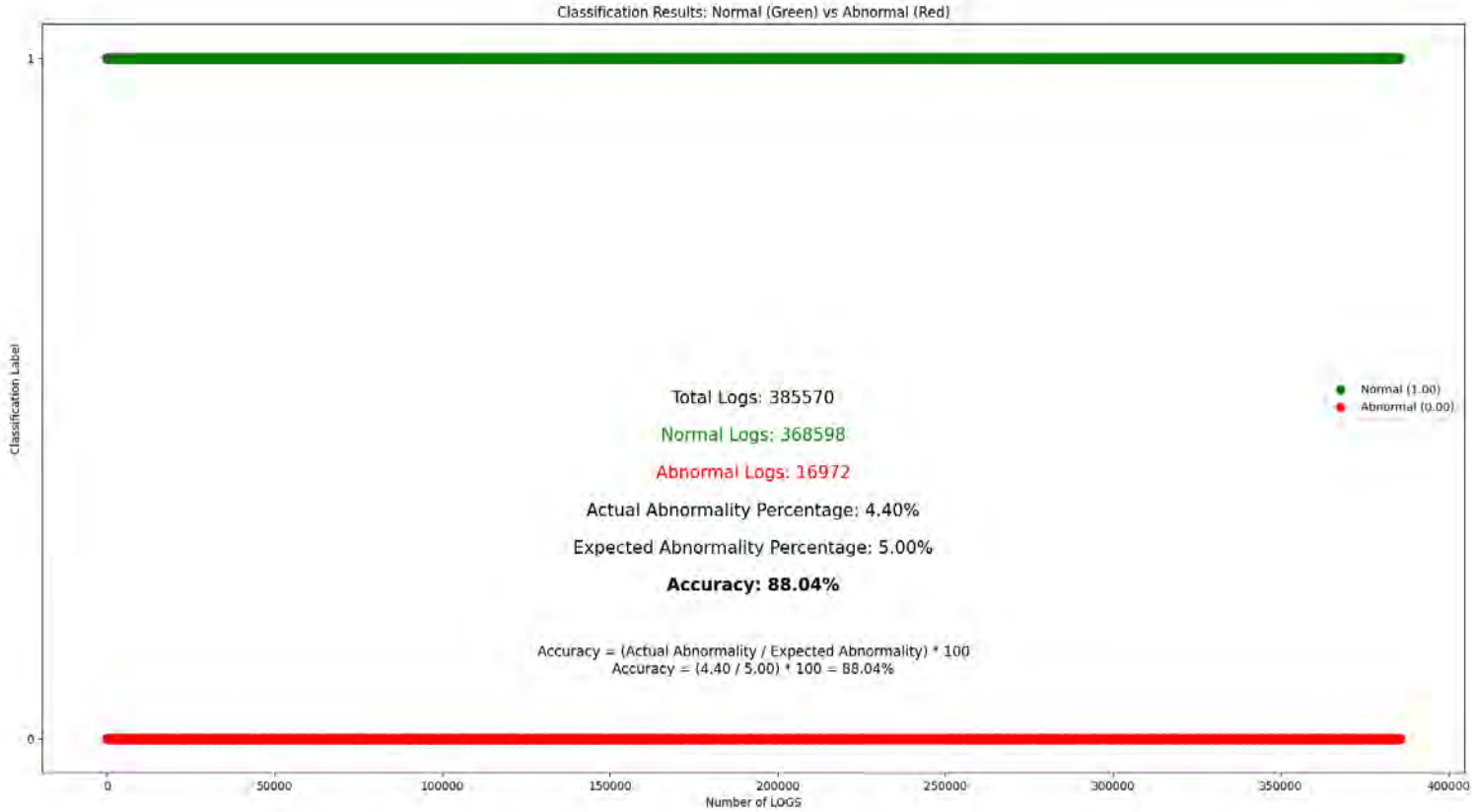

Figure 6.12: Evaluation with 5% of Dataset Randomly Shuffled

Upon examining the Figure 6.12, one might notice that the two lines representing normal logs and abnormal logs appear similar. However, the Figure 6.13 clearly indicates that the quantity of normal logs is significantly greater than that of abnormal logs. The similarity in the lines is due to the large volume of logs, which can obscure the differences between the two categories.

Figure 6.13: Evaluation with 5% of Dataset Randomly Shuffled

As observed, the model flagged 4.40% of the logs as abnormal, while the expected abnormality was equivalent to the proportion of logs shuffled in the dataset. The actual abnormality percentage can be determined using the following method:

$$\texttt{Actual Abnormality Percentage} = \frac{\texttt{Abnormal Logs}}{\texttt{Total Logs}} * 100 = \texttt{4.40\%} \quad (6.1)$$

This resulted in an accuracy of:

$$\texttt{Accuracy} = \frac{\texttt{Actual Abnormality Percentage}}{\texttt{Expected Abnormality Percentage}} * 100 = \textbf{88.04\%} \quad (6.2)$$

.

6.5.3 Evaluation with 10% of Dataset Randomly Shuffled

The third evaluation involves randomly shuffling 10% of the logs in the training dataset. Consequently, we expect an average abnormality rate of approximately 10%, as shuffling this portion of the logs alters 10% of the sequences as well.

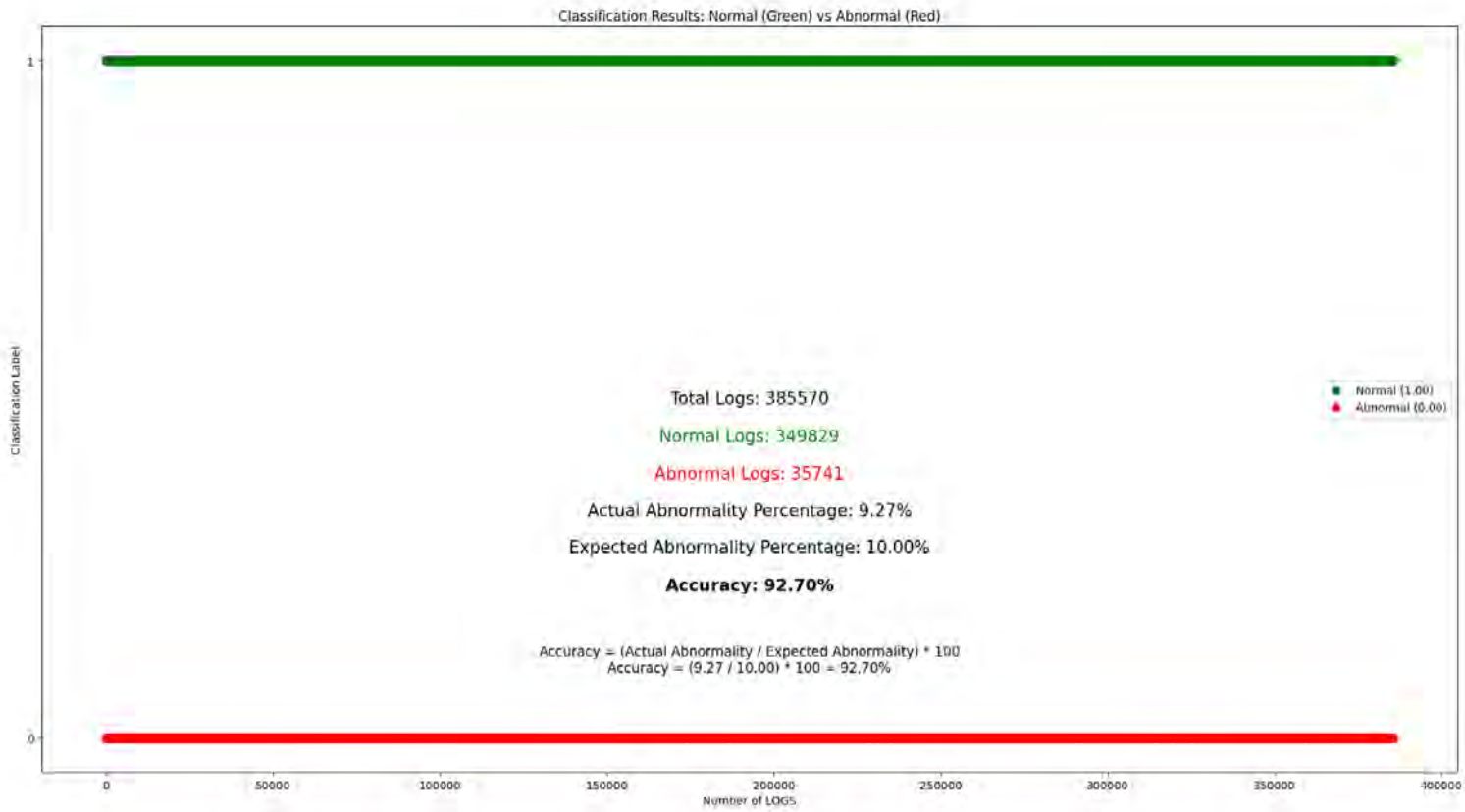

Figure 6.14: Evaluation with 10% of Dataset Randomly Shuffled

As illustrated in Figure 6.14, the actual abnormality percentage is calculated as before:

$$\texttt{Actual Abnormality Percentage} = \frac{\texttt{Abnormal Logs}}{\texttt{Total Logs}} * 100 = \texttt{9.27\%} \quad (6.3)$$

This yields an accuracy nearly equivalent to the second evaluation test:

$$\texttt{Accuracy} = \frac{\texttt{Actual Abnormality Percentage}}{\texttt{Expected Abnormality Percentage}} * 100 = \mathbf{92.70\%} \quad (6.4)$$

.

6.5.4 Evaluation with 20% of Dataset Randomly Shuffled

The final evaluation involves shuffling 20% of the logs in the training dataset. Consequently, we expect the average abnormality rate to be approximately 20%, as this shuffling alters 20% of the sequences.

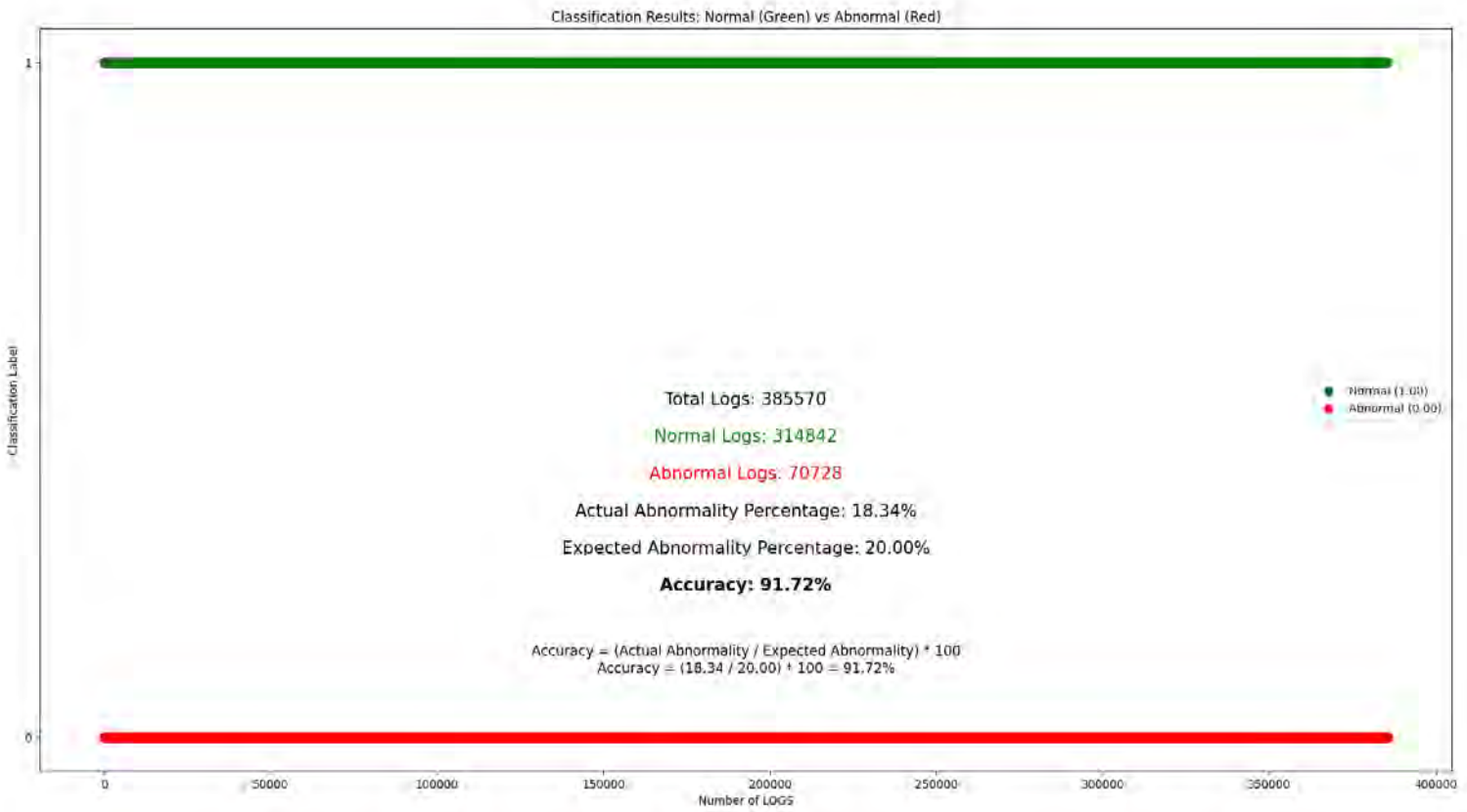

Figure 6.15: Evaluation with 20% of Dataset Randomly Shuffled

The actual abnormality percentage is now:

$$\texttt{Actual Abnormality Percentage} = \frac{\texttt{Abnormal Logs}}{\texttt{Total Logs}} * 100 = \texttt{18.34\%} \quad (6.5)$$

, resulting in an accuracy comparable to the previous evaluation tests:

$$\texttt{Accuracy} = \frac{\texttt{Actual Abnormality Percentage}}{\texttt{Expected Abnormality Percentage}} * 100 = \textbf{91.72\%} \quad (6.6)$$

.

6.5.5 Average Model's Accuracy

Given the variability in the evaluation test results, which yield relatively similar accuracies, the average accuracy of the model can be calculated by summing all accuracy values and dividing by the number of tests, as illustrated in the formula below:

$$\textbf{Average Accuracy} = \frac{88.04\% + 92.70\% + 91.72\%}{3} = \textbf{90.82\%} \quad (6.7)$$

Chapter 7

Conclusions

7.1 Conclusion and Results

In today's digital world, the demand for security measures has never been more critical due to the exposure of data to various threats. While the traditional authentication mechanisms discussed in Section 2.1 remain effective for initial user verification, they fail to authenticate users continuously. On the other hand, integrating these mechanisms with Continuous Authentication systems that leverage ML technologies, can significantly enhance security and accuracy in protecting user data. By monitoring user devices in real time, Continuous Authentication enables ongoing validation of user identity without requiring additional passwords or credentials. In scenarios where devices are stolen or credentials are compromised, they can detect deviations from normal user behavior over time, allowing the system to prompt for additional credentials or lock the device if suspicious activity is detected.

In conclusion, the combination of traditional and **Continuous Authentication** mechanisms presents a comprehensive solution for protecting sensitive data continuously, ensuring that users remain protected all the time, even in the face of evolving threats.

7.2 Future Work

The final step involves enhancing the System Architecture presented and analyzed in this thesis to ensure greater security. By utilizing the HTTPS protocol instead of HTTP, secure data transmission can be guaranteed through Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols.

Currently, user credentials are stored in local file-system storage, allowing for efficient management of sensitive data. To further enhance security, an integration of MongoDB as the database solution is in progress. This integration will not only boost the safety of user data but also provide greater flexibility in data handling. By utilizing MongoDB, a more secure infrastructure for managing user credentials can be established.

In terms of model accuracy, the BERT Large model is proposed, as discussed in Section 6.3. The BERT Large model features double the encoding layers and triple the trainable parameters compared to the Base Base model employed in this thesis. This increased capacity is expected to yield better representations and more effective embeddings. Consequently, the BERT Large model is expected to capture patterns within the dataset more accurately, leading to improved embedding generation.

Last, rather than solely relying on Long Short-Term Memory (LSTM) networks, the architecture can be enhanced by stacking multiple layers on top of the pre-trained model. A combination of multiple LSTMs with Convolutional Layers is currently recognized as an effective approach for such tasks nowadays.

Bibliography

[1] miniOrange. Different types of authentication. https://www.miniorange.com/blog/different-types-of-authentication-methods-for-security/.

[2] Continuous authentication: Definition benefits. https://www.incognia.com/the-authentication-reference/continuous-authentication.

[3] Javed Shah. Continuous authentication: A dynamic approach to user verification. https://www.1kosmos.com/authentication/continuous-authentication-guide/.

[4] Budhdi Sharma. Android log analysis. https://budhdisharma.medium.com/android-log-analysis-176f9b9dafaf.

[5] Log classes. https://stackoverflow.com/questions/7959263/android-log-v-log-d-log-i-log-w-log-e-when-to-use-each-one.

[6] Android debug bridge (adb). https://developer.android.com/tools/adb.

[7] View logs with logcat. https://developer.android.com/studio/debug/logcat.

[8] Rooting android devices. https://www.reddit.com/r/androidroot/comments/l6qasc/how_i_rooted_my_lenovo_tab_m8_research_on_mtk/.

[9] How to use su command over adb shell. https://stackoverflow.com/questions/27274339/how-to-use-su-command-over-adb-shell.

[10] jesama HelenGuohx. logparser. https://github.com/HelenGuohx/logbert/blob/main/logparser/Drain.py. Year: 2021.

[11] Kanika Thakral. Sliding window technique. https://www.geeksforgeeks.org/window-sliding-technique/.

[12] Yue Yin, Peng Li, and Jing Chen. A variable sliding window algorithm based on concept drift for frequent pattern mining over data streams*. In *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 818–825, 2023.

[13] Glatard T Shihab E Dehghani A, Sarbishei O. A quantitative comparison of overlapping and non-overlapping sliding windows for human activity recognition using inertial sensors. *Sensors (Basel)*, 19:5026, 2019.

[14] Muzna Yumman. Algorithm patterns- dynamic sliding window technique. https://www.linkedin.com/pulse/algorithm-patterns-dynamic-sliding-window-technique-muzna-yumman/.

[15] Wikipedia. Machine learning. https://en.wikipedia.org/wiki/Machine_learning.

[16] Sara Brown. Machine learning, explained. https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained.

[17] Tom Keldenich. 7 types of layers you need to know in deep learning and how to use them. https://inside-machinelearning.com/en/7-types-of-layers-you-need-to-know-in-deep-learning-and-how-to-use-them/.

[18] Layers. https://ml-cheatsheet.readthedocs.io/en/latest/layers.html.

[19] Wikipedia. Transformer (deep learning architecture). https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture).

[20] Amazon Web Services(AWS). What are transformers in artificial intelligence? https://aws.amazon.com/what-is/transformers-in-artificial-intelligence/.

[21] Nvidia. What is a transformer model? https://blogs.nvidia.com/blog/what-is-a-transformer-model/.

[22] Wikipedia. Gpt-3. https://en.wikipedia.org/wiki/GPT-3.

[23] Wikipedia. Generative pre-trained transformer. https://en.wikipedia.org/wiki/Generative_pre-trained_transformer,.

[24] Wikipedia. Bert (language model). https://en.wikipedia.org/wiki/BERT_(language_model),.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*, 2019.

[26] Sanna Persson. Paper summary — bert: Bidirectional transformers for language understanding. https://medium.com/analytics-vidhya/paper-summary-bert-pre-training-of-deep-bidirectional-transformers-for-language-understanding-861456fed1f9,.

[27] Senming Yan, Lei Shi, Jing Ren, Wei Wang, Yaxin Liu, Limin Sun, Xiong Wang, and Wei Zhang. Log-based anomaly detection with transformers pre-trained on large-scale unlabeled data. In *ICC 2024 - IEEE International Conference on Communications*, pages 1–6, 2024.

[28] Haixuan Guo, Shuhan Yuan, and Xintao Wu. Logbert: Log anomaly detection via bert. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2021.

[29] Xiaolin Chai, Hang Zhang, Jue Zhang, Yan Sun, and Sajal K. Das. Log sequence anomaly detection based on template and parameter parsing via bert. *IEEE Transactions on Dependable and Secure Computing*, pages 1–18, 2024.