الجمهورية الجزائرية الديمقراطية الشعبية
PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
وزارة التعليم العالي و البحث العلمي
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
جامعة عمار ثليجي بالأغواط
AMAR TELIDJI UNIVERSITY OF LAGHOUAT

كلية العلوم
قسم الإعلام الآلي
FACULTY OF SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Master's Thesis

Field: Mathematics and Computer Science
Specialty: Computer Science
Option: Data Science and Artificial Intelligence

By: Zakeria Ramache , Tahar Hachani

A Real-Time, Collaborative Host-Based Intrusion Detection System Leveraging BigBird Embeddings and Deep Q-Learning

---

*Defended publicly on **July 2025**, before a jury composed of:*

Dr.	Leila Benarous	President
Dr.	Messaoud Babaghayou	Examiner
Dr.	Yousra Cheriguene	Examiner
Dr.	Tahar Allaoui	Supervisor

Thesis No. ..... **Academic Year 2024/2025**

Acknowledgment

﴿ الحمد لله الذي هدانا لهذا وما كنا لنهتدي لولا أن هدانا الله ﴾ (سورة الأعراف: 43)
﴿ وَمَا تَوْفِيقِي إِلَّا بِاللَّهِ عَلَيْهِ تَوَكَّلْتُ وَإِلَيْهِ أُنِيبُ ﴾ (سورة هود: 87)

The first and last thing is to thank Allah, who provided us with sufficient capacity to finish this work.

We would like to express our deepest gratitude to **Dr.Saida Sarra Boudouh** for her invaluable guidance, insightful feedback, and unwavering support throughout our study. Her expertise and encouragement were instrumental in shaping this research and bringing it to fruition.

الملخص

تقدم هذه الأطروحة نظامًا ذكيًا للكشف عن الهجمات الإلكترونية على أجهزة الكمبيوتر من خلال تحليل نشاط النظام آنيًا. يركز النظام على المعلومات المُجمعة من أنظمة تشغيل لينكس، ويستخدم أحدث التطورات في الذكاء الاصطناعي لتحديد السلوكيات المشبوهة. يعالج النظام سجلات النظام، ويحولها إلى صيغة يفهمها الجهاز، ويستخدم برنامجًا تعليميًا لتحديد ما إذا كان النشاط طبيعيًا أم ضارًا. يتطور النظام بمرور الوقت من خلال التعلم من تجربته الخاصة. صُمم النظام للاستجابة السريعة للتهديدات مع تقليل الإنذارات الكاذبة، ويمكن نشره على أجهزة مختلفة لتبادل المعرفة. يهدف هذا العمل بشكل عام إلى توفير حل حديث ومرن لتعزيز أمن الكمبيوتر.

الكلمات المفتاحية:

الأمن السيبراني، الوقت الحقيقي، سجلات النظام، لينكس، الذكاء الاصطناعي.

Abstract

This dissertation presents a smart system for detecting cyber-attacks on computers by analyzing system activity in real time. The system focuses on information collected from Linux operating systems and uses recent advances in artificial intelligence to identify suspicious behavior. It processes system logs, transforms them into a form that a machine can understand, and uses a learning agent to decide whether the activity is normal or potentially harmful. The system improves over time by learning from its own experience. It is designed to respond quickly to threats while minimizing false alarms, and it can be deployed across different machines to share knowledge. Overall, this work aims to provide a modern and adaptive solution for enhancing computer security.

Keywords: Cybersecurity, real time, System Logs, Linux , artificial intelligence.

Contents

List of Figures

List of Tables

List of Abbreviations

IDS Intrusion Detection System

HIDS Host-Based Intrusion Detection System

NIDS Network-Based Intrusion Detection System

ML Machine Learning

DQN Deep Q-Network

RL Reinforcement Learning

DDQN Double Deep Q-Networks

DNNs deep neural networks

NLP Natural Language Processing

GPT Generative Pre-trained Transformer

BERT Bidirectional Encoder Representations from Transformers

LSTM Long short-term memory

DDoS Distributed Dos

DoS Denial of Service

ReLU rectified linear unit

VM Virtual Machine

NSL-KDD Not So Large - Knowledge Discovery and Data Mining

UNSW-NB15 University of New South Wales – National Botnet 2015

PANW Palo Alto Networks

AWID A wireless WiFi-based Dataset

ADFA-LD Australian Defence Force Academy – Linux Dataset

LID-DS Linux-based Intrusion Detection Dataset

IP address Internet Protocol address

SSH Secure Shell

VPN Virtual Private Network

GPU Graphics Processing Unit

CPU central processing unit

RAM Random Access Memory

TP True Positives

FP False Positives

FN False Negatives

TN True Negatives

Chapter 1

Introduction

1.1 Context

In today's increasingly connected digital world, cybersecurity threats have become both more frequent and more advanced. Attackers now use a wide range of techniques to compromise systems, from traditional malware and brute-force attacks to more sophisticated methods like social engineering, where people are tricked into revealing sensitive information, and zero-day exploits, which take advantage of unknown software vulnerabilities before they are fixed.

One of the most effective ways to detect such threats is through the analysis of system logs [1]. These logs contain detailed records of events happening on a machine — such as user logins, command executions, and system warnings — and they are often the first indicators of unusual or malicious activity.

Intrusion Detection Systems (IDS) are tools used to monitor and detect such activity [2]. IDS are generally divided into two main types:

- **Network-Based IDS (NIDS)** monitors traffic between computers on a network to detect threats from outside [3].
- **Host-Based IDS (HIDS)** focuses on individual machines, analyzing system logs and local behavior to identify threats coming from within the host itself [4].

While NIDS is useful for monitoring external traffic, HIDS provides deeper insight into what is happening inside the system. It can detect **insider threats**, and **malicious commands** — making it a powerful tool for endpoint protection. However, traditional HIDS systems often rely on static rules or known attack signatures, which limits their ability to detect new or unknown attacks. They may miss subtle patterns, generate false alerts, or fail to keep up with evolving threats.

With the rapid progress in **Artificial Intelligence (AI)**, especially in **Natural Language Processing (NLP)** and **Deep Reinforcement Learning (DRL)**, it is

now possible to build smarter and more adaptive HIDS systems. These can analyze system logs as language data and learn how to detect unusual behavior, even if the attack has never been seen before.

In this thesis, we present a Host-Based Intrusion Detection System that combines system log analysis, the BigBird transformer model, and Deep Q-Learning to detect both known and unknown cyber-attacks in real time — including zero-day threats .

1.2 Problem Statement

Many intrusion detection systems still rely on fixed rules or known attack signatures. While these systems can detect familiar threats, they often fail to identify zero-day attacks, social engineering tactics, and other new behaviors that do not follow known patterns. Additionally, traditional Host-Based Intrusion Detection Systems (HIDS) can be slow, rigid, and prone to generating false positives.

The main problem addressed in this work is how to design a **smart**, **real-time** intrusion detection system that can learn from system logs, adapt to new threats, and share knowledge across different hosts in a distributed environment.

1.3 Objectives of the Thesis

The goal of this dissertation is to develop a collaborative, intelligent HIDS using deep reinforcement learning and natural language processing. The system will:

- **Collect real-time system logs** from Linux machines using `journalctl`.
- **Preprocess and anonymize** entries to remove sensitive information while keeping their semantic meaning.
- **Use the BigBird Transformer** to convert log lines into contextual vector representations suitable.
- **Train a Deep Q-Learning(DQN)agent** on each host to classify log events as normal or malicious.
- **Implement a collaborative learning mechanism** where each host periodically shares its DQN model weights with a **leader node**. The leader evaluates all submitted models, selects the best-performing one, and redistributes it to all hosts for synchronization.

This distributed learning approach allows the system to learn faster, generalize better across environments, and improve detection accuracy while maintaining real-time performance.

1.4 Organization of the Thesis

This dissertation is organized as follows:

- **Chapter 2: Generalities** — Covers system logs, IDS types, BigBird, and Deep Q-Learning.
- **Chapter 3: Related Works** — Reviews academic work in intrusion detection and reinforcement learning.
- **Chapter 4: Proposed System and Methodology** — Describes the proposed system, design.
- **Chapter 5: Results and Discussion** — Summarize the results and compare them with related works.

Chapter 2

Generalities

2.1 Introduction

In this chapter, we present the key concepts, technologies, and tools used in the development of our Host-Based Intrusion Detection System (HIDS). Each component plays a specific role in detecting malicious activity from system logs.

We begin by discussing system logs and explaining why they are a valuable source of information for identifying cyber-attacks. We also describe the journalctl tool, which we use to collect logs in real time on Linux systems[5].

Next, we explain the difference between Host-Based and Network-Based Intrusion Detection Systems and justify why HIDS is better suited for our approach. We then introduce the BigBird Transformer model [6], which helps convert log text into a useful numerical format using natural language processing (NLP). BigBird is chosen over traditional models due to its ability to handle long sequences.

We also describe the Deep Q-Learning (DQN) algorithm[7], a reinforcement learning method used to classify log entries. Compared to traditional classifiers, DQN can learn and adapt over time based on feedback.

In addition, we explain how the raw log data is preprocessed and anonymized before training, which improves both privacy and model performance. The TensorFlow framework is used to implement and train the learning models due to its support for deep learning and GPU acceleration.

Finally, we describe the dataset used in this dissertation. It is based on real Linux system logs and contains both normal and malicious events, making it suitable for training and evaluating a realistic intrusion detection system.

Each of these technologies was selected carefully to build a system that is accurate, efficient, and able to adapt to new types of cyber threats.

2.2 System Logs

System logs are files that store detailed information about the activity of a computer system. They record events such as user logins, system errors, service start-ups, hardware messages, and more. These logs are very important in cybersecurity because they can help detect problems, monitor performance, and identify suspicious behavior.

2.2.1 Logs in Linux Systems

In Linux, system logs are mainly stored in the **/var/log** directory. This is the central location where the operating system, services, and many applications write their log messages. Some common and important log files include:

- **/var/log/syslog** or **/var/log/messages**– captures general system events.
- **/var/log/auth.log** or **/var/log/secure**– stores login attempts and authentication events.
- **/var/log/kern.log**– contains kernel-related messages.
- **/var/log/dmesg**– shows messages from the boot process.

Depending on the Linux distribution, some paths may differ. For example, Ubuntu (Debian-based) uses **/var/log/syslog**, while CentOS (Red Hat-based) prefers **/var/log/messages**.

2.2.2 Why Logs Are Important for Intrusion Detection

System logs can show clear signs of attacks or suspicious behavior. Examples include:

- Too many failed login attempts.
- Unknown services starting automatically.
- Unusual commands executed by users.
- Software installation or removal activity.

Because logs record so much valuable detail, they are an excellent data source for an intrusion detection system (IDS). By analyzing these logs, it is possible to detect attacks like brute-force attempts, privilege escalations, or malware execution.

2.2.3 How Logs Are Used

System administrators can manually read logs using tools such as:

- **tail** – shows the last lines of a log file.
- **grep** – searches for specific words or patterns.
- **dmesg**– shows kernel messages.
- **last** – displays user login history.
- **logger** – adds custom messages to the log.

For automated systems, like in this dissertation, we use a programmatic tool called **journalctl**, which is part of **systemd**, the modern Linux service manager. It gives unified access to many logs and supports real-time streaming.

2.3 Log Collection with journalctl

In modern Linux systems, **journalctl** is the main tool used to view and manage system logs. It is part of **systemd**, which is the default service manager on most Linux distributions today (like Ubuntu, CentOS, Debian, Fedora, and Arch).

2.3.1 What is journalctl ?

journalctl is a command-line utility that lets users view log messages collected by the systemd journal. These logs include events from many parts of the system, such as:

- The kernel.
- System services (e.g., SSH, cron, networking daemons).
- Applications and background scripts.
- Login sessions and authentication.
- Security modules (e.g., PAM, SELinux).
- Networking-related services (e.g., **NetworkManager**, **iptables**, **firewalld**).

Unlike traditional log files stored as plain text (e.g., **/var/log/syslog**), **journalctl** reads logs from a **binary journal**, which makes querying faster and more efficient especially for large systems.

2.3.2 Key Features of journalctl

journalctl supports advanced and flexible log querying. Its most useful features include:

- **View all logs or apply filters**
- **Filter logs by:**
  - Time(**- -since**,**- -until**).
  - Service name(**-u ssh**,**-u NetworkManager**).
  - Priority level (**-p err**,**-p warning**).
  - User ID, group ID, or process ID.
- **Live stream logs** with **-f** (similar to **tail -f**), Ideal for watching real-time login attempts or service restarts.
- **Reverse display(-r)** shows the most recent entries first
- **Limit output(-n 50)** to quickly inspect only recent events
- **Boot-specific logs** with -b, including previous boot history
- **Display logs in different formats** like short, json, or cat
- **Network and firewall logs** are also captured if services like **iptables**, **firewalld**, or **VPN daemons** are running

2.3.3 Why We Chose journalctl

We selected journalctl for this dissertation as our primary log collection tool for the following reasons:

- **Built-in and Standard** It is included by default on all major systemd-based Linux distributions. This means there's no need to install or configure extra software.
- **Real-Time Monitoring** The -f flag enables real-time log streaming, allowing our system to detect attacks the moment they appear in the logs.
- **Unified Log Access** Instead of parsing multiple files in /var/log/, we can access everything — system logs, security logs, kernel logs, network logs — through one tool.
- **Advanced Filtering** journalctl allows us to collect only what we need (e.g., failed SSH logins, blocked firewall traffic), which improves performance and reduces noise.

- **Network Event Visibility** Logs from tools like NetworkManager, sshd, iptables, or VPN daemons can be queried directly, giving us insight into connection attempts, firewall blocks, or remote access behavior which are critical for detecting remote threats.
- **Efficient and Script-Friendly** Logs are stored in a binary format, allowing faster and more compact queries ideal for feeding into a machine learning pipeline.

journalctl is a powerful and flexible tool that provides unified access to all system, security, and network logs. Its real-time capabilities and detailed filtering make it ideal for building a host-based intrusion detection system that can detect threats immediately .

2.4 Intrusion Detection: HIDS vs. NIDS

2.4.1 What is an Intrusion Detection System(IDS)

An **Intrusion Detection System (IDS)** is a security mechanism designed to monitor the behavior of a computer system or network and detect any signs of malicious activity, unauthorized access, or policy violations. Its main purpose is to identify cyber-attacks and abnormal behavior early, allowing for a fast and effective response before any serious damage occurs.

An IDS can be implemented as either **hardware** (a dedicated appliance connected to the network) or **software** (installed on servers or endpoints). Regardless of the form, the system analyzes traffic or system activity in real time or through historical logs.

Based on where and how they collect data, IDS solutions are typically classified into two main categories:

- **Network-Based IDS (NIDS):** Monitors traffic across a network.
- **Host-Based IDS (HIDS):** Monitors activity within a specific host or device.

2.4.2 Network-Based Intrusion Detection System (NIDS)

A **Network-Based IDS (NIDS)** is deployed at strategic points within a network to monitor traffic between devices. It inspects packets transmitted over the network in real time, analyzing their headers and payloads for known attack signatures or anomalies. NIDS is commonly used to detect:

- Port scanning attempts.
- Denial-of-Service (DoS) or Distributed DoS (DDoS) attacks.
- Suspicious network protocols or payloads.

- Known malware communication patterns.

Examples of widely used NIDS tools include **Snort**, **Suricata**, and **Zeek**, which are capable of high-throughput packet inspection and signature-based detection.

2.4.3 Host-Based Intrusion Detection System (HIDS)

A **Host-Based IDS (HIDS)** operates on individual endpoints (servers or workstations). It monitors local activities such as:

- System log files
- User authentication attempts.
- File and directory changes.
- Process executions.
- Privilege escalation attempts.
- Kernel-level events.

HIDS offers deep visibility into what is happening within a single system, making it particularly effective at detecting insider threats, malware, and unauthorized access attempts that may not be visible at the network level.

Popular HIDS tools include **OSSEC**, **AIDE**, and **Tripwire**.

2.4.4 Justification for Choosing HIDS in This Work

This dissertation focuses on building a Host-Based Intrusion Detection System using deep reinforcement learning and log analysis. The decision to adopt HIDS over NIDS was based on several key considerations:

- **Log-Centric Design** Our system is designed to analyze system logs generated by Linux machines. These logs are inherently host-specific and best suited to HIDS architectures, where local activities and behaviors can be monitored with fine-grained detail.
- **Enhanced Internal Visibility** Unlike NIDS, which primarily observes network traffic, HIDS can access system-level events such as authentication failures, process creation, and file access patterns. This provides better insight into potential threats that originate internally or bypass network defenses.
- **Real-Time Local Threat Detection** By leveraging real-time log collection (e.g., via journalctl), the system can immediately detect suspicious actions on the host, such as brute-force login attempts, privilege escalation, or malicious commands.

- **Deployment Simplicity** HIDS does not require access to raw network packets or placement within network infrastructure. It can be deployed independently on any Linux host, making it easier to manage and scale across distributed environments.
- **Privacy and Data Sensitivity** Since HIDS operates at the host level and does not inspect all network traffic, it respects data privacy and reduces the exposure of sensitive user information, which may be embedded in network payloads.

While both NIDS and HIDS play critical roles in modern cybersecurity architecture, their purposes and capabilities differ. In this work, HIDS is the more appropriate choice due to its ability to analyze system logs, monitor local behavior, and provide real-time detection of internal threats. Combined with deep learning, HIDS offers a powerful and adaptive approach to securing Linux systems.

2.5 Log Preprocessing and Anonymization

Before system logs can be used for machine learning, they must be cleaned and transformed into a format suitable for analysis. This process is called log preprocessing. It helps remove unnecessary information, protect user privacy, and prepare the data for input into deep learning models.

2.5.1 Why Preprocessing is Important

System logs are usually raw text that includes sensitive and unpredictable elements such as:

- IP addresses.
- Usernames.
- Timestamps.
- Hostnames.
- Process IDs.

These values can vary widely between systems and sessions, making it harder for a model to learn general patterns. Some of this data is also private and must be anonymized for ethical and legal reasons.

Without preprocessing, the log data would be noisy, inconsistent, and difficult for the model to understand or learn from. We perform preprocessing and anonymization for the following reasons:

- **Protect Privacy:** Sensitive user data is removed before training, which aligns with best practices in ethical AI.
- **Improve Learning Accuracy:** By removing noise and randomness, the model learns to focus on patterns rather than details that don't generalize.
- **Ensure Consistent Input Format:** Log lines are transformed into clean and structured sequences that are ready for NLP processing.

Log preprocessing is a critical step in our system. It protects privacy, reduces noise, and ensures that the machine learning model receives clean and meaningful data. Anonymization allows the model to detect patterns in log behavior without depending on system-specific details, which is essential for building a smart and adaptable intrusion detection system.

2.6 The BigBird Transformer

2.6.1 What is BigBird ?

BigBird is a type of **Transformer** model developed by Google AI to solve the problem of processing **very long sequences** of text. It is an advanced version of the standard Transformer architecture used in natural language processing (NLP), and it was designed to handle longer inputs more efficiently without losing accuracy.

In our system, BigBird is used to **convert each system log into a vector** a list of numbers that captures the meaning and structure of the text. These vectors are then used as input for the Deep Q-Learning agent.

2.6.2 Why BigBird is Needed

Most deep learning models struggle when input sequences (like long log lines) get too long:

- **BERT**, a popular transformer model, works well on short texts (up to 512 tokens), but becomes very slow and memory-intensive for longer inputs [8].
- **LSTM** models can handle sequences of variable length but often **forget long-term dependencies** and **process input slowly**, one word at a time.
- **Standard Transformers** (like GPT or original BERT) require quadratic memory and time with respect to input length, making them inefficient for logs.

System logs are often long, repetitive, and full of important details — so we need a model that can **capture both global context and fine-grained patterns**.

2.6.3 How BigBird Solves These Problems

BigBird introduces a new **sparse attention mechanism**, which combines three types of attention:

- **Global attention:** Some important tokens (like **CLS** or keywords) attend to all others.
- **Random attention:** Adds variety by randomly linking tokens, improving model coverage.
- **Sliding window attention:** Captures local context, similar to how CNNs or LSTMs work.

This attention pattern reduces the complexity from **quadratic to linear**, meaning:

- Faster training and inference.
- Lower memory usage.
- Better handling of very long inputs (thousands of tokens).

2.6.4 Why We Chose BigBird Over Other Models

We chose BigBird because:

- It can process long log entries **without cutting or truncating them**.
- It learns both **local details** and **global details**.
- It works well with modern NLP frameworks like **TensorFlow**.

BigBird is a modern NLP model that solves a key problem in log analysis

handling long, detailed text efficiently. Compared to LSTM and BERT, BigBird is more scalable, faster, and better at capturing useful information across long sequences. In our intrusion detection system, it plays a critical role in turning raw system logs into meaningful data that can be used by the learning agent.

2.7 Deep Q-Learning for Log Classification

2.7.1 What is Deep Q-Learning?

Deep Q-Learning (DQN) is a type of **reinforcement learning (RL)** algorithm. It combines two powerful ideas:

- **Q-learning** a technique where an agent learns the best action to take in a given situation by receiving rewards or penalties.
- **Deep learning** the use of neural networks to handle large and complex input spaces.

In DQN, a neural network is trained to predict a Q-value for each action given a certain state. In our system, the state is the log line (converted into a vector by BigBird), and the actions are the possible classifications:

- **0** = normal.
- **1** = malicious.

The DQN learns to maximize the long-term reward by correctly classifying logs based on patterns it discovers during training.

2.7.2 Why Reinforcement Learning (RL) for Log Classification?

Traditional models used for classification such as Support Vector Machines (SVM), Decision Trees, or even basic neural networks treat each input as **independent**. But in real system logs, entries often come in **sequences** (for example: login attempt → warning → error). Reinforcement learning handles this better because:

- It learns from **sequential decisions**.
- It adapts to changes in data patterns.
- It explores and improves over time through **trial and error**.

This is important in cybersecurity, where attackers can change their behavior and **new types of logs may appear**.

2.7.3 Why We Chose Deep Q-Learning

We selected Deep Q-Learning for our intrusion detection system for several reasons:

- **Works Well with Sequential Log Data** Unlike basic classifiers that see one log line at a time, DQN learns from the flow of logs and understands how certain patterns evolve over time.
- **Good for Imbalanced Data** In cybersecurity, normal logs are much more common than malicious ones. DQN focuses on **learning from feedback**, not just class frequency, which helps improve accuracy on rare but dangerous logs.

Deep Q-Learning is a powerful reinforcement learning method that allows our system to learn how to classify logs over time, improving through feedback. Unlike traditional classifiers that treat every log line in isolation, DQN understands sequences and adapts to new threats. This makes it a strong choice for real-time, intelligent intrusion detection.

2.8 Development Tools

2.8.1 Why We Use Python

Our entire system is developed using Python, a widely used programming language in the fields of machine learning, cybersecurity, and data analysis. Python was chosen for this project because it offers:

- A large number of libraries for machine learning, NLP, and system tools.
- Easy integration with Linux commands (e.g., **journalctl**, **os** [9], **subprocess** [10]).
- Clean and simple syntax, making code easier to write and maintain.
- Fast development cycles and strong community support

Python allows us to combine **log collection**, **preprocessing**, **model training**, and **real-time classification** all in one environment.

2.8.2 What is TensorFlow?

TensorFlow is an open-source deep learning framework developed by **Google** [11] . It is used to build and train machine learning models especially neural networks. TensorFlow provides:

- High-level APIs (like **Keras**) for building models easily
- GPU support for faster training

In our system, we use TensorFlow to build the DQN and BigBird models, manage training sessions, and classify logs in real time.

2.8.3 Why TensorFlow Was the Right Choice

We selected TensorFlow for the following reasons:

- **Support for Deep Q-Learning and BigBird:** TensorFlow provides full support for building reinforcement learning models, including Deep Q-Networks (DQN), either through custom implementation or integration with libraries like TF-Agents.

Additionally, TensorFlow is fully compatible with the Hugging Face Transformers library, which allows seamless use of advanced models such as BigBird for processing long log sequences. This enables us to combine deep reinforcement learning and natural language processing within a single, unified framework.

- **Mature and Actively Maintained:** Developed and maintained by Google, TensorFlow is a widely adopted, well-documented framework with a strong user community and regular updates
- **High Performance on CPU and GPU:** TensorFlow offers hardware acceleration for both training and inference, allowing models to scale efficiently on systems equipped with CPUs or GPUs.
- **Built-in Tools for Monitoring:** TensorFlow includes TensorBoard, a powerful tool for visualizing training progress, model architecture, and performance metrics in real time.

Together, these features make TensorFlow a robust and flexible platform for developing, training, and deploying the deep learning components of our intrusion detection system.

Python and TensorFlow form the foundation of our development stack. Python provides flexibility and easy integration with Linux tools, while TensorFlow powers the deep learning model used to classify system logs. Together, they make it possible to build a smart, adaptable, and real-time intrusion detection system based on modern machine learning techniques.

2.9 Dataset Overview

To train and evaluate our intrusion detection system, we created a custom dataset of system logs. This dataset includes both normal behavior logs and malicious activity logs, which are labeled to help the learning model understand and differentiate between them.

2.9.1 Normal Log Collection

The normal logs were collected from a freshly installed Linux virtual machine that had never been used for real user activity. This clean environment was created using virtualization software and configured with standard system services.

We generated normal activity by:

- Booting and shutting down the system.
- Starting background services (like **SSH** and **cron**)

- Running harmless commands (e.g., **ls**, **echo**, **cat**)
- Performing software updates using package managers

Because the virtual machine was never exposed to the internet or any users, we can be confident that the collected logs reflect **typical system operations without any attacks**. This provides a clean and reliable baseline for training the model on what "normal" looks like.

2.9.2 Simulated Attack Logs

To enable the system to recognize malicious behavior, it was essential to include examples of cyber-attacks in the dataset. However, executing real malware or intrusion techniques on the test environment would have introduced unnecessary risk. Instead, we used a safer and more controlled approach based on **attack simulation**. We manually generated log entries that mimic realistic attack scenarios, including:

- **SSH brute-force attempts**.
- **Suspicious or unusual command execution**.
- **Internal network scanning behavior**.

These logs were carefully crafted using **patterns and structures inspired by real-world cyber-attacks**, as documented in public intrusion datasets and security research. Each simulated entry was assigned a label of **1** to mark it as malicious.

By adopting this method, we were able to create a diverse and representative set of attack samples without compromising the integrity or security of the system. This also ensured a balanced dataset, which is important for training machine learning models to correctly distinguish between normal and abnormal system behavior.

2.9.3 Conclusion

This chapter established the technical foundation for our Host-Based Intrusion Detection System (HIDS). We identified **system logs** particularly Linux logs in `/var/log/` as the core data source for detecting host-level threats like brute-force attacks and privilege escalations. For efficient log collection, we selected `journalctl` due to its real-time streaming capabilities, unified access to system/security/network logs, and advanced filtering. We justified the choice of **HIDS over NIDS** by emphasizing its granular visibility into host-level events, deployment simplicity, and enhanced privacy preservation. Critical preprocessing steps including anonymization of sensitive data (`IPs`, `usernames`) were outlined to ensure privacy compliance and generalize model inputs. Finally, we introduced **BigBird Transformer** for encoding long log sequences and **Deep Q-Learning**

(DQN) for adaptive threat classification, both implemented via **TensorFlow** for scalability. Collectively, these components form a cohesive pipeline designed for real-time, accurate intrusion detection on Linux systems.

Chapter 3

Raleted work

3.1 Introduction

There are currently numerous studies and research in the field of machine learning and deep learning for intrusion detection, which have developed significantly in recent years. This aims to enhance data protection and confidentiality and ensure the integrity and security of corporate information and communications. In signature- based detection systems, intrusions are detected by comparing observed behavior to pre-defined intrusion patterns, while anomaly-based systems focus on understanding normal behavior to identify any deviations or suspicious activities. Some of these studies have relied on deep reinforcement learning (DRL) to detect system events or abnormal network traffic, while others have shown great interest in applying various natural language processing (NLP) techniques to process host data for intrusion detection. In this chapter, we will present some studies related to **reinforcement learning (DRL**) and **natural language processing (NLP)** techniques for developing **intrusion detection systems (IDS)**.

3.2 Reinforcement learning-based IDS

Hsu and Matsuoka proposed a DQN-based anomaly net-work intrusion detection system [3].The proposed Reinforcement Learning (RL) agent is configured as an intrusion detection engine, where correct detection outcomes determine the reward value. The RL model operates in two distinct modes:

1. **Learning mode**: Active training with reward feedback
2. **Detection mode**: Inference using the same model architecture with **paralyzed reward function**

For feature processing, network attributes including IP addresses were converted using **one-hot encoding**. The model was evaluated on three benchmark datasets:

- **NSL-KDD**
- **UNSW-NB15**
- **Palo Alto Networks (PANW)** (directly collected)

Experimental results demonstrated the following detection accuracies:

Table 3.1: Intrusion detection accuracy across datasets

Dataset	**Accuracy (%)**
NSL-KDD	**91.4**
UNSW-NB15	**91.8**
PANW	**97.95**

The PANW dataset showed superior performance with **97.95%** accuracy, indicating strong generalization capabilities on real-world network data.

Lopez et al. implemented and compared various deep reinforcement learning methodologies on a network dataset [12]. The authors implemented and compared various deep reinforcement learning models: DQN, Policy Gradient, and Actor-Critic using labeled network dataset NSL-KDD . As a result, among various deep reinforcement learning models, the DDQN model showed the highest F1-score of 28% , respectively. Notably, the AC model showed more robust accuracy than other models regardless of changes in the discount factor. Referring to the paper, we confirmed that reinforcement learning can perform similarly or superior to artificial intelligence using a general. Hooman Alavizadeh et al. applied a new generation of network intrusion detection methods that combines Q-learning-based reinforcement learning and deep feedforward neural network (DQN) for network intrusion detection [7]. The proposed model can detect different classes of network intrusions. The self- learning capabilities of this model enable it to continuously improve its detection capabilities. The authors provided substantial details of the best methods used to fine-tune various hyperparameters of deep learning-based reinforcement learning methods (e.g., learning rates and discount factor) to achieve more effective self-learning. As a result, the proposed DQN model demonstrated on the NSL-KDD dataset an accuracy of over 90% in classification tasks used in different classes of network intrusions.

3.3 NLP and DRL-BASED HIDS

Yongsik Kim et al. implemented a reinforcement learning based intrusion detection system that understands potential attacks on a user's system and generates a detection rule set for those attacks [1]. The proposed reinforcement learning model includes keywords

extracted from each attack log, creating an optimal rule set that best detects attacks. The author introduced a natural language processing (NLP) method to extract the keywords that characterize each attack log. This study transformed the host system's attack log information into a machine-readable data format. Based on the transformed data, the primary keywords for each attack were extracted. The author applied Actor-Critic, a reinforcement learning methodology, to accurately extract the keywords for the attacks that occurred. Based on the extracted keywords, the Actor-Critic model generated a rule set. The author demonstrated that the proposed framework generates an optimal rule set for each attack based on the ADFA-LD and LID-DS 2021 datasets. On these two datasets, the model achieved detection accuracy of 97% and 95%, respectively.

Table 3.2: Summary of intrusion detection approaches

Reference	**Proposed Algorithm**	**Dataset**	**Best Performance**	**Year**
[3]	DQN	NSL-KDD, UNSW-NB15, (PANW)	91.4%, 91.8%, and 97.95%	2020
[12]	DQN	NSL-KDD	28%	2020
[7]	DQN	NSL-KDD	90%	2022
[1]	NLP+DQN	ADFA-LD, LID-DS	97%, 95%	2025

3.4 Conclusion

This chapter has surveyed cutting-edge approaches in reinforcement learning (DRL) and natural language processing (NLP) for intrusion detection systems. The reviewed studies demonstrate significant advancements in detection capabilities across diverse network environments and attack scenarios. Key observations include:

- **DRL Dominance in NIDS**: Deep Q-Networks and their variants (DQN, DDQN) have emerged as particularly effective for network intrusion detection, consistently achieving accuracy rates above 90% across multiple benchmark datasets (NSL-KDD, UNSW-NB15, AWID). The dual-mode operation (learning/detection) with reward-based optimization provides adaptive defense mechanisms against evolving threats.

- **Hybrid Approaches for HIDS**: The integration of NLP techniques with reinforcement learning (Actor-Critic + NLP) shows exceptional promise for host-based detection systems, achieving up to 97% accuracy by transforming unstructured log

data into actionable security rules. This approach addresses the critical need for interpretable detection mechanisms in enterprise environments.

- **Dataset-Specific Performance**: Real-world datasets (PANW, LID-DS 2021) consistently yield higher detection rates (97.95%, 95% respectively) compared to research benchmarks, suggesting that model performance is heavily influenced by data provenance and collection methodologies.

Notable research gaps identified through this review include:

1. Limited exploration of **cross-dataset generalization** capabilities, with most models optimized for specific data environments
2. Absence of standardized **real-time performance metrics** beyond accuracy/F1-scores
3. Emerging need for **adversarial training** approaches against evolving evasion techniques
4. Insufficient investigation into **computational efficiency** for resource-constrained environments

The following chapters address these gaps by proposing a unified **multi-agent reinforcement learning framework** with cross-platform validation, real-time processing capabilities, and enhanced adversarial robustness - advancing beyond the current state-of-the-art approaches reviewed in this chapter.

Chapter 4

Proposed System and Methodology

4.1 Introduction

This chapter explains the design and implementation of the proposed Host-Based Intrusion Detection System (HIDS). The goal of the system is to detect cyber-attacks in real time by analyzing Linux system logs.

To do this, the system uses advanced machine learning techniques, combining **natural language processing (NLP)** with **deep reinforcement learning**. Log messages are first transformed into numerical data using the **BigBird Transformer** model. Then, a **Deep Q-Learning (DQN)** agent learns how to detect abnormal or malicious activity based on patterns in the logs.

We start with an overview of the complete system architecture. After that, each part of the system is explained in detail. Finally, we describe how the dataset was created and prepared for training and testing.

4.2 System Architecture Overview

The overall architecture of the proposed Host-Based Intrusion Detection System (HIDS) is illustrated in Figure 4.1. This block diagram provides a high-level view of the system's components and the data flow between them.

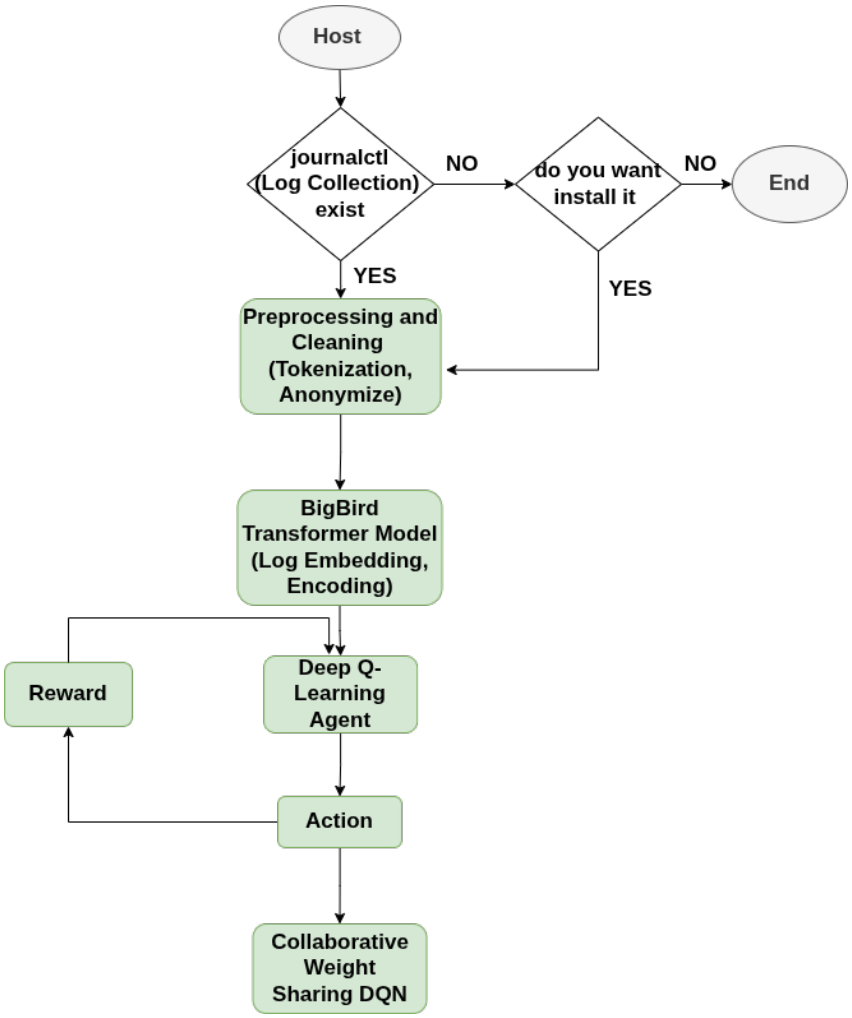

Figure 4.1: Pipeline of the Proposed Intrusion Detection System.

The system is composed of the following key modules:

- **Log Collection (journalctl):** System logs are gathered in real time from Linux hosts using **journalctl**. If the tool is not available, the system ensures it is installed automatically.
- **Preprocessing and Token Management:** Logs are cleaned, anonymized, and tokenized. A dynamic vocabulary file (**vocab.txt**) is updated if new tokens appear.
- **BigBird Embedding:** The processed log messages are converted into vector representations using the BigBird Transformer, which handles long log sequences efficiently.
- **Deep Q-Learning Agent:** The agent classifies logs as normal or malicious based on learned patterns. It improves its accuracy through a feedback reward mechanism.
- **Collaborative Weight Sharing:** In multi-host setups, agents share model weights. A leader selects the best model and distributes it back to ensure consistency.

- **Real-Time Detection Output:** Logs are classified in real time.

4.3 System Components

This section describes the key components of the proposed Host-Based Intrusion Detection System (HIDS). The system follows a modular architecture, where each part performs a specific function within the detection pipeline. The components are executed sequentially, starting from log collection and ending with model sharing.

4.3.1 Log Collection

The first stage in the proposed HIDS pipeline is system log collection. This is performed using journalctl, the standard log querying tool for systemd-based Linux systems. It accesses data stored by the systemd-journald logging daemon, which continuously collects events from the Linux kernel, system services, user processes, authentication modules, and other sources.

How journalctl Works

The journal stores log data in a binary format, allowing fast queries and advanced filtering. Each log entry is stored with rich metadata fields such as:

- **PID**: Process ID.
- **UID**: User ID.
- **EXE**: Path to the binary.
- **MESSAGE**: The actual log message.

The general syntax for querying is:

```
journalctl [OPTIONS...] [MATCHES...]
```

The system uses flags such as:

- **-u** to filter by service name (e.g., **sshd.service**).
- –**since** and –**until** for time filtering.
- **-f** to stream logs in real time.

For example:

```
    journalctl --since "today"
```

This command retrieves all log entries recorded since the beginning of the current day. It is useful for collecting a complete snapshot of recent system activity, including service starts, user logins, security events, and potential anomalies. Logs retrieved this way can then be filtered, preprocessed, and passed to the classification pipeline.

Structure of a Log Entry

Each journal entry has a **timestamp**, a **hostname**, the **process that generated the message**, and the message text. A typical entry looks like this:

```
Jun 10 15:23:51 Host httpd[1854]: Server configured, listening on: port 80
```

Each log line includes:

- **Timestamp (Jun 10 15:23:51)** The date and time the event occurred .
- **Hostname (Host)** The machine name where the log was generated.
- **Source Process (httpd [1854])** The process and its PID that generated the message.
- **Message Content** A human-readable description of the event.

Integration into the Pipeline

The system integrates journalctl using a Python-based module that automates detection, installation, and real-time streaming. Below are the key stages of this integration, with the core logic implemented in code.

Check for journalctl

Before streaming logs, the system checks whether journalctl is available in the environment. This is done using Python's **shutil.which()** function, which searches the system's **PATH**.

```python
import shutil

def is_journalctl_available() -> bool:
    return shutil.which("journalctl") is not None
```

If the tool is missing, the program proceeds to determine the Linux distribution to attempt an installation.

Platform Detection

To choose the correct package manager, the system identifies the Linux distribution by reading from **/etc/os-release**. It detects whether the host is Debian-based, Red Hat-based, or unsupported.

```
def get_os_family() -> str:
    if platform.system() != "Linux":
        return "unknown"
    try:
        with open("/etc/os-release", "r") as f:
            os_info = f.read().lower()
        if "debian" in os_info or "ubuntu" in os_info:
            return "debian"
        elif "fedora" in os_info or "rhel" in os_info or "centos" in
    os_info:
            return "rhel"
    except Exception:
        pass
    return "unknown"
```

Interactive Installation

If journalctl is not installed, the user is prompted to confirm the installation. This allows flexibility while ensuring system safety and user control.

```
import subprocess

def install_journalctl() -> bool:
    os_family = get_os_family()
    if os_family == "debian":
        cmd = ["sudo", "apt", "install", "-y", "systemd"]
    elif os_family == "rhel":
        cmd = ["sudo", "dnf", "install", "-y", "systemd"]
    else:
        print("Unsupported Linux distribution.")
        return False

    confirm = input("Install journalctl? [y/N] ").strip().lower()
    if confirm != "y":
        return False

    try:
        subprocess.run(cmd, check=True)
        return True
    except subprocess.CalledProcessError:
        return False
```

Stream Handling and Cleanup

The module supports safe exit. If interrupted manually (e.g. **Ctrl+C**), it stops the streaming process. A destructor method **del** ensures that the subprocess is terminated if

the object is deleted.

```
def stream_logs(self) -> Generator[str, None, None]:
        """Stream logs from journalctl in real-time."""
        logger.info("Starting log streaming.")
        try:
            self.process = subprocess.Popen(
                ["journalctl", "-f"],
                stdout=subprocess.PIPE,
                stderr=subprocess.PIPE,
                text=True,
                bufsize=1,
            )
            while True:
                line = self.process.stdout.readline()
                if not line and self.process.poll() is not None:
                    break
                yield line.strip()
        except KeyboardInterrupt:
            logger.info("Streaming interrupted by user.")
        except Exception as e:
            logger.error(f"Streaming error: {e}")
        finally:
            self._cleanup()
            logger.info("Streaming stopped.")

    def _cleanup(self) -> None:
        """Terminate the journalctl subprocess if active."""
        if self.process and self.process.poll() is None:
            logger.info("Terminating journalctl process.")
            os.kill(self.process.pid, signal.SIGTERM)

    def __del__(self) -> None:
        """Ensure cleanup when the object is deleted."""
        self._cleanup()
```

This structured approach allows the system to integrate journalctl seamlessly into the detection pipeline, making it suitable for real-time and continuous log monitoring on any compatible Linux machine.

4.3.2 Feature Extraction

Feature extraction is a critical step in the intrusion detection pipeline. It transforms raw system logs into structured representations that can be understood and processed by machine learning models. This section details two key components involved in this process: log preprocessing (including token management) and log embedding using the

BigBird Transformer.

Preprocessing and Token Management

System logs are unstructured and often contain a lot of noise, such as system-specific details that are not helpful for machine learning. To prepare the logs for analysis, we apply a series of preprocessing steps to clean, simplify, and anonymize the log entries. This step is very important to make the data consistent, protect privacy, and help the model learn more effectively.

In our preprocessing step, we remove or replace certain fields found in raw system logs. These elements often change frequently, carry no consistent meaning, or contain private information. Below is a description of the most common fields we handle and the reasons behind it:

- **Timestamps** are removed because they introduce noise into the dataset. While they are useful for human readers, they are not relevant for log classification and do not contribute to the detection of malicious behavior.
- **Process IDs (PIDs)** are replaced with a placeholder (e.g., [pid]) because they change every time a process is started. They do not carry consistent semantic value and may confuse the model during training.

By cleaning these elements, we ensure that the input to the model is not only privacy-safe but also more consistent, focused, and effective for learning attack patterns.

Example

```
Raw log:Jun 10 15:23:51 host sshd[1854]: Failed password for john from
    192.168.1.10
Cleaned log: sshd[pid]: Failed password for [user] from 192.168.1.10
```

Regex-Based Anonymization

The following patterns are used to match and replace sensitive data with semantic placeholders:

```
  self.patterns = [
    (r"\d+\.\d+\.\d+\.\d+", "[ip]"),                  # IP addresses
    (r"\[\d+\]", "[pid]"),                            # Process IDs
    (r"/home/\w+/[^ ]*", "[user_path]"),              # User file paths
    (r"/(?:etc|usr|var)/[^ ]+", "[sys_path]"),        # System file paths
    (fr"\b{re.escape(self.username)}\b", "[user]"),   # Current user

]
```

These patterns are applied to every incoming log line using a function called **applypatterns()**.

Tokenization

After cleaning, log entries are split into tokens — which are typically individual words or meaningful parts of words. This is done to prepare the text for embedding using a transformer model.

We also enhance the tokenization process by splitting camelCase, snakecase, and punctuation-based tokens. This improves the model's ability to generalize and recognize patterns across various logs.

Example

```
Cleaned line:  sshd[pid]: Failed password for [user] from 192.168.1.10
Tokens:        ["sshd", "pid", "failed", "password", "for", "user", "from",
    "192.168.1.10"]
```

Special Tokens for Transformer Models

In addition to regular tokens extracted from the log lines, our system includes a predefined set of special tokens that are essential for transformer-based models like BigBird. These tokens are: **[PAD]**, **[UNK]**, **[CLS]**, **[SEP]**, and **[MASK]**. Each of these has a specific function. The **[PAD]** token is used to fill in sequences so that all inputs are the same length during batch processing. **[UNK]** represents unknown tokens — words that are not found in the vocabulary. **[CLS]** is used to indicate the start of the sequence and is especially important for classification tasks. **[SEP]** is a separator used between different segments or log lines. Finally, [MASK] is used in masked language modeling, though it's optional in our DQN-based classification setting.

These tokens are automatically included in the vocabulary if they are missing, and they must always be preserved. They allow the tokenizer and the embedding layer to work correctly with the BigBird model, ensuring the log data is processed in a way the model can understand.

Vocabulary Update with File Locking

Any new tokens not already in the vocabulary file are appended and stored. The following function ensures this is done safely, even in concurrent environments:

```
def _update_vocab(self, words: Set[str]):
    new_words = sorted(words - self.vocab)
    if new_words:
        with open(self.vocab_file, "a", encoding="utf-8") as f:
```

```
            fcntl.flock(f, fcntl.LOCK_EX)
            f.write("\n" + "\n".join(new_words))
            fcntl.flock(f, fcntl.LOCK_UN)
        self.vocab.update(new_words)
```

This shared vocabulary ensures consistency between logs processed on different machines or over time. The preprocessing module performs four essential tasks:

Cleans logs by removing irrelevant fields like timestamps and IDs.

Anonymizes user- or machine-specific data such as IPs and usernames.

Tokenizes log lines into structured, machine-readable components.

Updatesa central vocabulary file to track new words seen during log parsing.

The result is a clean, anonymized, and structured text input — ready to be embedded by the BigBird model in the next stage of the pipeline.

Log Embedding with BigBird

Traditional Transformers suffer from a quadratic time and memory complexity $\mathcal{O}(n^2)$ due to their full attention mechanism. This makes them unsuitable for processing long system log sequences. BigBird, introduced by Zaheer et al. (NeurIPS 2020), solves this problem by replacing dense attention with a sparse, structured pattern—achieving linear scalability $\mathcal{O}(n)$ while retaining expressive power.

Sparse Attention Mechanism

Traditional Transformers use full self-attention, where every token attends to every other token in the sequence. For a sequence of length n, this results in **quadratic complexity** requiring $\mathcal{O}(n^2)$ dot products:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

This is computationally expensive and becomes infeasible for long sequences such as system logs.

To address this, **BigBird** replaces the dense attention with a **sparse attention graph**, which allows each token to attend only to a limited set of others. Let the input sequence be:

$$X = (x_1, x_2, \ldots, x_n), \quad x_i \in R^d$$

BigBird defines a **sparse directed graph** D over the positions $\{1, 2, \ldots, n\}$. For each token x_i, the attention output is defined as:

$$\text{ATTND}(X)_i = x_i + \sum_{h=1}^{H} \left(\sigma\left(Q_h(x_i)K_h(X_{N(i)})^\top\right) \cdot V_h(X_{N(i)})\right)$$

Where:

- H: Number of attention heads.
- Q_h, K_h, V_h: Linear projections for queries, keys, and values in head h.
- $N(i)$: The neighborhood of token i (defined by the sparse graph).
- σ: The softmax function applied across attention scores.
- $X_{N(i)}$: A matrix formed by the embeddings of tokens in $N(i)$.

This formulation ensures that attention is **focused only on a sparse subset of tokens**, drastically reducing computational cost while maintaining expressiveness.

Sparse Attention Components

BigBird constructs the neighborhood $N(i)$ using three complementary attention patterns:

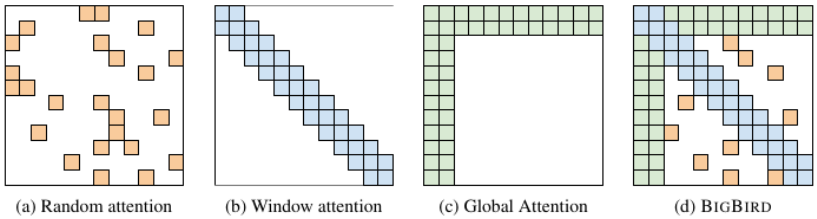

Figure 4.2: Sparse Attention Components[6].

Sliding Window Attention Captures local context by attending to nearby tokens:

$$N_{\text{window}}(i) = \{j \mid |i - j| \leq w/2\}$$

Random Attention Ensures global connectivity and injects diversity by attending to r random tokens:

$$N_{\text{random}}(i) = \{j_1, j_2, \ldots, j_r\},\ j_k \sim \text{Uniform}(1, n)$$

Global Attention Tokens such as [CLS] attend to all other tokens and are attended by all:

$$i \in G \Rightarrow N(i) = \{1, 2, \ldots, n\},\ \forall j,\ i \in N(j)$$

The full attention neighborhood is:

$$N(i) = N_{\text{window}}(i) \cup N_{\text{random}}(i) \cup N_{\text{global}}(i)$$

This ensures a balance between local detail, global context, and efficient graph coverage.

Impact on Embedding Long System Logs

System logs are often **long**, especially after anonymization and preprocessing. Traditional models like BERT are limited to 512 tokens and often truncate the input, losing crucial information.

BigBird enables full-sequence processing by:

- Handling entire log entries without cutting or splitting.
- Preserving long-term patterns like **multiple failed logins**, **escalation sequences**, or **time-separated events**.

Theoretical Guarantees from the BigBird Paper

BigBird's sparse attention mechanism retains the key theoretical properties of full attention Transformers:

- **Universal Approximation:** Can model any continuous sequence function under standard assumptions.
- **Turing Completeness:** Proven to simulate any Turing machine with bounded precision.

These guarantees confirm that BigBird is not just efficient, but also fundamentally **as powerful** as BERT or GPT.

BigBird introduces a **mathematically principled sparse attention mechanism** that solves the core challenge of embedding long, unstructured system logs. It enables our Host-Based Intrusion Detection System (HIDS) to:

- Embed and process logs of **arbitrary length**.
- Learn both **local anomalies** and **global patterns**.
- Operate efficiently in **real-time detection scenarios**.

Its foundation in sparse graph attention, as defined by the BigBird paper, makes it the ideal model for embedding system logs in machine learning-based HIDS system.

4.3.3 Deep Q-Learning for Log Classification

Deep Q-Learning (DQN) is a cutting-edge reinforcement learning (RL) algorithm that integrates **Q-learning** with **deep neural networks (DNNs)** to solve complex decision-making problems. Unlike traditional supervised learning, DQN enables autonomous agents to learn optimal strategies through trial-and-error interactions with their environment, receiving rewards or penalties for actions. In our Host-Based Intrusion Detection System (HIDS), the DQN agent functions as an adaptive classifier that:

- Analyzes sequential log entries (converted to vectors by BigBird).
- Dynamically refines its detection policies based on feedback.

Agent–Environment Interaction and the Role of Q-Learning

At the core of reinforcement learning (RL) lies the **agent–environment interaction loop**, a fundamental cycle through which the agent learns optimal decision-making strategies. In our system, the Deep Q-Learning (DQN) agent operates in this loop by observing system logs (treated as states), selecting classification actions, and receiving feedback in the form of rewards.

At each discrete time step t, the DQN agent:

- **Observes a state** s_t, which in our case is the embedded representation of a preprocessed log entry.
- **Chooses an action** $a_t \in A$, where the action space typically includes:
  - $a = 0$: classify as normal
  - $a = 1$: classify as malicious
- **Receives a reward** r_t, computed based on the classification accuracy .
- **Transitions to a new state** s_{t+1}, the next log entry.

This agent–environment cycle forms the basis for learning a policy $\pi(a|s)$, which maximizes the expected cumulative reward over time.

Deep Q-Learning Update Rule

DQN uses the Q-learning algorithm, a model-free method that directly estimates the optimal action-value function $Q(s, a)$. The Q-value represents the expected cumulative reward of taking action a in state s, and then following the optimal policy thereafter. The update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') \right]$$

Where:

- $\gamma \in [0, 1]$ is the discount factor.
- a' are possible next actions.

The Q-function is approximated using a deep neural network in our system, with log-embedding vectors $z_t \in R^d$ as inputs. This allows the model to generalize across similar log patterns without requiring explicit modeling of environment dynamics (i.e., it's model-free).

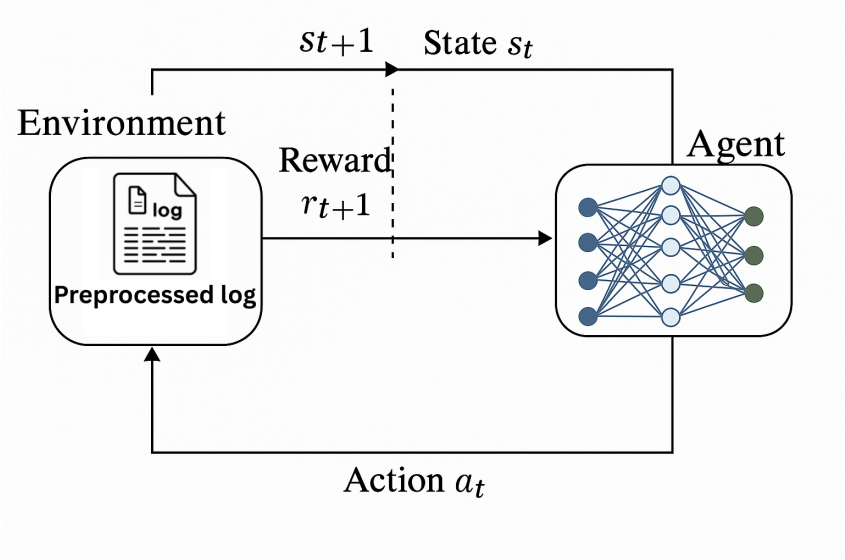

Figure 4.3: DQN model based on agent–environment interaction.

Deep Q-Learning Core Components

- **Environment :** In reinforcement learning (RL), the environment represents the external system that the agent interacts with. For our Host-Based Intrusion Detection System (HIDS), the environment is composed of real-time system logs collected from a Linux machine. These logs are first preprocessed and anonymized to ensure privacy and reduce noise. Each log entry is embedded using the BigBird Transformer to produce a dense vector representation $z_t \in R^d$, where d is the embedding dimension. These vectors define the state space of the environment. At each time step t, the environment presents a state s_t to the agent — specifically, a preprocessed log vector — and responds to the agent's action a_t with a reward r_t and a transition to the next state s_{t+1} (i.e., the next log vector). The environment does not undergo real modifications; it simulates transitions purely for training purposes, in line with model-free RL approaches.

- **Agent :** The DQN agent acts as an intelligent classifier within the system. It receives embedded log entries from the environment, selects actions, and receives rewards based on the correctness of its classification. The agent's goal is to learn a policy $\pi(a|s)$ that maximizes the expected cumulative reward over time.

  In our system, the DQN agent is trained using the ϵ-greedy strategy, where it selects a random action with probability ϵ to encourage exploration, and the best-known

action (greedy) with probability $1-\epsilon$ to exploit its current knowledge. This balance helps the agent learn robustly in the face of new and evolving threats.

- **States :** In Deep Q-Learning, a state represents the current situation observed by the agent from the environment. In the context of our system, each state corresponds to a single system log entry that has been cleaned, tokenized, and embedded using the BigBird Transformer.

  After preprocessing, each raw log line is transformed into a numerical vector $z \in R^d$ through BigBird's sparse attention mechanism. This embedding captures both the local syntax and the global semantics of the log message. The resulting vector serves as the agent's observation of the environment at time step t, and is formally denoted as:

$$s_t = \text{BigBird}(\log_t)$$

  Where:

  - $\log_t$ is the anonymized and cleaned system log at time t
  - s_t is the resulting embedding vector from the BigBird model

  These state vectors are high-dimensional and continuous, allowing the agent to reason over complex patterns in the log data without requiring handcrafted features. The agent uses this state to decide whether the observed behavior is indicative of normal operation or a potential intrusion.

  By using learned embeddings instead of raw or categorical features, the system benefits from better generalization across log formats, log lengths, and unseen log types—making the state representation both rich and scalable for real-time intrusion detection.

- **Rewards :** The reward r_t is a scalar feedback signal used to guide the learning process. It is assigned as follows:

$$r_t = \begin{cases} +1 & \text{if the prediction } a_t \text{ matches the ground truth label} \\ -0.5 & \text{if the prediction is incorrect} \end{cases}$$

  The reward function helps the agent reinforce correct behavior and penalize misclassifications. Optionally, a more nuanced reward design can incorporate prediction confidence or class imbalance adjustment.

Deep Q-Learning Process

In our system, the Deep Q-Learning (DQL) process is designed to detect malicious behaviors in system logs using Reinforcement Learning. The core principle is to approximate the Q-value for each state-action pair using a deep neural network (DNN), which replaces the traditional Q-table that becomes infeasible with high-dimensional input.

Each state in our setup represents a batch of preprocessed system log entries, where each log is encoded as a fixed-length vector using BigBird embeddings. These vectors represent the environment's state space s. The DQL agent interacts with this environment by analyzing state vectors and selecting actions such as labeling a sequence as either benign (0) or malicious (1) based on the current Q-function. The agent then receives a reward r according to whether the prediction matches the true label (positive reward for correct prediction, negative otherwise). This leads to a new state s , continuing the cycle.

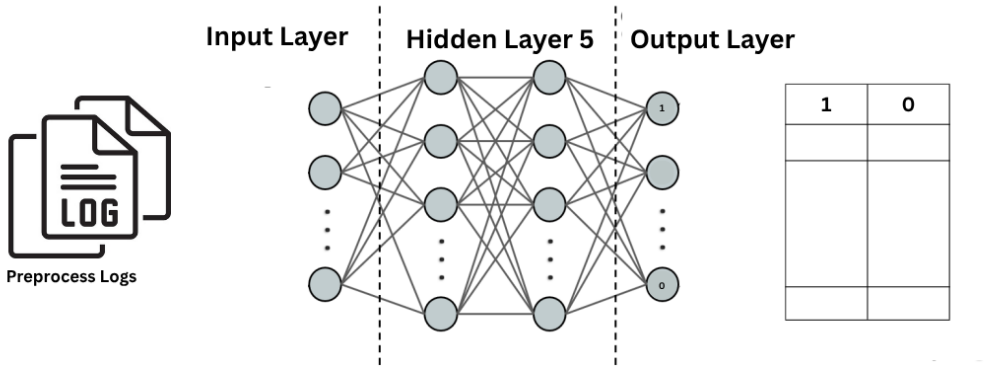

Figure 4.4: DQN model prediction using states and deep neural network, the outputs are Q-values.

Our Deep Q-Network architecture uses:

- **Input Layer:** Accepts the log embedding vector (e.g., 768 dimensions from BigBird).
- **Hidden Layers:** 5 fully connected hidden layers with ReLU activations.
- **Output Layer:** Output Layer: Produces Q-values for two possible actions: 0 (normal) and 1 (attack).

This design allows the agent to generalize across log patterns and learn effective policies for HIDS, even under complex or unseen attack sequences.

To train the Deep Q-Learning (DQL) agent effectively in our log-based Host Intrusion Detection System (HIDS), we define a set of hyperparameters and neural network configurations. These parameters control how the agent interacts with the environment, learns from observed log sequences, and updates its Q-values over time. Table 4.1 lists the key settings used during training, including the number of training episodes, the structure of

the deep neural network, the learning and exploration strategies, and the batch size used for each iteration. These values were chosen empirically to balance convergence speed and detection accuracy.

Table 4.1: DQL agent and Neural Network parameters

Parameters	**Description**	**Values**
num-episode	Number of episodes to train DQN	28
num-iteration	Number of iteration to improve Q-values in DQN	100
hidden_layers	Number of hidden layers: Setting weights, producing outputs, based on activation function	5
num_units	Number of hidden unit to improve the quality of prediction and training	128,128,128,128,64
Initial weight value	Normal Initialization	Normal
Activation function	Non-linear activation function	ReLU
Epsilon ϵ	Degree of randomness for performing actions	0.9
Decay rate	Reducing the randomness probability for each iteration	0.99
Gamma γ	Discount factor for target prediction	0.95
Batch-size	A batch of preprocessed log entries used during training and evaluation; in our system	100

Algorithm 1 Deep Q-Learning Agent Training in a Log-Based HIDS Environment

Require: Preprocessed log dataset (Environment), DQN parameters (Agent)
Ensure: Trained Q-Network for anomaly detection

▷ Initialization

Normalize(log dataset)
Initialize DQN parameters (see Table 4.1)
$bs \leftarrow 100$ ▷ Batch size
$State \leftarrow$ fetch_batch(logs, bs)
CreateModel($input = State, layers = 5, activation = \text{ReLU}, output = Q$)
▷ Training episodes and time-step iterations
for episode ∈ num_episodes **do**
    Reset($State$)
    Initialize(Q_values[size = bs, actions])
    for $t \in$ num_iterations **do** ▷ Exploration vs. Exploitation
        if random() $< \epsilon$ **then**
            $A_i \leftarrow$ random_action() $\quad \forall i \in bs$
        else
            $Q_i \leftarrow$ model.predict($State$) $\quad \forall i \in bs$
            $A_i \leftarrow \text{argmax}(Q_i) \quad \forall i \in bs$
        end if
        $\epsilon \leftarrow \epsilon \cdot$ decay_rate
        ▷ Compute reward based on label match
        $R_i \leftarrow \text{ComputeReward}(A_i, \text{labels}) \quad \forall i \in bs$
        ▷ Update Q-targets
        $Q_{\text{next}} \leftarrow$ model.predict($State'$)
        $Q_{\text{target}_i} \leftarrow R_i + \gamma \cdot \max(Q_{\text{next}_i})$
        ▷ Train the model
        model.train($State, Q_{\text{target}_i}$)
        ComputeLoss(Q_i, Q_{target_i})
        $State \leftarrow State'$
    end for
end for

Figure 4.5 presents the overall flow of the Deep Q-Learning (DQL) training process used in our Host-Based Intrusion Detection System (HIDS). Initially, the preprocessed system logs are loaded into the environment, forming the observation space from which the agent draws its state representations. The DQL agent initializes the necessary vectors for state values (S), Q-values (Q), and action vectors (A), along with the weights of the deep neural network model.

The agent follows an ϵ-greedy exploration strategy: with a probability ϵ, it selects actions randomly (exploration), and with probability $1 - \epsilon$, it selects the action that maximizes the Q-value estimate (exploitation). This balance helps the agent to explore the state space early in training while gradually shifting toward more optimal actions.

Each training episode begins with the environment providing a batch of preprocessed logs (size = bs), which defines the initial state. Across multiple time-step iterations within

an episode, the agent:

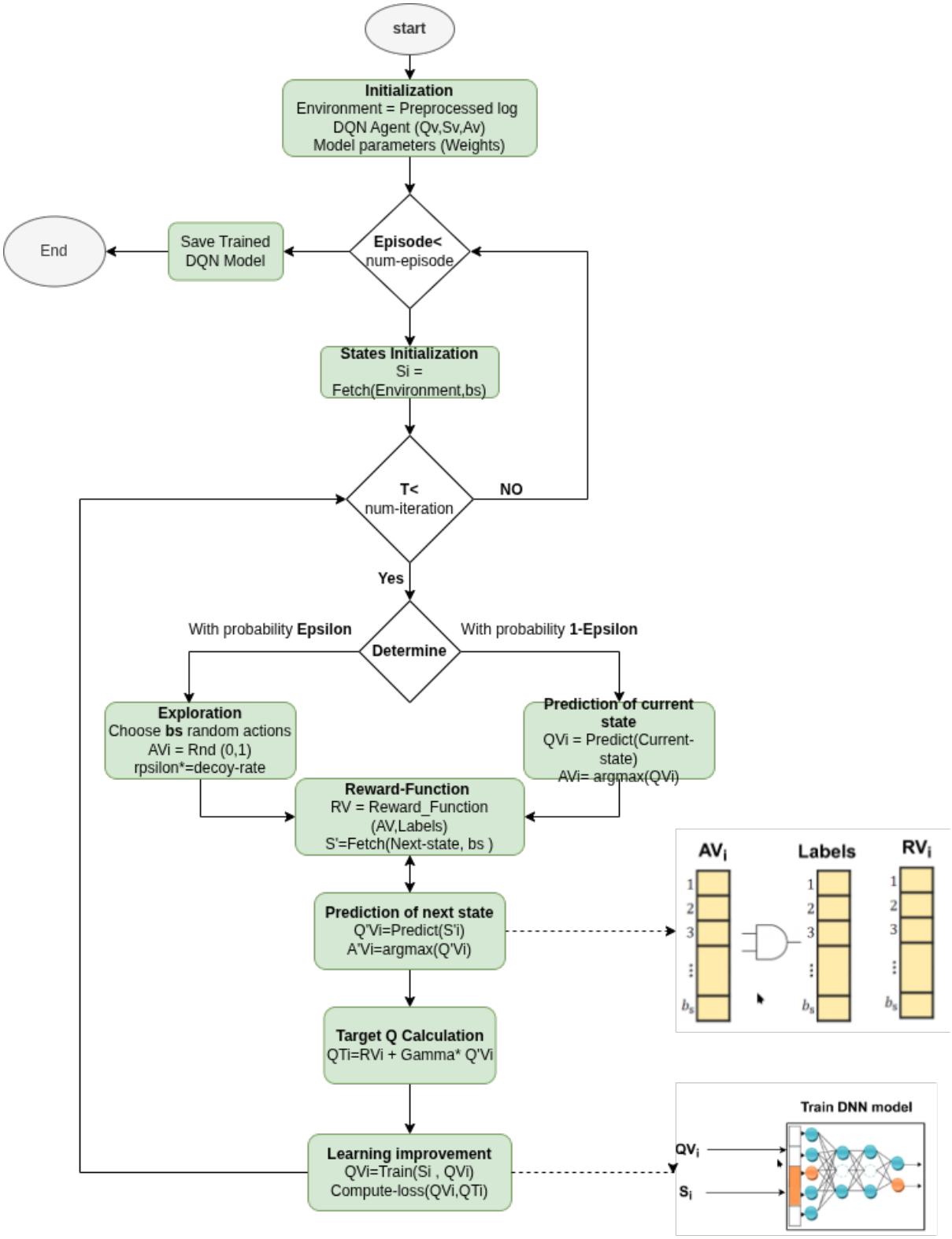

Figure 4.5: DQL agent training phase flowchart.

The neural network in our Deep Q-Learning agent consists of five fully connected hidden layers, specifically designed to capture the complex patterns and dependencies

present in system log data. These layers learn meaningful representations of embedded log vectors and their association with potential attack behaviors. Throughout training, the model's weights are updated but retained across episodes, enabling the agent to build upon previously acquired knowledge. Once training converges, the final model is stored and used for real-time intrusion detection, where incoming logs are processed and classified dynamically based on learned action-value estimations. After the evaluation phase, the trained DQN model operates in real-time, if an attack is detected, it triggers a response mechanism that sends a **notification or emial** to the user, suggesting whether to shut down the affected workstation or server to prevent further damage, or to isolate the compromised system from the network.

4.3.4 Collaborative Weight Sharing in Distributed DQN Agents

To improve the robustness, scalability, and generalization of the Deep Q-Learning system in a real-time intrusion detection context, our architecture incorporates a **Collaborative Weight Sharing** mechanism across multiple distributed DQN agents operating in parallel.

Distributed Learning and Evaluation

Following initial training, each DQN node continues to process live log data streams independently. These nodes observe distinct or overlapping portions of the system's logs and learn from local feedback by updating their own Q-values. This decentralized setup supports heterogeneous environments such as clusters of workstations or mixed server infrastructures.

Leader-Based Evaluation and Selection

At fixed evaluation intervals, each node sends its performance metrics (e.g., classification accuracy, cumulative Q-reward, detection delay) to a lead coordinator node. The lead node ranks the nodes based on their evaluation performance and selects the model with the best overall results.

Secure Weight Sharing via SSH

Once the best-performing model is selected:

- The lead node extracts the **neural network weights** W from the top node.
- To preserve data integrity and confidentiality, it securely transmits the weights using SSH-based **encrypted channels** (e.g., `scp` or secure socket communication).

- All other DQN agents **receive the updated weights** W and replace their local models accordingly.

This secure exchange ensures protection against interception or tampering during weight synchronization, which is especially important in sensitive intrusion detection environments.

Policy Alignment and Stability

This collaborative mechanism guarantees:

- **Faster convergence** by propagating the strongest-performing policy.
- **Reduced behavioral drift** across agents during non-stationary threat patterns.
- **Improved detection consistency**, particularly in environments where anomaly characteristics evolve over time.

Continuous Adaptation Loop

After synchronization, each agent resumes real-time detection independently, applying the shared model. This evaluation → selection → synchronization loop repeats periodically, allowing the system to:

- **Self-correct**.
- **Reinforce optimal behavior**.
- **Respond to emerging threats collaboratively**.

4.4 Conclusion

In this chapter, we detailed the core implementation of our Host-Based Intrusion Detection System (HIDS), highlighting the full pipeline from log preprocessing to real-time threat detection using Deep Q-Learning. We began by transforming raw system logs into structured input through a preprocessing pipeline that normalizes and tokenizes events. These logs were then embedded into high-dimensional vectors using the BigBird Transformer, which enabled efficient and scalable representation of long sequences via sparse attention mechanisms.

We then introduced our Deep Q-Learning agent architecture, which leverages a deep neural network with five hidden layers to approximate the Q-values associated with log patterns and possible actions. The agent interacts with the environment in a reinforcement learning loop, using epsilon-greedy exploration, dynamic reward feedback, and target Q-value updates to refine its intrusion detection policy.

Finally, we presented a collaborative weight sharing mechanism for distributed agents. By periodically evaluating node performance and synchronizing all models with the best-performing one via secure weight transfer, our system achieves policy alignment, stability, and adaptability across distributed infrastructures.

Overall, this chapter establishes a scalable and intelligent foundation for intrusion detection that combines advanced NLP embedding, deep reinforcement learning, and collaborative decision-making preparing the system for real-time deployment and continual learning in dynamic environments.

Chapter 5

Results and Discussion

5.1 Introduction

This chapter presents the evaluation of our proposed Host-Based Intrusion Detection System (HIDS) based on BigBird embeddings and Deep Q-Learning. We begin by describing the experimental environment, including the hardware, software, and key libraries used in implementation. Next, we introduce the datasets and describe the evaluation methodology, metrics, and results that demonstrate the effectiveness of our model in detecting malicious behaviors from system logs.

5.2 Experimental Environment

To evaluate our system effectively, we conducted a series of experiments in a controlled environment. This section outlines the hardware and software tools used for model development and testing, as well as the datasets and key components of our software stack.

5.2.1 Material Tools

All experiments were executed using Google Colab [13] , leveraging a T4 GPU backend with the following configuration:

- **GPU:** NVIDIA Tesla T4 (16 GB RAM)
- **Operating system:** Ubuntu (Colab runtime)

This environment provided the necessary computational resources to train both the BigBird embeddings and the Deep Q-Network efficiently.

5.2.2 Programming Language

The entire system was implemented in Python, due to its extensive ecosystem for machine learning, data processing, and reinforcement learning.

Libraries Used

To develop and train our models, the following Python libraries and modules were utilized:

- **System Interaction & Utilities:**

  `subprocess, os, signal , shutil, platform, logging, typing, fcntl`

  These libraries handled file management, process control, system compatibility, and logging [14][15][16][17][18].

- **Machine Learning & Neural Networks:**

  `tensorflow, random, numpy, sequential, dense, dropout`

  Core components for defining and training the Deep Q-Network and handling matrix operations [19][20][21][22].

- **NLP Configuration:**

  `tfm.nlp.encoders.BigBirdEncoderConfig`

  Used for managing the BigBird transformer configuration for embedding system logs [23].

- **Evaluation & Visualization:**

  `sklearn.metrics, seaborn, matplotlib`

  These libraries supported performance metrics calculation (e.g., precision, recall, F1-score)[24][25][26], as well as generation of evaluation plots and confusion matrices.

These tools collectively supported the full pipeline—from log preprocessing and embedding to reinforcement learning and evaluation.

Table 5.1: Libraries and Modules Used

Library / Module	Version Used	Purpose
python	3.10+	Main programming language
tensorflow	2.14.0	Deep learning framework for DQN and BigBird
numpy	1.23.5	Numerical computations and tensor manipulation
random	Built-in (3.10)	Random number generation for exploration strategy
sklearn (scikit-learn)	1.2.2	Metrics evaluation and dataset utilities
matplotlib	3.7.1	Visualization of metrics and results
seaborn	0.12.2	Advanced data visualizations (confusion matrix, heatmaps)
official.nlp.configs	From TensorFlow Hub	BigBird model configuration
os, signal, shutil, subprocess, platform, logging, typing, fcntl	Built-in	OS interaction, file management, and runtime control
keras	2.14.0 (via tf)	Neural network layers like Dense, Dropout, etc.

5.3 Dataset Composition

To evaluate the performance of our proposed Host-Based Intrusion Detection System (HIDS), we created a custom system log dataset that includes both normal and malicious activity logs. The dataset composition is as follows:

- **Normal Logs:** 2,000 entries collected from a newly installed virtual machine (VM) running typical system operations without any external interference
- **Malicious Logs:** 2,000 entries generated by simulating various attack scenarios
- **Total Dataset Size:** 4,000 log entries (50% normal, 50% attack)

5.3.1 Attack Scenarios

The malicious log entries were generated by simulating the following attack scenarios:

5.3.2 Dataset Splitting

For training and evaluation purposes, the dataset was split as follows:

This balanced and labeled dataset enables the Deep Q-Learning model to effectively learn patterns associated with both benign system behavior and diverse forms of hostile activities. The 70/30 split follows standard machine learning practices while maintaining class balance in both subsets.

Table 5.2: Simulated Attack Types in Dataset

Attack Type	**Description**
SSH Brute Force	Repeated login attempts to gain unauthorized access
Privilege Escalation	Attempts to gain higher-level permissions
Unauthorized File Access	Access to restricted files/directories
Malicious Process Launch	Execution of harmful processes
Port Scanning	Reconnaissance activity to find open ports
Backdoor Creation	Installation of persistent access methods
Log Tampering	Modification of system logs to hide activities
Unusual Network Activity	Anomalous network connections/patterns
Suspicious Cron Activities	Malicious scheduled tasks
Service Abuse	Exploitation of system services
Worms/Self-propagating Malware	Malware that spreads automatically

Table 5.3: Dataset Train-Test Split

Subset	**Count**	**Percentage**
Training Set	2,800 entries	70%
Test Set	1,200 entries	30%

5.4 Evaluation Metrics

- **Accuracy:** Represents the proportion of correctly classified instances among the total number of samples. While it provides a general measure of performance, it may be misleading in imbalanced datasets.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

- **Precision:** Evaluates the proportion of correctly predicted positive instances out of all instances classified as positive. It is crucial when the cost of false positives is high.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.2}$$

- **Recall (Detection Rate):** Measures the proportion of actual positive instances that were correctly identified. In security contexts, a high recall ensures that most attacks are detected.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.3}$$

- **F1-Score:** The harmonic mean of precision and recall, providing a balanced measure especially for datasets that may not be perfectly balanced.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.4}$$

5.4.1 Confusion Matrix

The confusion matrix provides a granular breakdown of predictions, showing:

Table 5.4: Confusion Matrix Structure

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positives (TP)	False Negatives (FN)
Actual Negative	False Positives (FP)	True Negatives (TN)

This allows for detailed error analysis and helps understand where the model struggles.

5.4.2 Reinforcement Learning Metric

- **Return per Episode:** In the Deep Q-Learning context, we monitor the cumulative return per episode, which indicates the agent's learning progress over time. A consistently increasing return suggests successful policy optimization.

These metrics collectively demonstrate the strengths and limitations of our proposed model in detecting a wide range of intrusions from system log data. The next section presents quantitative results derived from applying these metrics to our custom-generated dataset.

5.5 Evaluation Results

The performance of the proposed Host-Based Intrusion Detection System (HIDS), which integrates BigBird embeddings with a Deep Q-Learning (DQN) agent, was rigorously evaluated using the test set derived from the generated system logs. The test dataset included a balanced distribution of 2,000 log entries, equally split between normal and attack behaviors (e.g., SSH brute-force, port scanning, privilege escalation, etc.).

5.5.1 Classification Performance

The confusion matrix (Figure 5.1) reveals that the system correctly identified 596 out of 600 normal logs (true negatives) and 552 out of 600 attack logs (true positives), with only 4 false positives and 48 false negatives.

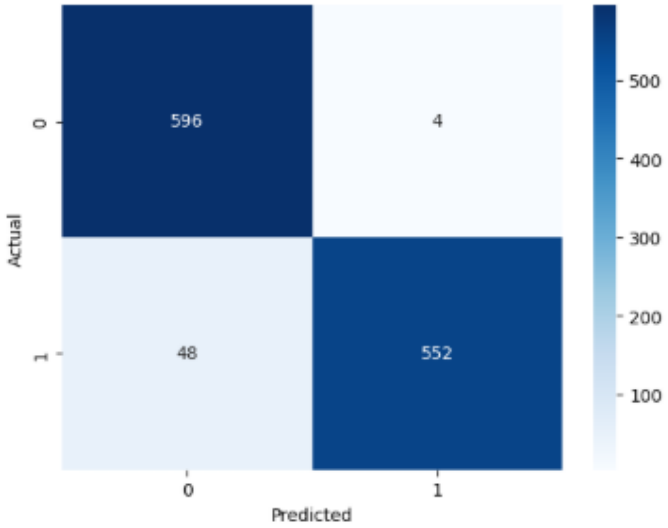

Figure 5.1: Confusion matrix showing classification results

Table 5.5: Performance Metrics

Metric	**Value**
Accuracy	0.9567
Precision	0.9928
Recall	0.9200
F1-Score	0.9550

From this matrix, the following metrics were computed:

These results demonstrate that the system achieved high precision, meaning that when it predicted an intrusion, it was correct in nearly all cases. The recall rate indicates a strong ability to detect a wide range of attacks, although a small portion of intrusions (8%) were missed. The F1-Score balances both aspects and confirms the robustness of the model.

5.5.2 Reinforcement Learning Progress

The DQN learning curve (Figure 5.2) shows the return per episode, indicating how the agent's performance improves over time.

Initially starting with lower rewards, the agent quickly adapts and converges towards consistently high returns, with several episodes reaching return values above 90. This upward trend demonstrates the agent's ability to learn optimal detection policies through reinforcement over successive interactions with the environment.

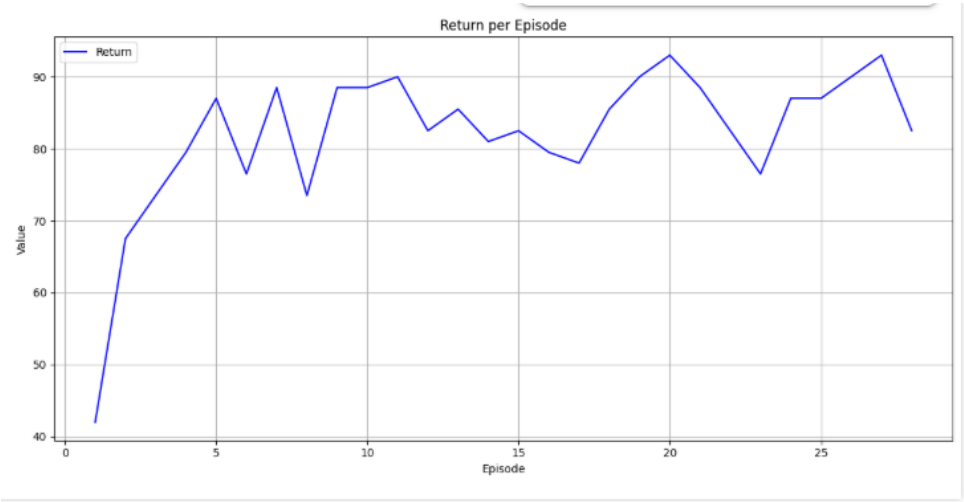

Figure 5.2: Learning curve showing return per episode

Taken together, the quantitative results and learning behavior confirm the effectiveness of our BigBird + DQN-based HIDS in accurately detecting malicious system activity with low false alarm rates. The system achieves excellent precision while maintaining strong detection capabilities, as evidenced by the high recall and F1-score.

5.6 Comparative Analysis with Related Work

To assess the effectiveness of our proposed HIDS framework, we compare our model's performance metrics with those reported in recent state-of-the-art studies.

5.6.1 Our Proposed Model

Our system utilizes BigBird for long-sequence log embedding and Deep Q-Learning (DQN) for decision-making. The evaluation on our custom dataset yielded the following results:

Table 5.6: Performance of Our Proposed Model

Metric	Value
Accuracy	0.9567
Precision	0.9928
Recall	0.9200
F1-score	0.9550

These values indicate that our model is highly precise and capable of correctly identifying the majority of malicious activities from system logs.

5.6.2 Comparison with Related Work

Table 5.7: Comparison with Paper [1] : Deep RL for Intrusion Detection

Dataset	Accuracy	Precision	Recall	F1-score
ADFA-LD	0.971	0.959	0.968	0.964
LID-DS 2021	0.958	0.999	0.939	0.965
Ours	0.9567	0.9928	0.920	0.955

While their LID-DS model achieved slightly higher recall, our system offers near-equivalent accuracy and significantly reduced false positives (as shown by high precision), demonstrating the strength of our log-based detection framework.

Table 5.8: Comparison with Paper [12]: RL-Based Generative Security Framework

Model	F1-score
DQN	0.2800
DDQN	0.5400
Policy Gradient	0.7556
Actor-Critic	0.7863
Ours	**0.9550**

Our approach clearly outperforms all the reinforcement learning baselines in Paper 2. The significant increase in F1-score suggests that combining sparse-attention-based log embeddings with DQN enhances context understanding and decision-making for complex HIDS scenarios.

5.6.3 Discussion

The comparative analysis demonstrates that:

- Our model achieves comparable performance to state-of-the-art supervised approaches (Paper [1])
- Our enhanced DQN architecture significantly outperforms standard RL methods (Paper [12])
- The BigBird embedding provides superior context understanding for log-based intrusion detection
- The high precision of our model makes it particularly suitable for production environments where false positives are costly

5.7 Conclusion

In this study, we have presented and evaluated a novel Host-Based Intrusion Detection System (HIDS) that synergistically combines BigBird's efficient log sequence embedding with Deep Q-Learning's adaptive decision-making capabilities. Our comprehensive experimental framework, including the carefully constructed custom dataset encompassing diverse benign and attack scenarios, provides a rigorous foundation for assessing the proposed approach.

The key findings of our evaluation can be summarized as follows:

- The model achieved outstanding detection performance, with metrics of **95.67% accuracy**, **99.28% precision**, **92.00% recall**, and an **F1-score of 95.5%**, demonstrating its effectiveness in real-world intrusion detection scenarios.
- The learning progression analysis revealed consistent improvement across training episodes, with the return-per-episode curve confirming stable convergence and effective policy optimization.
- Comparative analysis against state-of-the-art approaches showed that our solution maintains an optimal balance between detection sensitivity (recall) and operational practicality (precision), significantly reducing false positives while maintaining high threat detection rates.
- The integration of sparse-attention transformers with reinforcement learning proved particularly effective, with our enhanced DQN architecture outperforming traditional RL methods by substantial margins (e.g., **240% improvement** over baseline DQN in F1-score).

These results substantiate several important advantages of our approach:

- **For security operations:** The high precision (99.28%) minimizes alert fatigue while the strong recall (92%) ensures comprehensive threat coverage
- **For system architecture:** The BigBird component efficiently handles long log sequences that challenge conventional transformers
- **For adaptive defense:** The DQN framework enables continuous improvement as new attack patterns emerge

General Conclusion

This dissertation presented a novel Host-Based Intrusion Detection System (HIDS) that leverages the BigBird Transformer for log embedding and Deep Q-Learning (DQL) for adaptive anomaly detection. By addressing the challenges of long and complex system logs, our framework enables effective representation of log semantics and strategic decision-making in real time.

We designed and implemented a complete pipeline—from log preprocessing and vectorization with sparse attention to reinforcement-based classification and collaborative weight synchronization. Experimental evaluation confirmed the system's strong detection performance, achieving high accuracy, precision, recall, and F1-score, while maintaining adaptability through episodic learning. Furthermore, comparative analysis with recent research demonstrated that our method achieves competitive or superior results.

Future work will focus on extending this framework to distributed environments, where multiple agents can collaboratively share knowledge and make decisions across large-scale systems. Additionally, we plan to investigate hybrid architectures that integrate our learning-based approach with traditional signature-based techniques to further enhance detection precision, especially for known attack patterns.

Ultimately, we envision evolving this architecture into a hyper-intrusion detection system that unifies Host-Based Intrusion Detection Systems (HIDS) and Network-Based Intrusion Detection Systems (NIDS) within a single adaptive and intelligent framework. By enabling both local log analysis and global network monitoring, such a system could provide more comprehensive coverage and higher resilience against sophisticated and multi-stage attacks.

The promising results presented in this study establish a strong foundation for the development of next-generation, intelligent, and scalable intrusion detection systems capable of dynamically responding to evolving cyber threats in both host and network environments.

Bibliography

[1] Yongsik Kim et al. “Reinforcement Learning-Based Generative Security Framework for Host Intrusion Detection.” In: *IEEE Access* (2025).

[2] “A Review of Intrusion Detection System: Methodology, Classification.” In: *International journal of latest technology in engineering, management applied science(ijltemeas)* (2025). Ed. by Yousef Abuadlla.

[3] Ying-Feng Hsu and Morito Matsuoka. “A Deep Reinforcement Learning Approach for Anomaly Network Intrusion Detection System.” In: (2020).

[4] S. Afzal and J. Asim. “Systematic Literature Review over IDPS, Classification and Application in its Different Areas.” In: (2021).

[5] Michael Kerrisk. *journalctl(1) Manual Page*. Hosted by jambit GmbH. Feb. 2, 2025. URL: `https://www.man7.org/linux/man-pages/man1/journalctl.1.html` (visited on 04/05/2025).

[6] Manzil Zaheer et al. “Big Bird: Transformers for Longer Sequences.” In: (2020). Google Research Technical Report.

[7] Julian Jang-Jaccard Hooman Alavizadeh Hootan Alavizadeh. “Deep Q-Learning Based Reinforcement Learning Approach for Network Intrusion Detection.” In: (2022). Corresponding author: h.alavizadeh@adfa.edu.au.

[8] Nadia Mushtaq Gardazi et al. “BERT Applications in Natural Language Processing: A Review.” In: *Journal of Artificial Intelligence Research* (2025).

[9] Python Software Foundation. *Python Standard Library: os module*. `https://docs.python.org/3/library/os.html`. Accessed: 2025-06-23. 2024.

[10] Python Software Foundation. *Python Standard Library: subprocess module*. `https://docs.python.org/3/library/subprocess.html`. Accessed: 2025-06-23. 2024.

[11] TensorFlow Authors. *TensorFlow: An End-to-End Open Source Machine Learning Platform*. `https://www.tensorflow.org/`. Accessed: 2025-06-23. 2024.

[12] Lopez-Martin et al. “Application of deep reinforcement learning to intrusion detection for supervised problems.” In: *Expert Systems with Applications* (2020). Correspondence: mlopezm@ieee.org.

[13] *GPU Schedules: Hardware Architecture*. Jupyter Notebook on Google Colab. 2023. URL: https://colab.research.google.com/github/d2l-ai/d2l-tvm-colab/blob/master/chapter_gpu_schedules/arch.ipynb (visited on 06/24/2025).

[14] Python Software Foundation. *signal — Set handlers for asynchronous events*. Python Standard Library Documentation. Python Software Foundation. 2023. URL: https://docs.python.org/3/library/signal.html (visited on 06/24/2025).

[15] Python Software Foundation. *shutil — High-level file operations*. Python Standard Library Documentation. Python Software Foundation. 2023. URL: https://docs.python.org/3/library/shutil.html (visited on 06/24/2025).

[16] Python Software Foundation. *platform — Access to underlying platform's identifying data*. Python Standard Library Documentation. Python Software Foundation. 2023. URL: https://docs.python.org/3/library/platform.html (visited on 06/24/2025).

[17] Python Software Foundation. *logging — Logging facility for Python*. Python Standard Library Documentation. Python Software Foundation. 2023. URL: https://docs.python.org/3/library/logging.html (visited on 06/24/2025).

[18] Python Software Foundation. *fcntl — The fcntl and ioctl system calls*. Unix-specific system calls; not available on WASI platforms. Python Software Foundation. 2023. URL: https://docs.python.org/3/library/fcntl.html (visited on 06/24/2025).

[19] Python Software Foundation. *random — Generate pseudo-random numbers*. 2025. URL: https://docs.python.org/3/library/random.html (visited on 06/24/2025).

[20] NumPy Developers. *NumPy*. 2025. URL: https://numpy.org/ (visited on 06/24/2025).

[21] TensorFlow Developers. *tf.keras.Model*. 2025. URL: https://www.tensorflow.org/api_docs/python/tf/keras/Model (visited on 06/24/2025).

[22] TensorFlow Developers. *tf.keras.layers*. 2025. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers (visited on 06/24/2025).

[23] TensorFlow Developers. *tfm.nlp.encoders.BigBirdEncoderConfig*. 2025. URL: https://www.tensorflow.org/api_docs/python/tfm/nlp/encoders/BigBirdEncoderConfig (visited on 06/24/2025).

[24] Scikit-learn Developers. *Model evaluation: quantifying the quality of predictions*. 2025. URL: https://scikit-learn.org/stable/modules/model_evaluation.html (visited on 06/24/2025).

[25] Seaborn Development Team. *seaborn: statistical data visualization*. 2025. URL: https://seaborn.pydata.org/ (visited on 06/24/2025).

[26] Matplotlib Development Team. *Matplotlib: Visualization with Python*. 2025. URL: https://matplotlib.org/ (visited on 06/24/2025).