GPT-based Self-supervised Anomaly Detection in Command Lines

Miles Q. Li[1], Julien Keutchayan[2], François Charest[2], Benjamin C. M. Fung[1*]

[1*]School of Information Studies, McGill University, 3661 Peel Street, Montreal, H3A 1X1, Quebec, Canada.
[2]Direction Valorisation des données, Desjardins, 150 Sainte-Catherine Street, Montreal, H2X 3Y2, Quebec, Canada.

*Corresponding author(s). E-mail(s): ben.fung@mcgill.ca;

Abstract

Detecting anomalies in command lines is essential for system security, as malicious or unintended commands may cause severe issues like system corruption or data breaches. Traditional methods rely on predefined rules or labeled datasets, which are time-consuming to develop and struggle to adapt to evolving command patterns. This paper introduces a novel self-supervised method for detecting anomalies in command lines using a GPT model trained from scratch. Our approach focuses on restoring corrupted command lines by leveraging context from both preceding and succeeding tokens, enabling the detection of anomalies caused by incorrect, missing, or extra tokens.
Experimental results demonstrate that our method achieves a detection accuracy of 93.74%, significantly outperforming previous anomaly detection methods on the evaluated benchmark. Furthermore, the model provides insights into anomalies at the token level, helping security personnel verify issues by generating corrected versions of anomalous command lines. The proposed method offers a practical solution for continuous monitoring and improvement of system security.

Keywords: Anomaly Detection, Command Lines, GPT, Self-supervised Learning, Transformer, Cybersecurity

1 Introduction

As computer systems grow in complexity, ensuring their security and integrity becomes ever more critical [1]. Command-line interfaces (CLIs) are powerful tools frequently used for system administration and maintenance tasks [2]. However, the flexibility and capabilities of command lines also make them potential vectors for misuse, leading to catastrophic consequences such as system corruption, data leakage, or tampering [3]. Consequently, identifying anomalous or malicious command lines and preventing their execution is imperative for safeguarding computer systems.

Traditional rule-based anomaly detection methods suffer from limited recall because it is impractical to enumerate all potential misuse scenarios of command lines [4]. Supervised machine learning approaches for anomalous command line detection require large volumes of labeled data, which are both time-consuming and costly to obtain [5]. Moreover, these methods may be biased and often fail to adapt to the evolving nature of command-line usage [6]. Consequently, there is a growing interest in self-supervised methods that can detect anomalies without relying on labeled data, providing a more scalable and flexible solution.

The advent of the Transformer architecture [7] has revolutionized natural language processing and sequential data modeling. Self-supervised learning approaches using Transformers, such as GPT [8], GPT-2 [9], and RoBERTa [10], have demonstrated outstanding performance in various applications. These models learn representations from large unlabeled datasets, capturing complex patterns and dependencies in the data. Inspired by these advancements, researchers have begun exploring self-supervised learning for anomaly detection in sequential data [11, 12].

In this paper, we present a novel self-supervised approach for anomaly detection in command lines using a GPT-like model trained from scratch. Unlike existing Transformer-based approaches that rely solely on bidirectional encoders (e.g., BERT) or autoregressive decoders (e.g., GPT), our method introduces a novel context-based sequence validation task. This approach utilizes the full context—preceding and succeeding tokens—for evaluating and restoring corrupted command lines, addressing limitations in both types of existing models. Our method focuses on restoring corrupted command lines and identifying anomalies by comparing the restored command lines with the original inputs. This approach offers several benefits: it eliminates the need for labeled datasets, thereby enhancing scalability, and it enables security personnel to validate anomalies by analyzing token-level predictions against generated corrected versions of command lines.

The practical application of this approach is evident in real-world scenarios where command lines are extensively utilized. Anomalies can arise from various factors, and creating a dataset that covers all possible anomalies is impractical. However, anomalous command lines typically have a lower probability of occurrence, making them suitable candidates for anomaly detection models [13]. By collecting and analyzing both historical and new command lines, security teams can prioritize investigating command lines with the highest anomaly scores. Safe command lines identified through this process can be added to the training dataset, enabling continuous model improvement. The source code associated with this research will be made publicly available upon publication.

The main contributions of this paper are:

1. Introducing a self-supervised anomaly detection method for command lines using a GPT-like model trained from scratch, addressing critical cybersecurity needs.
2. Demonstrating the method's effectiveness in detecting both artificially corrupted and naturally occurring anomalies through comprehensive experiments.
3. Providing a framework that includes interpretability and restoration capabilities, enhancing the practical utility of the approach in real-world security contexts.
4. Proposing a dedicated token-wise probability calculation method that mitigates the cascading effects of corrupted tokens, significantly improving the accuracy of anomaly detection in command lines.

The structure of the paper is as follows: Section 2 reviews related work in anomaly detection and command line security; Section 3 details the problem formulation; Section 4 outlines our methodology, including data preparation, model training, and inference processes; Section 5 presents experimental results; and Section 6 concludes the paper and suggests future research directions.

2 Related Work

Anomaly detection in sequential data is a critical area of research with applications in cybersecurity, system administration, and fraud detection [13]. In this section, we review traditional methods for anomaly detection, deep learning approaches, and specific works related to command-line anomaly detection.

2.1 Traditional Anomaly Detection Methods

Traditional anomaly detection methods, such as Principal Component Analysis (PCA) [14], Isolation Forest (iForest) [15], and One-Class Support Vector Machines (OCSVM) [16], have been widely used for detecting anomalies in sequential data. These techniques typically rely on statistical models or distance measures to capture deviations from normal behavior. However, they often struggle to handle high-dimensional data and fail to capture the complex, dynamic patterns found in real-world sequences [4].

2.2 Deep Learning Approaches

Deep learning methods have emerged as powerful tools for anomaly detection due to their ability to model complex temporal dependencies. Recurrent Neural Networks (RNNs), particularly Long Short-Term Memory (LSTM) networks [17], have been applied to sequential anomaly detection tasks. DeepLog [18] leverages LSTMs to model normal patterns in log sequences and detect deviations as anomalies. While effective in handling sequential data, RNN-based methods are limited in capturing long-range dependencies and often suffer from vanishing gradient problems [19].

The introduction of the Transformer architecture [7] has led to significant advancements in modeling sequential data. Transformer-based models can capture long-range dependencies and parallelize computations, making them suitable for large-scale data.

LogBERT [12] applies a BERT-based model [20] to log anomaly detection by utilizing self-supervised tasks such as Masked Log Key Prediction (MLKP) and Volume of Hypersphere Minimization (VHM). This approach leverages bidirectional context to detect subtle anomalies. LogGPT [21] employs a GPT-based model [9] to predict the next token in log sequences using autoregressive modeling. It enhances anomaly detection capability through reinforcement learning with a Top-K reward mechanism, improving the model's robustness in identifying uncommon but valid sequences. Liu et al. [22] explore anomaly detection for command-line sessions using DistilBERT [23], a smaller and faster variant of BERT. Their approach demonstrates that transformer-based models can effectively detect anomalies in command sequences, highlighting the adaptability of these models to command-line specific tasks. Le et al. [24] provide an in-depth review of log parsing, log vectorization, and anomaly detection models. While their focus is on structured log data and supervised learning techniques, the insights from their work highlight the challenges and advancements in using deep learning for anomaly detection in sequential data.

2.3 Limitations of Existing Methods

While the aforementioned models have shown promising results, they have limitations when applied to command-line anomaly detection. RNN-based methods like DeepLog [18] are limited by their unidirectional context and difficulty in capturing long-range dependencies [19]. Transformer-based models like LogBERT [12] rely on bidirectional encoders that cannot generate sequences of different lengths—limiting their ability to handle missing or extra tokens and LogGPT [21] use autoregressive models that consider only the preceding context.

Moreover, existing methods often lack interpretability and restoration capabilities, which are crucial for practical applications in cybersecurity. Security personnel need to understand the nature of anomalies and have tools to restore or suggest legitimate command lines for verification.

To address these limitations, we propose a self-supervised anomaly detection method using a GPT model trained from scratch. Our approach focuses on restoring corrupted command lines and identifying anomalies through comparison with the original inputs. By leveraging a context-based sequence validation task, our model can utilize both preceding and succeeding context, effectively handling anomalies due to wrong, missing, or extra tokens. Additionally, our method provides token-level interpretability and restoration capabilities, enhancing its practical utility in real-world security contexts.

3 Problem Formulation

A command line can be represented as a sequence of characters:

$$\text{com} = \langle c_1, c_2, \ldots \rangle$$

Consider two collections of unlabeled command lines, $COM1$ and $COM2$. The command lines in $COM2$ are executed later than those in $COM1$ by the same group of users. The *self-supervised anomaly detection problem* involves building a machine learning model M based on $COM1$, such that M can be used to compute the anomaly score of commands in $COM2$.

In addition to the basic anomaly detection task, we want the model M to have interpretability and restoration capabilities.

The interpretability feature of the model M is defined as the ability to show the anomaly score of each token in the command line, allowing manual examination of the model's predictions.

The restoration capability involves generating the closest legitimate command line from an anomalous one. This feature aids security personnel in validating and understanding the issues associated with anomalies.

This setting is practical for real-world scenarios where employees in companies use command lines for their work. Anomalies can arise from various reasons, making it challenging to create a diverse enough dataset to cover all potential anomalies. However, a common characteristic of anomalous command lines is their lower probability of occurrence, which aligns with the problem setting as an anomaly detection task.

In practice, security staff can collect past command lines in $COM1$ and new command lines in $COM2$. Based on the ranking of command lines in $COM2$ by their anomaly scores, they can investigate those with the highest scores. Command lines labeled as safe can be added to $COM1$, and the model can be regularly updated. Consequently, similar command lines in the future will have lower anomaly scores.

4 Methodology

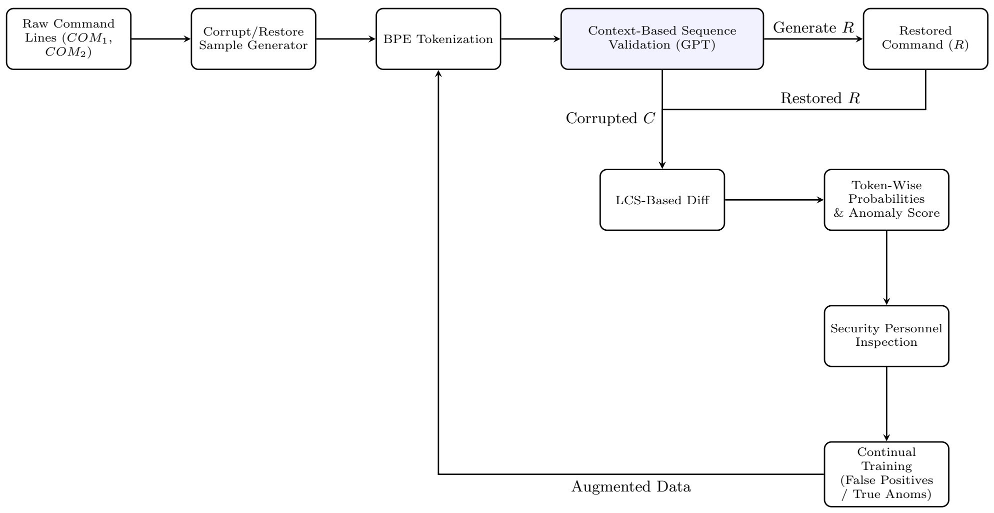

Raw Command Lines (COM_1, COM_2)
Corrupt/Restore Sample Generator
BPE Tokenization
Context-Based Sequence Validation (GPT)
Generate R
Restored Command (R)
Restored R
Corrupted C
LCS-Based Diff
Token-Wise Probabilities & Anomaly Score
Security Personnel Inspection
Continual Training (False Positives / True Anoms)
Augmented Data

Fig. 1 Framework of Context-Based Sequence Validation for Anomaly Detection in Command Lines.

Anomalous command lines are typically anomalous for three major reasons: 1) they contain wrong tokens, 2) they are missing some tokens, or 3) they have extra tokens. Command lines exhibiting these issues are less likely to occur naturally. Given the notion that anomalous command lines have a lower probability of appearing, we aim to model the probability of generating a complete command line.

4.1 Main Idea

GPT models are trained to predict the probability of the next token in a sequence given the preceding context, a technique known as next-token prediction, causal language modeling, or autoregressive language modeling. By using the probability of generating each token in a command line, we can derive an anomaly score for the entire command line.

However, a limitation of this approach in our task is that it only considers the preceding tokens when calculating the probability of generating a specific token. This is problematic because the validity of a token in a command line should depend on the context from both sides, not just the preceding tokens.

To address this issue, we need the probability of generating each token to depend on the context of the complete command. Masked language modeling is a training objective that allows Transformer encoders, such as BERT [20] and RoBERTa [10], to use context from both sides. However, encoders have the limitation that they cannot change the length of the input text. Therefore, it can handle wrong tokens but cannot handle missing tokens or extra tokens. Therefore, decoder of Transformer is a better choice. To resolve the one side context issuse with autoregressive language modelling, we propose the task ***context-based sequence validation*** to train a GPT model.

This task involves providing the model with the entire command line and asking it to validate the sequence. If the command line appears to be correct, the model generates the same sequence. If the command line looks incorrect, the model generates a legitimate version that is closest to it. This approach allows the model to utilize context from both sides of each token, ensuring a more accurate evaluation and correction of command lines. By incorporating this context-based sequence validation, our model can effectively identify and rectify anomalies, leading to improved anomaly detection in command lines.

4.2 Data Preparation

The training data consists of pairs of corrupted and original command lines. We created three types of corrupted samples: (1) missing tokens, (2) additional random tokens, and (3) substituted tokens where a random token replaces an existing one. These types represent common command-line anomalies encountered in practice. The data is formatted as follows:

$$[Beg\_Cor]\ \textit{Corrupted command}\ [End\_Cor]$$
$$[Beg\_Res]\ \textit{Restored command}\ [End\_Res]$$

$[Beg\_Cor]$, $[End\_Cor]$, $[Beg\_Res]$, and $[End\_Res]$ represent the beginning and ending tokens of the corrupted command line and restored command line. This task extends the standard causal language modeling paradigm by incorporating a restoration objective. While autoregressive decoders typically use only preceding context, our approach leverages both preceding and succeeding context by framing the task as sequence validation and restoration. To the best of our knowledge, this is the first application of such a reconstruction-based task for anomaly detection in command lines.

To handle the unique syntax and structure of command lines, we trained a Byte Pair Encoding (BPE) [25] tokenizer from scratch on the command-line dataset. This approach ensures that the tokenizer is tailored to the specific patterns and tokens present in the data, such as paths, options, and arguments. The BPE tokenizer splits command lines into subwords or tokens that capture meaningful units, enabling the model to process sequences effectively without being overwhelmed by rare or unknown tokens.

The BPE tokenizer was trained with a vocabulary size of 30,000, balancing the need for a rich vocabulary with computational efficiency. During preprocessing, all command lines were tokenized using the trained BPE tokenizer, ensuring consistency across training, validation, and inference phases. This step is crucial because it allows the model to generalize better to unseen command lines by representing their syntax and semantics in a compact and interpretable manner.

4.3 Model Training

We utilize the causal language modeling objective to train the model on the formatted data samples. The training process focuses solely on predicting the tokens in the restored part of the sequence ($Restored\ command\ [End_Res]$). To ensure this, the labels corresponding to the tokens in the corrupted portion ($[Beg\_Cor]\ Corrupted\ command\ [End\_Cor]$) are masked by setting their values to '-100' in PyTorch. This effectively instructs the model to ignore these tokens during the optimization process.

The AdamW optimizer is employed with an initial learning rate of 5×10^{-4}. We also use mixed precision training to improve efficiency, storing model weights in FP32 precision but performing forward and backward passes in FP16. Furthermore, we apply a cosine learning rate decay schedule, beginning with a warmup phase over the first 100 updates. This strategy ensures smooth training and helps the model learn to restore command lines more effectively.

4.4 Inference

For inference, given an unknown command line x which might be an anomaly, we first feed the prefix $wrap(x) = [Beg\_Cor]x[End\_Cor][Beg\_Res]$ to the model$f$, then we ask the model to complete the rest part. The ending criterion is the generation of the token $[End\_Res]$ and it is not included in the restored command line $f(wrap(x))$. Thus, we obtain the most likely legitimate command line given the input command line. If it is the same as the input command line $f(wrap(x)) = x$, then the model predicts it is not an anomaly (i.e., no anomalous tokens in it). Otherwise $f(wrap(x)) \neq x$, the output command line is compared with the input command line to identify the differences, and inspected by the security personnel to verify the decision made by the model.

Table 1 Comparison of Token Probabilities between Straightforward Method and Improved Algorithm.

Straightforward Method				Improved Algorithm			
Token	Prob	Max Token	Max Prob	Token	Prob	Max Token	Max Prob
/	99.99			/	99.99		
user	100.00			user	100.00		
/	100.00			/	100.00		
grads	99.99			grads	99.99		
/	99.99			/	99.99		
xxx	0.00	xxxxx	96.53	xxx	0.00	xxxxx	96.53
rlog	0.00	rlogin	91.36	rlog	0.00	rlogin	99.07
on	54.38			on	0.00	vaxc	99.52
vaxc	0.96	-	73.52	vaxc	99.52		
-	99.01			-	99.96		
8	99.45			8	99.97		

If $f(wrap(x)) \neq x$, we proceed by feeding the sequence $[Beg\_Cor]x[End\_Cor][Beg\_Res]x[End\_Res]$ to the model. During the generation of $[Beg\_Res]x[End\_Res]$, there may be steps where the token in x does not have the highest probability. For each such step, we record the token with the highest probability $p_{\max}$ along with its probability and the probability of the token in x (p_{origin}). We then calculate the ratio $r = \frac{p_{\text{origin}}}{p_{\max}}$. The smallest $\max(r)$ across all generation steps indicates the degree of anomaly. This anomaly score can be used to rank command lines where $f(wrap(x)) \neq x$ for further inspection. It is important to note that some of these command lines may simply exhibit patterns that are less common in the training set, rather than being truly anomalous. Therefore, an appropriate threshold for the anomaly score is necessary to determine whether a command line is genuinely anomalous. This threshold is highly context-dependent. We propose using $\sigma\theta$ as the threshold, where σ represents the standard deviation of the token probability distribution at a given position, and θ is a manually defined parameter. The use of the standard deviation accounts for positions where multiple tokens may be equally valid, allowing for a more relaxed threshold in such cases.

4.5 Continual Training

As the model is used and the results are inspected by security personnel, potentially anomalous command lines are labelled as real anomaly or false positives. Those commands can form labeled data to continue training the model.

The false positive command lines should be formulated as `[Beg_Cor]x[End_Cor] [Beg_Res]x[End_Res]` to train the model. The real anomaly command lines should be formed as `[Beg_Cor] x [End_Cor] [Beg_Res] y [End_Res]` in which y is a legitimate command line that is close to the input command line given by the security personnel. If no y is specified, then `[Beg_Cor]x[End_Cor] [Beg_Res]f(wrap(x)) [End_Res]` should be used as a training sample to reinforce the decision made by the model.

4.6 Token Level Probability Analysis

The most straightforward method to calculate the token-level probability of each token in the input command involves feeding the following sequence to the model:

$$[Beg\_Cor]\ input\ command\ [End\_Cor]$$
$$[Beg\_Res]\ input\ command\ [End\_Res]$$

In this approach, the input command line is treated as the restored command line, and the probability of each token is directly extracted from this sequence. However, this method has a significant drawback: the presence of the first anomalous token can influence the probabilities of subsequent tokens, leading to inaccurate predictions.

For example, consider the command line "/user/grads/xxxxx rlogin vaxc -8", which is a legitimate input. If we corrupt it as "/user/grads/xxx rlogon vaxc -8", the probability of each token calculated with the straightforward method is shown in the left part of Table 1. As can be seen, the corruption of "rlogin" to "rlogon" negatively impacts the probability of the token "vaxc," despite "vaxc" being a legitimate token. This issue arises because the model's context has already been disrupted by the earlier corruption, causing a cascading effect on subsequent token predictions.

This example illustrates the limitations of the straightforward method, where a single corrupted token can propagate errors through the entire sequence, leading to a decrease in the accuracy of the calculated probabilities for later tokens. As a result, it becomes challenging to accurately distinguish between legitimate and corrupted tokens based on their calculated probabilities alone. This challenge motivates the need for a more refined approach.

Our proposed algorithm takes a different approach to handling the effects of corrupted tokens on subsequent predictions. First, we tokenize the corrupted command line, then ask the model to generate the most likely restored command line. This restored command line represents the sequence that the model believes is the closest approximation of the original, uncorrupted input.

To identify discrepancies between the corrupted and restored command lines, we employ the Longest Common Subsequence (LCS) algorithm, commonly used in tools like the "git diff" command. The LCS algorithm identifies the longest sequence of tokens that appear in both the corrupted and restored command lines without rearranging their order. The difference between these sequences reveals which tokens have been modified, deleted, or inserted.

Our application of the LCS algorithm differs from traditional string-based approach by focusing on tokens rather than characters or substrings. This token-level analysis allows for a more granular understanding of the differences between the corrupted and restored command lines, especially in the context of command line syntax where even minor changes can significantly alter the meaning.

In cases where the LCS algorithm identifies tokens present in the corrupted command line but missing in the restored command line, we report the probability of these deleted tokens alongside the probability of the most likely token that could have replaced them (if any). Importantly, these deleted tokens are not reinserted into the model's input for subsequent token predictions. This omission prevents the introduction of potential errors that could arise from including corrupted or low-probability tokens in the prediction sequence, which could otherwise negatively impact the model's accuracy in predicting subsequent tokens.

Conversely, when the LCS algorithm finds tokens in the restored command line that are absent from the corrupted command line, these tokens are directly inserted into the model's input for the prediction of the following tokens. By including these high-probability tokens, we ensure that the model maintains a more accurate context for subsequent predictions, enhancing the overall reliability of the restored command line.

Finally, for tokens that appear in both the corrupted and restored command lines, we predict their probability and include them in the model's input for future token predictions. This approach allows the model to maintain consistency and leverage its understanding of the correct sequence structure.

By applying the LCS algorithm at the token level rather than the string level, we achieve a more accurate and context-aware restoration of corrupted command lines. This method effectively mitigates the issue of error propagation that can occur with straightforward probability calculations, ensuring that subsequent tokens are predicted based on the most likely and contextually appropriate sequence. This improvement is demonstrated by the more accurate token probabilities reported by our algorithm, as shown in the comparison table. Our improved method addresses the issues observed with the straightforward approach, particularly the cascading effect of an early corruption on subsequent token predictions. As demonstrated in the right part of Table 1, our method successfully isolates the influence of corrupted tokens, ensuring that legitimate tokens in the restored command line maintain their proper context and probability.

In the earlier example, the token "vaxc," which was previously affected by the corruption of "rlogin" to "rlogon," now retains a high probability in our improved method. Specifically, the probability of "vaxc" is 99.52%, as shown in the right part of the table. This result highlights that "vaxc" is no longer negatively impacted by the preceding anomalous tokens, thanks to our algorithm's careful handling of token predictions.

By not reintroducing deleted tokens into the model's input and selectively adding high-probability tokens from the restored command line, our approach preserves the integrity of the subsequent token predictions. This method prevents the propagation of errors and ensures that the model can focus on accurately predicting legitimate tokens, even when earlier tokens in the sequence are corrupted.

Overall, the results displayed in the right part of Table 1 provide strong evidence that our improved algorithm significantly enhances the accuracy of token probability calculations. By mitigating the influence of corrupted tokens on subsequent predictions, we achieve a more reliable and contextually accurate restoration of command lines, which is crucial for effective anomaly detection.

Algorithm 1 Token-wise Probability Calculation for Anomaly Detection

```
1: Input: Corrupted command line $C = \{t_1, t_2, \dots, t_n\}$, GPT-like model $M$
2: Output: Token probabilities $P = \{p_1, p_2, \dots, p_k\}$, Max tokens $M = \{m_1, m_2, \dots, m_k\}$, Max probabilities $P_{max} = \{p_{m1}, p_{m2}, \dots, p_{mk}\}$
3: Generate the most likely restored command line $R = \{r_1, r_2, \dots, r_m\}$ using model $M$
4: Apply LCS algorithm to find the sequence of tuples representing differences between $C$ and $R$
5: Initialize token probabilities $P \leftarrow \emptyset$, max tokens $M \leftarrow \emptyset$, max probabilities $P_{max} \leftarrow \emptyset$
6: for each tuple $(type, token_c, token_r)$ in LCS sequence do
7:     if $type$ = Common then
8:         Compute $p_i$ = probability of $token_c$ (or $token_r$) using model $M$
9:         Append $p_i$ to $P$
10:        Include $token_c$ (or $token_r$) in the input for predicting the next token
11:    else if $type$ = Deleted from C then
12:        Compute $p_i$ = probability of $token_c$ using model $M$
13:        Find the most likely token $m_i$ and its probability $p_{mi}$
14:        Append $p_i$ to $P$, $m_i$ to $M$, and $p_{mi}$ to $P_{max}$
15:    else if $type$ = Added from R then
16:        Include $token_r$ in the input for predicting the next token
17:    else if $type$ = Modified from C to R then
18:        Compute $p_i$ = probability of $token_c$ using model $M$
19:        Find the most likely token $m_i$ and its probability $p_{mi}$
20:        Append $p_i$ to $P$, $m_i$ to $M$, and $p_{mi}$ to $P_{max}$
21:        Include $token_r$ in the input for predicting the next token
22:    end if
23: end for
24: Return: $P$, $M$, $P_{max}$
```

5 Experiments

In this section, we present the details of our experimental setup, including the dataset, evaluation process, results, and analysis. Our goal is to validate the effectiveness of the proposed method for anomaly detection in command lines using a GPT model.

5.1 Dataset

We utilize the Greenberg Dataset [26] for our experiments. This dataset contains Unix command-line session information from 168 users divided into four groups: novice-programmers, experienced-programmers, computer-scientists, and non-programmers. The dataset comprises 303,628 lines of commands, anonymized by replacing user names with placeholders. Each entry includes:

- **S:** Start time of the login session.
- **E:** End time of the login session (may be NIL).

- **C:** Command line entered by the user.
- **D:** Current working directory.
- **A:** Alias invoked by the command line (may be NIL).
- **H:** Indicator if the line was retrieved through history (T if true, else NIL).
- **X:** Indicator of an error with a letter and number code.

We formulate each command line as the current working directory followed by the exact command line entered by the user, providing contextual information that can affect the anomaly detection process.

5.2 Data Preparation

Table 2 Performance comparison between our method and other anomaly detection approaches.

Method	Acc	Precision	Recall	F1
LogGPT	76.51	76.51	75.56	76.03
LogBERT	72.00	76.20	63.71	69.40
LSTM LM	88.84	83.46	91.91	87.48
GPT LM	83.53	80.66	88.23	84.27
One-Class SVM	55.92	54.84	67.03	60.33
Isolation Forest	46.73	46.20	39.80	42.76
Local Outlier Factor	67.92	72.50	57.75	64.29
Our Approach	**93.74**	**91.73**	**96.24**	**93.92**

To closely mirror real-world scenarios, we chronologically split the dataset. Commands from each group are sorted and then divided into a training set (90%) and an evaluation set (10%). The evaluation set is splitted into a validation set and a test set, with the former used to select the anomaly score threshold for evaluation on the test set. This approach enables the model to learn from historical data while assessing its ability to generalize to newer command lines.

During training, we generate pairs of corrupted and original command lines. Corruptions are introduced by randomly removing, inserting, or masking tokens within the command lines. The formatted data is structured as follows:

`[Beg_Cor]` Corrupted command `[End_Cor]`
`[Beg_Res]` Restored command `[End_Res]`

5.3 Evaluation Process

For evaluation, we create two versions of each sample: the original and a randomly corrupted version. The original sample is automatically labeled as normal, while the corrupted version is labeled as anomalous. We then use the model to determine whether each sample is anomalous. By applying various anomaly score thresholds, we assess the model's sensitivity and performance across different decision criteria.

It is important to note that the original samples in the Greenberg Dataset may not always represent normal behavior. Some command lines may include illegal operations, typographical errors, or commands that significantly deviate from typical usage patterns. Therefore, we manually review the original command lines flagged as anomalous by the model to verify whether they genuinely constitute anomalies.

5.4 Methods for Comparison

We evaluated the performance of our proposed method against several baseline models, including traditional anomaly detection algorithms and recent deep learning approaches. The methods we used are as follows:

Table 3 Attribution of false negatives across different thresholds. The table categorizes false negatives by the type of corruption (masked, inserted, or removed tokens) that the model failed to detect as anomalies.

Threshold	False Negatives	Masked	Insert	Remove
1e-05	1385 (23.53%)	360 (25.99%)	368 (26.57%)	657 (47.44%)
0.001	514 (8.73%)	148 (28.79%)	137 (26.65%)	229 (44.55%)
0.01	335 (5.69%)	100 (29.85%)	86 (25.67%)	149 (44.48%)
0.1	222 (3.77%)	67 (30.18%)	55 (24.77%)	100 (45.05%)
0.5	161 (2.74%)	46 (28.57%)	41 (25.47%)	74 (45.96%)
0.6	152 (2.58%)	42 (27.63%)	40 (26.32%)	70 (46.05%)
0.7	142 (2.41%)	41 (28.87%)	37 (26.06%)	64 (45.07%)
0.8	134 (2.28%)	40 (29.85%)	36 (26.87%)	58 (43.28%)
0.9	131 (2.23%)	39 (29.77%)	35 (26.72%)	57 (43.51%)
1.0	128 (2.17%)	36 (28.12%)	35 (27.34%)	57 (44.53%)
10.0	70 (1.19%)	19 (27.14%)	17 (24.29%)	34 (48.57%)
100.0	49 (0.83%)	11 (22.45%)	11 (22.45%)	27 (55.10%)

- **LogBERT** [12]: A self-supervised Transformer-based model inspired by BERT, trained from scratch on command-line sequences. LogBERT employs two training tasks: Masked Command Token Prediction (similar to masked language modeling) and a spherical objective to minimize the volume of normal command lines in the embedding space. The model learns to predict masked tokens in command lines, capturing the underlying structure of normal sequences. During evaluation, a command line is considered anomalous if the predicted tokens deviate significantly from the actual tokens. A validation set is used to determine the threshold for detecting anomalous command lines, which is then applied to the test set.
- **LogGPT** [21]: A GPT-based model specifically designed for log anomaly detection. LogGPT is pre-trained on command-line sequences to predict the next token, capturing normal patterns in the data. The model is then fine-tuned using reinforcement learning with a Top-K reward metric to enhance its ability to detect anomalies. During evaluation, a command line is flagged as anomalous if any of its tokens fall outside the Top-K predicted tokens at each position. The threshold for anomaly detection is derived from a validation set and applied to the test set.
- **One-Class SVM**: A popular method for anomaly detection that learns a decision boundary around the normal class in the feature space. We used a TF-IDF vectorizer

to extract features from the command lines and trained the One-Class SVM on the training set.

- **Isolation Forest**: An ensemble method that isolates observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. This method was also applied to the TF-IDF feature vectors.
- **Local Outlier Factor (LOF)**: A method that identifies outliers by comparing the local density of a sample to that of its neighbors. LOF was trained on the TF-IDF features extracted from the command lines.
- **RNN Language Model**: A sequential baseline using a Recurrent Neural Network (RNN) trained as a language model on the command lines. The model learns to predict the next token in a sequence, and during evaluation, the log-probabilities of the tokens in a command line are summed to obtain the overall probability for the line. The threshold for anomaly detection was determined by evaluating different thresholds on a validation set to maximize accuracy, and the best threshold was then applied to the test set. The RNN core we employ is LSTM.
- **GPT Language Model**: A Transformer-based baseline using a GPT (Generative Pretrained Transformer) architecture trained as a language model on the command lines. Similar to the RNN model, the GPT model predicts the next token in a sequence. During evaluation, the log-probabilities of the tokens in a command line are summed to derive the overall probability for the line. The threshold for anomaly detection was determined on a validation set, with the best threshold applied to the test set. This model allows for a direct comparison between Transformer-based and RNN-based sequential models for anomaly detection.

Table 4 Impact of varying corruption ratios on false negatives.

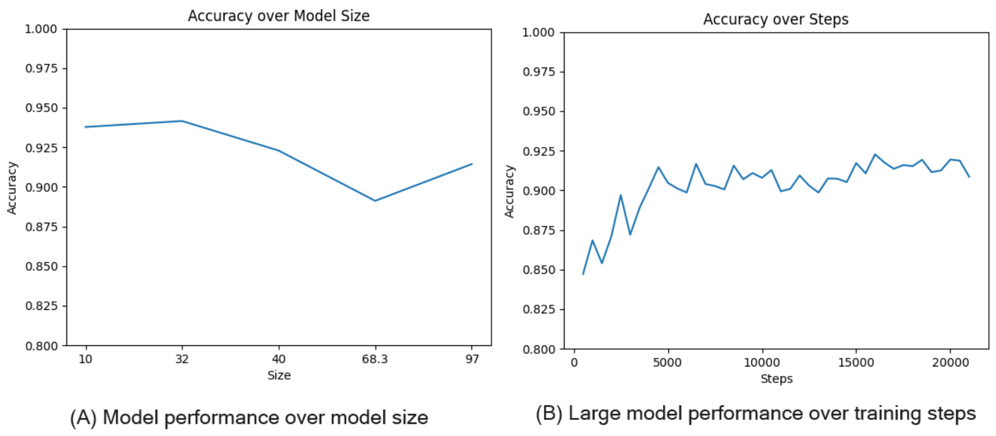
Ratio	FN	Masked	Insert	Remove
1:1:1	142	41 (28.87%)	37 (26.06%)	64 (45.07%)
1:1:3	120	32 (26.67%)	20 (16.67%)	68 (56.67%)
1:1:5	143	22 (15.38%)	12 (8.39%)	109 (76.22%)
2:2:1	140	33 (23.57%)	27 (19.29%)	80 (57.14%)

5.5 Results

Table 2 presents the performance metrics of our method compared with existing methods. Our method outperformed all baseline models, achieving an accuracy of 93.74% and an F1 score of 93.92%. In contrast, traditional methods like One-Class SVM, Isolation Forest, and LOF showed significantly lower performance, with accuracy scores below 70%. The deep learning-based baselines, including the RNN and GPT language models, performed better than traditional methods but were still outperformed by our approach.

Notably, our method also outperformed recent Transformer-based models designed for log anomaly detection, such as LogBERT and LogGPT. LogBERT achieved an

Fig. 2 (A) Model performance as a function of model size. The figure illustrates the classification accuracy at $\theta = 0.7$ across models of different sizes. (B) Training progression of the largest model (97m) over different training steps, showing accuracy improvements followed by a slight decrease, indicating overtraining.



accuracy of 72.00% and an F1 score of 69.40%, while LogGPT achieved an accuracy of 76.51% and an F1 score of 76.03%. This demonstrates the effectiveness of our context-based sequence validation task and the benefits of utilizing both preceding and succeeding context in anomaly detection.

We analyzed the causes of false negative samples by categorizing them based on the types of model failures. As presented in Table 3, the most prevalent issue is the model's inability to capture anomaly command lines that are missing tokens, as opposed to those containing unexpected tokens (either masked or inserted random tokens). The presence of random tokens is more discernible, as they are contextually inappropriate. Conversely, missing tokens are less apparent, as subsequent tokens are expected, and the missing elements might not be critical (e.g., a folder name in a path or an optional argument).

We attempted to address this issue by increasing the proportion of token-removed command lines during training (Table 4). However, increasing the ratio of token removal corruption did not improve the model's ability to detect corrupted (anomalous) command lines with missing tokens. Counterintuitively, these types of anomalies were more frequently missed. This suggests that the difficulty in detecting command lines with missing tokens cannot be easily addressed through more intensive training. In an attempt to counter this issue, we also reduced the proportion of token-removed command lines during training (as seen in the 2:2:1 ratio), but this adjustment did not yield any significant improvement.

ormance as a function of model size. The figure illustrates the classification accuracy at $\theta = 0.7$ across models of different sizes, highlighting the performance peak and subsequent decline with increased model size. (B) Training progression of the largest model (97m) over different training steps, showing accuracy improvements followed by a slight decrease, indicating over-training.

Table 5 Examples of false positives identified by the model.

Command Line	Anomaly Cause	Description
/user/profs/xxxxx ian/bin/roff -v text/**alter**	**alter** 0.00%	"alter" never appeared in the training set
/user/grads/xxxxx/651 rm *.BAK *,CKP	**,** 0.00%	Typo: "," should be "."
/user/profs/xxxxx/**art**/sculpt find. -name logfile -exec rm	**art** 0.00%	The path "/user/profs/xxxxx/art" never appeared in the training set
/user/cpsc211/l03b31/xxxxx **ddd** ffgdgjhgjjkhgkjlkjlhkjl	**ddd** 0.00%	Erroneous command line
/user/grads/xxxxx/cactus kill -9 **247**92 24793 24794 24795	**247** 0.00%	The PID to kill is statistically random
/user/profs/xxxxx/courses/211/now/notes/ work rm m17.* mar17 **M**17	**M** 0.15%	Unknown reason

5.6 Performance with Model Size

To study how well the performance of the model grows with the size of the model, we configure the model in different sizes and report their classification accuracy with $\theta = 0.7$. Figure 2 (A) shows the result. We can see that the model performance does not always grow with the model size but with a peak. We believe this is due to the facts that command lines are not as complicated as natural language and the diversity of the dataset is limited. To mitigate the possibility that the large model's lower performance may be because of insufficient training, we studied the performance of it over different training steps. Figure 2 (B) shows the performance of the large model (97m) over different training steps. We can see that the model's performance improves as it trains and then the drops slightly which indicates over-training. This confirms the fact that the lower performance of the largest model than smaller ones is not due to insufficient training.

5.7 Qualitative Evaluation

As mentioned earlier, not all samples in the dataset may be legitimate since they are collected from real-life commands used by four distinct groups of users. We conducted a qualitative evaluation of the top 100 most anomalous false positive samples identified by our model. Upon closer inspection, these samples were indeed found to be anomalous when compared with the training samples for the following reasons:

- Executed in a path that has never or rarely appeared in the training set.
- Use of arguments that have never or rarely appeared in the training set.
- Contain typos or misspellings.
- Contain random patterns that are unlikely to repeat.
- Anomalies for unknown reasons.

Table 5 presents representative examples of these false positives, illustrating how the model identifies anomalies at the token level along with the underlying causes.

This evaluation demonstrates that the original samples identified as anomalies by our model were flagged for valid reasons, validating the model's performance in these cases.

5.8 Discussion

Our experimental results demonstrate that the proposed self-supervised anomaly detection method effectively identifies anomalies in command lines, outperforming both traditional and recent deep learning-based methods. The use of context-based sequence validation and improved probability calculation enables the model to leverage full context and mitigate the effects of corrupted tokens, leading to better detection of anomalies caused by wrong, missing, or extra tokens.

While our model shows strong performance, detecting anomalies due to missing tokens remains challenging. Future work may explore incorporating additional contextual information or employing more sophisticated techniques to address this issue.

Finally, we acknowledge that our experimental analysis relies primarily on synthetically corrupted command lines of historical user sessions. While this approach proves effective for detecting structural anomalies and deviations from established usage patterns, the model's specific performance against real-world attacks, such as privilege escalation attempts, obfuscated shellcode, or destructive commands, is not yet fully known. That said, our qualitative evaluation (Table 5) revealed that the model successfully identified naturally occurring anomalies in the original dataset, including typographical errors, erroneous commands, and unusual usage patterns, partially demonstrating its capability to detect real-world deviations beyond synthetic corruptions. Future work will focus on extending the evaluation to datasets containing verified security incidents to empirically quantify the model's robustness in adversarial scenarios.

6 Conclusion

In this paper, we introduced a novel self-supervised anomaly detection method for command lines using a GPT-based restoration framework. Recognizing the critical role of command lines in system administration and the potential risks posed by anomalous commands, our approach leverages a GPT model trained from scratch to restore corrupted command lines and detect anomalies by comparing the restored outputs with the original inputs.

Our method addresses the limitations of traditional rule-based and supervised learning approaches by eliminating the need for labeled datasets and predefined rules. By utilizing both preceding and succeeding context through the context-based sequence validation task, the model effectively handles anomalies caused by incorrect, missing, or extra tokens. The dedicated token-wise probability calculation method further enhances the model's ability to identify anomalies by mitigating the cascading effects of corrupted tokens on subsequent predictions.

Experimental results demonstrated that our method significantly outperforms traditional anomaly detection algorithms and recent deep learning-based models designed for log and command-line anomaly detection on the Greenberg dataset. The model also provides token-level interpretability and restoration capabilities, enabling security personnel to understand and verify anomalies by examining specific tokens and comparing restored command lines.

Despite the strong performance, detecting anomalies due to missing tokens remains a challenge, as these anomalies often result in subtler deviations that are harder to detect. Our attempts to address this issue by adjusting the training data did not yield significant improvements, suggesting that more advanced techniques or additional contextual information may be necessary.

The proposed GPT-based restoration framework presents a practical and effective solution for anomaly detection in command lines, addressing critical cybersecurity needs in system administration. By providing a self-supervised approach with interpretability and restoration capabilities, our method enhances the ability of security personnel to maintain system integrity and prevent malicious activities. We believe that this work lays a foundation for future advancements in anomaly detection and encourages further exploration of self-supervised learning techniques in cybersecurity applications.

Acknowledgments. The research is supported in part by Desjardins, MITACS (IT28623/FR89481), Discovery Grants (RGPIN-2024-04087) from the Natural Sciences and Engineering Research Council of Canada (NSERC), and Canada Research Chairs Program (CRC-2019-00041).

References

[1] Stouffer, K., Pillitteri, V., Lightman, S., Abrams, M., Hahn, A.: Guide to industrial control systems (ics) security. Technical report, National Institute of Standards and Technology (2011)

[2] Brun, Y., Le, D., Ernst, M.: An empirical investigation into the efficiency of the command line interface. Empirical Software Engineering **25**(1), 36–65 (2020)

[3] Kiss, C., *et al.*: Software vulnerabilities: Discover, exploit, patch, and repeat. In: 2015 IEEE International Conference on Software Quality, Reliability and Security, pp. 515–520 (2015). IEEE

[4] Ahmed, M.M., Mahmood, A., Hu, J.: A survey of network anomaly detection techniques. Journal of Network and Computer Applications **60**, 19–31 (2016)

[5] Chalapathy, R., Chawla, S.: Deep learning for anomaly detection: A survey. arXiv preprint arXiv:1901.03407 (2019)

[6] Ruff, L., Vandermeulen, R.A., Görnitz, N., Deecke, L., Siddiqui, S.A., Binder, A., Müller, E., Kloft, M.: A unifying review of deep and shallow anomaly detection. Proceedings of the IEEE **109**(5), 756–795 (2021)

[7] Vaswani, A., Shazeer, N., Parmar, N., *et al.*: Attention is all you need. In: Advances in Neural Information Processing Systems, pp. 5998–6008 (2017)

[8] Radford, A., Narasimhan, K., Salimans, T., Sutskever, I.: Improving language understanding by generative pre-training. OpenAI Blog (2018)

[9] Radford, A., Wu, J., Amodei, D., *et al.*: Language models are unsupervised multitask learners. OpenAI Blog **1**(8), 9 (2019)

[10] Liu, Y., Ott, M., Goyal, N., et al.: Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692 (2019)

[11] Shen, W., Zhang, Q., Xin, R., et al.: Time-series anomaly detection service at microsoft. arXiv preprint arXiv:2009.05222 (2020)

[12] Guo, H., Yuan, S., Wu, X.: Logbert: Log anomaly detection via bert. In: 2021 International Joint Conference on Neural Networks (IJCNN), pp. 1–8 (2021). IEEE

[13] Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: A survey. ACM computing surveys (CSUR) **41**(3), 1–58 (2009)

[14] Shyu, M.-L., Chen, S.-C., Sarinnapakorn, K., Chang, L.: A novel anomaly detection scheme based on principal component classifier. IEEE Foundations and New Directions of Data Mining Workshop, 172–179 (2003)

[15] Liu, F.T., Ting, K.M., Zhou, Z.-H.: Isolation forest. In: 2008 Eighth IEEE International Conference on Data Mining, pp. 413–422 (2008). IEEE

[16] Schölkopf, B., Platt, J.C., Shawe-Taylor, J., Smola, A.J., Williamson, R.C.: Estimating the support of a high-dimensional distribution. In: Neural Computation, vol. 13, pp. 1443–1471 (2001). MIT Press

[17] Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**(8), 1735–1780 (1997)

[18] Du, M., Li, F., Zheng, G., Srikumar, V.: Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1285–1298 (2017)

[19] Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks **5**(2), 157–166 (1994)

[20] Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. NAACL-HLT **1**, 4171–4186 (2019)

[21] Han, X., Yuan, S., Trabelsi, M.: Loggpt: Log anomaly detection via gpt. In: 2023 IEEE International Conference on Big Data (BigData), pp. 1117–1122 (2023).

IEEE

[22] Liu, Z., Buford, J.: Anomaly detection of command shell sessions based on distilbert: Unsupervised and supervised approaches. arXiv preprint arXiv:2310.13247 (2023)

[23] Sanh, V., Debut, L., Chaumond, J., Wolf, T.: Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108 (2019)

[24] Le, V.-H., Zhang, H.: Log-based anomaly detection with deep learning: How far are we? In: Proceedings of the 44th International Conference on Software Engineering, pp. 1356–1367 (2022)

[25] Sennrich, R., Haddow, B., Birch, A.: Neural machine translation of rare words with subword units. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL), pp. 1715–1725 (2016)

[26] Greenberg, S.: Using unix: Collected traces of 168 users. Advanced Technologies, The Alberta Research Council, 1–13 (1988)