GlitchProber: Advancing Effective Detection and Mitigation of Glitch Tokens in Large Language Models

Zhibo Zhang*
Huazhong University of Science and Technology
Wuhan, China
zhangzhibom@hust.edu.cn

Wuxia Bai*
Huazhong University of Science and Technology
Wuhan, China
wuxiabai@hust.edu.cn

Yuxi Li*
Huazhong University of Science and Technology
Wuhan, China
yuxili@hust.edu.cn

Mark Huasong Meng
Technical University of Munich
Munich, Germany
huasong.meng@gmail.com

Kailong Wang†
Huazhong University of Science and Technology
Wuhan, China
wangkl@hust.edu.cn

Ling Shi
Nanyang Technological University
Singapore, Singapore
ling.shi@ntu.edu.sg

Li Li
Beihang University
Beijing, China
lilicoding@ieee.org

Jun Wang
Beihang University
Beijing, China
junwang.lu@gmail.com

Haoyu Wang
Huazhong University of Science and Technology
Wuhan, China
haoyuwang@hust.edu.cn

ABSTRACT

Large language models (LLMs) have achieved unprecedented success in the field of natural language processing. However, the black-box nature of their internal mechanisms has brought many concerns about their trustworthiness and interpretability. Recent research has discovered a class of abnormal tokens in the model's vocabulary space and named them "glitch tokens". Those tokens, once included in the input, may induce the model to produce incorrect, irrelevant, or even harmful results, drastically undermining the reliability and practicality of LLMs.

In this work, we aim to enhance the understanding of glitch tokens and propose techniques for their detection and mitigation. We first reveal the characteristic features induced by glitch tokens on LLMs, which are evidenced by significant deviations in the distributions of attention patterns and dynamic information from intermediate model layers. Based on the insights, we develop GlitchProber, a tool for efficient glitch token detection and mitigation. GlitchProber utilizes small-scale sampling, principal component analysis for accelerated feature extraction, and a simple classifier for efficient vocabulary screening. Taking one step further, GlitchProber rectifies abnormal model intermediate layer values to mitigate the destructive effects of glitch tokens. Evaluated on five mainstream open-source LLMs, GlitchProber demonstrates higher efficiency, precision, and recall compared to existing approaches, with an average F1 score of 0.86 and an average repair rate of 50.06%. GlitchProber unveils a novel path to address the challenges posed by glitch tokens and inspires future research toward more robust and interpretable LLMs. Our code is available at https://github.com/LLM-Integrity-Guard/GlitchProber.

CCS CONCEPTS

• **Computing methodologies** → *Knowledge representation and reasoning*.

KEYWORDS

LLM security, Glitch token, LLM analysis

ACM Reference Format:
Zhibo Zhang, Wuxia Bai, Yuxi Li, Mark Huasong Meng, Kailong Wang, Ling Shi, Li Li, Jun Wang, and Haoyu Wang. 2024. GlitchProber: Advancing Effective Detection and Mitigation of Glitch Tokens in Large Language Models. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695060

1 INTRODUCTION

In the field of Natural Language Processing (NLP), large language models (LLMs) like GPT-4 [2], Gemini [31, 34], and Claude 3 [4] have demonstrated near-human-level text generation capabilities. Their exceptional performance has led to widespread adoption [16, 35, 41]. When using these models, users provide a prompt, which the model's tokenizer breaks down into a series of discrete tokens. These tokens are the fundamental units of information processing for the model, playing a crucial role in the usage of LLMs. Recent research [12, 13, 26, 28, 29, 32], however, has shown that some "glitch tokens" exist in the vocabulary of LLMs. Once included in a prompt,

*Co-first author with equal contribution.
†Corresponding Author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695060

these special tokens can potentially lead to model errors, such as misunderstanding user intent, refusing to answer, or generating irrelevant or harmful text. Therefore, thorough analysis and detection of these glitch tokens are crucial to ensure the reliability and safety of LLMs.

To tackle the glitch tokens issue, one notable method recently is presented by Li et al. [21]. This typical and intuitive solution involves studying the characteristics of glitch tokens in the word embedding space of LLMs and accordingly developing detection techniques. They discovered that glitch tokens tend to cluster in the embedding space and proposed an iterative clustering-based technique called GlitchHunter for efficient glitch token detection.

Although there has been progress in detecting glitch tokens, there still lacks an efficient and precise detection of glitch tokens universally applicable in different LLMs. Furthermore, existing approaches primarily focus on detection, however, how to fix the issues caused by glitch tokens in the usage of LLMs remains an open question. Several limitations contribute to the aforementioned challenges:

(1) The exhaustive search method of checking vocabulary is simple and intuitive but incurs significant time costs with large token sets, making it inefficient for practical use.
(2) Existing detection methods primarily identify glitch tokens based on features like word frequency and word vectors. However, these features do not deeply explore the mechanisms by which glitch tokens impact model behaviors, resulting in poor detection accuracy and generalization performance.
(3) Current research primarily focuses on detecting glitch tokens rather than how to fix them. While detection can identify issues, it does not eliminate the negative impact of glitch tokens on model performance, limiting its practical value.

Our work. To address these existing challenges and bridge the gap, in this work, we investigate the internal structure of LLMs to explore the differences between glitch tokens and normal tokens. Specifically, through empirical study, we discovered significant differences between glitch tokens and normal tokens in terms of the attention patterns and dynamic information of multi-layer perceptron (MLP) modules within transformer-based LLMs. This discovery reveals the adversarial impact of glitch tokens on the internal mechanisms of the model, indicating that glitch tokens introduce abnormal interference and noise to neural networks. This hinders the model from correctly understanding and processing the semantic information carried by these tokens, ultimately leading to erroneous outputs.

From these findings, we gain an insight that the glitch tokens can be efficiently detected due to the deviated distributions of intermediate layers' outputs caused by them, and accordingly their impact can be effectively mitigated by proactively rectifying those abnormal outputs. Based on this insight, we propose a new method for glitch token detection and fix called GlitchProber. For glitch token detection, GlitchProber first samples a small subset of manually labeled glitch tokens as the sample set, and extracts the outputs of these tokens from the intermediate layers, specifically, the attention scores of the attention patterns and MLP status. It then applies Principal Component Analysis (PCA) [1] dimensionality reduction to the intermediate layers' outputs and obtains a feature representation matrix for the sample set. This matrix, along with the corresponding class labels, is used to train a Support Vector Machine (SVM) [9] classifier, which can subsequently be employed for glitch token detection. To fix the glitch tokens, GlitchProber analyzes the activation value range of normal tokens in the intermediate MLP status and rectifies the activation states of glitch tokens. Specifically, it aims to adjust the activation patterns of glitch tokens to be closer to those of normal tokens, and thereby minimize their impact on the model's output. Our work is published on our website [15].

Contributions. We summarize our key contributions as follows:

- **Empirical Study Exploring the Internal Impact of Glitch Tokens on LLMs.** We conduct a comprehensive and systematic empirical study on how glitch tokens and normal tokens manifest at the structural level across different LLMs. One of our key findings is that glitch tokens can trigger abnormal values in a model's attention patterns and MLP status.
- **Effective Glitch Token Detection.** Our evaluation on five representative open source LLMs demonstrates that GlitchProber can save approximately 40% of time in glitch token detection compared to the state-of-the-art approaches. Additionally, GlitchProber exhibits a significant improvement in detection accuracy.
- **Effective Glitch Token Fixing.** In terms of fix, GlitchProber successfully repairs an average of 7,758 tokens across the five LLMs. It achieves an average repair rate of 50.06%, significantly outperforming the baseline approach. Our results demonstrate the effectiveness of the proposed fix strategy by adjusting the intermediate values of glitch tokens in the intermediate layers.

Ethical Consideration. In this work, we recognize that glitch tokens can cause abnormal responses from LLMs, potentially affecting their usage. However, we strictly adhere to ethical principles and do not condone any abuse or exploitation of these findings. Our research aims to raise awareness of these risks and contribute to a more secure LLM community. We have reported our findings to the respective LLM developers and are committed to collaborating with them to develop effective defenses and mitigation strategies. By working cooperatively, we promote responsible research practices and ensure the safe and beneficial use of LLMs.

2 BACKGROUND AND RELATED WORK

2.1 Transformer-based LLMs

Self-attention [7, 25] is a core component of Transformer-based models, demonstrating strong modeling capabilities across various tasks. Given an input $X \in \mathbb{R}^{n \times d}$, where n denotes the sequence length and d denotes the dimension, self-attention linearly projects X into query, key, and value representations, i.e., Q, K, and V. The attention scores matrix A is then computed by taking the dot product between the query and key matrices, followed by a softmax normalization. The attention output is obtained by multiplying the attention scores with the value matrix.

$$A = \text{softmax}(\frac{QK^T}{\sqrt{d}}) \tag{1}$$

$$\text{Attention}(Q, K, V) = A \cdot V \tag{2}$$

To analyze the behavior of Transformer-based models during sequence processing, we introduce the concept of *attention patterns*,

which can be extracted from the corresponding row $A[n]$ of the attention scores matrix A. In autoregressive generation tasks, the attention patterns only contain the attention weights between the current token and previously generated tokens.

The MLP module in Transformer-based models employs a gating mechanism similar to that of Gated Multi-Layer Perceptrons (gMLPs) [23]. Given an input $Y \in \mathbb{R}^{n \times d}$, where n denotes the sequence length and d denotes the dimension, the MLP first projects Y to a higher-dimensional space using a linear transformation:

$$Z = YU \tag{3}$$

where $U \in \mathbb{R}^{d \times d_m}$ is a learnable weight matrix. The transformed representation Z is then split along the feature dimension into two matrices, $Z_1, Z_2 \in \mathbb{R}^{n \times d_m/2}$:

$$Z_1, Z_2 = \text{split}(Z) \tag{4}$$

An activation function σ is applied element-wise to Z_1 to obtain the *MLP gate* $\sigma(Z_1)$. The MLP gate is then multiplied element-wise with the *MLP data* Z_2 to produce the gated output $\tilde{Z}$:

$$\tilde{Z} = \sigma(Z_1) \odot Z_2 \tag{5}$$

Finally, another linear transformation is applied to map $\tilde{Z}$ back to the original dimension.

$$\text{Output} = \tilde{Z}W \tag{6}$$

where $W \in \mathbb{R}^{d_m/2 \times d}$ is another learnable weight matrix.

The MLP gate $\sigma(Z_1)$ and MLP data Z_2 in the MLP can be seen as a special variant of the spatial gating unit in Transformer-based models. These two components work together to control the information flow and capture dependencies between tokens. By extracting and analyzing the MLP gate and MLP data, we can gain insights into how the model processes and responds to different types of input within the module.

2.2 Glitch Token Phenomenon

Tokenizer plays a key role in an LLM as it transforms a continuous text sequence into a list of discrete values called tokens [39]. The tokens transformed from the training corpus form the vocabulary dictionary of LLMs, and the vocabulary dictionary in turn determines the capacity of LLMs to produce diverse and comprehensive output. The rapid advancement of LLMs has brought attention to various anomalous phenomena [10, 11, 19, 20, 22, 24, 40], one of which is the existence of "glitch tokens". These tokens exhibit anomalies in constructing the expected semantics, and are subsequently reflected in the abnormal and unexpected decoding in the LLM's output.

The glitch token phenomenon, first explored on the Lesswrong website, refers to anomalous tokens such as "SolidGoldMagikarp" and "petertodd" that cause unexpected and inaccurate results in language models like GPT-2 and GPT-J [26, 28, 29, 32]. Subsequent research examined the characteristics and instability of these tokens, revealing that even subtle changes in prompts can lead to significant differences and hallucinations in the generated results. The discovery of "polysemous" tokens, which produce different responses to repeated requests, further highlighted the prevalence and variability of the glitch token phenomenon in LLMs [27].

Recently, Li et al. [21] systematically investigated the glitch tokens with a proposed taxonomy covering their types and symptoms. They proposed three tasks, namely repetition, length and spelling, in their study to recognize glitch tokens. Their observation of the clustering distribution of glitch tokens in the word embedding spaces offers a novel perspective on the automatic identification of glitch tokens, making systematic detection feasible in LLMs containing billions, or even tens of billions of parameters.

The glitch token phenomenon uncovers the limitations and instability of LLMs when processing specific tokens. In this work, we aim to conduct a systematic and in-depth investigation of this phenomenon to gain a deeper understanding of the internal mechanisms of these models. This will provide valuable insights that can contribute to enhancing the robustness and reliability of LLMs in future applications.

3 EMPIRICAL STUDY

Our empirical study aims to explore an intuitive method for detecting glitch tokens. To this end, we investigate the differences in the model's behaviors when processing glitch tokens versus normal tokens. Two research questions are raised to guide the study:

- **RQ1 (Characteristics): What differences are exhibited between glitch tokens and normal tokens at the structural level of an LLM?**
- **RQ2 (Ubiquity): Are the differences discovered in RQ1 prevalent in most LLMs?**

To address the two RQs, we investigate the internal mechanisms of LLMs by analyzing the status of each layer in the transformer forward process of prediction. This involves examining the data flow of the intermediate layers as the model processes inputs.

3.1 Experiment Setup

To better understand the impact of glitch tokens on the model's internal output generation process, we conduct a series of experiments on the Llama-2-7b-chat model [36], shortly written as Llama2. Llama2 is a language model based on the Llama architecture. In our experiments, we set the temperature to 0 to eliminate randomness and ensure consistency in the model's responses. All other configurations are default.

We employ a unified approach to determine whether each token is glitchy or normal in this study based on existed definition of glitch tokens [12]. While the symptoms of glitch tokens may vary across different tasks, we consistently utilize a repetition task to construct input sequences for glitch token identification.

In the context of our work, a **repetitive task** refers to a specifically designed experimental procedure to test the fidelity of a language model in reproducing input tokens. This task is utilized primarily for the identification and categorization of tokens based on their performance when repetitively prompted. Specifically, the repetitive task involves the following steps:

(1) Formulating a prompt that requires the model to duplicate a specific token. The typical prompt structure is, "Can you repeat the token '{*token*}' and return it back to me?" This format is deliberately chosen to minimize contextual influence and focus purely on the token reproduction capability of the model.

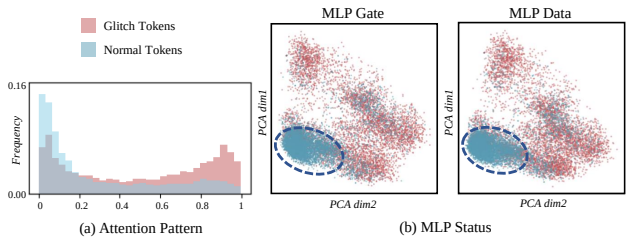


Figure 1: The distribution of attention patterns and MLP status for glitch tokens (shown in red color) and normal tokens (in blue color) in Llama2 model

(2) Submitting the prompt to the model, which is configured by setting the temperature parameter to zero.
(3) Observing and analyzing the model's output to determine whether it accurately reproduces the input token. The output is deemed successful if the model returns the exact token as requested; otherwise, the token is classified as a glitch token.

We traverse all the 32,000 tokens of the Llama2 model and eventually identify 6,425 glitch tokens from the entire vocabulary. To precisely capture intermediate layer outputs within the model, we resort to a transformer mechanistic interpretability tool named *Transformer-lens* [30]. Its hook technique enables real-time access of the activation values at all layers and allows code insertion into specific intermediate layers of the model. In this study, we insert hooks into all intermediate layers during the first forward of the tested model. This approach is chosen because the first forward comprehensively reflects the model's understanding of the input sequence and highlights the differences between normal and glitch tokens.

We select two key features to represent the model's internal output, i.e., *attention patterns* and *MLP status*. The attention patterns capture the relative importance and relationships between tokens, while MLP status is composed of MLP gate and MLP data (Section 2.1), providing insights on how the model synthesizes and modulates new representations within the MLP module.

3.2 RQ1: Glitch Token Characteristics

We compare the extracted intermediate results of Llama2 model when processing prompts containing normal tokens and glitch tokens and observe significant disparity in attention patterns and MLP status. The distribution of attention patterns for glitch tokens in some attention heads differs significantly from that of normal tokens. To quantitatively analyze these differences, we randomly sample normal tokens in size of the same number as the pre-identified glitch tokens. To analyze the attention patterns, we create two sets of prompts: one containing 6,425 prompts with glitch tokens and another containing 6,425 prompts with normal tokens. We then compute the frequency distribution of attention patterns generated by these prompts, categorizing them into different value ranges. We visualize the results using a histogram, as shown in Figure 1 (a). The attention patterns of normal tokens (shown in blue color) generally cluster around lower ranges and exhibit a relatively smooth distribution. In contrast, the attention patterns for glitch tokens (in red color) display a comparably divergent and chaotic distribution.

Furthermore, the distribution characteristics of glitch tokens in the MLP status show deviations compared to normal tokens. Due to the high dimension of MLP status values, we cannot visualize them in the same method used for attention patterns. Instead, we resort to PCA algorithm to convert the captured MLP status, i.e., MLP gate and MLP data, to two dimension values. We present the distribution of MLP gate and MLP data in scatter plots, as shown in Figure 1 (b). It can be observed that both two representations of MLP status for normal tokens tend to cluster towards a centroid, forming a relatively dense and bounded distribution. In contrast, the MLP status of glitch tokens are highly dispersed and scattered.

Finding 1

Llama2 shows a significant disparity in attention patterns and MLP status when dealing with glitch tokens and normal tokens.

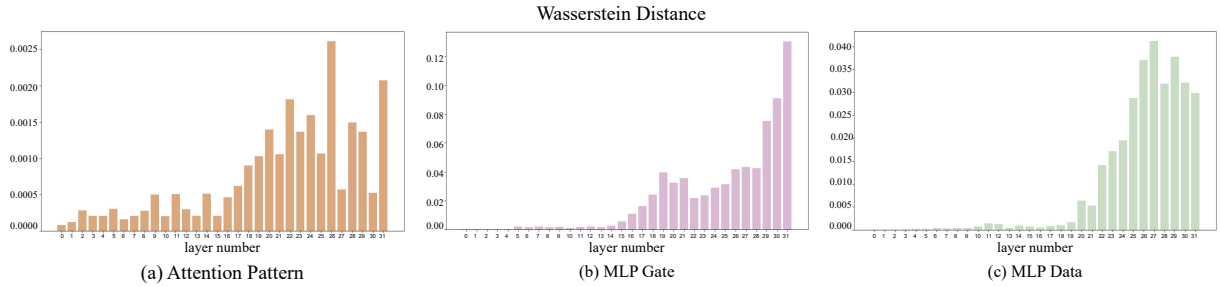
To illustrate the anomalies across layers, we resort to the Wasserstein distance [38] to measure the magnitude of the differences in the intermediate layers outputs produced by normal and glitch tokens, and thereby reveal the distinctions in the model's internal mechanisms when processing these two groups of tokens. In this study, a larger Wasserstein distance indicates a greater distributional difference. Figure 2 shows the Wasserstein distance in attention patterns and MLP status between normal and glitch tokens across different layers of the Llama2 model. We find that the differences caused by normal and glitch tokens per layer are not uniformly distributed. The attention patterns and MLP status exhibit greater differences in the downstream layers closer to the output, e.g., layers 19-31. This finding suggests that the impact of glitch tokens, although may result in negligible erroneous results in front layers, is amplified along with the propagation, leading to unexpected outputs in the end.

Finding 2

The anomalous intermediate results caused by glitch tokens are not uniformly distributed across all layers of the model but are concentrated and amplified in specific key layers.

3.3 RQ2: Ubiquity

In order to verify whether our previous findings exist in other LLMs, two additional LLMs, namely Qwen-7B-Chat model and Mistral-7B-Instruct model (shortly as Qwen and Mistral), are selected to complement our empirical study. As shown in Figure 3, the experimental results on these two models are similar to those of Llama2. The attention patterns of glitch tokens and normal tokens exhibit inconsistent distribution. For example, the attention patterns of normal tokens mainly fall within the range of $[0, 0.2]$ in the Qwen model, while the attention patterns of glitch tokens show a different shape and concentrates in the range of $[0.8, 1]$. Such inconsistency can also be observed in the Mistral model, evidenced by the attention values of normal tokens and glitch tokens primarily approximating around 0.5 and 0.7, respectively. In terms of MLP status, the intermediate layers tend to produce outputs in a centroid-based cluster for normal tokens. However, the MLP status values of glitch tokens are overall chaotic and disseminated.



(a) Attention Pattern (b) MLP Gate (c) MLP Data

Figure 2: Wasserstein distance of the probability distributions between glitch tokens and normal tokens in different intermediate layers of Llama2 model

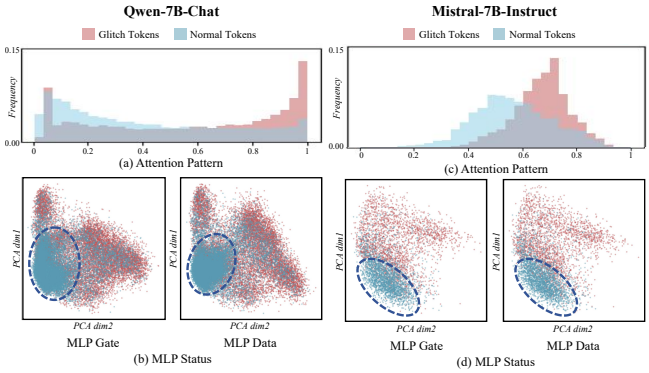


Figure 3: The example distribution of attention patterns, MLP gate and MLP data for glitch tokens (shown in red color) and normal tokens (in blue color) in Qwen-7B-Chat and Mistral-7B-Instruct.

Finding 3

We identify similar differences exhibited between normal and glitch tokens at the intermediate layers of different LLMs.

The findings of RQ1 provide insights for the subsequent detection and fix of glitch tokens, while the finding in RQ2 offers factual evidence for the broad application of our approach in LLMs.

4 METHODOLOGY

Based on the findings from our empirical study, we propose the GlitchProber algorithm, which aims to achieve automatic detection and fix of glitch tokens by analyzing the internal activation states of LLMS. Our approach consists of two main ideas:

(1) Leveraging the differences in model activation values when processing glitch tokens and normal tokens to achieve rapid screening of glitch tokens. By designing an anomaly detection algorithm, we can identify and label potential glitch tokens based on their activation features extracted from specific layers, which we refer to as key layers (detailed in Section 4.3). These key layer features are crucial for detecting glitch tokens.
(2) Fixing errors caused by glitch tokens by adjusting the model's intermediate results. We designed a series of experiments where we automatically adjusted the output values of the model's intermediate layers and observed the impact on the final output.

As elucidated in Section 3.2, glitch tokens in the model predominantly affect the downstream layers close to the output, notably impacting attention patterns and MLP status in specific key layers. This revelation is instrumental in shaping our approach of GlitchProber. By strategically focusing our efforts on key layers, we can significantly reduce computational overhead while achieving a comparable level of detection and fix efficacy to traversing all layers. Thus, this approach not only optimizes the efficiency of our methods but also ensures that our interventions are targeted where they are most needed. For a detailed introduction to the selection strategy of key layers, please refer to Section 4.3.

Based on these ideas, we designed GlitchProber, which integrates detection and fix algorithms. GlitchProber's detection algorithm identifies and locates glitch tokens causing model output errors by analyzing intermediate layer activation states. It randomly samples a subset of tokens from the vocabulary, tests them using repetitive tasks, and extracts activation features from key layers. These features undergo dimensionality reduction and are labeled based on repetitive task outcomes. A classifier is then trained with the labeled data to assess unknown tokens. Finally, the remaining tokens are detected individually using the trained SVM classifier, and predictions are verified through repetitive tasks.

GlitchProber's fix algorithm corrects the anomalous activation patterns of glitch tokens by adjusting activation values in the model's intermediate layers, eliminating their negative impact on the model output. It first calculates activation statistics of normal tokens in key layers and identifies neurons that are consistently activated or silent in most normal tokens. Then, it compares these neurons' activations between normal and glitch tokens, calculating suppression ratio coefficients for anomalous activations and activation values to promote silent neurons. Finally, it rectifies the activation values of glitch tokens in key layers based on these coefficients and values, automatically fixing the glitch tokens.

4.1 Detecting Glitch Tokens via GlitchProber

GlitchProber identifies and detects glitch tokens that cause model output errors by analyzing the intermediate layer activation states of Transformer language models when processing tokens. The main workflow of the algorithm is shown in Figure 4. The detection algorithm of GlitchProber consists of three main steps: feature extraction and dimension reduction, SVM-based glitch token classifier, and glitch token identification and validation.

4.1.1 ***Feature Extraction and Dimension Reduction.*** GlitchProber adopts a random sampling strategy to select samples from

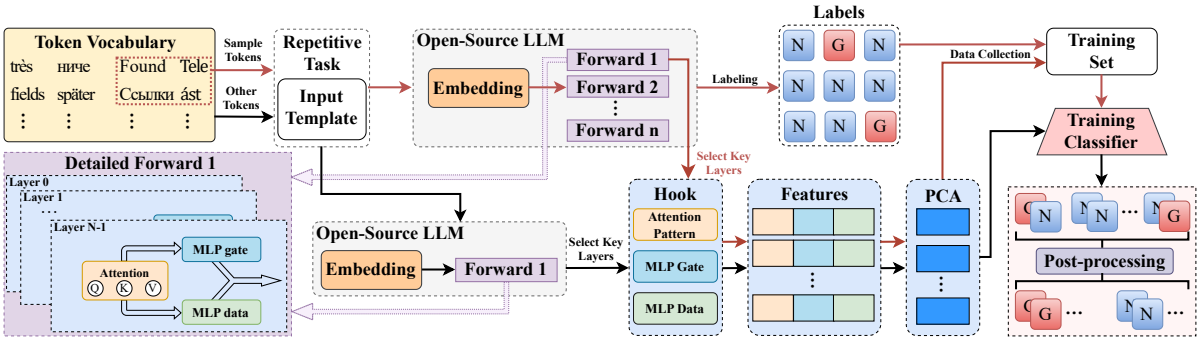


Figure 4: GlitchProber workflow for detecting glitch tokens. The red arrows represent the data flow during the training process, while the black arrows represent the data flow during the detection process.

the model's token vocabulary V to form the sample set S. The choice of sampling rate γ needs to balance between sample size and computational efficiency. A larger γ leads to a larger sample size and more accurate detection results but also incurs higher computational costs. Conversely, a smaller γ results in a smaller sample size and faster computation but may affect the detection performance. Through experiments, we determined that when γ is in the range of [0.1, 0.3], GlitchProber achieves a good balance between detection performance and efficiency.

GlitchProber uses a unified repetitive task to construct input sequences for glitch token identification in the sample set S. For tokens in the sample set S, the algorithm assigns corresponding category labels according to the output results of the repetitive task. At the same time, we extract the attention pattern, MLP gate and MLP data features in the model's first forward process. However, the dimension of the original feature tensors is high, and directly using them to train the classifier would lead to excessive computational costs. To improve computational efficiency, we adopt a dimension reduction strategy. The PCA algorithm [1] is applied to reduce the dimension, mapping the original high-dimensional features to a low-dimensional subspace while maximally preserving the discriminative information of the features. Through experiments, we found that when the dimension P of the reduced features is in the range of [50, 200], a good balance between computational efficiency and information retention can be achieved. Therefore, we set $P = 75$ as the default dimension reduction parameter.

4.1.2 ***SVM-based Glitch Token Classifier.*** GlitchProber trains an SVM classifier using the low-dimensional feature representation matrix F of the sample set S and the corresponding category labels. The trained SVM classifier will be used for subsequent glitch assessment of unknown tokens. SVM is a binary classification algorithm that is particularly suitable for problems with high-dimensional feature spaces, which aligns well with the characteristics of the glitch token detection task in GlitchProber. Besides, SVM has higher computational efficiency and shorter training time when handling high-dimensional features compared to other binary classification algorithms. That helps achieve rapid real-time detection of glitch tokens in GlitchProber. Therefore, we adopt SVM as the classifier in the detection algorithm of GlitchProber.

4.1.3 ***Glitch Token Identification and Validation.*** The tokens in the token vocabulary that were not sampled are individually detected. The token to be detected is input into the same repetitive task as in the training phase, and its features attention patterns, MLP gate, and MLP data are extracted in the model's first forward module. Subsequently, the trained SVM classifier is used to make predictions based on the extracted features.

If a token is predicted as "glitchy", the algorithm will input this token into the model and further use the repetitive task to validate it. If the model can correctly repeat the token, we consider it as a potential normal token, and the SVM classifier may have made a false positive prediction. Through this post-processing step, GlitchProber can effectively reduce the false positive rate of glitch token detection and improve the precision of detection. Finally, GlitchProber outputs two sets: the set of glitch tokens G and the set of normal tokens N.

Algorithm 1: GlitchProber (Detection)

Input: Token Vocabulary V; PCA Dimension P; Sample Rate γ; KeyLayers[]
Output: Glitch token set G; Normal token set N

$S \leftarrow$ randomSample(V, γ);
foreach *Token* $\in S$ **do**
  SampleFeatures $\leftarrow$ hookModel(Token, KeyLayers);
  SampleLabels $\leftarrow$ validateGlitch(Token);
end
$F \leftarrow$ PCA(SampleFeatures, P);
Classifier $\leftarrow$ trainClassifier(F, SampleLabels);
foreach *Token* $\in V$ **do**
  if *Token* $\notin S$ **then**
    Feature $\leftarrow$ PCA((hookModel(Token, KeyLayers)), P);
    if *classify*(*Classifier*, *Feature*) == *'Normal'* **then**
      $N \leftarrow$ Token;
    else
      if *validateGlitch*(*Token*) == *'Glitch'* **then**
        $G \leftarrow$ Token;
      else
        $N \leftarrow$ Token;
      end
    end
  end
end

4.1.4 ***Detection Process Algorithm.*** The pseudocode for the detection algorithm of GlitchProber is shown in Algorithm 1.

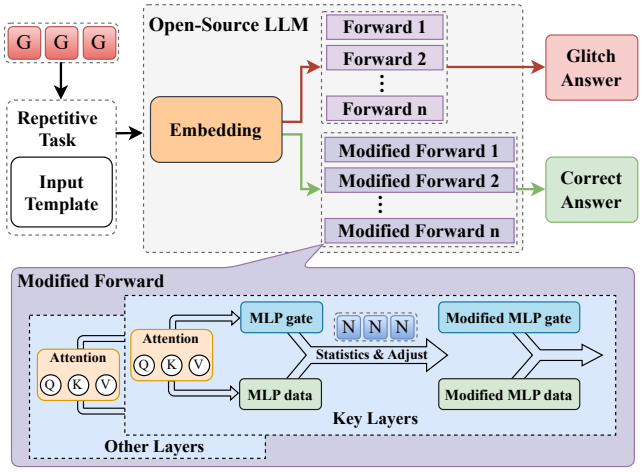


Figure 5: GlitchProber workflow for fixing glitch tokens.

Initially, the algorithm samples a subset of tokens (S) from the vocabulary (V) based on a predefined sampling rate (γ) (line 1). For each token in this subset, the algorithm extracts relevant features using a transformer model over specified key layers and assigns labels indicating whether each token is glitchy (lines 2-5).

Following the feature extraction, the algorithm applies PCA to reduce the dimension of these features to a lower-dimensional space (P) (line 6). The reduced feature set (F) is then used to train a SVM classifier with the assigned glitch labels (line 7).

For tokens not included in the initial sample, the algorithm uses the trained SVM classifier to predict whether each token is normal or a glitch (lines 8-12). Tokens classified as glitches will undergo a further validation step to confirm their status, effectively minimizing the false positive rate (lines 14-18). Finally, the algorithm compiles and outputs two sets, namely glitch token set (G) and normal token set (N).

4.2 Fixing Glitch Tokens via GlitchProber

We further explore the possibility of correcting the anomalous activation patterns of glitch tokens to normal patterns by adjusting the activation values of the model's intermediate layers, thereby eliminating their negative impact on the model output. Based on the idea of adjusting neuron activation values to eliminate the influence of glitch tokens, we focus on two types of neurons in normal tokens, i.e., the neurons that are activated in the vast majority of normal tokens, and the neurons that are not activated in any normal tokens. Then we compare the differences in activation values of these key neurons between normal tokens and glitch tokens. By simulating those normal activation patterns, we achieve adaptively adjustment of the activation values of glitch tokens. The overall approach is illustrated in Figure 5.

4.2.1 ***Normal Token Activation Value Statistics.*** GlitchProber first randomly samples a subset of normal tokens, i.e., $N' \subset N$, to calculate the activation value distribution of normal tokens in the target layers. We set the sampling rate to γ, which is consistent with the sampling rate in the glitch token detection phase.

For the MLP module in each layer, we calculate the activation statistics of the tokens in the normal token set N'. We define two sets of neuron indices as $Neun^{\uparrow}$ and $Neun^{\downarrow}$ based on their activation patterns, where $Act[i]$ represents the activation value of the i-th neuron in the MLP module, and m is the predefined threshold.

$$Neun^{\uparrow} = \{i \,|\, Act[i] > m \text{ for over 99\% of tokens in } N'\} \quad (7)$$

$$Neun^{\downarrow} = \{i \,|\, Act[i] \leq m \text{ for all tokens in } N'\} \quad (8)$$

$Neun^{\uparrow}$ denotes the set of key neurons that exhibit high activation levels, surpassing predefined threshold m, across sample token set N'. Given that activated neurons constitute a small proportion of the total neurons, we consider a neuron to be a key neuron if it is activated in over 99% of the tokens. Conversely, $Neun^{\downarrow}$ represents the set of key neurons that exhibit consistently low activation levels, falling below m. We consider a neuron to be a key neuron in $Neun^{\downarrow}$ if its activation level remains below the threshold m for all tokens in N'. These neurons can be considered as the key features for suppressing noise, as they are consistently inactive for normal tokens. By identifying these two sets of neuron indices based on their activation patterns, we can create a profile of the expected behavior of normal tokens at each MLP module. Those methods ensure the recorded neurons take the most informative features when we mitigate the influence of noise and irrelevant information caused by glitch tokens.

Note that we only adjust the activation values of the MLP module and not the attention patterns. Attention patterns capture the relative importance between tokens, and modifying them may disrupt token dependencies and introduce noise. In contrast, MLP activation values reflect the model's understanding of each token independently, and adjusting these values has less impact on token relationships. By only adjusting MLP activation values, we aim to fix glitch tokens without introducing additional noise.

4.2.2 ***Activation Value Adjustment.*** When the hooked model processes detected glitch tokens, GlitchProber intervenes in this process, making trend-based adjustments to the neurons identified by $Neun^{\uparrow}$ and $Neun^{\downarrow}$. For neurons in $Neun^{\uparrow}$ that should be activated but have insufficient activation in glitch tokens, the algorithm uses β as an amplification factor to promote their activation. Conversely, for neurons in $Neun^{\downarrow}$ that should be suppressed but are abnormally activated in glitch tokens, the algorithm uses α as a reduction factor to suppress their anomalous activation. The factors β and α are named adjustment factors, which play a crucial role in the glitch token fixing process. For the calculation of β and α, we first calculate the average activation value difference $\Delta Act^{\uparrow}$ of glitch tokens relative to normal tokens on highly activated neurons ($Neun^{\uparrow}$), and the average activation value ratio $\Delta Act^{\downarrow}$ on lowly activated neurons ($Neun^{\downarrow}$).

$$\Delta Act^{\uparrow} = \frac{1}{|Neun^{\uparrow}|} \sum_{i \in Neun^{\uparrow}} (Act_{\text{normal}}[i] - Act_{\text{glitch}}[i]) \quad (9)$$

$$\Delta Act^{\downarrow} = \frac{1}{|Neun^{\downarrow}|} \sum_{i \in Neun^{\downarrow}} \left(\frac{Act_{\text{glitch}}[i]}{Act_{\text{normal}}[i]} \right) \quad (10)$$

Subsequently, through linear transformation and range restriction, the algorithm maps $\Delta Act^{\uparrow}$ and $\Delta Act^{\downarrow}$ to appropriate numerical intervals to obtain the values of β and α, respectively.

$$\beta = k_1 \cdot \Delta Act^{\uparrow} + b_1 \tag{11}$$

$$\alpha = k_2 \cdot \Delta Act^{\downarrow} + b_2 \tag{12}$$

The constants k_1, b_1, k_2, and b_2 are derived through an adaptive process tailored to the specific dynamics of each model. A set of default values is provided, which can be adjusted based on empirical data to optimize the correction process for different types of models. They are crucial for ensuring β and α effectively modulate neuron activations while maintaining system stability and performance.

After the adjustment of the MLP activation values is completed, we input the corrected activation values back into the subsequent layers, allowing the model to continue the forward propagation until the final fixed result is output. This process corrects on each token in the detected glitch token set G.

Algorithm 2: GlitchProber (Fix)

```
Input: Glitch token set G; Normal token set N; Sample Rate γ;
       Threshold m; KeyLayers[]
1  N' ← randomSample(N, γ);
2  Neun↑, Neun↓ ← statisticsNeuron(N', m);
3  β ← statisticsBeta(N', Neun↑);
4  α ← statisticsAlpha(N', Neun↓);
5  foreach Token ∈ G do
6      for Layer ∈ KeyLayers do
7          Activation ← hookModel(Token, Layer);
8          foreach Neuron ∈ Neun↑ do
9              Act[Neuron] ← Act[Neuron] + β;
10         end
11         foreach Neuron ∈ Neun↓ do
12             Act[Neuron] ← Act[Neuron]/α;
13         end
14         hookModel(Token, Layer) ← Activation;
15     end
16 end
```

4.2.3 ***Fixing Process Algorithm.*** Based on Section 4.2.2, we present the pseudocode for the fix algorithm of GlitchProber in Algorithm 2. The algorithm begins by sampling a subset (N') of normal tokens from the full set of normal tokens (N) (line 1). Then it computes the statistical distribution of activation values across key neurons in the subset, distinguishing $Neun^{\uparrow}$ and $Neun^{\downarrow}$ (line 2). After that the algorithm calculates the adjustment factors β and α for the identified key neurons (line 3-4).

For each glitch token in the set G, the algorithm iteratively applies these adjustments across specified layers (i.e., KeyLayers) of the model (lines 5-7). Neurons in $Neun^{\uparrow}$ have their activation values increased by β, amplifying their response to mimic normal activation patterns (lines 8-10). Conversely, neurons in $Neun^{\downarrow}$ have their activation values reduced by dividing by α, suppressing any abnormal activations (lines 11-13). Each adjusted activation is reintegrated into the model's processing flow, allowing it to continue with forward propagation with the corrected values (line 14).

4.3 Key Layers Selection

In the design of our GlitchProber's detection and fix algorithms, we focused on exploiting the attention pattern and MLP status features within certain *key layers*. The rationale behind selecting these key layers stems primarily from our empirical findings outlined in Finding 2 (Section 3.2), which highlighted that glitch tokens predominantly affect the model's downstream layers closer to the output. For instance, in Llama2, this pertains to layers 19 to 31. Further refining our selection, we encountered a counter-intuitive discovery: modifying features in layers exceedingly close to the output paradoxically diminished the effectiveness of our fix algorithms.

The layers preceding the final output are crucial for tailoring responses based on preceding computations; alterations in these layers can disrupt representational balances, leading to degraded performance. In our approach for Llama2, we designated layers 19 to 28 as key layers, optimally positioned in the middle to lower sections of the model's architecture. This strategic placement ensures that our interventions effectively mitigate the effects of glitch tokens while preserving the model's robustness.

5 EVALUATION OF GLITCH TOKEN DETECTION

5.1 Experiment setup

Experiment Environment. All experiments are performed on a workstation with Ubuntu 22.04.3 LTS and 250GB memory, and 2 A100 GPU with 80GB memory each.

LLM Selection. We thoroughly evaluated our proposed method using a diverse set of computational models. We selected five widely recognized, open-source models, including Llama-2-7b-chat [36], Mistral-7B-Instruct-v0.1 [18], Qwen-7B-Chat [5], Gemma-2b-it [14], and Yi-6B-Chat [3]. These models served as the subjects for our in-depth analysis, allowing us to assess the versatility and effectiveness of our method across various real-world applications. Table 1 provides an overview of these models' parameters.

Evaluation of Detection Baselines. To evaluate the performance of GlitchProber, we compared it with two implemented benchmark schemes and a recent testing method, GlitchHunter.

(1) **Exhaustive Search:** Each token in the token list is individually fed into the model, which performs tasks such as paraphrasing, spelling, and length calculation for each token.
(2) **Rule-based Random Sampling:** First, randomly select half of the tokens from the language model to form a candidate set. Since common English words typically do not become glitch tokens, use the Natural Language Toolkit (NLTK) to remove high-frequency English words from the candidate set. The remaining tokens are considered potential glitch tokens.
(3) **GlitchHunter:** This is the state-of-the-art automated detection method for glitch tokens [21].

Evaluation Metrics of Detection. For efficiency evaluation, we consider the **Time Cost** required to process all glitch tokens in the complete token list of a model. For GlitchProber, it encompasses the total duration including feature extraction, classifier training, identification and validation. For effectiveness evaluation, we consider **True Positive**, **Precision**, **Recall** and **F1-Score**.

Evaluation Settings. In our detection experiments, we evaluated the performance of GlitchProber with SVM regularization parameter and degree[33, 37] set to $C = 1, degree = 3$. We employ $\gamma = 0.1$ for random sampling and principal components to $P = 75$

Table 1: Summary of models in evaluation

Model Name	Number of Parameters	Vocabulary Size	Hidden Layers	Intermediate Size	Attention Heads
Llama-2-7b-chat	6.74B	32,000	32	11,008	32
Mistral-7B-Instruct-v0.1	7.24B	32,000	32	14,336	32
Qwen-7B-Chat	7.72B	151,936	32	22,016	32
Gemma-2b-it	2.51B	256,000	18	16,384	8
Yi-6B-Chat	6.06B	64,000	32	11,008	32

Table 2: Time cost comparison of GlitchProber and other baselines on different LLMs.

Test Model	Exhaustive Search	GlitchHunter	GlitchProber (ours)
Llama-2-7b-chat	619min 43s	74min 11s	**61min 38s**
Mistral-7B-Instruct-v0.1	651min 17s	64min 26s	**42min 39s**
Qwen-7B-Chat	2,228min 23s	720min 42s	**92min 48s**
Gemma-2b-it	3,575min 9s	681min 16s	**96min 43s**
Yi-6B-Chat	974min 4s	825min 25s	**140min 57s**
Average Time Cost	1,609min 42s	473min 11s	**89min 9s**

for PCA. For the rule-based random sampling methods, we conducted 100 independent experiments and averaged the results to obtain statistically significant conclusions. For the GlitchHunter method, we used the default settings from the original paper [21].

5.2 RQ3 (Efficient Detection): How efficient is our approach in identifying glitch tokens across different LLMs?

To evaluate the efficiency of GlitchProber, time overhead and the accuracy comparison results of various methods on five large models are shown in Table 2 and Table 3.

Our experimental results demonstrate that GlitchProber reached a detection efficiency advantage. Meanwhile, GlitchProber achieves a 100% precision which matches the performance of both GlitchHunter and the exhaustive search benchmark method. This signifies GlitchProber's minimal false positive rate. Furthermore, GlitchProber achieves a recall rate of 64.47%, surpassing GlitchHunter's 26.52%. The F1-score indicates that GlitchProber strikes a fine balance between precision and recall, efficiently detecting glitch tokens while maintaining high accuracy.

Answer to RQ3

GlitchProber achieves perfect accuracy across all test cases with low time overhead, exhibiting superior stability and performance in glitch token detection. Its efficiency gains are primarily attributed to its strategic adoption of small-scale sampling and intermediate layer feature extraction techniques, significantly enhancing detection efficacy.

5.3 RQ4 (Ablation Study): How do the different components of GlitchProber affect the detection results?

To assess the importance of the components in GlitchProber, we performed an ablation study across five models. We developed two variants: GlitchProber-No-PCA and GlitchProber-No-Post. GlitchProber-No-PCA omits the PCA during feature processing, while GlitchProber-No-Post eliminates the final token validation steps in the original GlitchProber. The comprehensive results are shown in Table 4.

Table 3: Performance comparison of GlitchProber and other baselines on different LLMs

Test Model	Metric	Rule-based Random Sampling	GlitchHunter	GlitchProber
Llama-2-7b-chat	TP	2,936	1,955	4,446
	Precision	24.74%	100.00%	100.00%
	Recall	45.70%	30.43%	69.22%
	F1-score	0.3210	0.4724	0.8181
Mistral-7B-Instruct-v0.1	TP	1,288	1,233	1,873
	Precision	11.44%	100.00%	100.00%
	Recall	46.35%	44.37%	67.41%
	F1-score	0.1836	0.6147	0.8053
Qwen-7B-Chat	TP	15,419	4,031	19,366
	Precision	21.04%	100.00%	100.00%
	Recall	50.24%	14.42%	63.08%
	F1-score	0.2966	0.2521	0.7736
Gemma-2b-it	TP	13,777	3,240	17,387
	Precision	11.30%	100.00%	100.00%
	Recall	49.27%	10.56%	62.18%
	F1-score	0.1838	0.1910	0.7668
Yi-6B-Chat	TP	3,215	2,662	4,900
	Precision	13.10%	100.00%	100.00%
	Recall	39.67%	32.84%	60.45%
	F1-score	0.1969	0.4944	0.7535
Average Performance	Precision	16.32%	100.00%	100.00%
	Recall	46.24%	26.52%	64.47%
	F1-score	0.2364	0.4049	0.7835

Table 4: Comparison of performance and memory usage for GlitchProber with different feature configurations across various language models.

Model	Metrics	GlitchProber	GlitchProber-No-Post	GlitchProber-No-PCA
Llama-2-7b-chat	F1-Score	0.8529	0.6097	–
	Memory	103.71GB	101.22GB	250.00GB
Mistral-7B-Instruct-v0.1	F1-Score	0.8652	0.5429	–
	Memory	107.11GB	102.67GB	250.00GB
Qwen-7B-Chat	F1-Score	0.8854	0.5510	–
	Memory	109.03GB	101.20GB	250.00GB
gemma-2b-it	F1-Score	0.8143	0.4226	–
	Memory	131.04GB	127.96GB	250.00GB
Yi-6B-Chat	F1-Score	0.8718	0.4507	–
	Memory	83.32GB	82.76GB	250.00GB

Note: '–' denotes incomplete experiment due to exceeding the maximum memory of our server of 250.00GB.

Table 4 offers a detailed comparative analysis of GlitchProber alongside its variants. With respect to the F1-score, GlitchProber markedly surpasses GlitchProber-No-Post, demonstrating average improvements of 0.32. These findings underscore the vital importance of implementing robust post-classification enhancements to boost overall performance. Furthermore, the absence of PCA in GlitchProber-No-PCA leads to substantial increases in memory usage beyond the maximum capacity of our server, resulting in the incompletion of GlitchProber-No-PCA. This starkly highlights the necessity of dimensionality reduction as a means to optimize resource allocation and ensure system stability.

Answer to RQ4

The PCA dimensionality reduction and the post-process are crucial components for GlitchProber. Omitting either of these components leads to a significant decrease in effectiveness, undermining the utility of GlitchProber.

6 EVALUATION OF GLITCH TOKEN FIX

6.1 Experiment setup

Evaluation Baselines of Fix. Due to the lack of existing methods for fixing glitch tokens, we compared GlitchProber with a benchmark scheme: a rule-based fix method, to verify its effectiveness. Specifically, the rule-based fix method does not rely on specific activation value differences to determine α and β. Instead, it directly

Table 5: Performance comparison of GlitchProber and Rule-based method on different models.

Model	Metric	Method	
		Rule-based Fix	GlitchProber
Llama-2-7b-chat	Repaired Tokens	3,805	4,021
	Repair Rate	59.22%	62.58%
Mistral-7B-Instruct-v0.1	Repaired Tokens	359	1,045
	Repair Rate	12.92%	37.60%
Qwen-7B-Chat	Repaired Tokens	10,645	14,765
	Repair Rate	34.68%	48.11%
Gemma-2b-it	Repaired Tokens	9,865	13,638
	Repair Rate	35.28%	48.77%
Yi-6B-Chat	Repaired Tokens	3,390	4,317
	Repair Rate	41.83%	53.26%
Average	Repaired Tokens	5,613	7,758
	Repair Rate	36.79%	50.06%

uses a fixed value to adjust activation values at the same neuron positions as GlitchProber.

Evaluation Metrics of Fix Experiments. We used two test metrics to evaluate the performance of the fix methods: the number of repaired glitch tokens (Repaired Tokens) and the repair rate (Repair Rate). The repair rate represents the proportion of glitch tokens successfully repaired out of all glitch tokens. It is calculated using the following formula:

$$\text{Repair Rate} = \frac{\text{Repaired Token Number}}{\text{Total Glitch Token Number}} \tag{13}$$

These two metrics can intuitively reflect the actual effectiveness of the fix methods.

Evaluation Settings. In our fix experiments, we remain the same $\gamma = 0.1$ for random sampling and choose the same key layers. For the threshold, we set $m = 1$ to determine $Neun^{\uparrow}$ and $Neun^{\downarrow}$.

6.2 RQ5 (Effective Fix): How effective is our approach in fixing glitch tokens across different LLMs?

We compared the performance of the GlitchProber fix algorithm with the rule-based fix method in terms of repaired tokens and repair rate under the same conditions to evaluate the effectiveness of GlitchProber. Table 5 presents the performance comparison of the two methods across different test models.

The results indicate that although the rule-based fix method uses fixed α and β values ($\alpha = 4$ and $\beta = 1.5$) and lacks flexibility, it can still direct the activation or inhibition of neurons associated with glitch tokens, achieving a certain degree of fix. This demonstrates that adjusting neuron activation values to correct the abnormal behavior of glitch tokens is a viable and effective fix strategy. However, due to its lack of specificity, the fix effect of this method is relatively limited. In contrast, GlitchProber precisely calculates α and β, and selectively adjusts the activation patterns of key feature neurons, achieving an average repair rate of 50.06% across the five models, outperforming the rule-based method.

Answer to RQ5

GlitchProber has effectively fixed glitch tokens across five LLMs. Compared to the rule-based method, its key improvement is the precise calculation of α and β, allowing better application to different models.

Table 6: Post process time of GlitchProber using various SVM parameter configurations (in seconds)

Feature Type	C=0.1 degree=2	C=0.5 degree=2	C=0.5 degree=3	C=1 degree=3
Attn_pattern	1,413.88	1,453.21	1,360.12	1,357.70
MLP_gate	1,407.36	1,458.88	1,436.23	1,445.11
MLP_data	1,473.43	1,541.10	1,579.08	1,581.00
Attn_pattern + MLP_gate	1,410.79	1,438.57	1,444.74	1,472.50
Attn_pattern + MLP_data	1,427.43	1,479.34	1,514.72	1,513.41
MLP_gate + MLP_data	1,453.72	1,504.36	1,546.09	1,530.88
Attn_pattern + MLP_gate + MLP_data	1,444.23	1,471.85	1,502.46	1,500.23

7 DISCUSSION

7.1 Hyperparameters Choice of GlitchProber

In this section, an experiment is conducted to illustrate the selection process for the principal features in LLMs and the hyperparameters in SVM. Specifically, the parameters C and *degree* in the SVM's polynomial kernel require elucidation. The parameter C, commonly referred to as the regularization parameter, controls the trade-off between achieving a low error on the training data and minimizing the model complexity for better generalization to new data. A higher value of C tries to fit the training set as well as possible (higher model complexity), while a lower value leads to a model that might not perform as well on the training set but is better at generalizing. On the other hand, *degree* pertains to the degree of the polynomial kernel function and is crucial for defining the complexity of the decision surface. A higher *degree* results in more complex decision boundaries, capable of capturing more intricate patterns in the data. However, this also increases the risk of overfitting, particularly in scenarios with noise and limited data samples [33, 37].

As depicted in Table 6, no significant differences are observed in the time consumption across various groups of hyperparameters and features. Therefore, the average F1-score, as illustrated in Figure 6, is considered for comparison. The comparison of the F1-score without post-processing is chosen as it more accurately reflects the inherent effectiveness of the different features and hyperparameters. Figure 6 clearly shows that the hyperparameter group in the lower right corner achieves the highest F1-score of 0.6117 without post-processing, which is noteworthy. Consequently, $C = 1$ and degree = 3 are selected as the hyperparameters for SVM. All three features are chosen for the detection process, and two MLP-based features are chosen for the fix process.

We further examine the rationale behind the selection of the hyperparameter γ, which determines the sampling rate from the model's token vocabulary V for the sample set S in GlitchProber. The parameter γ is pivotal in balancing detection accuracy against computational efficiency. Empirical analysis, as shown in Table 7, demonstrates that a γ increment from 0.1 to 0.3 improves recall from 0.6922 to 0.7457 and raises the F1 score from 0.8181 to 0.8543, with a reasonable increase in computational time. However, further increasing γ to 0.7, while boosting recall and F1 scores to 0.7812 and 0.8772 respectively, significantly extends processing times to over 100 minutes. Therefore, a γ range of 0.1 to 0.3 is recommended, as it optimally balances performance gains with computational efficiency, ensuring that GlitchProber remains practical for operational use.

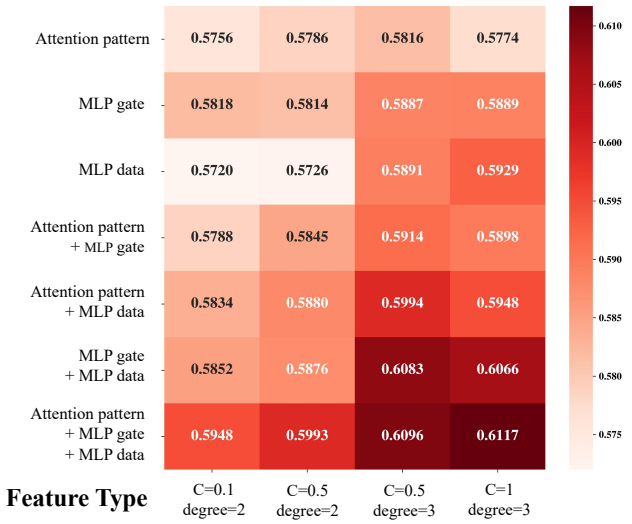


Figure 6: Different Feature Combination Comparison for GlitchProber

Table 7: Model performance across different gamma values on Llama2 model

Sampling Rate	$\gamma = 0.1$	$\gamma = 0.2$	$\gamma = 0.3$	$\gamma = 0.5$	$\gamma = 0.7$
Precision	100%	100%	100%	100%	100%
Recall	69.22%	72.81%	74.57%	76.65%	78.12%
F1 Score	81.81%	84.26%	85.43%	86.78%	87.72%
Time	61min 38s	68min 14s	74min 30s	86min 52s	100min 41s

Table 8: Performance comparison of original and modified model using Finetuning and GlitchProber for Mitigation

Dataset	Original Model	GlitchProber	Finetuning
GSM8K	0.315	0.301	0.238
HumanEval pass@1	0.129	0.103	0.009
HumanEval pass@5	0.190	0.161	0.024
MMLU	0.453	0.417	0.229

7.2 Rationale of GlitchProber versus Exhaustive Search

In the context of exhaustive search mechanisms, LLMs are required to generate, on average, **twenty** tokens at a zero temperature setting for every individual token processed. This computational method is both intensive and inefficient. Conversely, the implementation of GlitchProber necessitates merely a **single** forward processing step to capture the intermediate features of the LLM for each token. This approach substantially reduces 95% of redundant operations of those required by the traditional exhaustive search method.

7.3 Glitch Token Mitigation By Fine-tuning

GlitchProber adaptively modifies the process of model calculation without altering the model parameters, thereby minimizing the mitigation impact on model performance and preserving the basic abilities of LLMs. In contrast, we also construct a dataset with Q&A for the repetition task and attempt to mitigate the glitch token phenomenon by fine-tuning LLMs. However, compared with GlitchProber, fine-tuning LLMs alters the parameters of the model, potentially compromising its basic abilities. For example, we fine-tune the Llama-2-7b-chat with a dataset containing 3,000 Q&A pairs of repetition tasks. To evaluate the model's basic skills, we use three widely accepted datasets namely `GSM8K` [8], `HumanEval` [6], and `MMLU` [17].

Detailed results presented in Table 8 indicate that the model's ability in code writing and solving math problems post GlitchProber is comparable with the original model. and significantly better compared to the fine-tuned model, which notably diminished the basic abilities of original model.

7.4 Threats to Validity

Internal. Our primary concern involves the selection and computation of hyperparameters in both the detection and fixing phases of GlitchProber. For the detection phase, we detail the rationale behind our choices through an experiment described in Section 7.1. Various feature types and hyperparameters for the SVM consistently outperform the established baselines. In the fixing phase, the linear computation of α and β proves untenable. Enhanced discussion and manipulation of the activation value are recommended as future research directions.

External. Threats are associated with our experimental framework. For the performance of glitch tokens in intermediate layers, we experiment on three LLMs with different parameters and vocabulary sizes. Based on these findings, GlitchProber experiment on five different LLMs, showing its generalizability. Furthermore, as GlitchProber is required to access the intermediate data of the LLM, GlitchProber is only applicable to open-source LLMs. The transferability of GlitchProber from open-source LLMs to closed-source LLMs like GPT-4 could further be explored.

8 CONCLUSION

In this work, we reveal that glitch tokens trigger abnormal activation characteristics in the model's attention patterns and MLP status through systematic empirical study. Inspired by this, we propose methods for detecting and fixing glitch tokens. GlitchProber employs a sampling strategy, extracting features from attention patterns and MLP status, and achieves rapid screening of the vocabulary through PCA dimensionality reduction and SVM classification. Experiments on LLMs demonstrate that GlitchProber saves 40% of the time compared to existing methods while achieving higher accuracy. Another important contribution is our strategy to fix glitch tokens by adjusting the activation values of the model's intermediate layers. Experiments on LLMs confirm the effectiveness of this fix strategy, and the average repair rate of GlitchProber was improved by 13.27% compared with the baseline method.

ACKNOWLEDGEMENT

This work was supported by the National NSF of China (grants No.62302176, No.62072046, 62302181), the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), and the Knowledge Innovation Program of Wuhan-Basic Research (2022010801010083).

REFERENCES

[1] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2, 4 (2010), 433–459.
[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal

Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] 01. AI, :, Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Tao Yu, Wen Xie, Wenhao Huang, Xiaohui Hu, Xiaoyi Ren, Xinyao Niu, Pengcheng Nie, Yuchi Xu, Yudong Liu, Yue Wang, Yuxuan Cai, Zhenyu Gu, Zhiyuan Liu, and Zonghong Dai. 2024. Yi: Open Foundation Models by 01.AI. arXiv:2403.04652 [cs.CL]

[4] Anthropic. 2024. The Claude 3 Model Family: Opus, Sonnet, Haiku. Online. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf (Accessed: 2024-04-24).

[5] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. arXiv:2309.16609 [cs.CL]

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[7] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. 2020. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794* (2020).

[8] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168* (2021).

[9] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[10] Gelei Deng, Yi Liu, Yuekang Li, Kailong Wang, Ying Zhang, Zefeng Li, Haoyu Wang, Tianwei Zhang, and Yang Liu. 2024. MasterKey: Automated Jailbreak Across Multiple Large Language Model Chatbots. In *NDSS*.

[11] Gelei Deng, Yi Liu, Kailong Wang, Yuekang Li, Tianwei Zhang, and Yang Liu. 2024. Pandora: Jailbreak GPTs by Retrieval Augmented Generation Poisoning. *NDSS AISCC* (2024).

[12] Martin Fell. 2023. A Search for More ChatGPT / GPT-3.5 / GPT-4 "Unspeakable" Glitch Tokens. Online. https://www.lesswrong.com/posts/kmWrwtGE9B9hpbgRT/a-search-for-more-chatgpt-gpt-3-5-gpt-4-unspeakable-glitch (Accessed: 2024-05-05).

[13] Jonas Geiping, Alex Stein, Manli Shu, Khalid Saifullah, Yuxin Wen, and Tom Goldstein. 2024. Coercing LLMs to do and reveal (almost) anything. *arXiv preprint arXiv:2402.14020* (2024).

[14] Thomas Mesnard Gemma Team, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, and et al. 2024. Gemma. (2024). https://doi.org/10.34740/KAGGLE/M/3301

[15] GlitchProber. (Accessed on 06/07/2024). https://sites.google.com/view/glitchprober/.

[16] Hao Guan, Guangdong Bai, and Yepang Liu. 2024. Large Language Models Can Connect the Dots: Exploring Model Optimization Bugs with Domain Knowledge-Aware Prompts. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1579–1591. https://doi.org/10.1145/3650212.3680383

[17] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300* (2020).

[18] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL]

[19] Haodong Li, Gelei Deng, Yi Liu, Kailong Wang, Yuekang Li, Tianwei Zhang, Yang Liu, Guoai Xu, Guosheng Xu, and Haoyu Wang. 2024. Digger: Detecting Copyright Content Mis-usage in Large Language Model Training.

[20] Ningke Li, Yuekang Li, Yi Liu, Ling Shi, Kailong Wang, and Haoyu Wang. 2024. Drowzee: Metamorphic Testing for Fact-conflicting Hallucination Detection in Large Language Models. In *OOPSLA (To Appear)*.

[21] Yuxi Li, Yi Liu, Gelei Deng, Ying Zhang, Wenjia Song, Ling Shi, Kailong Wang, Yuekang Li, Yang Liu, and Haoyu Wang. 2024. Glitch Tokens in Large Language Models: Categorization Taxonomy and Effective Detection. In *FSE*.

[22] Yuxi Li, Yi Liu, Yuekang Li, Ling Shi, Gelei Deng, Shengquan Chen, and Kailong Wang. 2024. Lockpicking LLMs: A Logit-Based Jailbreak Using Token-level Manipulation.

[23] Hanxiao Liu, Zihang Dai, David R. So, and Quoc V. Le. 2021. Pay Attention to MLPs. arXiv:2105.08050 [cs.LG]

[24] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. 2023. Prompt Injection attack against LLM-integrated Applications. *arXiv preprint arXiv:2306.05499* (2023).

[25] Haoneng Luo, Shiliang Zhang, Ming Lei, and Lei Xie. 2020. Simplified self-attention for transformer-based end-to-end speech recognition. *arXiv preprint arXiv:2005.10463* (2020).

[26] mwatkins. 2023. The petertodd phenomenon. Online. https://www.lesswrong.com/posts/jkY6QdCfAXHJk3kea/the-petertodd-phenomenon (Accessed: 2024-05-05).

[27] mwatkins. 2023. A Search for More ChatGPT/GPT-3.5/GPT-4 "Unspeakable" Glitch Tokens. Online. https://www.lesswrong.com/posts/kmWrwtGE9B9hpbgRT/a-search-for-more-chatgpt-gpt-3-5-gpt-4-unspeakable-glitch (Accessed: 2024-05-03).

[28] mwatkins and Jessica Rumbelow. 2023. SolidGoldMagikarp II: technical details and more recent findings. Online. https://www.lesswrong.com/posts/Ya9LzwEbfaAMY8ABo/solidgoldmagikarp-ii-technical-details-and-more-recent (Accessed: 2024-05-05).

[29] mwatkins and Jessica Rumbelow. 2023. SolidGoldMagikarp III: Glitch token archaeology. Online. https://www.lesswrong.com/posts/8viQEp8KBg2QSW4Yc/solidgoldmagikarp-iii-glitch-token-archaeology (Accessed: 2024-05-05).

[30] Bloom J Nanda N. 2022. TransformerLens. Online. https://github.com/neelnanda-io/TransformerLens (Accessed: 2024-05-05).

[31] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530* (2024).

[32] Jessica Rumbelow and mwatkins. 2023. SolidGoldMagikarp(plus, prompt generation). Online. https://www.lesswrong.com/posts/aPeJE8bSo6rAFoLqg/solidgoldmagikarp-plus-prompt-generation (Accessed: 2024-05-05).

[33] Bernhard Scholkopf and Alexander J. Smola. 2002. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press.

[34] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[35] Oguzhan Topsakal and Tahir Cetin Akinci. 2023. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. In *International Conference on Applied Engineering and Natural Sciences*, Vol. 1. 1050–1056.

[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[37] Vladimir N. Vapnik. 1995. *The Nature of Statistical Learning Theory.* Springer-Verlag.

[38] Leonid Nisonovich Vaserstein. 1969. Markov processes over denumerable products of spaces, describing large systems of automata. *Problemy Peredachi Informatsii* 5, 3 (1969), 64–72.

[39] Dixuan Wang, Yanda Li, Junyuan Jiang, Zepeng Ding, Guochao Jiang, Jiaqing Liang, and Deqing Yang. 2024. Tokenization Matters! Degrading Large Language Models through Challenging Their Tokenization. *arXiv preprint arXiv:2405.17067* (2024).

[40] Guanyu Wang, Yuekang Li, Yi Liu, Gelei Deng, Tianlin Li, Guosheng Xu, Yang Liu, Haoyu Wang, and Kailong Wang. 2024. MeTMaP: Metamorphic Testing for Detecting False Vector Matching Problems in LLM Augmented Generation. *FORGE* (2024).

[41] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155* (2023).