LLMs Have Rhythm: Fingerprinting Large Language Models Using Inter-Token Times and Network Traffic Analysis

Saeif Alhazbi (**b***, Ahmed Mohamed Hussain (**b**†, Gabriele Oligeri (**b***, Panos Papadimitratos (**b**† *College of Science and Engineering (CSE), Hamad Bin Khalifa University (HBKU) – Doha, Qatar {salhazbi, goligeri}@hbku.edu.qa

[†]Networked Systems Security Group, KTH Royal Institute of Technology – Stockholm, Sweden ahmed.hussain@ieee.org, papadim@kth.se

Abstract-As Large Language Models (LLMs) become increasingly integrated into many technological ecosystems across various domains and industries, identifying which model is deployed or being interacted with is critical for the security and trustworthiness of the systems. Current verification methods typically rely on analyzing the generated output to determine the source model. However, these techniques are susceptible to adversarial attacks, operate in a post-hoc manner, and may require access to the model weights to inject a verifiable fingerprint. In this paper, we propose a novel passive and non-invasive fingerprinting technique that operates in real-time and remains effective even under encrypted network traffic conditions. Our method leverages the intrinsic autoregressive generation nature of language models, which generate text one token at a time based on all previously generated tokens, creating a unique temporal pattern-like a rhythm or heartbeat-that propagates even when the output is streamed over a network. We find that measuring the Inter-Token Times (ITTs)-time intervals between consecutive tokens-can identify different language models with high accuracy. We develop a Deep Learning (DL) pipeline to capture these timing patterns using network traffic analysis and evaluate it on 16 Small Language Models (SLMs) and 10 proprietary LLMs across different deployment scenarios, including local host machine (GPU/CPU), Local Area Network (LAN), Remote Network, and VPN. The experimental results confirm that our proposed technique is effective and maintains high accuracy even when tested in different network conditions. This work opens a new avenue for model identification in realworld scenarios and contributes to more secure and trustworthy language model deployment.

Index Terms—Large Language Models, Small Language Models, Fingerprinting, Network Traffic Analysis, Deep Learning

I. INTRODUCTION

In recent years, language models, particularly Large Language Models (LLMs), have experienced accelerated advancement and widespread adoption across various fields. Their remarkable capabilities in language comprehension, reasoning, text, and code generation have unlocked new possibilities for automating and solving complex cognitive tasks [1], [2]. Due to their computational complexity and resource requirements, these models are typically deployed on expensive specialized hardware in cloud data centers and accessed as a cloud service through web applications or APIs provided by the vendors [3]. Building upon these services, third-party platforms integrate multiple state-of-the-art LLMs and offer them as a single subscription service to clients.

However, this remote deployment paradigm introduces critical security and privacy challenges, especially from the client's perspective. A fundamental security concern in this ecosystem is the lack of mechanisms for clients to verify the identity and integrity of the language models they interact with. This becomes critical as individuals and organizations increasingly rely on these models for sensitive tasks, raising concerns about service providers potentially modifying or substituting models without client knowledge or consent. Even in the case of reputable service providers, clients must place considerable trust in them to deliver exactly what they claim to offer in terms of model performance.

Existing solutions for LLMs identification broadly fall into two main categories: watermarking and fingerprinting. Watermarking works by embedding imperceptible markers within the model output during the generation process, which can later be detected through algorithmic analysis tools. Yet, this technique is vulnerable to adversarial attacks through paraphrasing or text modification, which can obscure the watermark [4], [5]. Unlike watermarking, fingerprinting does not embed markers but rather identifies the model based on the unique inherent characteristics in its generated output. Fingerprinting can be passive, analyzing the model output looking for any statistical or stylistic pattern. However, this is susceptible to text manipulation attacks [6]. Active fingerprinting, on the other hand, requires carefully designed prompts to elicit a specific model response for its identification [7], [8]. While robust, the active fingerprinting technique is computationally intensive and requires access to the model weights for finetuning and response alignment. Overall, both verification techniques operate in a post-hoc manner, analyzing the model output after generation.

In this paper, we investigate whether autoregressive language models can be uniquely identified from their token generation timing patterns as they stream their responses. Specifically, we measure and analyze Inter-Token Times (ITTs) the temporal intervals between consecutive tokens—that arise from the autoregressive generation process to determine if they can serve as a reliable fingerprint. Through extensive



Fig. 1: ITTs of six different SLMs. Each SLM was given the same prompt and instructed to repeat identical text to ensure the generated output is identical across all models. All SLMs were run on the same Graphics Processing Unit (GPU) and machine. Notice the distinct differences in latency, spikes, periodicity, and jitter, which together form each model's unique timing "fingerprint".

analysis, we show that these ITTs form a unique "rhythm" or signature that depends on the model's architecture, parameter size, and underlying hardware. As illustrated in Fig. 1, Small Language Models (SLMs) with similar parameter sizes or from the same family exhibit distinct ITTs pattern different from other models despite generating identical tokens and running on the same GPU. Each model has a unique temporal signature, characterized by variations in latency, periodicity, timing spikes, and subtle fluctuations in ITTs across the generated tokens sequence.

When autoregressive language models are accessed remotely via encrypted channels, these unique token generation timing patterns are preserved and propagated through the network as the models stream their responses to clients in real-time. Despite varying network conditions (e.g., latency, jitter, routing) and protocol overhead (e.g., encryption, packetization), these timing characteristics remain observable in network traffic and provide a reliable fingerprint for model identification.

To capture this fingerprint, we design and implement a DL- base pipeline that processes network traffic data and performs feature engineering extracting 36 features. These features are then passed to a hybrid DL architecture consisting of Bidirectional Long Short-Term Memorys (BiLSTMs) layers with a multi-head attention mechanism to identify the model. We conduct extensive evaluations of our proposed technique on both open-source SLMs and proprietary LLM.

Our experiments span a wide range of deployment scenarios, including local GPU/CPU deployment, LAN, remote networks, and VPN. Across these experiments, the results consistently demonstrate the effectiveness and robustness of our approach in identifying language model families and distinguishing between model variants. These findings provide a new perspective on model identification and ensure greater trust and integrity in their usage.

Our contributions can be summarized as follows:

- We demonstrate that autoregressive language models exhibit unique temporal patterns during token generation and propose a novel passive, real-time fingerprinting technique that leverages these unique patterns for model identification in both local and remote network scenarios.
- We design and implement an end-to-end pipeline that processes network traffic, extracts 36 timing and size features to capture the language model's fingerprint, and employs a hybrid BiLSTM-attention model to classify language models based on these features.
- We validate our approach through comprehensive experiments on 16 SLMs and 10 proprietary LLMs across various deployment scenarios (local host, LAN, remote network, VPN). Our results demonstrate the technique's effectiveness and robustness in identifying both model families and specific variants, even under different network conditions.

Paper Organization. Section II provides background and related work; Section III formulates the problem; Section IV presents our scenario and adversary model; Section V details the proposed methodology and experimental setup; Section VI reports the experimental results and analysis; and finally, Section VII concludes the paper.



Fig. 2: The autoregressive token generation process in LLMs. At each step (1-5), the model uses all previous tokens (shown in dashed boxes) as a context to predict the next token, then append the new token to the sequence. Generated tokens are shown with corresponding timestamps ($t_1...t_5$). Δt represents inter-token times between consecutive tokens generations.

II. BACKGROUND AND RELATED WORK

The development of language models is built upon decades of research and technological advancement [9]. In particular, the early 2000s and 2010s marked a significant milestone in the evolution and progress of language models. Starting with the groundbreaking work in [10], which introduced the first probabilistic language model based on neural networks, and laid the foundation of using word embeddings and applying DL in Natural Language Processing (NLP). Following that, the work in [11], [12] enhanced word representation to capture semantic relationships more efficiently than earlier techniques. By 2017, the field of NLP witnessed a paradigm shift with the introduction of the Transformer architecture [13]. Traditional architectures such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) suffered primarily from limitations in capturing long-term dependency and had to process text sequentially which made their training and inference slow and computationally expensive [14]. In contrast, the Transformer architecture addressed these issues through self-attention and positional encoding, which enabled parallelization and effective handling of long dependencies. This breakthrough has laid the groundwork for modern LLMs such as GPT and LLaMA, which scale to billions of parameters.

At their core, LLMs operate in an autoregressive manner, generating text one token at a time based on previously generated tokens and learned contextual embeddings. A token (e.g., a word or subword) is the basic unit of text processed and generated by the language models and is typically selected based on the highest probability of being next in the sequence. This iterative process continues until either a maximum sequence length is reached or a special end-of-sequence token is generated. Fig. 2 shows the sequential autoregressive token generation process in LLMs. At inference time, the generation speed of a language model is primarily determined by factors such as its architecture, parameter size, and hardware parallelization capabilities. Although these large

models achieve state-of-the-art performance in a wide range of NLP tasks, their computational complexity and massive size require specialized hardware to run efficiently. Hence, most LLMs are deployed in cloud data centers that can supply the required computational resources. As the adoption of LLMs such as ChatGPT, Gemini, and LLaMA becomes widespread, concerns about the security and privacy implications associated with these models have also grown [15]–[18]. Issues such as distinguishing between machine- and human-authored content, safeguarding intellectual property, and training models on stolen data are among the primary concerns. Researchers have been actively developing a variety of LLMs fingerprinting and detection techniques aiming to address these challenges.

Two primary approaches have emerged in response to these issues: watermarking and fingerprinting. Watermarking embeds identifying markers into the model-generated output to trace its origin and verify its authenticity. These markers are imperceptible to humans yet detectable through algorithmic methods. For instance, pioneering work by authors in [4] introduced a watermarking framework that modifies the output distribution by selecting a randomized set of "green" tokens before generating each word. Then, during the sampling process, the model softly promotes the use of these green tokens to create a statistical pattern without degrading the text quality. However, such methods are vulnerable to adversarial attacks, including text paraphrasing and token manipulation, which can compromise the watermark's integrity and reliability [19]. In contrast, fingerprinting focuses on detecting inherent patterns or characteristics in the model outputs to identify the generative model. Fingerprinting techniques can be categorized broadly into two main approaches: passive and active fingerprinting.

Passive fingerprinting analyzes the intrinsic characteristics of the model output by examining its lexical and stylistic patterns. For example, authors in [6] found that LLMs produce unique linguistic patterns and writing styles, such that even simple n-gram and part-of-speech distribution analysis can serve as effective fingerprints. These subtle variations in the frequency of specific lexical and syntactic features can differentiate between human- and machine-generated texts, as well as between different model families, such as GPT and LLaMA. The study also found that models within the same family often share similar fingerprints, even across different model sizes. Although this method requires no model modification or special queries, it is vulnerable to adversarial attacks that obscure the fingerprint through text manipulation. Another passive fingerprinting strategy introduced in [20] exploits memory usage patterns to identify the architectural family of LLMs deployed on edge and embedded devices. The key idea is that different LLM architectures exhibit distinct memory usage patterns during inference. By collecting high-resolution memory usage traces via the tegrastats tool and training a ROCKET model from the Sktime library, the authors effectively classified even previously unseen LLM families with an average accuracy of 92%. Alternatively, active fingerprinting methods, which require direct interaction with the model, can be further categorized based on their level of access.

In black-box active fingerprinting, specific queries or prompts are sent to the model to elicit responses that aid in its identification. These methods typically assume limited access to the model-often via an API or by observing its outputs. For example, [7] proposed LLMmap, a novel blackbox active fingerprinting technique that sends carefully crafted queries to the target application and analyzes the responses to identify the specific LLM version in use. Their query selection is informed by domain expertise on how LLMs generate uniquely identifiable responses to thematically varied prompts. The targeted query families include banner grabbing, alignment-based prompts, and malformed queries. With as few as 8 interactions, LLMmap achieves over 95% accuracy in identifying 42 different LLM versions, including both opensource and proprietary models. Similarly, [8] proposed Hide and Seek, a black-box approach using an Auditor-Detective framework. In this method, one LLM (the Auditor) generates discriminative prompts, while another LLM (the Detective) analyzes the responses to determine family relationships. This approach achieved 72% accuracy in distinguishing between popular architectures such as LLaMA, Mistral, and Gemma.

On the other hand, white-box active fingerprinting assumes full access to a model's weights and architecture. This level of access enables researchers to deliberately embed a unique signature or pattern into the model during the training or finetuning phase for ownership verification. For example, authors in [21] introduced novel cryptographic fingerprinting techniques called Chain and Hash to prove the ownership of LLM models. Their approach involves generating a set of special questions and answers, then concatenating each question with all questions, all potential answers, and a secret key using a secure hashing function like SHA-256. This fingerprint is then incorporated into the model through fine-tuning, with additional measures like meta-prompts and random padding to enhance the robustness. The verification process is performed by querying the model with the fingerprinting questions and checking whether it produces the expected responses. Similarly, [22] proposed a lightweight instruction tuning approach to embed verifiable fingerprints in LLMs. This technique trains models to generate predetermined outputs when presented with carefully crafted multilingual character sequences (secret keys). Remarkably, using only 60 training instances, this technique demonstrated perfect fingerprint retention across 11 different LLMs, even after extensive fine-tuning.

Despite these advances, existing LLMs watermarking and fingerprinting methods have several limitations. Watermarking techniques are often vulnerable to adversarial attacks, while some fingerprinting methods are computationally intensive and require access to model weights. To address these challenges, we propose a novel passive fingerprinting technique that identifies language models in real-time by leveraging their intrinsic ITTs characteristics, that are inherently difficult to manipulate or obscure via adversarial attacks. Furthermore, our approach is computationally efficient and can be deployed for online detection and monitoring, offering fine-grained fingerprinting capabilities that accurately distinguish between model families and even their variants.

III. PROBLEM FORMULATION

In this section, we establish the theoretical foundation of the proposed fingerprinting technique. We begin by formalizing the autoregressive generation process inherent in LLMs and defining the concept of token. We then demonstrate how these tokens are transmitted in network packets, showing how timing patterns and packet sizes can serve as distinctive fingerprints of the underlying model. Finally, we define our classification framework and focal loss objective.

Token Definition. A token is the basic unit of input or output text processed by a language model. Let \mathcal{M} denote a language model with a vocabulary \mathcal{V} , such that each token $s_i \in \mathcal{V}$ may represent a word, subword, or individual character, depending on the model's tokenizer.

Token Generation Process. For each finite output sequence of N tokens generated by \mathcal{M} :

$$\mathcal{S} = \{s_1, s_2, \dots, s_N\} \subset \mathcal{V}$$

where each s_i corresponds to the *i*-th token in the generated text. Each token is generated at a specific time t_i so that the overall model output, annotated with timestamps, is given by:

$$(s_1, t_1), (s_2, t_2), \ldots, (s_N, t_N)$$

Let \mathcal{T} denote the sequence of timestamps associated with the token generation, defined as:

$$\mathcal{T} = \{t_1, t_2, \dots, t_N\} \in \mathbb{R}^N_+$$

The Inter-Token Time (ITT) between consecutive tokens is defined as:

$$\Delta t_i = t_{i+1} - t_i, \quad i = 1, 2, \dots, N - 1.$$

Here, Δt_i measures the time required for the model (and its underlying hardware) to generate the next token. In general, in an autoregressive language model \mathcal{M} each token s_i is drawn from the model's vocabulary \mathcal{V} according to:

$$P(s_i|s_{< i}) = \mathcal{M}(s_1, \dots, s_{i-1}) \tag{1}$$

with $s_{\langle i}$ representing all tokens generated prior to s_i . At inference time, the autoregressive language model selects one token at each step using different sampling techniques (i.e., greedy decoding, temperature sampling, or top-k sampling) until an ending criterion is met.

From Tokens to Network Packets. While Δt_i captures the generation pattern between tokens on the model side, the output is typically observed via a streaming API or web interface. Hence, in a networking scenario, tokens are bundled into packets for transmission over the network (e.g., LAN or Internet). Therefore, in this practical scenario, packet is the measurable unit on the client side rather than the token.

Let $\mathcal{P} = \{p_1, p_2, \dots, p_M\}$ denote the sequence of packets received by the client. Each packet p_i arrives at timestamp t_i^a and may contain one or more tokens from S such that:

$$p_i = f_{pack}(s_i, s_{i+1}, ..., s_{i+m})$$

where f_{pack} represents the packetization function and m the number of tokens in packet i. Each packet p_i is encrypted before transmission:

$$p_i^{enc} = E(p_i, key)$$

with E denoting the encryption function. The packet arrival time t_i^a is different from the generation time due to network latency (l) and jitter (ϵ) :

$$t_i^a = t_i + l + \epsilon$$

On the client side, the inter-arrival time of packets is measured as:

$$\Delta t_i^p = t_{i+1}^a - t_i^a, \quad i = 1, 2, \dots, M - 1$$

his value is affected by both the language model's inter-token times Δt_i and network and protocol overhead (i.e., encryption, packetization, jitter, delays, etc.). Packet sizes are denoted as $\{|p_i|\}$.

Despite all these added complexities, we hypothesize that the language model's token generation pattern is preserved and remains detectable from these inter-arrival times Δt_i^p and packet sizes $\{|p_i|\}$ unless an obfuscation technique is applied at the server side.

Feature Extraction. We define a feature-extraction function that maps the raw $\{\Delta t_i^p\}$ and $\{|p_i|\}$ to higher-level features (e.g., burst rate, timing entropy, size-time correlation):

$$\phi(\{\Delta t_i^p\}, \{|p_i|\}) \longrightarrow \mathbb{R}^d$$

In our implementation, we extract 36 engineered features from the raw inter-arrival times $\{\Delta t_i^p\}$ and packet sizes $\{|p_i|\}$ (see Appendix A for details). For each sliding window wover the network data, we map these raw measurements to a 36-dimensional feature vector x using a feature extraction function ϕ , i.e.,

$$x = \phi(\{\Delta t_i^p\}, \{|p_i|\}) \in \mathbb{R}^{36}$$

Classification Framework. Given a set of K candidate LLMs, $\{M_1, \ldots, M_K\}$, our goal is to learn a classifier

$$f: \mathbb{R}^d \longrightarrow \{1, 2, \dots, K\}$$

Here \mathbb{R}^d denotes the *d*-dimensional feature space derived from the feature-extraction map. We collect a dataset *D* and label it as:

$$\mathcal{D} = \{ (\mathbf{x}_i, y_i) \}_{i=1}^N,$$

with $\mathbf{x}_i \in \mathbb{R}^d$ representing the *i*th sample corresponding to the extracted features vector for a window w of observed network packet sequence (comprising inter-arrival times and sizes), and $y \in \{1, \ldots, K\}$ denoting the true label corresponding to the model \mathcal{M}_k that generated that traffic snippet.

Given Dataset D, we train a deep neural network model $f(\mathbf{x}; \theta)$ (parameterized by θ) to output a probability distribution over K classes:

$$f(\mathbf{x};\theta) = (p_1(\mathbf{x};\theta), p_2(\mathbf{x};\theta), \ldots, p_K(\mathbf{x};\theta))$$

where $p_k(\mathbf{x}; \theta)$ represents the predicted probability that the feature vector \mathbf{x} originates from model \mathcal{M}_k , and the probabilities satisfy $\sum_{k=1}^{K} p_k(\mathbf{x}; \theta) = 1$.

Focal Loss Objective. To handle any potential class imbalance, we use focal loss [23], which is defined as:

$$\mathcal{L}_{\text{focal}}(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} \alpha_k \left(1 - p_k(\mathbf{x}_i; \theta) \right)^{\gamma} \mathbf{1}_{\{y_i = k\}} \log(p_k(\mathbf{x}_i; \theta))$$
(2)

where, for each sample *i*:

- $p_k(\mathbf{x}_i; \theta)$ is the predicted probability of class k.
- $\alpha_k \in [0, 1]$ is an optional weighting factor to mitigate class imbalance.
- γ ≥ 0 is the focusing parameter that increase emphasis on misclassified samples.
- $\mathbf{1}_{\{y_i=k\}}$ is the indicator that is 1 if $y_i = k$ and 0 otherwise.

By minimizing this focal loss, the classifier focuses on hardto-classify and underrepresented examples while reducing the weight on easily classified examples. The final prediction for a feature vector x is given by:

$$\hat{y} = \arg \max_{k \in \{1, \dots, K\}} p_k(\mathbf{x}; \theta^*)$$

where the optimal model parameters θ^* are found by minimizing Equation (2) via gradient-based methods (e.g., Adam or SGD) during training:

$$\theta^* = \arg\min_{\theta} \mathcal{L}_{\text{focal}}(\theta)$$

Once trained, $f(\cdot; \theta^*)$ provides a mapping from the extracted network-traffic features to the predicted LLM identity.



Fig. 3: Man-in-the-Middle attack scenario where an active adversary (A) provides a deceptive LLM API service, forwarding the user's prompt requests to a legitimate LLM provider (P) and relaying the responses back to the user (U).

IV. SCENARIO AND ADVERSARY MODEL

We consider a scenario involving three entities: the User (\mathcal{U}) , typically a normal user or developer, who interacts with what they believe to be a legitimate LLM API service to perform tasks such as text completion or function-calling; the Legitimate LLM Provider (\mathcal{P}) (e.g., OpenAI, Anthropic, or Mistral), who operates a production LLM service through authenticated APIs; and an Adversary (A) acting as a Manin-the-Middle (MitM) in an active role. Fig. 3 illustrates the adversarial scenario. The adversary A runs a deceptive API service that claims to provide proprietary, state-of-the-art LLM functionality, but in reality \mathcal{A} forwards prompts from \mathcal{U} to \mathcal{P} and simply relays the responses back to \mathcal{U} . To avoid being unmasked, A implements a filtering mechanism that detects and blocks any attempts to query model identity before forwarding prompts to \mathcal{P} . In this scenario, since the LLM operates as a streaming service, we assume A is incapable of obfuscating the LLM fingerprint (e.g., by randomizing response timing) as such obfuscation would degrade the quality of service, raise suspicion, and compromise the effectiveness of their deception [24]. By intercepting the prompts and responses of \mathcal{U}, \mathcal{A} not only compromises the security of \mathcal{U} , but also infringes on \mathcal{P} 's intellectual property rights and term of services. Furthermore, \mathcal{A} profits by charging users higher fees compared to \mathcal{P} 's pricing under the false claim of providing cutting-edge model. In this context, \mathcal{U} needs to verify the authenticity of the provided LLM service before interacting with the API system and sharing sensitive information. Our proposed fingerprinting techniques utilizes a DL classifier trained on known LLM fingerprints to identify the true underlying LLM. This approach reveals whether the service is truly a proprietary state-of-the-art model as claimed, or merely proxying requests to an existing commercial LLM provider's model.

V. METHODOLOGY

In this section, we discuss in detail our comprehensive experimental framework to study and analyze the unique ITT patterns of a variety of LLMs. We first describe the two types of language models used in our study, along with the different experimental scenarios to evaluate them. Then, we outline the data collection and processing phase, followed by the training and evaluation methodology. Finally, we report on our hardware and software experimental setup. Our investigation involves two primary categories of models: (i) open-source SLMs running locally and (ii) proprietary LLMs accessed via their Graphical User Interface (GUI) platforms. For each category of language models, we conducted experiments across different scenarios to evaluate the impact of factors such as hardware configurations and network conditions on the models' token generation timing patterns.

A. Open-Source SLMs

To understand how LLMs generate their patterns and establish a foundation for our experiment, we collected clean fingerprints from open-source SLMs deployed locally. Our experiment includes 16 SLMs spanning five model families: Gemma from Google [25], Granite3 from IBM [26], LLaMa from Meta [2], Mistral7B and Ministral from Mistral [27], and Phi from Microsoft [28], [29]. These models represent diverse architectures and parameter sizes, ranging from 1B to 9B parameters. Initially, we systematically analyzed SLMs fingerprints by running identical models locally on both GPU and Centeral Processing Unit (CPU). Through this comparative hardware analysis, we aimed to investigate how underlying hardware infrastructure impacts these temporal fingerprints. Then, using GPU deployment, we evaluated the SLMs across three distinct configurations simulating different real-world scenarios. Starting with SLMs allowed us to establish a controlled experimental foundation to analyze and trace their temporal fingerprints from source to destination while simultaneously examining the factors impacting these fingerprints. We hypothesize that if we can successfully capture fingerprints generated by these SLMs, then LLMs fingerprints would be more distinctive and detectable. This is because LLMs have significantly more complex architecture and larger parameter sizes (ranging from tens to hundreds of billions) compared to SLMs, which significantly increases their computational footprint and as a result creates more distinct temporal patterns in their token generation process. Our deployment setups include three different scenarios:

Local Host Deployment. In this setup, the client and server are running on the same Linux Ubuntu machine. This configuration eliminates external network overhead since all data transfer occurs within the system, providing an ideal environment for capturing clean SLMs fingerprints without network variability or latency.

LAN Deployment. In this setup, the client and server where SLMs are deployed operate on two separate machines and are connected via the same LAN. On the client side, a Python script automates the process of sending predefined prompts and listening to the streamed response as it is being generated from the Ollama server on the other machine. Meanwhile, *Tshark* captures the incoming packets and filters them according to their IP addresses. The communication between the two devices is not encrypted, and tokens are packetized during transmission. This configuration allows us to observe how the SLMs fingerprint behaves in a controlled local network environment with minimal latency and variability.

Remote Deployment via Internet. In this setup, we aim to investigate the impact of real-world internet conditions on SLMs fingerprint. To achieve this, we enabled remote access to the Ollama server over the internet using the Cloudflare tunneling service. The client machine is located in Sweden, while the Ollama server where the SLMs are hosted and running is located in Qatar. This geographical distribution introduces typical internet-based network conditions to the SLMs fingerprints.

SLMs Prompting. We crafted and used a diverse set of prompts, ranging from simple factual questions to complex analytical topics of different lengths, complexity, and response types. Through this methodology, we aimed to simulate realistic usage of these SLMs and ensure that each model demonstrates its unique token generation behavior under different prompting conditions.

B. Proprietary LLMs

Next, we extended our experiment to include real-world scenarios by accessing proprietary LLMs over the internet. We selected three families of widely-used proprietary models: OpenAI's GPT series (including GPT-4, GPT-4o, and GPT-4o mini), Anthropic's Claude (Sonnet 3.5, Opus, and Haiku), and Mistral (Pixtral Large, Mistral Large2, Mistral Small, and Mistral Nemo). At the time of writing, these models represent state-of-the-art LLMs, featuring diverse architectures, parameter sizes, and operational infrastructures. We captured the network traffic of these models and trained a hybrid DL model, then tested the same trained model under the following different scenarios:

Different Day. To evaluate the consistency and persistence of the LLMs fingerprints over time, we collected the training data on one day and the test data on a different day. Through this experimental design, we aim to examine whether LLMs fingerprints remain stable despite temporal variations in the models' operational environments, including variability in server load and network conditions at both the client and server endpoints.

Different Network. Network conditions can significantly affect the quality of observed LLMs fingerprints. To evaluate the impact of network conditions and to validate the networkagnostic nature of LLMs fingerprints, we trained our detection model using data collected from one network location and tested it using data collected from a different geographical location. This cross-network evaluation assesses the model ability to generalize across different network environments. In particular, we investigate whether the fingerprint characteristics remain distinguishable despite inherent variations in network conditions such as latency, jitter, and bandwidth between different locations.

VPN. To further validate our proposed technique, we conducted an additional experiment under more complex network conditions. Specifically, we accessed the proprietary LLMs using Virtual Private Network (VPN) with an exit node located in a different geographical region. By introducing this proxy layer, we aimed to determine whether the LLMs fingerprint remains observable despite the additional encryption, routing, and processing overhead introduced by VPN. In essence, we evaluated how well the detection model trained under normal network traffic conditions performs in this more challenging scenario.

LLMs Prompting. Similar to open-source SLMs we crafted diverse prompts to interact with the proprietary LLMs through their GUI website. These prompts varied in length, complexity, and topic, ranging from simple question-answer exchanges to extended chains of prompts that establish deep contextual dependencies. With this comprehensive prompting technique, we aimed to capture LLMs fingerprints under realistic usage scenario that reflect typical user interaction with such models.

The primary goal of performing these experimental scenarios is to demonstrate that LLM fingerprints are inherent characteristics of the models themselves, persisting regardless of temporal variations, network configurations, and routing infrastructures.

C. Data Collection and Processing

In the data collection phase, we acquired data using a wired connection throughout the different experiments, as it provides greater stability and reliability in measuring traffic patterns. We applied filters in Tshark and Wireshark to capture only inbound, data-only traffic-excluding server-related and control packets-to focus solely on the data packet patterns relevant to the analysis. The captured network traffic raw data was stored in a PCAP file and consisted of traffic packet sizes and their arrival times. As described in Algorithm 1, the data processing pipeline begins with the extraction and preprocessing of raw network traffic data. Specifically, we extract two fundamental components from the raw network traffic: packet arrival timestamps (\mathcal{T}_i) and corresponding packet sizes \mathcal{P}_i . From the timestamps, we compute the inter-arrival times (ΔT_i) between consecutive packets to capture the temporal signature of the LLMs and form the foundation for our fingerprinting methodology's feature extraction process. Following this phase, we perform data de-noising to remove anomalies and address any missing or invalid values. This preprocessing step ensures that our subsequent feature extraction phase is based on clean and reliable data that accurately represent each model's generation pattern.

D. Features Engineering

Next, we apply feature engineering to the extracted raw network traffic data. As the LLMs responses are transmitted over networks, they experience various noise factors and delays, making it infeasible to rely only on raw data (i.e., packet inter-arrival times and packet sizes) for accurate model identification. To address this limitation, we implement a feature engineering process as illustrated in Algorithm 2. The primary objective of feature engineering is to enhance data representation, reduce noise, and better capture the underlying patterns in the stream of LLMs packets. Using an iterative empirical process based on the two essential features– inter-arrival times and packet sizes–we extract a total of

Algorithm 1 Data Extraction and Preprocessing

1: Input:

- 2: N: Number of LLMs
- 3: \mathcal{D}_i : Raw network traffic data for each LLM L_i , where i = 1, 2, ..., N
- 4: Output:
- 5: Cleaned inter-arrival times ΔT_i and packet sizes \mathcal{P}_i for each LLM
- 6: for each LLM L_i , i = 1 to N do
- 7: Extract packet arrival times $\mathcal{T}_i = \{t_1, t_2, \dots, t_n\}$ from \mathcal{D}_i .
- 8: Extract packet sizes $\mathcal{P}_i = \{p_1, p_2, \dots, p_n\}$ from \mathcal{D}_i .
- 9: Compute inter-arrival times $\Delta T_i = \{\Delta t_1, \Delta t_2, \dots, \Delta t_{n-1}\}$ where $\Delta t_i = t_{i+1} t_j$.
- 10: Clean and preprocess data:
- 11: Remove anomalies and outliers.
- 12: Handle missing or invalid values.
- 13: end for

36 engineered features that focus on metrics revealing the model's token generation behavior. These features span six main categories: rate and throughput metrics (e.g., maximum burst rate and packet rate); inter-arrival time statistics (e.g., mean inter-arrival time and percentile distributions); pattern and regularity metrics (e.g., timing regularity and permutation entropy); timing change dynamics (e.g., mean time change and timing acceleration); correlation and combined metrics (e.g., size-time correlation and size-time products); and burstiness and entropy metrics (e.g., burstiness measure and inter-arrival time entropy). All these features collectively measure different aspects of the model's token generation behavior. Our aim is to recognize the "rhythm" of each LLM by detecting and analyzing the temporal changes in network activity within a small time window. We apply a sliding window of 0.5 seconds with a step size of 0.1 seconds to the raw network traffic data, which consists of packet sizes and inter-arrival times. These parameters were determined through extensive empirical evaluation, during which we tested various combinations. Within each data window, we compute a set of statistical and temporal features for that sample of data and label the resulting feature vector with the corresponding LLMs class L_i . The complete mathematical formulas and detailed calculations for all derived features are provided in Appendix A.

E. Training

We designed a hybrid DL architecture that combines multiple BiLSTM blocks with a multi-head attention mechanism to capture sequential long-term dependencies in network traffic patterns while focusing on the most discriminative features. This hybrid architectural approach has been successfully adopted in several contributions to network traffic analysis [30], [31]. As illustrated in Fig. 4 our model consists of three BiLSTM blocks with decreasing units to progressively refine the temporal features throughout the network. After each block, we added batch normalization to stabilize training and

Algorithm 2 Feature Engineering

1: Input:

- 2: Cleaned inter-arrival times ΔT_i and packet sizes \mathcal{P}_i for each LLM
- 3: w: Sliding window size
- 4: s: Step size for sliding window
- 5: Output:
- 6: Feature vectors \mathbf{x}_i labeled with LLM L_i
- 7: for each LLM L_i , i = 1 to N do
- 8: for each window w over ΔT_i and \mathcal{P}_i with size w and step s do
- 9: Extract features:
- 10: Statistical features (mean, variance, percentiles) of Δt and p.
- 11: Temporal features (burstiness, entropy).
- 12: Advanced features (permutation entropy, LIS).
- 13: Label feature vector \mathbf{x}_i with LLM L_i .
- 14: **end for**
- 15: end for

dropout layers (rate = 0.3) to prevent overfitting. Following the first BiLSTM block, an 8-head attention mechanism with a key dimension of 128 is applied, followed by batch normalization layer. A residual connection then adds the attention output back to the initial BiLSTM block's output to preserve the original temporal features and ensure efficient gradient flow before passing them to the second BiLSTM block. Finally, the architecture concludes with two dense layers (128 and 64 units) followed by batch normalization and a softmax layer to produce the final classification probabilities.

The training pipeline, outlined in Algorithm 3 consists of several stages optimized to handle network traffic data. The process starts with data preparation, where the input dataset \mathcal{D} containing the feature vector \mathbf{x}_i and corresponding labels \mathbf{L}_i is partitioned into training and validation sets. Next, the data is preprocessed using a per-sample normalization technique to standardize each input within a time window independently, preserving the relative temporal patterns while mitigating the effects of varying network conditions. To address the class imbalance problem where some LLMs have more data samples than others we used two strategies: class weighting and focal loss. This combination prevents bias towards overrepresented classes and maintains sensitivity to undersampled events. During data preparation, we again use the sliding window technique with window size w = 128 and step size s = 4to segment the temporal sequences. This creates overlapping windows that preserve the temporal continuity of the LLM generation patterns and provide sufficient training samples. We selected a batch size of 64, and trained the model for 30 epochs, as the standard deviation of accuracy in the last 10 epochs remained below 0.01, indicating convergence and training stability. It is worth noting that upon completion of the training phase, we tested the model on a newly collected dataset for each testing scenario.



Fig. 4: Attention-based BiLSTM architecture for SLMs and LLMs traffic classification.

Algorithm 3 Model Training

Require:

- 1: Dataset $\mathcal{D} = \{(\mathbf{x}_i, L_i)\}$ of feature vectors and labels
- 2: Number of training epochs T
- 3: Hyperparameters (learning rate η , batch size, etc.) **Ensure:**
- 4: Trained classification model f
- 5: Split \mathcal{D} into training set \mathcal{D}_{train} and validation set \mathcal{D}_{val}
- 6: Preprocess data with per-sample normalization and adaptive noise
 - ▷ Compute class weights for imbalanced data
- 7: Calculate class frequencies from training labels
- 8: Set $w_i = \sqrt{\max(\text{freq})/\text{freq}_i}$ for each class $i \\ \triangleright$ Model initialization and compilation
- 9: Initialize model f with BiLSTM-Attention architecture
- 10: Initialize Focal Loss with $\alpha = 0.25$, $\gamma = 2.0$, and class weights w_i
- 11: Set Adam optimizer with learning rate η
- 12: for epoch t = 1 to T do
- 13: Train model f on $\mathcal{D}_{\text{train}}$
- 14: Validate model f on \mathcal{D}_{val}
- 15: Update model parameters based on loss
- 16: end for

F. Evaluation

Given the imbalanced nature of the dataset after applying the preprocessing phase, using traditional accuracy metric can be misleading to report on the model's performance. Instead, we consider the following evaluation metrics to assess the model's classification performance:

$$Precision = \frac{TP}{TP + FP}$$
(3)

where precision measures the proportion of correct positive predictions, reflecting the model's accuracy in identifying specific LLM x such that out of all times the model identified x how often it was correct.

$$\operatorname{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{4}$$

where recall measures the model's ability to find all instances of a specific LLM x, such that out of all actual occurrences

of x in the data, how many were correctly identified by the model.

F1 Score =
$$\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$
 (5)

where the F1 score provides a balanced metric for the harmonic mean of precision and recall, especially in cases when evaluating model performance across different LLMs with varying frequencies in the dataset. In addition to the previous metrics, we present a confusion matrix for each conducted experiment to provide a comprehensive evaluation of the model's classification performance across all classes. The evaluation process, as shown in Section VI, is conducted on a completely new collected data.

G. Experimental Setup

We evaluated our proposed solutions using both open-source SLMs and proprietary LLMs. Specifically, we deployed 16 SLMs running locally on consumer-grade GPUs and CPUs. These open-source lightweight models were installed using Ollama (version 0.3.14), an open-source platform that simplifies the installation and running of LLMs locally. In contrast, the proprietary models, including ChatGPT, Mistral, and Claude, were accessed through their web-based GUI.

1) Hardware Configuration: For data collection, processing, deployment of open-source SLMs, and training of the classification model, we used a local machine with the following specifications. Our system was equipped with an NVIDIA RTX 4090 GPU, providing 24 GB of VRAM to support highspeed inference and efficient handling of SLMs parameters and architectures. This GPU was used for both SLM deployment and training of the classification models at a later stage. The system's processor was an Intel Core i9-13900K featuring 24 cores (8 performance cores and 16 efficiency cores) with a maximum clock speed of 5.8 GHz. To meet the memory demands for our computational task, the machine was equipped with 128 GB of DDR5 RAM operating at 5200 MHz.

2) Software Setup: The host machine was running Ubuntu 24.04.1 LTS. Automation of tasks such as prompting the SLMs, data collection, and processing was implemented using scripts written in Python 3.11.5. Several Python libraries were used, including *requests* for HTTP communication with the Ollama server running on port 11434 (configured to serve

all 16 models), *json* for parsing model responses, *csv* for data logging and storage, and *datetime* for precise timestamp handling. To capture network traffic, we integrated *tshark* directly within the Python script for the locally deployed SLMs, while *Wireshark* was used for proprietary models during the prompting process [32]. For implementing and training the DL model, we used *TensorFlow 2.15.0* managed through *Miniconda* environment manager (version 24.1.2). As for remote access to the Ollama server and to prompt the locally deployed open-source SLMs over the internet, we used the Cloudflare Tunneling service to establish a secure connection to the local machine. During VPN-based testing, we used Surfshark VPN to introduce network obfuscation and evaluate VPN network impact on the model classification performance.

VI. EXPERIMENTAL RESULTS

In this section, we present our experimental results by evaluating our proposed fingerprinting technique on two categories of language models: open-source SLMs running locally, and proprietary LLMs accessed through their websites. For SLMs, we first investigated their temporal patterns across different deployment scenarios, including local hardware configurations (GPU/CPU), LAN environment, and remote network access, where we developed a DL classification model to establish the feasibility of fingerprinting in challenging network conditions. For proprietary LLMs, we trained a classification DL model and evaluated its robustness by testing it on data collected on a different day, in a different network, and through VPN connection.

A. Open-Source SLMs

As an experimental baseline to understand and measure the unique ITTs that serve as a fingerprint for generative language models, we conducted experiments on 16 opensource SLMs from five leading companies in language model development. Due to hardware constraints and the high memory and GPU requirements of larger open-source LLM (e.g., Gemma 27B [25], or LLaMA 90), we experimented with model sizes ranging from 1B to 9B parameters. These models are considered small and are computationally feasible to run on consumer-grade hardware. We studied these SLMs across three different deployment scenarios: (i) local host where both client and server run on the same machine, (ii) LAN where both run on separate machines within the same network and (iii) remote deployment where client and server communicate over the internet. By adopting this controlled incremental approach, the objective is to establish a ground truth for our investigation and study the phenomenon analytically and progressively before applying any complex machine-learning techniques.

Local Host Deployment. In this scenario, both the server where the SLMs inference is performed, and the client prompting the models reside in the same machine. This setup allowed us to collect a clean fingerprint by eliminating any networkrelated factors and focusing purely on the intrinsic ITTs specific to each language model. To study the hardware impact on model generation characteristics, we ran the same experiment



Fig. 5: Mean ITTs for different SLMs families deployed on GPU. Colors represent models' family. Error bars represent standard deviation from the mean.



Fig. 6: Probability density distribution of ITTs for SLMs local host deployment.

on both GPU and CPU installed in the same machine. By running SLMs on the CPU, we considered the worst-case scenario where such models may be deployed on hardware with significant resource constraints. All SLMs were prompted with identical prompts to ensure fairness, consistency, and validity in the comparison across all models. During the response generation process, the Ollama server recorded each token with *created_at* field indicating the creation timestamps. We used this feature to compute the overall mean ITTs for each SLM across all generated outputs.

Deployment on GPU: First, we analyzed ITTs for the 16 SLMs deployed on the NVIDIA RTX 4090 GPU. As shown in Fig. 5 each model exhibits a distinct ITT profile different from the other models, even those within the same family demonstrate distinguishable temporal signature. Furthermore, we observed a clear correlation between model size and mean ITT such that models with larger parameter sizes tend to have longer mean ITT as they require more computational resources. This trend is particularly prominent within models of the same family and holds true across all families, though with varying degrees of scaling proportional to model size. For example, Gemma family which has the widest temporal range, demonstrate the most pronounced size-dependent linear scaling relationship with Gemma2 (9B), showing the longest mean ITT of approximately 0.0099 seconds nearly three times that of its smaller (2B) parameter variants Gemma1 (2B), which averages around 0.0031 seconds. Similarly, the LLaMA



Fig. 7: Mean ITTs for different SLMs families deployed on CPU. Colors represent models' family. Error bars represent standard deviation from the mean.

models demonstrate scaling with a moderate slope from 0.0029 seconds for LLaMA3.2 (1B) to 0.0070 seconds for LLaMA3 (8B). On the other hand, the Phi model's family shows a remarkable efficiency consistency across their variants with ITTs differences (Phi1 (2.7B) 0.0038s, Phi3 (3.8B): 0.0042s, Phi3.5 (3.8B): 0.0044s).

The standard deviations for GPU deployments are notably low, indicating stable and consistent ITTs. To complement our analysis and obtain a deeper understanding of the temporal patterns, we plotted the distribution of ITTs as probability density functions as illustrated in Fig. 6. As confirmed by the previous figure's result, each model exhibits a unique temporal distribution pattern with minimal overlap, even among models of similar parameter sizes from different families. The shapes of these distributions reveal another unique characteristic such that some models like LLaMA3.2 (1B) and Gemma (2B) show sharp, narrow peaks indicating very consistent timing , while others like LLaMA2 (7B) show broader distributions indicating greater timing variability. These variations in distribution shapes are observable both across models of similar parameter sizes and among variants within the same model family. This result reveals several key insights about model generation patterns.

While architectural similarities within a model family can create shared temporal behavior patterns, each model still produces its own unique, identifiable signature due to differences in parameter sizes and specific optimization techniques. Furthermore, the unique distribution shapes observed between models of similar parameter sizes suggest that these differences stem fundamentally from the architectural design of these models. Overall, these findings establish a strong foundation for our hypothesis that autoregressive language models can be identified through a novel lens using their temporal pattern during the token generation process. This approach enables discrimination not only between model families but also among specific variants within them, potentially serving as a complementary method to current fingerprinting techniques.

1) Deployment on CPU: As anticipated, due to the limited parallel processing capabilities of the CPU, the mean ITTs for SLMs inference are significantly longer compared to GPU execution times, as shown in Fig. 7. Specifically, the mean ITTs on CPU range from 0.02 to 0.14 seconds, wich is approximately an order of magnitude slower than GPU execution times of 0.003 to 0.01 seconds. Furthermore, the relation between model parameter size and inference speed is amplified on CPU, with larger models experiencing proportionally greater slowdowns compared to their performance on GPU. Moreover, CPU performance shows more timing variability demonstrated by larger error bars in the measurement. This trend is more noticeable in larger model families like Gemma and LLaMA. These results demonstrate that hardware configuration is a fundamental significant factor in determining ITTs of any language model alongside other characteristics such as model architecture, number of parameters, and the optimization techniques used. However, this hardware dependency does not prevent the practical applicability of our approach, as fingerprinting can be calibrated to the deployed hardware, which typically remains unchanged in the production environment, and can be recalibrated if hardware changes occur.

Network Impact: Most language models, particularly those with large weights, are accessed remotely over the internet as a cloud-based services since they require specialized highend GPU and computing infrastructure, which is typically not available to end-users. Therefore, we aim to investigate how protocol overhead and network conditions may affect the differentiability of language models based on their generation timing patterns when their responses are transmitted to clients. We test SLMs in two networking scenarios where language models could be deployed: (i) a controlled LAN where both the server running the model and the client reside within the same network, and (ii) a remote deployment environment where the models are hosted on an external network and accessed through the internet.

LAN Deployment: As we move beyond local host inference, network conditions and protocol overhead introduce additional complexity. We progressively evaluate how this may impact ITTs by testing that within a controlled LAN environment with minimal latency and variability. Such an environment closely resembles edge computing and Internet of Things (IoT) networks, where devices communicate locally. In this scenario, two machines communicate directly within the same subset network without any encryption, with one acting as a client and the other one as a server. Tokens generated by the Ollama server are packetized before transmission, simulating a realtime streaming scenario. In this context, a packet might contain one or more tokens generated by the SLM; as a result, on the client side, we instead shift our analysis to measure the inter-arrival times between consecutive packets. Fig. 8 displays the probability distribution function of the inter-arrival time of packets for each SLM. Compared to the previous result obtained within the local host machine (GPU based scenario) in Fig. 5, the LAN network environment introduces minimal additional variability, resulting in a slight shift and overlaps in the timing distributions. However, the general model groupings and relative performance characteristics remain recognizable despite these network effects.

Remote Network (Internet): In our final deployment scenario, we move from a controlled LAN to a fully remote



Fig. 8: Probability density distribution of SLMs packet interarrival times for LAN deployment.

network where real-world network conditions and protocol overhead are introduced including encryption, packetization, latency, jitter, routing variability, and potential congestion. Using Cloudflare's tunneling service, we enabled remote access over the internet to the Ollama server hosting the SLMs. The client machine was located in Sweden, while the server hosting the models was located in Qatar.

Our network path analysis revealed an 8-hop route traversing multiple infrastructure, including regional ISP networks, international transit points, and Cloudflare servers, each adding a potential source of jitter, latency, and packet reordering. This setup evaluates the models' behavior under typical internet deployment conditions where users access LLM services across geographical distances. However, this scenario represents the most challenging case for model identification, as network effects can potentially mask or distort the underlying modelspecific timing patterns. Similar to the LAN deployment, we leverage packet inter-arrival times to characterize models' behavior.

A simple statistical analysis of the probability density distribution depicted in Fig. 10 reveals a significant overlap in packet inter-arrival time distributions among different SLMs. Compared to both the local host and LAN scenarios, the distributions are notably broader, and the distinction between smaller and larger models is no longer visually apparent. This makes it difficult to distinguish models' identities based solely on a simple statistical analysis. This observation is quantitatively supported by Table I, which shows that local host deployment models exhibit consistent performance, with mean latencies ranging from 2.910ms (LLaMA3.2 1B) to 9.935ms (Gemma2 9B) and remarkably low standard deviations (0.235-1.141ms), indicating stable token generation. These timing patterns strongly correlate with model size, as larger models consistently show higher latencies.

On the other hand, the LAN environment introduces minimal additional overhead, with mean latencies increased by only 0.2-0.8ms compared to the Local Host scenario, while maintaining relatively low standard deviations. In contrast, remote deployment dramatically impacts both latency and consistency with mean latencies increased by factors of 15-20x (ranging from 57.043ms to 176.024ms) and standard deviations growing by two orders of magnitude (100-306ms).



Fig. 9: Training and validation accuracy curve during model training over 100 epochs on raw network traffic data versus feature engineered data.

These findings indicate that real-world network conditions such as those occurring over the internet introduce a significant layer of complexity and variability into the timing patterns of language models. Therefore, these results suggest that simple timing-based identification methods are likely impractical for real-world environments where clients access LLMs over geographically distributed infrastructure. Reliable model identification under such conditions requires more sophisticated approaches, combining machine learning techniques with advanced feature engineering to overcome these network-induced confounding effects.

Fingerprinting SLMs Using Network Traffic and ML. Network traffic analysis is an active research area to manage and secure networks based on their traffic flow characteristics [33]–[35]. Prior studies have leveraged statistical features extracted from network traffic flow, particularly focusing on packet size and inter-arrival time patterns, to classify application data [36]-[38]. Machine Learning (ML) techniques have demonstrated superior capability in handling noisy and complex data, effectively extracting subtle and nonlinear features that are typically beyond the reach of simple statistical models. In our experiments, model-specific patterns were clearly distinguishable in local-host and LAN deployment scenarios; however these patterns became obscured in the remote deployment scenario due to significant network noise and variability. To overcome this issue, we explore the application of ML techniques to learn and extractSLMs patterns from the data, even under challenging scenarios characterized by high noise and variability.

Using Network Traffic Raw Data: Before implementing any advanced feature engineering techniques, we first investigate whether raw network traffic data alone could effectively be used to identify SLMs in remote deployment scenarios. This baseline analysis aims to determine whether simple inherenet patterns in raw traffic are sufficient to distinguish between different SLMs, without relying on complex preprocessing or feature extraction methods.

Model	Local Host		LAN		Remote Network	
	Mean (ms)	Std (ms)	Mean (ms)	Std (ms)	Mean (ms)	Std (ms)
Gemma2:2b	4.757	0.532	5.073	5.992	92.491	140.935
Gemma2:9b	9.935	0.739	10.281	6.938	176.024	306.142
Gemma:2b	3.113	0.500	3.533	6.774	60.179	105.225
Gemma:7b	7.308	0.739	7.722	7.445	122.211	186.898
Granite3-dense:2b	4.264	0.592	4.536	5.808	77.297	124.723
Granite3-dense:8b	8.033	0.699	8.254	6.109	141.283	241.541
Granite3-moe:3b	7.229	0.235	7.478	5.443	130.655	200.566
Llama2:7b	6.197	0.577	6.388	5.013	125.487	186.944
Llama3.2:1b	2.910	0.446	3.203	5.741	57.043	100.862
Llama3.2:3b	4.353	0.535	4.682	6.115	80.970	125.695
Llama3:8b	7.044	0.628	7.349	5.957	124.175	184.154
Ministral:8b	8.612	0.649	8.678	5.406	114.694	158.620
Mistral:7b	6.291	0.603	6.568	5.547	115.432	187.360
Phi3.5:3.8b	4.416	1.141	4.833	6.479	88.051	137.776
Phi3:3.8b	4.223	0.811	5.045	9.410	83.558	134.219
Phi:2.7b	3.783	0.638	4.141	6.172	78.295	133.684

TABLE I: Comparison of model statistics across Local Host, LAN, and Remote Network. Each environment reports the mean and standard deviation of model timing.

Fig. 9 demonstrates that our DL model is overfitting the training data but fails to generalize to the validation, which was split from the same data according to an 80/20 ratio. This suggests that the neural network model with generalization, as it overfits (memorizes) noisy training data but fails to perform well on unseen samples even though they are from the same distribution [39]. These findings demonstrate that raw network traffic features alone are insufficient for reliable SLM fingerprinting using ML in remote deployment scenario. To address this limitation, a more advanced feature engineering approach is required to extract robust and discriminative characteristics from the network traffic. After applying feature engineering to raw network traffic data, the training and validation accuracies show significant improvement and convergence as depicted by the two red curves in Fig. 9. This convergence indicates that the model is effectively learning generalizable patterns rather than memorizing noise.



Fig. 10: Probability density distribution of SLMs packet interarrival times for remote deployment.

Advanced Feature Engineering: Given the limitations of raw network features and the non-linear separability shown in the probability distribution, we developed a comprehensive feature engineering pipeline to capture the underlying distinctive characteristics of SLM behavior. Specifically, our feature engineering approach transforms raw network traffic data into a rich set of discriminative higher-level statistical



Fig. 11: SLMs classification results when accessed remotely.

and temporal features designed to be robust against networkinduced noise. Using a sliding window size of 0.5 seconds and step size of 0.1 seconds over the sequence of raw network traffic data, we compute 36 engineered features. These features represent a mixture of rate-based metrics (e.g., maximum average rate, burst rate), temporal statistics (inter-arrival times, entropy), correlation measures (size-time correlation), and complexity indicators (permutation entropy, entropy rate). The complete mathematical formulas of these features are provided in Appendix A. Next, we trained our model, as described in Section 3, using the extracted data with an input size of 128, step size of 4, and batch size of 64. These parameters were selected based on iterative tuning experiments that yielded the best performance. The model was trained for 30 epochs and then evaluated on an entirely new dataset. The effectiveness of this feature engineering and training pipeline is shown in the confusion matrix in Fig. 11. The high



Fig. 12: Precision, Recall, and F1-Score for SLMs classification when accessed remotely.

values along the diagonal elements, indicate that the classifier successfully recognizes most models. Several key observations can be drawn from the classification results. Most models are identified with high precision, with many achieving over 90%, such as Gemma2-9B, Granite3-moe-3B, and LLaMA3-3.1B. Some confusion occurs between closely related models from the same family, such as Phi3:3.8B and Phi3.5 3.8B, which indicates similar behavior likely due to architectural similarities. Further examination of the models' performance metrics in Fig. 12 reveals that the majority of models maintain balanced precision, recall, and F1-scores above 0.85. The few exceptions occur primarily within architecturally similar model families, such as Phi models, where performance metrics drop to around 0.6-0.7. These findings confirm the robustness of the feature engineering approach and deep learning model in accurately identifying SLMs fingerprints, even when the models are hosted on remote networks environment characterized with significant latency, jitter, and network-induced noise that could potentially mask the underlying token generation patterns.

B. Proprietary LLMs

Building on our successful fingerprinting of open-source SLMs, we validate our proposed solution on several popular proprietary LLMs. These models present more challenging scenarios as they are typically accessed through APIs and web interfaces, which introduces additional complexities due to server load balancing, content delivery networks, and network condition variability. We selected 10 LLMs developed by three leading companies in the field: OpenAI's ChatGPT models (ChatGPT-4, ChatGPT-4o, and ChatGPT-4o Mini), Anthropic's Claude models (Haiku, Sonnet, and Opus), and Mistral's models (Mistral-Small, Large2, Nemo, and Pixtral-Large). It is worth noting that these models are configured to stream their responses to clients, generating output in chunks (e.g., word by word or token by token) and delivering it as it is generated. For each model, we collected approximately 3,000 seconds of network traffic data for training. Following that, we trained our model using the same configuration described in Section 3. To evaluate the robustness of our fingerprinting technique, we tested the model under three different challeng-



Fig. 13: Model classification results on a different day.

ing scenarios. Each represents 1,000 seconds of network traffic from realistic deployment scenarios that could potentially impact the model's generalizability and detection performance:

Different Day. First, we tested the model's ability to maintain its performance consistency across test data collected on a different day. This experiment aimed to determine whether temporal factors, such as varying server load and network condition variability, would significantly impact classification accuracy. Interestingly, as shown in the confusion matrix in Fig. 13, the model maintains strong classification performance across most LLMs confirming its robustness against temporal variability. Notably, the test results reveal several insights. OpenAI's models show some inter-family confusion, particularly between ChatGPT-40 and ChatGPT-4, suggesting shared architectural characteristics. Mistral's models demonstrate robust identification with minimal cross-family confusion. On the other hand, Anthropic's models maintain clear separation from other providers. This suggests that our fingerprinting technique captures the underlying fundamental characteristics of each model's signature that remain consistent even when tested on different days.

Different Network. Next, we tested the model on data collected from a network in different geographic locations. Specifically, while the training data was collected from a network located in Qatar, the test data was gathered from a network in Sweden. This way we can assess the model's generalizability and identify any potential bias toward the network conditions where the training was performed. Figure 14 shows that despite the network change, most of the predicted models are correctly classified along the main diagonal indicating strong generalization across different environments. In particular, certain models such as Mistral-Small, Pixtral-Large, and Opus remain distinctive and consistently recognizable regardless of the testing environment. However, when misclassifications occur among models, they tend to be from within the same model family or from the same company.



Fig. 14: Model classification results across the different networks.



Fig. 15: Model classification result when VPN is enabled.

In other words, similarities in model architecture or originating from the same company's servers may introduce some occasional confusion. Overall, these results suggest that the extracted fingerprints are not bound to a specific network setup but rather maintain their discriminative power across different network environments.

VPN. Another validation scenario involves testing the model on data collected while the VPN is enabled and configured to connect to a static server in Germany. In this setup, the VPN introduces additional routing complexity and potential timing pattern distortion. We evaluate the model's resilience to this type of network obfuscation by accessing the LLMs through a VPN connection. As shown in the confusion matrix

in Fig. 15 the model continues to accurately recognize most LLMs correctly. However, as observed in previous tests, there is a slight increase in misclassification, particularly among models from the same vendor or family. This effect is most noticeable in Mistral models (Large2 vs. Nemo) and OpenAI models (ChatGPT-40 Mini vs. ChatGPT-4). This suggests that our fingerprinting approach remains largely effective, despite some confusion among closely related models due to VPN masking.

Model Comparison in Different Scenarios. As reported earlier, the overall performance of the classification model remains relatively high and robust even when tested in different conditions. To evaluate the model performance in detail using different metrics, Fig. 16 presents a comparative analysis of the model's Precision, Recall, F1-score, and weighted metrics across all three scenarios considered in our testing experiment. This comparative evaluation highlights how changing the infrastructural and obfuscation-related conditions may impact fingerprinting performance.

Different Day: The different day scenario achieves the strongest performance across all metrics. In particular, F1 scores for most models surpass 0.85, with several achieving around 0.95. Similarly, Precision and Recall follow a similar trend, suggesting that temporal variations impact fingerprint detection minimally. However, notable exceptions are Mistral-Large2 and Sonnet 3.5, where their performance across all three metrics is the lowest. This result indicates that the classifier's performance remains relatively high despite temporal variations.

Different Network: In the different network scenario, where the test data was collected in a different geographic location, we observe in Fig. 16 that most models experience varying degrees of resilience. In particular, models such as Mistral-Nemo, Mistral-Small, and Opus demonstrate remarkable robustness. On the contrary, models such as ChatGPT-40 Mini exhibit the most significant degradation across all metrics. In general, the results reveal that network conditions on the client side indeed affect the model fingerprint, but the degree of impact varies across models.

VPN: The VPN scenario represents a more challenging scenario as the model performance degrades further compared to both the different day and different network scenarios. Several models that previously showed strong performance now experience varying degrees of decline due to the additional complexity, latency, and routing obfuscation caused by the VPN. For example, models such as Mistral-Small and Mistral-Nemo managed to maintain relatively strong performance. In contrast, models such as ChatGPT-40 and Sonnet 3.5 experienced more degradation under the VPN conditions. This suggests that while VPN may make model identification more challenging, certain models retain their reliable distinct pattern even under this scenario. Overall, as shown in Fig. 16d, the weighted metrics demonstrate that the different day scenario achieves the best performance (approximately 0.85), followed by the different network scenario (approximately 0.75), while the VPN scenario shows the lowest performance (approximately 0.7) across all weighted metrics (Precision, Recall, and F1-score).

16



Fig. 16: Performance comparison across different experimental scenarios. The plots show how classification performance varies across different deployment conditions and model families, with (a) F1-score, (b) precision, (c) recall, and (d) weighted average metrics.

Feature Visualization. While the classification metrics showed the effectiveness of the model in recognizing LLMs identities, it is critical to inspect how models process and represent their distinctive characteristics internally. To visualize these high-dimensional feature representations learned by the network, we apply t-SNE dimensionality reduction to the final layer outputs, projecting them into 2D space. As shown in Fig. 17, t-SNE reveals that data points belonging to the same model cluster together, forming spatially distinguishable groups. This suggests that these non-overlapping clusters are unique even after undergoing complex dimensionality reduction, indicating that these are inherent features of LLMs. Moreover, the compactness of these clusters indicates stability and consistency in the learned features, with tighter clusters suggesting lower variability in the model's behavior patterns. Furthermore, models from the same provider tend to exhibit similar behavior patterns, as shown by their close proximity in the visualization space. For example, the ChatGPT family models (ChatGPT-4, ChatGPT-4o, ChatGPT-4o Mini - shown in blue shades) cluster near each other, with particularly tight grouping between ChatGPT-4 and ChatGPT-40. Anthropic's Opus and Haiku (brown shades) show some regions of overlap, and Sonnet appears in close proximity to other Anthropic models in certain areas. In general, this distinct separation confirms the effectiveness of our feature engineering and training pipeline in successfully capturing and distinguishing the fingerprints of the LLMs.



Fig. 17: t-SNE visualization using base model.

VII. CONCLUSION

In this paper, we demonstrated that autoregressive language models have unique temporal generation patterns that can be used for model identification even when responses are encrypted and transmitted over network remotely. We proposed a feature engineering and training pipeline to capture the underlying model's signature and evaluated its effectiveness on both SLMs and LLMs. Our method successfully identified models with weighted F1 score 85% on different day, 74% across different network, and 71% when accessed through a VPN, maintaining robustness despite network variability and temporal changes. This work provides a robust, real-time technique for language model identification at the network layer and enhances the security and trustworthiness of systems where language models are deployed.

REFERENCES

- [1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [2] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.
- [3] J. S. G. Y. Z. Wen, P. C. Ng, Z. Wang, I. McLoughlin, A. B. Ng, and S. See, "On-device llms for smes: Challenges and opportunities," *arXiv* preprint arXiv:2410.16070, 2024.
- [4] J. Kirchenbauer, J. Geiping, Y. Wen, J. Katz, I. Miers, and T. Goldstein, "A watermark for large language models," in *International Conference* on Machine Learning. PMLR, 2023, pp. 17061–17084.
- [5] X. Zhao, S. Gunn, M. Christ, J. Fairoze, A. Fabrega, N. Carlini, S. Garg, S. Hong, M. Nasr, F. Tramer *et al.*, "Sok: Watermarking for ai-generated content," *arXiv preprint arXiv:2411.18479*, 2024.
- [6] H. McGovern, R. Stureborg, Y. Suhara, and D. Alikaniotis, "Your large language models are leaving fingerprints," arXiv preprint arXiv:2405.14057, 2024.
- [7] D. Pasquini, E. M. Kornaropoulos, and G. Ateniese, "Llmmap: Fingerprinting for large language models," arXiv preprint arXiv:2407.15847, 2024.
- [8] D. Iourovitski, S. Sharma, and R. Talwar, "Hide and seek: Fingerprinting large language models with evolutionary learning," *arXiv preprint* arXiv:2408.02871, 2024.
- [9] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv* preprint arXiv:2303.18223, 2023.
- [10] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," Advances in neural information processing systems, vol. 13, 2000.
- [11] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.
- [12] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.
- [13] A. Vaswani, "Attention is all you need," Advances in Neural Information Processing Systems, 2017.
- [14] S. Hochreiter, "Long short-term memory," Neural Computation MIT-Press, 1997.
- [15] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (llm) security and privacy: The good, the bad, and the ugly," *High-Confidence Computing*, p. 100211, 2024.
- [16] B. C. Das, M. H. Amini, and Y. Wu, "Security and privacy challenges of large language models: A survey," arXiv preprint arXiv:2402.00888, 2024.
- [17] B. Yan, K. Li, M. Xu, Y. Dong, Y. Zhang, Z. Ren, and X. Cheng, "On protecting the data privacy of large language models (llms): A survey," *arXiv preprint arXiv:2403.05156*, 2024.
- [18] S. Neel and P. Chang, "Privacy issues in large language models: A survey," arXiv preprint arXiv:2312.06717, 2023.
- [19] V. S. Sadasivan, A. Kumar, S. Balasubramanian, W. Wang, and S. Feizi, "Can ai-generated text be reliably detected?" *arXiv preprint* arXiv:2303.11156, 2023.

- [20] N. Nazari, F. Xiang, C. Fang, H. M. Makrani, A. Puri, K. Patwari, H. Sayadi, S. Rafatirad, C.-N. Chuah, and H. Homayoun, "Llm-fin: Large language models fingerprinting attack on edge devices," in 2024 25th International Symposium on Quality Electronic Design (ISQED). IEEE, 2024, pp. 1–6.
- [21] M. Russinovich and A. Salem, "Hey, that's my model! introducing chain & hash, an llm fingerprinting technique," arXiv preprint arXiv:2407.10887, 2024.
- [22] J. Xu, F. Wang, M. D. Ma, P. W. Koh, C. Xiao, and M. Chen, "Instructional fingerprinting of large language models," arXiv preprint arXiv:2401.12255, 2024.
- [23] T. Lin, "Focal loss for dense object detection," arXiv preprint arXiv:1708.02002, 2017.
- [24] G. Serazzi, "Impact of variability of interarrival and service times," in *Performance Engineering: Learning Through Applications Using JMT*. Springer, 2023, pp. 63–72.
- [25] G. Team, M. Riviere, S. Pathak, P. G. Sessa, C. Hardin, S. Bhupatiraju, L. Hussenot, T. Mesnard, B. Shahriari, A. Ramé *et al.*, "Gemma 2: Improving open language models at a practical size," *arXiv preprint arXiv:2408.00118*, 2024.
- [26] I. Granite Team, "Granite 3.0 language models," October 2024. [Online]. Available: https://github.com/ibm-granite/granite-3.0-language-models/
- [27] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. I. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.
- [28] M. Javaheripi, S. Bubeck, M. Abdin, J. Aneja, S. Bubeck, C. C. T. Mendes, W. Chen, A. Del Giorno, R. Eldan, S. Gopi *et al.*, "Phi-2: The surprising power of small language models," *Microsoft Research Blog*, vol. 1, no. 3, p. 3, 2023.
- [29] M. Abdin, J. Aneja, H. Awadalla, A. Awadallah, A. A. Awan, N. Bach, A. Bahree, A. Bakhtiari, J. Bao, H. Behl *et al.*, "Phi-3 technical report: A highly capable language model locally on your phone," *arXiv preprint arXiv:2404.14219*, 2024.
- [30] M. Wang, C. Chen, X. Zhang, and H. Qiu, "Bcba: An iiot encrypted traffic classifier based on a serial network model," *Future Generation Computer Systems*, vol. 164, p. 107603, 2025.
- [31] H. Yao, C. Liu, P. Zhang, S. Wu, C. Jiang, and S. Yu, "Identification of encrypted traffic through attention mechanism based long short term memory," *IEEE transactions on big data*, vol. 8, no. 1, pp. 241–252, 2019.
- [32] Wireshark Foundation, "Wireshark," git v4.2.2 packaged as 4.2.2-1.1build3 (Linux build). [Online]. Available: https://www.wireshark.org/
- [33] E. Papadogiannaki and S. Ioannidis, "A survey on encrypted network traffic analysis applications, techniques, and countermeasures," ACM Computing Surveys (CSUR), vol. 54, no. 6, pp. 1–35, 2021.
- [34] A. Azab, M. Khasawneh, S. Alrabaee, K.-K. R. Choo, and M. Sarsour, "Network traffic classification: Techniques, datasets, and challenges," *Digital Communications and Networks*, vol. 10, no. 3, pp. 676–692, 2024.
- [35] S. Rezaei and X. Liu, "Deep learning for encrypted traffic classification: An overview," *IEEE communications magazine*, vol. 57, no. 5, pp. 76– 81, 2019.
- [36] A. M. Hussain, G. Oligeri, and T. Voigt, "The dark (and bright) side of iot: Attacks and countermeasures for identifying smart home devices and services," in Security, Privacy, and Anonymity in Computation, Communication, and Storage: SpaCCS 2020 International Workshops, Nanjing, China, December 18-20, 2020, Proceedings 13. Springer, 2021, pp. 122–136.
- [37] M. Seydali, F. Khunjush, B. Akbari, and J. Dogani, "Cbs: A deep learning approach for encrypted traffic classification with mixed spatiotemporal and statistical features," *IEEE Access*, 2023.
- [38] M. Caprolu, S. Raponi, G. Oligeri, and R. Di Pietro, "Cryptomining makes noise: Detecting cryptojacking via machine learning," *Computer Communications*, vol. 171, pp. 126–139, 2021.
- [39] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning (still) requires rethinking generalization," *Communications of the ACM*, vol. 64, no. 3, pp. 107–115, 2021.

Appendix

NOTATION

TABLE II: Description of symbols used

Symbol	Description			
B_i	Size of the <i>i</i> -th packet in bytes			
Δt_i	Inter-arrival time between packet i and $i + 1$			
N	Number of packets			
w	Window size			
b	Burst window size			
\bar{x}	Mean of x			
σ_x	Standard deviation of x			
$P_k(x)$	kth percentile of x			

36 Features Derived from Raw Network Traffic Data

RATE-BASED FEATURES

1) Maximum Average Rate

$$\max_{t} \frac{8 \cdot \sum_{i=t}^{t+w} B_i}{w}$$

2) Maximum Burst Rate

$$\max_{t} \frac{8 \cdot \sum_{i=t}^{t+b} B_i}{b}$$

3) Maximum Arrival Rate

$$\max_{t} \frac{1}{\operatorname{mean}(\Delta t)}$$

4) Maximum Bytes per Second

$$\max_{t} \sum_{i=t}^{t+1} B_i$$

5) Bytes per Window

$$\frac{\sum_i B_i}{w}$$

 $\frac{N}{w}$

6) Packet Rate

INTER-ARRIVAL TIME (IAT) FEATURES

1) Mean IAT

$$\bar{\Delta t} = \frac{1}{N-1} \sum_{i=1}^{N-1} \Delta t_i$$

2) Standard Deviation IAT

$$\sqrt{\frac{1}{N-1}\sum_{i=1}^{N-1} (\Delta t_i - \bar{\Delta t})^2}$$

3) Coefficient of Variation IAT

$$\frac{\sigma_{\Delta t}}{\bar{\Delta t}}$$

4) IAT 25th Percentile

$$P_{25}(\Delta t)$$

5) IAT 75th Percentile

$$P_{75}(\Delta t)$$

6) IAT 90th Percentile

$$P_{90}(\Delta t)$$

7) IAT 90th to 10th Ratio

$$\frac{P_{90}(\Delta t)}{P_{10}(\Delta t)}$$

8) Large IAT Ratio

$$\frac{\left|\left\{\Delta t_i : \Delta t_i > \bar{\Delta t}\right\}\right|}{N-1}$$

9) IAT Entropy
$$-\sum_i p_i \log p_i$$

- TIME SERIES FEATURES
- 1) Mean Size-Time Product

$$\frac{1}{N-1}\sum_{i=1}^{N-1}B_i\,\Delta t_i$$

2) CV Size-Time Product

$$\frac{\sigma_{B\Delta t}}{\overline{B\Delta t}}$$

3) Timing Regularity

$$\frac{1}{1 + \sigma_{\Delta^2 t}}$$

4) Relative Time Pattern Entropy

$$-\sum_i p_i \log p_i$$

CHANGE AND ACCELERATION FEATURES

1) Mean Time Change

$$\frac{1}{N-2} \sum_{i=1}^{N-2} (\Delta t_{i+1} - \Delta t_i)$$

2) Standard Deviation Time Change

$$\sqrt{\frac{1}{N-2}\sum_{i=1}^{N-2} (\Delta^2 t_i - \overline{\Delta^2 t})^2}$$

3) Mean Time Acceleration

$$\frac{1}{N-3} \sum_{i=1}^{N-3} (\Delta^3 t_i)$$

4) Standard Deviation Time Acceleration

$$\sqrt{\frac{1}{N-3}\sum_{i=1}^{N-3} (\Delta^3 t_i - \overline{\Delta^3 t})^2}$$

CORRELATION AND ENTROPY FEATURES

1) Size-Time Correlation

$$\frac{\operatorname{cov}(B,\Delta t)}{\sigma_B \, \sigma_{\Delta t}}$$

2) Time Entropy Rate

 $\frac{H(w)}{|w|}$

3) Longest Increasing Size Sequence

$$\frac{\mathrm{LIS}(B)}{N}$$

4) Longest Increasing Time Sequence

$$\frac{\mathrm{LIS}(\Delta t)}{N-1}$$

5) Size Permutation Entropy

$$-\sum_{\pi} p(\pi) \log p(\pi)$$

6) Time Permutation Entropy

$$-\sum_{\pi} p(\pi) \log p(\pi)$$

STATISTICAL FEATURES

1) Time Autocorrelation

$$\frac{\sum_{i=1}^{N-2} (\Delta t_i - \bar{\Delta}t) (\Delta t_{i+1} - \bar{\Delta}t)}{\sum_{i=1}^{N-1} (\Delta t_i - \bar{\Delta}t)^2}$$

2) IAT Skewness

$$\frac{\frac{1}{N-1}\sum_{i=1}^{N-1}(\Delta t_i - \bar{\Delta t})^3}{\left(\sqrt{\frac{1}{N-1}\sum_{i=1}^{N-1}(\Delta t_i - \bar{\Delta t})^2}\right)^3}$$

3) IAT Kurtosis

$$\frac{\frac{1}{N-1}\sum_{i=1}^{N-1}(\Delta t_i - \bar{\Delta}t)^4}{\left(\frac{1}{N-1}\sum_{i=1}^{N-1}(\Delta t_i - \bar{\Delta}t)^2\right)^2}$$

RATE VARIABILITY FEATURES

1) Rate Variability

$$\frac{\sigma_R}{\bar{R}}$$

2) Peak Data Rate

$$\max_{i} \frac{8B_i}{\Delta t_i}$$

3) Burst Rate

$$\max_{t} \frac{8\sum_{i=t}^{t+b} B}{b}$$

4) **Burstiness**

$$\frac{\sigma_{\Delta t} - \bar{\Delta t}}{\sigma_{\Delta t} + \bar{\Delta t}}$$