Training LLMs to Speak Network

Iván González[1], Jorge E. López de Vergara[1], Javier Aday Delgado-Soto[2], Daniel Perdices[1], and Luis de Pedro[1]

[1] Department of Electronic and Communication Technologies, School of Engineering, Universidad Autónoma de Madrid, Spain
{ivan.gonzalez,jorge.lopez_vergara,daniel.perdices,luis.depedro}@uam.es
[2] Secretary of State for Digitalization and Artificial Intelligence, Ministry for Digital Transformation and Public Administration, Spain
javieraday.delgado@digital.gob.es

Abstract. The synthesis of realistic network traffic is essential for the evaluation of communication infrastructures, particularly in contexts such as performance benchmarking, security validation, and scalability testing. Traditional methodologies, such as traffic replay systems and statistical traffic models, often fall short in replicating the intricate and dynamic characteristics of real-world network behavior. In this study, we propose an innovative approach that leverages the fine-tuning of Large Language Models (LLMs) to generate synthetic network conversations that closely mirror authentic traffic patterns. Our methodology supports the structured generation of protocol-specific traffic, including ARP, ICMP, DNS and HTTP. Experimental results reveal that the proposed method achieves high accuracy and behavioral fidelity, with performance largely dependent on the capabilities of the underlying language model. These outcomes highlight the promise of LLM-based traffic generation as a versatile and effective tool for advancing research in network simulation, protocol behavior analysis, and cybersecurity experimentation.

Keywords: Network traffic generation · Generative AI · LLM · Fine-tuning.

1 Introduction

The integration of Machine Learning (ML) into network and service management is transforming the landscape of cybersecurity and operational efficiency. These modern approaches offer significant advantages over traditional systems; however, they are heavily reliant on large-scale datasets, particularly in the case of Deep Learning (DL) models. In this context, data quality becomes a critical factor in enhancing model performance, yet the volume of data must also be sufficient to capture and generalize underlying patterns. Unfortunately, high-quality datasets suitable for training are often limited and scarce, which poses a challenge to the generalization capabilities of these models.

To address the challenge of limited and outdated network traffic datasets, the generation of realistic synthetic traffic emerges as a promising solution. This approach not only augments existing datasets but also ensures high-quality data that can simulate diverse network scenarios. Furthermore, synthetic traffic generation mitigates key limitations of current datasets, such as class imbalance, poor feature quality, and obsolescence [26,5]. By expanding the volume of available traffic, researchers can create novel test scenarios that improve network management strategies. This capability is particularly beneficial for applications such as packet classification [1], traffic policing [2,8], and cybersecurity [9,4]. Without the cohesive integration of realistic traffic generation techniques, the full potential of AI-driven cybersecurity solutions cannot be realized.

In this work, we propose a novel methodology for generating realistic network traffic by fine-tuning LLMs on Scapy-based packet generation scripts. In this case, fine-tuning is the process of deriving a new model from an existing one adapted to a particular problem or dataset, in our case, network traffic generation. Training an LLM from scratch is a massive task that requires hundreds to thousands of GPUs, while fine-tuning involves a set of more delicate procedures that try to get the most out of the model while modifying a rather small number of parameters. Scapy [6] is a powerful Python library that allows for the creation, manipulation, and analysis of network packets. By training LLMs on curated Scapy code that represents diverse protocol-specific conversations, we enable the model to learn the syntax, semantics, and sequence of protocol interactions. This approach allows the model to generate with small prompts new Scapy scripts that simulate realistic network traffic across various protocols.

The remainder of this paper is organized as follows: Section 2 reviews related work and emphasizes the novelty of our approach. Section 3 presents the proposed methodology for packet generation, including dataset construction and the evaluation framework. The experimental results are provided in Section 4. Finally, Section 5 summarizes the main contributions and outlines potential directions for future work.

2 State of the Art

The generation of realistic network traffic is a longstanding objective in the domains of network security and performance evaluation. Traditional methods, including traffic replay systems [12] and statistical modeling, have been widely used to simulate network behavior. These approaches can be categorized by their level of abstraction: metadata-based [27], flow-based [25], and packet-based generation [20]. While effective in specific contexts, such methods often lack adaptability and fail to capture the complexity of evolving network conditions.

Recent advances in deep learning have significantly enhanced synthetic traffic generation [3,28]. Generative models such as Generative Adversarial Networks (GANs) [10,20,29] and LLMs [7] have introduced dynamic, context-aware capabilities. In metadata-based generation, deep models replicate statistical distributions of traffic features. Flow-based approaches leverage RNNs, such as LSTMs [19], to model temporal dependencies, while packet-based methods employ CNNs and RNNs to emulate protocol-specific behaviors [19].

GAN variants like PAC-GAN [10] and WGAN-GP [25] have been used to generate labeled traffic for intrusion detection. Diffusion and state-space models [16,11] have shown promise in reducing statistical divergence from real data, although challenges in protocol conformance remain open.

Transformer-based models, particularly GPT variants, have emerged as powerful tools for traffic generation. PAC-GPT [17] demonstrated GPT-3 ability to generate ICMP and DNS packets, albeit with limitations in request–response modeling. *GPT on the wire* [13], based on a few-shot learning approach, offers a cost-effective alternative for traffic synthesis, making it ideal for rapid prototyping and experimentation. TrafficGPT [22], in contrast, adapts the model architecture to enable a deeper understanding of traffic data.

Building on these developments in LLM-driven traffic generation, our work introduces a methodology that bridges the gap between prompt-based generation and architecture-level adaptations. Our method leverages fine-tuning of existing LLMs on curated Scapy datasets to provide a practical and extensible solution to generate realistic network traffic, supporting both research and evaluation efforts.

3 Methodology

The proposed methodology for traffic generation consists of four key stages:

1. **Dataset Generation:** A curated dataset of Scapy scripts is constructed to represent a diverse set of network interactions, including ARP, DNS (over UDP), ICMP, and HTTP (over TCP). Each script is paired with a natural language summary that describes the parameters of the conversation (e.g., IP addresses, ports, domain names), along with the corresponding Scapy code used to generate the packet sequence.
2. **LLM Fine-Tuning:** A pre-trained LLM is fine-tuned on the dataset using supervised learning. The model learns to translate natural language descriptions of network conversations into valid Scapy code, capturing both protocol logic and Python syntax. Unlike few-shot approaches [13], this method enables the model to internalize protocol semantics and generate complete executable scripts.
3. **Code Execution and Traffic Generation:** The generated Scapy code is executed to synthesize real network traffic, which is then stored in `.pcap` format for further analysis.
4. **Validation and Analysis:** This stage involves both static and dynamic validation. It includes syntax checking of the generated code and protocol compliance verification using Wireshark dissectors.

The next subsections detail each step of the methodology.

3.1 Dataset Generation

The dataset consists of plain-text instruction–output pairs, designed to train the model to generate Scapy-based network scripts from natural language descriptions. Each entry includes:

– **Instruction:** A natural language prompt describing the task: the generation of a Scapy script for a specific network interaction. It includes a concise summary of the communication scenario, specifying protocol details, IP addresses, ports, and other relevant parameters.

– **Output:** A complete Python script utilizing the Scapy library that fulfills the instruction and adheres to the expected protocol behavior.

This format enables the model to learn how to interpret natural language descriptions of network behavior and translate them into executable code. It emphasizes linguistic generalization and aligns with the pretraining objectives of LLMs, facilitating effective fine-tuning for code generation tasks.

An example entry from the dataset is shown below:

```
Below is an instruction that describes a task. Write a response
    that appropriately completes the request.

### Instruction:
Generate Python code to create packets with the scapy framework
    based on the following summary of a DNS conversation:
Source: IP= "37.24.70.161" // Destination: IP= "37.25.117.164"
    // Others: id=0x1017 , resource= "www.company23.io" ,
    response = "151.48.69.115"
Don’t explain the code, just generate the code block itself.
The last code line of your code must be ’pkt_list = [pkt_1,
    pkt_2, ...]’; If not, make the appropriate corrections to
    the code.

### Output:
import random
from scapy.all import *
import time

RANDOM_MAC1="02:00:00:%02x:%02x:%02x" % (random.randint(0, 255),
    random.randint(0, 255),random.randint(0, 255))
RANDOM_MAC2="02:00:00:%02x:%02x:%02x" % (random.randint(0, 255),
    random.randint(0, 255),random.randint(0, 255))
RANDOM_PORT = random.randint(4097, 65530)
pkt_1 = Ether(src=RANDOM_MAC1, dst=RANDOM_MAC2)/IP(src
    ="37.24.70.161", dst="37.25.117.164", proto=17) / UDP(sport=
    RANDOM_PORT, dport=53) / DNS(id=0x1017, qr=0, rd=1, opcode
    =0, qdcount=1, ancount=0, nscount=0, arcount=0, qd=DNSQR(
    qname="www.company23.io", qtype="A", qclass="IN"))
time.sleep(abs(random.gauss(0, 0.03)))
pkt_2 = Ether(src=RANDOM_MAC2, dst=RANDOM_MAC1)/IP(src
    ="37.25.117.164", dst="37.24.70.161", proto=17) / UDP(sport
    =53, dport=RANDOM_PORT) / DNS(id=0x1017, qr=1, opcode=0, ra
    =1, rcode=0, qdcount=1, ancount=1, nscount=0, arcount=0, qd=
    DNSQR(qname="www.company23.io", qtype="A", qclass="IN"), an=
    DNSRR(rrname="www.company23.io", type="A", rclass="IN", ttl
    =255, rdata="151.48.69.115"))

pkt_list = [pkt_1, pkt_2]
```

The dataset was constructed using a semi-synthetic methodology. A subset of real network packets was initially captured to ensure alignment with authentic protocol behavior and converted to Scapy code. Then, more samples were added by generating new conversations through controlled modifications of network parameters, tailored to the characteristics of each protocol. To simulate realistic traffic dynamics, temporal features such as randomized inter-packet delays (e.g., `time.sleep(abs(random.gauss(0, 0.03)))`) were incorporated, introducing variability that mimics natural network timing.

3.2 LLM Fine-Tuning

The next stage involves selecting one or more language models for fine-tuning, based on three primary criteria: (i) their performance on Python code generation benchmarks [14,23,21]; (ii) their compatibility with the available hardware, specifically a 40 GB NVIDIA A100 GPU; and (iii) our prior experience working with these models [13].

Given the memory constraints of the Nvidia A100 GPU, we evaluated whether each model could be fine-tuned with or without the use of advanced optimization techniques such as quantization and Low-Rank Adaptation (LoRA). Quantization reduces memory usage and accelerates computation by lowering the precision of model weights, while LoRA enables efficient fine-tuning by injecting small trainable matrices into the attention layers, typically less than 1-10% of the original size.

The fine-tuning process is conducted using supervised learning, where a pre-trained LLM is trained on a dataset of instruction–output pairs. Each instruction, as shown in subsection 3.1, is a natural language description of a network communication scenario, and the corresponding output is a complete Scapy script that implements the described behavior. Through this process, the model learns to translate natural language prompts into valid Python code that complies with protocol specifications.

Based on the selection criteria, we chose two models: Codestral by Mistral and Deepseek-Coder by DeepSeek, both available via the Hugging Face platform (*Codestral-22B-v0.1* and *deepseek-coder-1.3b-instruct*, respectively) [15]. *Codestral-22B-v0.1* is a state-of-the-art 22B parameter model specifically designed for code generation tasks. Its high capacity offers strong reasoning and code synthesis capabilities, making it a suitable candidate for advanced fine-tuning and downstream applications. However, due to its large size, it requires optimization techniques such as 4-bit quantization and Low-Rank Adaptation (LoRA) to fit within available hardware resources, specifically the A100 GPU memory limits. On the other hand, *deepseek-coder-1.3b-instruct* is a significantly smaller model (1.3B parameters) that strikes a balance between efficiency and performance. It is lightweight enough to be fine-tuned directly without the need for aggressive memory optimizations. This makes it particularly suitable for rapid prototyping and experimentation, especially in environments where computational resources are limited.

3.3 Code Execution and Traffic Generation

In this stage, the fine-tuned LLM receives a natural language prompt similar in structure to those used during training, that requests network packets based on a protocol and a summary that describes the parameters of the conversation. The model responds by generating Scapy code that constructs the corresponding packet sequence as a list of packets, stored in a variable named `pkt_list`. This list encapsulates the entire sequence of packets representing the described interaction (see Subsection 3.1).

The code is executed using `exec()` Python function, which dynamically runs the generated script and populates the `pkt_list` variable with Scapy packet objects. Once the packet list is constructed, it is serialized and saved to a `.pcap` file using `wrpcap()` Scapy function.

The resulting `.pcap` file can then be analyzed using tools such as Wireshark to inspect packet structures, protocol fields, and sequence logic. This approach ensures safe, reproducible traffic generation and supports detailed offline analysis of the generated network behavior.

3.4 Validation and Analysis

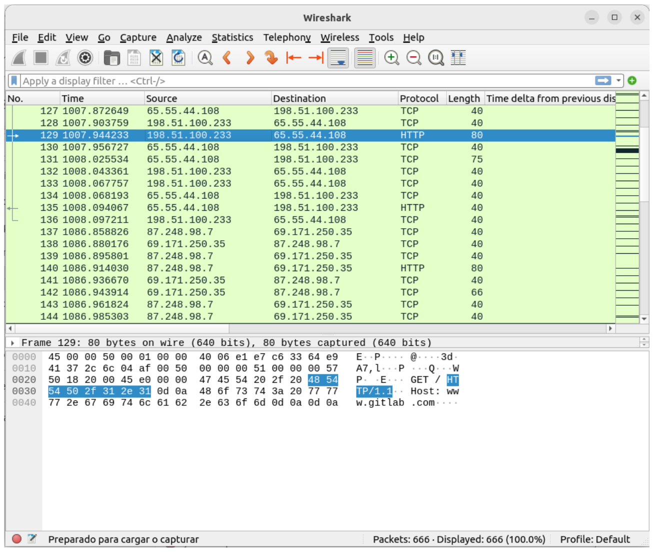
To assess the quality and reliability of the generated Scapy scripts, we adopt a comprehensive evaluation strategy that combines textual similarity metrics with functional traffic validation. This methodology ensures that the outputs are both syntactically accurate and semantically valid from a network protocol perspective.

Textual Similarity Metrics. We evaluated the similarity between generated scripts and reference implementations using two complementary metrics:

– **ROUGE-1 [18]:** Measures unigram (word-level) overlap between the generated and reference code. It provides precision, recall, and F1-score, capturing surface-level lexical similarity.
– **CodeBLEU [24]:** A code-specific metric that extends BLEU by incorporating programming language structure. It combines n-gram matching, abstract syntax tree (AST) similarity, data flow consistency, and semantic embeddings. CodeBLEU is particularly useful for evaluating whether the generated code preserves the logic and structure of the reference implementation.

Packet-Level Validation with Wireshark. To verify the functional validity of the generated scripts, the generated packets are saved in `.pcap` format. We then analyzed the packets using Wireshark to detect protocol-level issues (see Figure 1). We extract:

– **Warnings and errors:** Reported by Wireshark protocol dissectors, including malformed headers, checksum errors, and unexpected field values.
– **Protocol compliance:** Whether the packets conform to the expected structure and behavior of the specified protocols (e.g., valid DNS query/response pairs, correct TCP handshakes).


Wireshark
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.    Time         Source           Destination      Protocol  Length  Time delta from previous dis
   127 1007.872649  65.55.44.108     198.51.100.233   TCP       40
   128 1007.903759  198.51.100.233   65.55.44.108     TCP       40
   129 1007.944233  198.51.100.233   65.55.44.108     HTTP      80
   130 1007.956727  65.55.44.108     198.51.100.233   TCP       40
   131 1008.025534  65.55.44.108     198.51.100.233   TCP       75
   132 1008.043361  198.51.100.233   65.55.44.108     TCP       40
   133 1008.067757  198.51.100.233   65.55.44.108     TCP       40
   134 1008.068193  65.55.44.108     198.51.100.233   TCP       40
   135 1008.094067  65.55.44.108     198.51.100.233   HTTP      40
   136 1008.097211  198.51.100.233   65.55.44.108     TCP       40
   137 1086.858826  87.248.98.7      69.171.250.35    TCP       40
   138 1086.880176  69.171.250.35    87.248.98.7      TCP       40
   139 1086.895801  87.248.98.7      69.171.250.35    TCP       40
   140 1086.914030  87.248.98.7      69.171.250.35    HTTP      80
   141 1086.936670  69.171.250.35    87.248.98.7      TCP       40
   142 1086.943914  69.171.250.35    87.248.98.7      TCP       66
   143 1086.961824  87.248.98.7      69.171.250.35    TCP       40
   144 1086.985303  87.248.98.7      69.171.250.35    TCP       40

Frame 129: 80 bytes on wire (640 bits), 80 bytes captured (640 bits)
0000  45 00 00 50 00 01 00 00  40 06 e1 e7 c6 33 64 e9   E··P···· @····3d·
0010  41 37 2c 6c 04 af 00 50  00 00 00 51 00 00 00 57   A7,l···P ···Q···W
0020  50 18 20 00 45 e0 00 00  47 45 54 20 2f 20 48 54   P· ·E··· GET / HT
0030  54 50 2f 31 2e 31 0d 0a  48 6f 73 74 3a 20 77 77   TP/1.1·· Host: ww
0040  77 2e 67 69 74 6c 61 62  2e 63 6f 6d 0d 0a 0d 0a   w.gitlab .com····

Preparado para cargar o capturar      Packets: 666 · Displayed: 666 (100.0%)      Profile: Default
```

Fig. 1. Packet-Level analysis with Wireshark.

4 Results

4.1 Dataset Composition

The dataset generated for this study comprises network traffic conversations categorized into four protocol types: ARP, DNS, HTTP, and ICMP. It is divided into training, validation, and test subsets. Each protocol is represented with a similar number of conversations across these subsets, making the dataset relatively balanced. This balance is crucial for ensuring fair and unbiased model training and evaluation. Out of a total of 2 595 samples, 2 333 were used for training, 262 for testing, of which 50 have been chosen for validation. Although modest in size compared to large-scale pretraining corpora, the dataset is sufficiently rich and diverse to enable the model to learn the structure and behavior of the targeted protocols effectively. The distribution of conversation types and the dataset splits are shown in Table 1.

Table 1. Summary of the number of conversations per protocol in the training, test, and validation datasets.

Protocol	**Training**	**Test(Validation)**	**Total**
ARP	687	77(15)	764
DNS	648	73(14)	721
HTTP	393	44(8)	437
ICMP	605	68(13)	673
Total	2 333	262(50)	2 595

4.2 Fine-tuning

The training configuration was kept consistent across both models, with the hyperparameters detailed in Table 2. The models were fine-tuned using Supervised Fine-Tuning (SFT) on the training subset with a batch size of 4, ensuring a fair comparison while maximizing GPU memory utilization for each configuration. Evaluation was conducted using batches of size 1 on 50 samples from the test set, which were designated as a validation subset during training to monitor performance and mitigate overfitting. Both models were trained for 8 epochs, maintaining consistency in training duration and allowing for a direct comparison of learning dynamics under equivalent conditions.

The distinction between the two training setups lies in the use of LoRA and quantization. Codestral was trained using 4-bit quantization and LoRA adapters, which significantly reduced memory usage and training time. In contrast, DeepSeek-Coder was trained in full precision without parameter-efficient fine-tuning techniques, resulting in higher memory consumption but potentially more direct gradient updates across the full model. This difference in training strategy likely contributed to the observed variations in training dynamics and final loss, rather than differences in model scale or pretraining exposure.

Table 2. Comparison of Training Configurations and Results

Parameter	Codestral	Deepseek-Coder
Batch Size (Train / Eval)	4 / 1	4 / 1
Gradient Accumulation Steps	1	1
Epochs	8	8
Optimizer	`adamw_8bit`	`adamw_8bit`
LR Scheduler	Linear	Linear
Learning Rate	2e-5	2e-5
Warmup Steps	5	5
Precision	bf16	bf16
Weight Decay	0.001	0.001
Peak Reserved Memory (GB)	19.104	32.932
LoRA/Quantization	Yes / 4-bit	No / Full
Training Runtime (s)	19 121.67	9 120.04
Final Training Loss	0.055700	0.014000

4.3 Evaluation

The evaluation phase is designed to assess the models ability to generalize from the training data and accurately generate Scapy scripts based on natural language instructions. This is performed using a dedicated test subset of the dataset, which consists of 262 samples. The evaluation is conducted in batches, allowing for efficient processing while maintaining consistency across model comparisons. During evaluation, the model output is compared against the ground truth Scapy script, and the generated packets are analyzed using Wireshark to identify protocol-level issues, as described in Section 3.

Both Codestral and DeepSeek-Coder demonstrate strong capabilities in generating network protocol Scapy code, though their strengths vary by protocol. Codestral consistently achieves a 100% success rate across all protocols, indicating robust and reliable performance. In contrast, DeepSeek-Coder shows notably lower success on ARP.

Table 3. Average generation time (seconds) of conversations per protocol on GPU

Protocol	Codestral	DeepSeek-Coder
ICMP	26.09	6.61
DNS	40.63	10.20
ARP	24.34	6.16
HTTP	119.55	30.67
Mean	52.65	13.41

Table 3 reveals that DeepSeek-Coder significantly outperforms Codestral in generation time across all evaluated network protocols, averaging approximately 13.41 seconds compared to Codestral 52.65 seconds. A key factor contributing to this disparity is the difference in model size. Smaller models like *deepseek-coder-1.3b-instruct* typically require fewer computations and can fully utilize GPU memory, resulting in faster and more efficient inference. In contrast, larger models such as Codestral may offer greater contextual understanding or higher accuracy, but at the cost of increased latency due to their computational demands.

We analyze each LLM results in detail below.

Codestral. Codestral generated 262 conversations from a total of 262 entries in the test dataset, all of which successfully passed the Textual Similarity evaluation and resulted in valid packet generation (Table 4). The similarity metric results reveal distinct performance patterns across protocols. The ICMP protocol achieved perfect scores in both ROUGE (1.000) and CodeBLEU (1.000), indicating exceptional lexical fidelity and structural accuracy. ARP followed closely with a ROUGE score of 0.996 and a strong CodeBLEU score of 0.991, suggesting highly faithful and well-structured code generation. HTTP also performed well, with a ROUGE score of 0.935 and a CodeBLEU score of 0.922, reflecting consistent quality. DNS demonstrated balanced performance with a ROUGE score of 0.908 and a CodeBLEU score of 0.900, making it a reliable protocol for code generation. Overall, Codestral exhibits robust capabilities in generating accurate and semantically rich code across multiple network protocols, with particularly outstanding results for ICMP and ARP.

DeepSeek-Coder. DeepSeek-Coder generated 262 conversations from the test dataset, all of which successfully passed the Textual Similarity evaluation and resulted in valid

Table 4. Codestral: Average Textual Similarity Metrics

Protocol	#Conversations	Success Rate	ROUGE	CodeBLEU
ARP	77	100%	0.996	0.991
DNS	73	100%	0.908	0.900
HTTP	44	100%	0.935	0.922
ICMP	68	100%	1.000	1.000

packet generation (Table 5). When compared to Codestral, which also achieved a perfect success rate across all protocols, DeepSeek-Coder demonstrates nearly identical performance in terms of ROUGE and CodeBLEU scores.

Table 5. DeepSeek-Coder: Average Textual Similarity Metrics

Protocol	#Conversations	Success Rate	ROUGE	CodeBLEU
ARP	77	100%	0.995	0.991
DNS	73	100%	0.908	0.900
HTTP	44	100%	0.935	0.922
ICMP	68	100%	1.000	1.000

4.4 Analysis with Wireshark

To complement the textual evaluation of generated network conversations, we conducted a packet-level validation to assess the functional correctness and protocol compliance of the generated traffic. This analysis is critical for verifying that the generated scripts not only resemble real network interactions textually but also produce valid packets. Using Wireshark, we examined `.pcap` files generated from the models outputs. The validation focused on identifying malformed packets, protocol-specific warnings, and deviations from expected communication patterns. This process provides a deeper understanding of the model ability to generate operationally valid network traffic across different protocols.

Codestral. The packet-level analysis of Codestral confirmed a high degree of structural and semantic fidelity across all supported protocols, with no observed errors. The only warnings arose from the use of non-standard MAC addresses, specifically those with first-byte values that violate IEEE-assigned organizationally unique identifier (OUI) conventions. This non-compliance results from the data augmentation strategy employed to increase MAC address diversity during dataset generation. Although such addresses may trigger heuristic warnings in strict validation environments, they do not impact protocol correctness or traffic interpretability. Overall, the results demonstrate that Codestral generates high-quality, protocol-compliant network traffic.

Deepseek-Coder. The packet-level analysis of DeepSeek-Coder confirmed a high degree of structural and semantic fidelity across all supported protocols, consistent with the results observed for Codestral. As with Codestral, no protocol-specific errors were flagged by Wireshark, and all generated traffic was considered valid and interpretable. Additionally, DeepSeek-Coder exhibited the same issue related to non-standard MAC addresses, which stems from the data augmentation strategy used during dataset generation.

5 Conclusions and future work

This study demonstrates the effectiveness of the proposed methodology for generating network packets using LLMs fine-tuned on protocol-specific conversations synthesized with Scapy. The proposed approach enables network packets to be represented and generated as structured code, making them more accessible to LLMs. By representing packets as structured code, the approach leverages the strengths of LLMs in learning syntactic and behavioral patterns across diverse network protocols. The strong performance of both Codestral and DeepSeek-Coder in the textual similarity assessment highlights the viability of this representation for producing coherent and protocol-compliant outputs. The subsequent packet-level validation, conducted using Wireshark, confirmed the structural integrity of the generated traffic, with all packets conforming to protocol specifications, reinforcing the broader applicability of LLMs in network traffic generation tasks. The only minor deviation involved the use of non-standard MAC addresses, which did not affect protocol correctness or traffic interpretability.

In summary, the integration of Scapy-based protocol modeling with LLM fine-tuning represents a promising methodology for generating realistic and operationally meaningful network traffic. While the effectiveness of this approach depends on the capabilities of the underlying language model, our results demonstrate that, when paired with a sufficiently capable model, it can achieve high structural fidelity and protocol compliance. This methodology shows strong potential for applications in network simulation, automated testing, and cybersecurity research.

Future work should focus on expanding protocol coverage and increasing the complexity of generated network interactions. In particular, we aim to explore the use of reasoning-capable LLMs to enable the hierarchical learning of complex protocols from foundational ones. This direction opens the possibility of compositional protocol synthesis, where models generalize from basic primitives to more sophisticated, stateful behaviors. Additionally, extending the approach to encompass more complex and dynamic network scenarios, including video streaming, file sharing, and large-scale downloads, would provide a robust testbed for evaluating the models capacity to manage multi-protocol flows, session persistence, and real-time data exchange. Integrating this methodology with full-featured network simulation environments could also enable the development of interactive, LLM-driven tools for cybersecurity training, automated protocol analysis, and intelligent traffic generation in realistic settings, thereby advancing both research and practical applications in network systems.

Acknowledgements. This work was supported in part by the R&D activity program with reference TEC-2024/COM-504 and acronym RAMONES-CM, granted by the Comunidad de Madrid through the Directorate General for Research and Technological Innovation via Order 5696/2024, and by the Comunidad de Madrid under the project LINO (SI4/PJI/2024-00221) through the direct grant agreement for the promotion of research and technology transfer at Universidad Autónoma de Madrid.

References

1. Aceto, G., Ciuonzo, D., Montieri, A., Persico, V., Pescapé, A.: AI-powered internet traffic classification: Past, present, and future. IEEE Communications Magazine **62**(9), 168–175 (2024)
2. Aceto, G., Ciuonzo, D., Montieri, A., Pescapé, A.: Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges. IEEE Transactions on Network and Service Management **16**(2), 445–458 (2019)
3. Adeleke, O.A., Bastin, N., Gurkan, D.: Network traffic generation: A survey and methodology. ACM Comput. Surv. **55**(2) (jan 2022)
4. Agrawal, G., Kaur, A., Myneni, S.: A review of generative models in generating synthetic attack data for cybersecurity. Electronics **13**(2) (2024)
5. Ahmed, L.A.H., Hamad, Y.A.M., Abdalla, A.A.M.A.: Network-based intrusion detection datasets: A survey. In: 2022 International Arab Conference on Information Technology (ACIT). pp. 1–7 (2022)
6. Biondi, P.: Scapy: A python library for packet manipulation. `https://scapy.net/` (2025)
7. Bovenzi, G., Cerasuolo, F., Ciuonzo, D., Di Monda, D., Guarino, I., Montieri, A., Persico, V., Pescapè, A.: Mapping the landscape of generative ai in network monitoring and management. arXiv preprint arXiv:2502.08576 (2025)
8. Bovenzi, G., Cerasuolo, F., Montieri, A., Nascita, A., Persico, V., Pescapé, A.: A comparison of machine and deep learning models for detection and classification of android malware traffic. In: 2022 IEEE Symposium on Computers and Communications (ISCC). pp. 1–6 (2022)
9. Bovenzi, G., Di Monda, D., Montieri, A., Persico, V., Pescapè, A.: Classifying attack traffic in iot environments via few-shot learning. Journal of Information Security and Applications **83**, 103762 (2024)
10. Cheng, A.: PAC-GAN: Packet generation of network traffic using generative adversarial networks. In: 2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON). pp. 0728–0734 (2019)
11. Chu, A., Jiang, X., Liu, S., Bhagoji, A., Bronzino, F., Schmitt, P., Feamster, N.: Feasibility of state space models for network traffic generation. In: Proceedings of the 2024 SIGCOMM Workshop on Networks for AI Computing. pp. 9–17 (2024)
12. Chu, W., Guan, X., Cai, Z., Gao, L.: Real-time volume control for interactive network traffic replay. Computer Networks **57**(7), 1611–1629 (2013)
13. Delgado-Soto, J.A., López de Vergara, J.E., González, I., Perdices, D., de Pedro, L.: Gpt on the wire: Towards realistic network traffic conversations generated with large language models. Computer Networks **265**, 111308 (2025)
14. Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X., Lou, Y.: Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation (2023), `https://arxiv.org/abs/2308.01861`
15. Hugging Face: Hugging face – the ai community building the future. `https://huggingface.co/` (2024)

16. Jiang, X., Liu, S., Gember-Jacobson, A., Bhagoji, A.N., Schmitt, P., Bronzino, F., Feamster, N.: Netdiffusion: Network data augmentation through protocol-constrained traffic generation. Proceedings of the ACM on Measurement and Analysis of Computing Systems **8**(1), 1–32 (2024)
17. Kholgh, D.K., Kostakos, P.: PAC-GPT: A novel approach to generating synthetic network traffic with gpt-3. IEEE Access **11**, 114936–114951 (2023)
18. Lin, C.Y.: Rouge: A package for automatic evaluation of summaries. In: Text Summarization Branches Out: Proceedings of the ACL-04 Workshop. pp. 74–81. Association for Computational Linguistics (2004)
19. Meslet-Millet, F., Mouysset, S., Chaput, E.: Necstgen: An approach for realistic network traffic generation using deep learning. In: GLOBECOM 2022 - 2022 IEEE Global Communications Conference. pp. 3108–3113 (2022)
20. Nukavarapu, S.K., Ayyat, M., Nadeem, T.: Miragenet - towards a gan-based framework for synthetic network traffic generation. In: GLOBECOM 2022 - 2022 IEEE Global Communications Conference. pp. 3089–3095 (2022)
21. Papers with Code: State-of-the-art code generation on humaneval. `https://paperswithcode.com/sota/code-generation-on-humaneval` (2024)
22. Qu, J., Ma, X., Li, J.: TrafficGPT: Breaking the token barrier for efficient long traffic analysis and generation (2024)
23. Ravkine, M.: Can AI code? - results. `https://huggingface.co/spaces/mike-ravkine/can-ai-code-results` (2023)
24. Ren, S., Tu, Z., Feng, S., Lou, J., Yin, D.G., Zhang, S., Svyatkovskiy, A., Fu, S., Xu, M., Zhang, D., et al.: Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297 (2020)
25. Ring, M., Schlör, D., Landes, D., Hotho, A.: Flow-based network traffic generation using generative adversarial networks. Computers & Security **82**, 156–172 (2019)
26. Ring, M., Wunderlich, S., Scheuring, D., Landes, D., Hotho, A.: A survey of network-based intrusion detection data sets. Computers & Security **86**, 147–167 (2019)
27. Ruiz, M., Ruiz, M., Tabatabaeimehr, F., Gifre, L., López-Buedo, S., López de Vergara, J.E., González, Ó., Velasco, L.: Modeling and assessing connectivity services performance in a sandbox domain. Journal of Lightwave Technology **38**(12), 3180–3189 (2020)
28. Xuying Meng, Chungang Lin, Y.W., Zhang, Y.: NetGPT: Generative pretrained transformer for network traffic. arXiv (2023), `https://arxiv.org/pdf/2304.09513`
29. Yin, Y., Lin, Z., Jin, M., Fanti, G., Sekar, V.: Practical gan-based synthetic ip header trace generation using netshare. In: Proceedings of the ACM SIGCOMM 2022 Conference. pp. 458–472 (2022)