Master Thesis

Modelling IoT Network Behaviour and Enhancing Security Using AI Technologies

Submitted by: Kushal Prakash

First examiner: Prof. Markus Miettinen

Second examiner: Maurizio Petrozziello

Date of start: 09.12.2024

Date of submission: 16.06.2025

Statement

I confirm that I have written this thesis on my own. No other sources were used except those referenced. Content which is taken literally or analogously from published or unpublished sources is identified as such. The drawings or figures of this work have been created by myself or are provided with an appropriate reference. This work has not been submitted in the same or similar form or to any other examination board.

Signature: 

Date: 16.06.2025

Contents

List of Figures

List of Tables

Abbreviations

API Application Programming Interface

AUC–ROC Area Under the Receiver Operating Characteristic Curve

BLE Bluetooth Low Energy

CoAP Constrained Application Protocol

CPU Central Processing Unit

CSV Comma-Separated Values

GAN Generative Adversarial Network

HIDS Host-Based IDS

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IDS Intrusion Detection System

IoT Internet of Things

JSD Jensen–Shannon Divergence

KLD Kullback–Leibler Divergence

LaTeX Lamport TeX

LLM Large Language Model

LSTM Long Short-Term Memory

LoRaWAN Long Range Wide Area Network

ML Machine Learning

mDNS Multicast DNS

MQTT Message Queuing Telemetry Transport

NIDS Network-Based IDS

NS-2 Network Simulator 2

NS-3 Network Simulator 3

PCAP Packet Capture

REST Representational State Transfer

RNN Recurrent Neural Network

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

UI User Interface

VAE Variational Autoencoder

Wi-Fi Wireless Fidelity

XML eXtensible Markup Language

AML Adversarial Machine Learning

FPR False Positive Rate

FNR False Negative Rate

HMM Hidden Markov Model

PEFT Parameter-Efficient Fine-Tuning

WandB Weights & Biases

Chapter 1

Introduction

The proliferation of Internet of Things (IoT) devices from smart thermostats and security cameras in our homes to industrial sensors on factory floors has transformed both daily life and critical infrastructure. Yet these tiny, resource constrained systems, often relying on lightweight protocols such as MQTT, CoAP, or Zigbee, bring a host of security challenges. Attacks ranging from device spoofing and denial-of-service floods to subtle protocol abuses can slip past traditional rule-based defenses. To address this, many modern Intrusion Detection Systems (IDS) employ machine learning to spot anomalies in traffic patterns. However, these ML-based approaches typically suffer from high false-alarm rates when trained on limited or unrepresentative data [52][3].

Compounding the problem, real-world IoT traffic captures are often sparse, siloed by privacy concerns, or lack the diversity needed to expose uncommon or emerging threats. Classical traffic generators can fill in some gaps, but they struggle to reproduce the rich temporal and semantic correlations found in real deployments. Public testbeds like FIT IoT-LAB (Serot et al. [40]) offer valuable platforms for experimentation, yet it still rely on handcrafted scenarios rather than truly open-ended traffic synthesis.

Despite their sophistication, both modern ML-based traffic generators and classical, rule-driven tools often miss the mark. They tend to produce packet traces with rigid timing, low variability, and a polish that real IoT communications rarely exhibit. In practice, this means simulations can feel sterile missing the unexpected bursts, protocol quirks, and device jitters that make real networks so challenging to model and secure.

In this thesis, we introduce a new approach that harnesses large language models (LLMs) to capture and reproduce IoT device network signals, using those models to generate high-fidelity synthetic traffic. By fine-tuning architectures like LLaMA, Qwen, and Gemma on our own packet captures and, where available, established benchmark datasets, we assemble rich, hybrid corpora blending real and model synthesized logs. Crucially, our experiments demonstrate that augmenting IDS training data with LLM-generated samples not only lowers false-alarm rates but also measurably boosts detection accuracy confirming the practical value of synthetic traffic for real-world anomaly detection.

Beyond IDS augmentation, synthetic IoT traffic has many other applications: stress-testing network stacks, validating protocol implementations, powering digital twins of smart cities, and even assisting in automated firmware analysis. Because transformer-based LLMs excel at learning long-range dependencies and structured templates, they

offer a degree of flexibility and realism unmatched by traditional generators. Over the course of this work, we explore both the capabilities and limitations of LLM-based traffic synthesis, and demonstrate its impact on downstream security tasks in IoT environments.

By learning patterns from real IoT datasets, LLMs can:

- Simulate normal device behavior over time (e.g. sensor readings, heartbeat messages and state changes).
- Generate malicious or anomalous traffic to simulate attacks (e.g. spoofing, DDoS, flooding, etc).
- Emulate multi-device interaction patterns in smart environments.

This makes LLMs a powerful tool for network simulation, testbed creation, and IDS evaluation, offering greater flexibility, realism, and adaptability than traditional rule-based generators.

1.1 IoT Security Challenges.

Unlike traditional computers or servers, IoT devices are often deployed in large numbers like smart meters, sensors, cameras each with minimal CPU, memory, and power budgets. This scale and resource constraint make it difficult to ship and maintain robust security controls: many devices arrive with hard-coded credentials, unpatched firmware, or unencrypted communication simply because manufacturers optimize for cost and convenience. On top of that, the IoT landscape is highly fragmented, with wildly different hardware and software stacks lacking a universal security standard. Once these devices are plugged into the Internet often via open ports or weak protocols like plain HTTP they become attractive low-hanging fruit for attackers. Together, these factors (massive scale, insecure defaults, heterogeneous platforms, and public exposure) combine to make IoT environments far more vulnerable than conventional IT networks.

Research Contributions. This work makes several key contributions to both the scientific and development communities. First, we demonstrate that open-source large language models when fine-tuned on real IoT packet captures can generate synthetic traffic with the temporal quirks and structure needed to train more accurate anomaly detectors. Second, we provide an open-source framework (including code, model checkpoints, and tokenized datasets) that practitioners can use to reproduce our experiments or extend them to new device fleets and protocols. Third, by rigorously benchmarking multiple LLM architectures and downstream IDS performance, we offer concrete guidelines on model selection, sampling strategies, and data augmentation ratios insights that will accelerate future research in synthetic traffic generation and cybersecurity analytics. Ultimately, our thesis bridges a critical gap between cutting-edge techniques and the practical challenges of securing IoT ecosystems at scale.

1.2 Target State / Target Objectives

This research project, while grounded in academic inquiry, also has practical implications for cybersecurity, IoT system design, and AI-driven network monitoring. By identifying key stakeholders and end users, we clarify how this framework can create value and drive impact.

1.2.1 Researchers and Academics

Academic researchers in network security, artificial intelligence, and Internet of Things (IoT) stand to benefit directly from the framework developed in this work. By introducing a novel method for generating synthetic IoT traffic with open-source Large Language Models (LLMs), we offer:

- A reproducible and extensible architecture for exploring LLMs beyond traditional natural language tasks.
- Synthetic datasets that augment scarce real-world IoT captures, especially useful for anomaly detection and IDS training.
- Comparative evaluations of different LLMs in terms of signal fidelity and attack simulation, laying the groundwork for future research.

In this way, the system contributes both methodological advancements and valuable dataset resources to the research community.

1.2.2 Cybersecurity Professionals and Threat Analysts

For practitioners in cybersecurity and network defense, the ability to generate high-fidelity synthetic traffic is invaluable. The proposed framework enables:

- Simulation of complex attack scenarios, including zero-day exploits, in a controlled and safe environment.
- Improved IDS performance through training on synthetic signals that mimic real traffic patterns while avoiding privacy constraints.
- Benchmarking of anomaly detection models against a synthetic baseline that includes adversarially generated traffic(C. Stanford [6]).

These capabilities are particularly relevant for security testing, red teaming, and penetration testing when diverse, labeled datasets are otherwise unavailable.

1.2.3 IoT Developers and System Architects

Developers and architects of IoT systems often face challenges when testing device behavior under varied network conditions. This framework allows them to:

- Rapidly prototype network scenarios and evaluate device resilience under different loads and attack vectors.
- Emulate realistic communication patterns—spanning MQTT, TCP, UDP, and mDNS—helpful for designing scalable, robust IoT deployments.
- Validate security controls (e.g., rate limiting, anomaly detection, QoS enforcement) early in the development cycle without requiring a large fleet of live devices.

By generating synthetic traffic that reflects real-world behavior, IoT teams can identify vulnerabilities and tune performance before deploying hardware at scale.

1.2.4 AI and ML Engineers in Network Simulation

As LLMs are adapted for time-series and structured-signal modeling, this project demonstrates a compelling use case: treating network packets as tokens in a sequence. AI practitioners can use the framework to:

- Train or fine-tune LLMs on non-textual, structured data extending language models to new domains.
- Experiment with few-shot learning or prompt-tuning techniques for task-specific simulations.
- Apply similar methods to other cyber-physical systems (e.g., SCADA networks, smart grids), exploring the generalizability of sequence modeling beyond IoT.

In this way, the work bridges advanced AI methods with practical network simulation needs.

1.2.5 Intended Use: Academic Foundation with Applied Scalability

Although this system was developed primarily for academic exploration, its modular design and open-source components make it adaptable to production-scale environments. Key design features include:

- Seamless integration with real-world IDS platforms, network simulation testbeds, or digital-twin infrastructures(Tsai and Yang [45]).
- A path from proof-of-concept to deployment-ready synthetic traffic pipelines, thanks to reproducible training scripts and interpretable metrics.
- Opportunities for collaboration with research institutions, government cyber defense agencies, or industrial IoT testbeds to extend and refine the framework.

In summary, this work serves both as a research toolkit advancing our theoretical understanding of AI-driven network simulation and as a practical asset for building, testing, and securing IoT ecosystems.

1.3 Thesis Structure

This thesis unfolds over six chapters, each building on the last to guide you from problem statement to conclusions. In Chapter 1, we introduce the research goals, motivations, and scope of the study. Chapter 2 then dives into the necessary background, covering both IoT traffic analysis techniques and the fundamentals of large language models. In Chapter 3, we outline our system design, detailing how data is collected, processed, and prepared for model training. Chapter 4 takes you through the implementation phase, showing how the models were fine-tuned, how synthetic signals were generated, and how the IDS was constructed. Chapter 5 presents the key results, performance evaluations, and comparisons across real and synthetic datasets. Finally, Chapter 6 wraps up with a summary of our findings, reflections on limitations, and a look ahead at future research directions.

Chapter 2

Background

This chapter lays the theoretical foundation for the concepts central to this thesis, which investigates the use of Large Language Models (LLMs)(Vaswani et al. [47]) to simulate the behavior of IoT devices and enhance intrusion detection mechanisms. It begins by introducing the fundamental principles of IoT systems, including device communication protocols, network architectures, and the typical signal patterns generated by IoT devices. The chapter then explores the evolution and capabilities of open-source LLMs (LLaMA, Qwen, Phi-2, Gemma and Granite) with a focus on their application beyond natural language processing, particularly in structured data generation. Following this, the discussion highlights the role of synthetic data in network simulation, the principles behind intrusion detection in IoT contexts.
Through this exploration, the chapter aims to provide a comprehensive understanding of the technological and conceptual landscape underpinning this research. By situating the use of LLMs within the broader context of IoT security and simulation, it establishes a solid theoretical basis for the experimental methods, datasets, and evaluation metrics employed in later chapters.

2.1 Internet of Things (IoT) Architecture and Communication

The Internet of Things (IoT) encompasses a distributed network of 'smart' devices sensors, actuators, micro-controllers, and radios that collect, process and exchange data. By embedding these devices within physical objects, IoT systems enable automation, real-time monitoring, and adaptive control across diverse domains (for example, home automation, healthcare, industrial systems, smart cities). To simulate their behavior or secure their operations effectively, it is helpful to view IoT as a layered framework, each layer governed by particular functions and protocols.

2.1.1 Definition and Components of IoT

IoT architectures are often described in four conceptual layers:

- **Sensing Layer**: At the physical edge, sensors measure environmental parameters (temperature, humidity, motion, voltage), and actuators carry out commands (e.g., flipping a relay or rotating a servo). This layer produces the time-series signals or discrete events that drive IoT communication.
- **Network Layer**: This layer moves data from edge devices toward processing units or cloud services. It involves wireless or wired radios (Wi-Fi, Ethernet, LTE, etc.) and transport protocols (TCP, UDP, HTTP, TLS, MQTT). For instance, TCP or TLS-encrypted TCP (TLS 1.2) ensure reliable, ordered delivery for firmware updates or sensitive telemetry, while UDP handles quick, loss-tolerant data bursts. mDNS (Multicast DNS) often appears in local networks, allowing devices—say, a smart speaker—to announce themselves without manual configuration.
- **Data Processing Layer**: This layer aggregates, filters, and transforms those raw time-series readings or discrete events into higher-level features: downsampling noisy streams, computing rolling statistics, extracting protocol fields, or applying lightweight anomaly filters before transmission. The **Network Layer** then takes over, reliably moving this processed information from the edge toward fog or cloud services via Wi-Fi, Ethernet, LTE, or low-power radios. Here, transport protocols (TCP, UDP, HTTP, TLS, MQTT, mDNS, etc.) ensure everything from ordered delivery of firmware updates to low-latency telemetry bursts.
- **Application Layer**: Sitting atop the network, this layer comprises user-facing interfaces or cloud platforms that visualize data, manage devices, and execute analytics or control logic. Examples range from smartphone apps that toggle smart lights to enterprise dashboards orchestrating hundreds of industrial sensors.

Figure 2.1 illustrates these four layers stacked from edge to cloud. Notice how the Data Processing Layer bridges the raw, high-frequency signals of the Sensing Layer with the more structured, protocol-oriented flows of the Network Layer, enabling both efficient transport and more meaningful downstream analysis.

In this research, we concentrate primarily on the network layer, since it is at this level that packet flows and protocol interactions take shape making it essential for accurately simulating realistic traffic patterns and understanding the subtleties of communication behavior.

2.1.2 IoT Communication Protocols

Our traffic captures encompassed five key protocols—`TCP`, `UDP`, `HTTP/HTTPS`, `TLS 1.2`, and `mDNS`—each of which plays a distinct role in IoT communications and therefore shapes the synthetic data we generate.

- **TCP (Transmission Control Protocol)**: A connection-oriented, reliable protocol used when guaranteed delivery matters (e.g., pushing configuration updates or uploading logs to the cloud).
- **UDP (User Datagram Protocol)**: Connectionless and lightweight, ideal for time-sensitive data like frequent sensor readings, where occasional packet loss is tolerable.

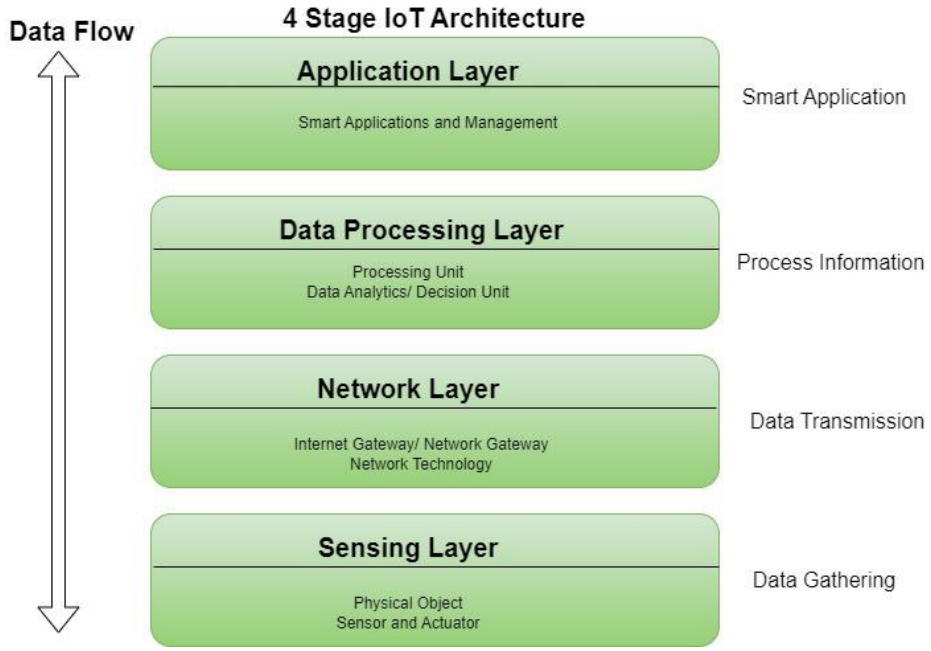


Figure 2.1: Four-layer IoT architecture: Sensing, Data Processing, Network, and Application.

Source: (GeeksforGeeks [15])

- **HTTP/HTTPS (HyperText Transfer Protocol over TLS)**: Common in RESTful API calls between IoT devices and cloud services; easy to implement but relatively verbose, so less efficient for constrained networks.
- **TLS 1.2 (Transport Layer Security)**: When endpoints exchange sensitive data (e.g., health metrics), TLS 1.2 wraps TCP sessions to encrypt payloads and prevent eavesdropping.
- **mDNS (Multicast DNS)**: Enables "zero-configuration" device discovery on a LAN. Devices like IP cameras frequently use mDNS to broadcast their presence and service endpoints.

2.1.3 IoT Network Characteristics

IoT networks differ significantly from traditional computing networks in several key ways:

- **Resource Constraints:** IoT devices often operate with minimal processing power, memory, and battery life. This limits the complexity of onboard security or protocol handling and necessitates optimized communication strategies(Weber [51]).
- **Communication Patterns:** Communication patterns in IoT networks vary, some devices send data at regular intervals such as a temperature sensor reporting every 10 seconds while others only transmit when a state change occurs, for

example, a motion detector triggering an alert. These differing behaviors can be modeled in LLMs either as structured time-series (for periodic communication) or as conditional generation tasks (for event-driven communication).

- **Device Heterogeneity:** IoT ecosystems include a vast array of devices, each with different capabilities, firmware, and protocols. This heterogeneity complicates network modeling and detection of anomalies.

These characteristics make realistic simulation particularly important for training intrusion detection systems, and this is where the use of LLMs to generate synthetic, protocol-aware traffic can provide valuable contributions.

2.2 Synthetic Network Traffic Generation and Simulation Methods

Simulating network traffic is a cornerstone of designing, testing, and evaluating modern network infrastructures. Whether we are benchmarking performance, validating protocols, or conducting cybersecurity research, having realistic traffic models helps ensure that our systems behave as expected before they are deployed. In the realm of the IoT, this task becomes even more critical and more challenging due to the sheer diversity of devices, the scale of deployments, and the constantly changing behavior of connected endpoints. Accurately modeling device activity and producing representative traffic traces are especially important when training and evaluating intrusion detection mechanisms. In the following sections, we review classical simulation tools and statistical generators, then explore how modern machine learning (ML) models particularly deep learning and Generative Adversarial Networks (GANs) have been applied to synthetic traffic generation, highlighting both their strengths and their pitfalls.

2.2.1 Consequences of Unrealistic Traffic Simulation

Training an IDS on synthetic traffic that doesn't faithfully mirror real-world behavior is like teaching a guard dog to bark at shadows: the more detached the training scenarios, the less reliable the watchdog becomes when real threats arrive(Fortinet [14]). If our simulated IoT traces lack the subtle timing variations, protocol quirks, or even the occasional malformed packet found in actual deployments, several problems emerge:

- **Blind Spots to Novel Attacks.** An IDS tuned only on "clean," overly regular traffic will struggle to recognize real attack patterns that fall outside its narrow training envelope. Sophisticated adversaries can exploit these gaps, slipping malicious payloads past a detector that has never seen the true complexity of device communications.

- **Explosion of False Alarms.** Conversely, the IDS may overreact to benign but unexpected behaviors—routine firmware updates, network hiccups, or unusual but harmless sensor bursts—triggering an avalanche of false positives. This noise buries genuine alerts and erodes operator trust, leading teams to ignore or disable the detector altogether.

- **Misallocation of Resources.** Every false alarm demands investigation automated or human, which wastes time, compute cycles, and operational budget. In critical environments like industrial control systems or health monitoring networks, chasing phantom threats can delay responses to real emergencies.
- **Poor Generalization Across Devices.** Without exposure to the full diversity of packet sizes, inter-arrival times, and protocol mixes, an IDS may only work well on the narrow set of devices it "knows." Deploying that same model to a different sensor type or firmware version often requires retraining from scratch.
- **Eroded Confidence in Machine Learning.** High error rates and brittle detectors reinforce skepticism about ML-based security. If synthetic data can't deliver reliable outcomes, practitioners may revert to manual rules or abandon advanced analytics altogether.

By contrast, when LLM-augmented simulations capture the messy reality of IoT traffic random jitter, protocol fallbacks, sporadic retransmissions, and more a trained IDS becomes both more sensitive to true anomalies and more robust against false alarms. In this way, realistic synthetic data isn't just a convenience; it's the cornerstone of trustworthy, scalable IoT security.

2.2.2 Traditional Network Simulation Methods

For decades, discrete-event simulators and traffic replay tools have formed the bedrock of network behavior modeling:

- **Packet-Level Simulators:** NS-2, NS-3, OMNeT++, and Cooja provide detailed, packet-by-packet modeling of network protocols, device interactions, and mobility patterns. Researchers use these simulators to craft complex topologies and observe protocol performance under various configurations.
- **Hybrid Testbeds:** Platforms like EmuFog, iFogSim, and FIT IoT-Lab combine virtualized network environments with real IoT hardware. This hybrid approach brings simulations closer to reality, but testbeds are often constrained by physical scale, hardware availability, and reproducibility challenges.
- **Traffic Replay Tools:** Utilities such as `tcpreplay` allow researchers to reinject previously captured packet traces into a live network. While this approach faithfully recreates known scenarios, it cannot generate new, unseen behaviors limiting its utility when exploring rare events or novel attack vectors.

Although these tools excel at deterministic, protocol level experiments, they struggle to produce the diverse, adaptive, and context-sensitive traffic patterns that reflect real-world IoT environments. In particular, simulating emergent behaviors such as devices adapting to network congestion or responding to external triggers often requires manual scripting or cumbersome parameter tuning.

2.2.3 Statistical and Rule-Based Synthetic Traffic Generators

Early synthetic traffic generators aimed to bridge the gap between raw simulators and real network data. By relying on statistical or rule-based methods, these approaches offer lightweight, controllable traffic models:

- **Poisson Processes:** A classic choice for modeling packet arrival rates, Poisson processes assume that events occur independently and at a constant average rate. While simple to implement, this model ignores burstiness and correlation between packets.
- **Markov Models:** By representing network states as nodes and transitions between them as probabilities, Markov chains capture some sequence information such as switching between "idle" and "active" states. However, they can become unwieldy when trying to encode the many contextual dependencies found in multi-protocol IoT traffic.
- **Rule-Based Generators:** Tools like Ostinato or Scapy let researchers script predefined packet flows with user-defined fields and timing. These generators are powerful for crafting specific scenarios (e.g., sending a custom TCP handshake), but they require deep domain knowledge to cover every nuance making it difficult to reproduce the variability of real device behavior.

Overall, statistical and rule-based methods provide good baseline traffic patterns, yet they often fail to capture nonlinear dynamics, encrypted flows, or device specific quirks that emerge in large scale, heterogeneous IoT deployments.

2.2.4 Machine Learning Approaches to Network Simulation

Recent advances in machine learning and deep learning have opened new avenues for synthetic network traffic generation. Instead of hand crafting models, these approaches learn IoT netowrk traffic patterns directly from real data and then generate novel sequences that mirror actual device behaviors. Below are some of the most prominent ML techniques:

- **Autoencoders:** By compressing packet features into a lower-dimensional representation, autoencoders can learn the "signature" of normal traffic. At inference time, they can reconstruct typical flows or highlight anomalies as deviations from the learned baseline.
- **Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) Models:** These sequence oriented architectures predict the next packet attributes—such as timing, size, or protocol fields based on historical context. RNNs/LSTMs excel at modeling temporal dependencies, making them well-suited for capturing periodic or bursty IoT traffic(Elman [11]).
- **Variational Autoencoders (VAEs):** Extending basic autoencoders with probabilistic latent spaces, VAEs generate diverse traffic samples by sampling from learned distributions. They offer more variety than deterministic models but sometimes produce samples of lower fidelity.

- **Generative Adversarial Networks (GANs):** GANs consist of two neural networks—a *generator* that proposes synthetic traffic samples and a *discriminator* that tries to distinguish generated traffic from real captures. Over repeated training loops, the generator improves until it produces traces that are increasingly difficult to differentiate from genuine traffic. Variants such as SeqGAN, TimeGAN, and NetGAN have been designed specifically for sequence modeling or graph-based network structures, addressing some of the challenges inherent in packet level data(Goodfellow et al. [16]).

Use Cases for GANs in Network Security

- Synthesizing both normal and malicious traffic to enrich intrusion detection training sets.
- Simulating adversarial scenarios such as stealthy scans or evasion attacks—to evaluate IDS robustness.
- Generating rare or zero-day attack patterns for supervised learning frameworks, where labeled data are otherwise unavailable.

2.2.5 Limitations of Current ML- and GAN-Based Methods

While ML techniques show great promise, they also introduce new challenges:

- **Structural and Semantic Consistency:** GANs and other generative models often reproduce statistical features (e.g., packet size distributions) but neglect protocol conformance resulting in invalid headers or broken TCP handshakes that would not occur in real networks.
- **Mode Collapse in GANs:** During training, GANs sometimes converge on a narrow subset of behaviors—producing synthetic traffic that lacks diversity and fails to cover the full spectrum of real-world events.
- **Training Complexity and Data Requirements:** Effective GAN training demands large, labeled datasets and careful tuning of hyperparameters. In many IoT contexts especially for malicious or rare behaviors labeled data are sparse, imbalanced, or proprietary.
- **Temporal Coherence:** Simple generative models without explicit time modeling struggle to maintain realistic timestamps, periodicities, or device-level state transitions (e.g., battery drain or sensor drift over time).
- **Modeling Multi-Device Interactions:** Most ML models focus on a single device's packet logs or flattened features. As a result, they cannot generate coordinated, multi-device traffic such as a smart home scenario where several devices interact through a local gateway.

In summary, although ML- and GAN-based traffic generators represent a significant step forward compared to purely statistical or rule-based approaches, they are not a panacea. Future work must blend classical simulation tools, domain specific knowledge, and advanced learning algorithms to create hybrid frameworks that faithfully reproduce both the micro-level details and macro-level patterns of IoT networks.

2.2.6 Opportunities and Motivation for Using LLMs

The challenges faced by existing machine learning based traffic generators underline a clear opportunity for Large Language Models (LLMs). Unlike GANs or RNNs, which are often constrained by narrow distributions or temporal coherence issues, LLMs are pretrained on vast corpora of both structured and unstructured sequences. This broad exposure equips them with an intuitive grasp of long range dependencies, discrete formatting rules, and syntactic structures qualities that are crucial when modeling network packets and protocol exchanges.

Moreover, LLMs can be steered with minimal examples or carefully crafted prompts, allowing them to produce traffic that closely mirrors real-world IoT behavior. For instance, a well-tuned prompt can coax an LLM into generating:

- Complete protocol sessions such as a multi-packet TCP/HTTP handshake with realistic headers and payloads.
- Time-series streams that resemble sensor outputs, capturing modalities like periodic heartbeat messages or bursty status reports.
- Malicious traces simulating specific attack campaigns ranging from simple port scans to more sophisticated injection attempts.

In this thesis, we explore how open-source LLMs (e.g., LLaMA, Phi-2, Qwen, Granite, and Gemma) can be fine-tuned or prompted to generate structured synthetic IoT network data. By simulating benign workflows, we aim to assess whether these models can fill in gaps left by traditional simulators and GAN frameworks ultimately improving the breadth and realism of datasets used for intrusion detection training and evaluation.

2.3 Large Language Models (LLMs)

Large Language Models (LLMs) have emerged as transformative tools in artificial intelligence, demonstrating remarkable capabilities not only in natural language understanding and generation but also in handling structured data, time-series modeling, code synthesis, and complex reasoning. Their strength comes from learning long range dependencies, discrete formatting rules, and contextual relationships traits that are especially valuable in IoT domains, where data are often sequential, heterogeneous, and noisy. In this section, we offer a concise overview of LLM architectures, explain key components of the Transformer model, and introduce the open-source LLMs we leverage in this research, highlighting why they are well suited for simulating IoT network behavior and generating synthetic traffic.

2.3.1 Overview of LLMs

At their core, LLMs are deep neural networks trained on massive corpora of text or structured sequences using self-supervised objectives (e.g., next-token prediction). By ingesting billions of tokens during pretraining, these models learn to encode syntax, semantics, and long-range dependencies into high dimensional representations. As a result, they can generalize from limited examples and adapt to new tasks through fine tuning or prompt engineering. This adaptability makes LLMs particularly attractive for domains like IoT, where data may be sparse or partially labeled, but patterns and protocols follow underlying structural rules.

2.3.2 Transformer Architecture Basics

The breakthrough enabling modern LLMs was the Transformer architecture (Vaswani et al. [47]), which replaced recurrent computations with fully parallel self-attention mechanisms. Key components include:

- **Multi-Head Self-Attention:** Each attention head learns to focus on different positions within the input sequence, capturing diverse contextual relationships simultaneously. This allows the model to "remember" how earlier packets in a traffic flow might influence later packets, even across long distances.
- **Positional Encoding:** Since attention alone does not encode token order, positional encodings inject explicit information about sequence position.
- **Feedforward Layers:** After attention, dense feedforward layers transform each token embedding independently, increasing the network's capacity to model complex patterns.
- **Layer Normalization and Residual Connections:** These architectural choices stabilize training for very deep models, ensuring that gradient signals propagate effectively through dozens or even hundreds of layers.

Figure 2.2 shows the Transformer as two mirror-image "stacks" of layers: an encoder on the left and a decoder on the right. The encoder begins by turning each input token into an embedding, adds a bit of positional "flavor," and then repeatedly applies self-attention (so every token can peek at every other token) and a small feed-forward network each wrapped in a residual "Add & Norm" block. The decoder does much the same but first masks out future tokens and then also attends to the encoder's outputs. Finally, its top layer is projected through a linear+softmax to produce the next token probabilities(Vaswani et al. [47]).

By stacking multiple Transformer blocks, researchers can build models containing billions of parameters that excel at learning both local and global sequence patterns.

2.3.3 Attention Mechanisms and Sequence Modeling

Self-attention allows LLMs to learn dependencies between distant elements in a sequence—a critical feature when modeling network traffic and device communication.

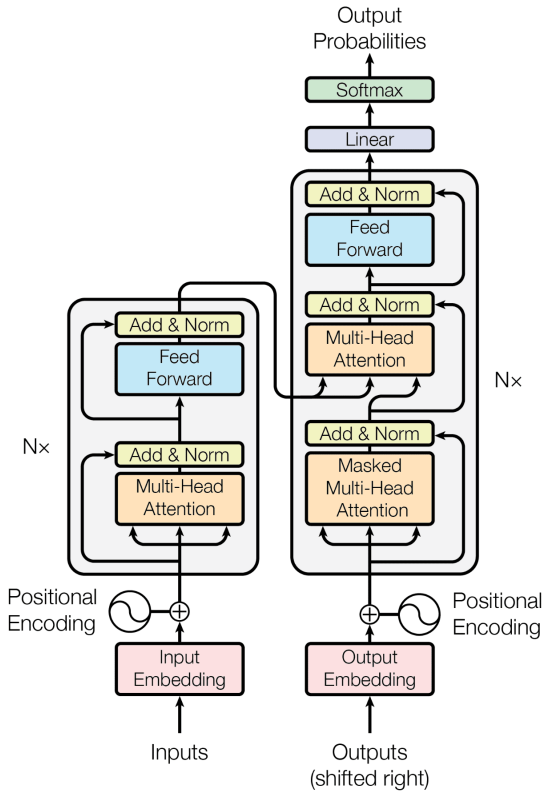


Figure 2.2: A classic Transformer encoder–decoder architecture

Source: (Vaswani et al. [47])

For instance, a temperature sensor's periodic heartbeat at one timestamp may trigger downstream control signals several seconds later. Attention mechanisms enable the model to link these events across a flexible context window. Consequently, LLMs can generate plausible IoT traffic patterns whether periodic telemetry or event-driven bursts without needing handcrafted rules for every edge case(Shaw et al. [42]).

2.3.4 Open-Source LLMs Used in This Work

Our experiments draw on a suite of modern, open-source LLMs chosen for their balance of performance, accessibility, and fine-tuning flexibility. Each model brings unique strengths in sequence modeling, structured data generation, and resource efficiency:

- **LLaMA (Meta):** LLaMA models are trained on carefully curated text and code datasets and optimized for general-purpose tasks. Their architecture scales from a few billion to tens of billions of parameters, offering strong reasoning and generation abilities. LLaMA's efficient implementation makes it easy to fine-tune on domain specific sequences, such as IoT packet logs.

  Figure 2.3 illustrates the Open-LLaMA transformer block, repeated N times. Each block begins by normalizing the input, then runs rotary positional encodings. A second sub-layer applies feed-forward network and another residual connection. After N of these layers, the model produces next-token probabilities.

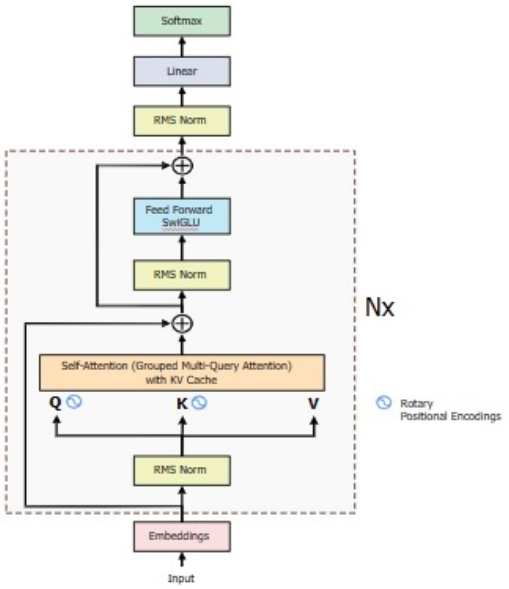


Figure 2.3: Open LLaMa architecture

Source: (Vignesh [48])

- **Qwen 2.5 (Alibaba):** Qwen is an open-source LLM family developed by Alibaba, featuring bilingual training on both Chinese and English corpora. Qwen's training regimen emphasizes robust performance on long context sequences, making it particularly adept at emulating multi-packet sessions. Furthermore, Qwen's architecture incorporates mixed attention patterns combining global and sliding window attention to handle high-throughput sequence generation with lower memory footprints. This design ensures that Qwen can capture both local packet-level details and broader session-level context, ideal for simulating realistic IoT traffic flows.

  Figure 2.4 illustrates the Qwen2.5-3B transformer block. The input first passes through a small Conv3D projection to form initial embeddings. This is followed by M layers of *windowed* multi-head self-attention each consisting of the attention sublayer. Finally, the output is normalized once more, projected by a linear layer, and fed into a softmax to produce next-token probabilities.

- **Phi-2 (Microsoft):** A compact yet powerful model (2.7 billion parameters), Phi-2 is trained on high quality text, code, and mathematics datasets with a focus on reasoning capabilities. Its modest size and efficiency make it well suited for rapid prototyping in resource-constrained environments, such as edge devices that require on-the-fly synthetic trace generation.

  Figure 2.5 depicts the Phi-2 decoder-only block, repeated N times. This simple, two-sublayer design with its combination of normalization, dropout, and residuals helps Phi-2 achieve stable training and efficient inference.

- **Gemma (Google):** Gemma is an instruction-tuned LLM optimized for respon-

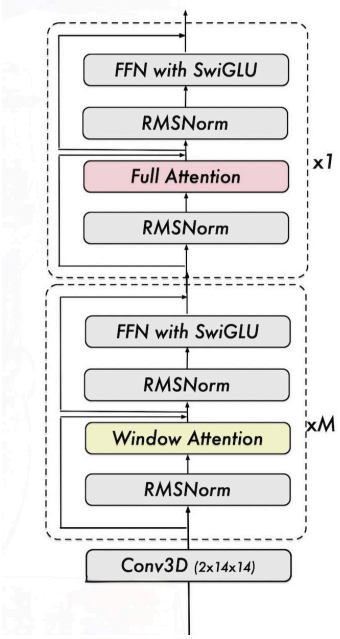


Figure 2.4: Qwen 2.5 architecture

Source: (Team [44])

sible AI development and downstream alignment. It excels at structured task completion through prompt engineering and reinforcement learning. In our work, Gemma's ability to follow complex instructions helps generate customized protocol interactions, such as crafting specific CoAP exchanges or simulating TLS handshakes with realistic certificate fields

Figure 2.6 depicts the Gemma-2B decoder-only block, which is stacked N times. Each block starts by mapping tokenized text through a token embedding layer and adding sinusoidal positional embeddings. After all N layers, a final LayerNorm precedes a linear output layer and softmax, yielding the next-token probabilities. This simple, uniform pattern of normalization, attention, and feed-forward with consistent residual connections gives Gemma-2B its balance of stability and generative power.

- **Granite (IBM Research):** The Granite family focuses on enterprise grade AI with an emphasis on structured, domain-specific, and procedural text. These models are trained to preserve protocol semantics and maintain anomaly detection capabilities, making them particularly suited for simulating both benign and adversarial IoT signals. Granite's training process also incorporates reproducibility and safety measures, aligning with ethical guidelines for dual-use LLMs in cybersecurity research(Walter [50]).

  Figure 2.7 shows the Granite data-ingestion and pre-processing pipeline, which transforms a raw text corpus into high-quality, model-ready inputs. This rigorous pipeline guarantees that Granite trains on diverse, non-redundant, and policy compliant text.

Each of these open-source LLMs provides a different trade-off between model size,

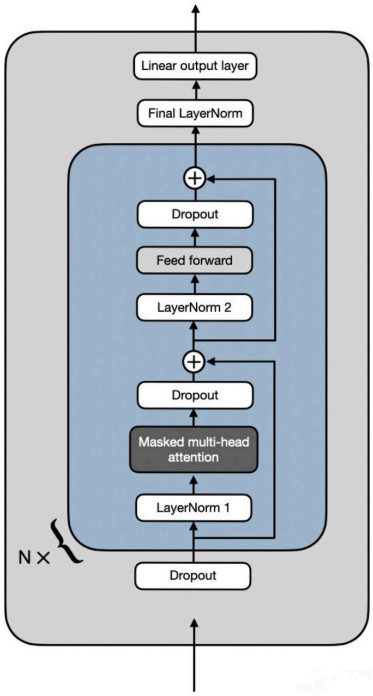


Figure 2.5: Phi-2 architecture

Source: (Microsoft Research [32])

inference speed, and sequence modeling prowess. By experimenting with LLaMA, Qwen, Phi-2, Gemma, and Granite, we aim to identify which architectures best capture the nuances of IoT packet flows, from periodic telemetry to stealthy intrusion patterns.

2.3.5 Application of LLMs Beyond Natural Language

While Large Language Models (LLMs) are most famous for their prowess in generating and understanding human language, they have demonstrated surprising versatility in other domains where data can be represented as sequences of tokens. In particular:

- **Structured Data Generation:** By learning how fields and delimiters interact, LLMs can output valid JSON, CSV, or other structured formats. In the IoT context, this means they can recreate the layered structure of packet logs or device telemetry generating headers, payloads, and metadata that conform to real-world formats.
- **Code Generation and Parsing:** Models such as Codex or CodeLLaMA, which are trained on large code repositories, develop an inductive bias toward syntax and logical structure. Since network protocols often resemble mini-programs (for example, a sequence of bytes representing a TCP handshake), these LLMs can simulate protocol exchanges by "writing" packets much like they would generate function calls or loops in code.
- **Time-Series Forecasting:** When numerical sensor readings or discretized telemetry data are tokenized, LLMs can learn temporal dependencies and forecast future

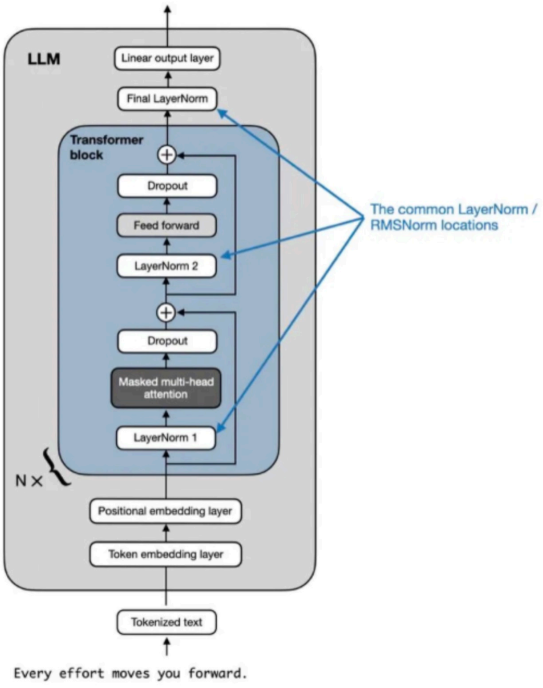


Figure 2.6: Gemma 2B architecture

Source: (Koninti [23])

values. For example, a model fine-tuned on hourly temperature readings can produce realistic sequences that mimic daily fluctuations or occasional spikes making them useful for simulating periodic or bursty IoT signals.

By leveraging these capabilities, we can ask an LLM to produce realistic packet traces, mimic the structure of a JSON-encoded status report, or predict sensor outputs over time—tasks that go far beyond standard text completion.

2.3.6 Few-Shot Learning and Prompt Tuning for Domain Specific Tasks

One of the most appealing features of modern LLMs is their ability to generalize from a handful of examples, a phenomenon often called *few-shot learning* or *in-context learning*. In IoT security scenarios where labeled datasets are scarce, attack types are heterogeneous, and individual devices behave differently this few-shot capability becomes especially valuable. Through careful prompt design or lightweight fine-tuning, an LLM can be guided to:

- **Generate Protocol-Specific Traffic:** For example, a prompt such as

  ```
  "Generate network traffic with following parameters....."
  ```

  can produce a sequence of HTTP request packets with realistic headers, timestamps, and JSON-formatted payloads.

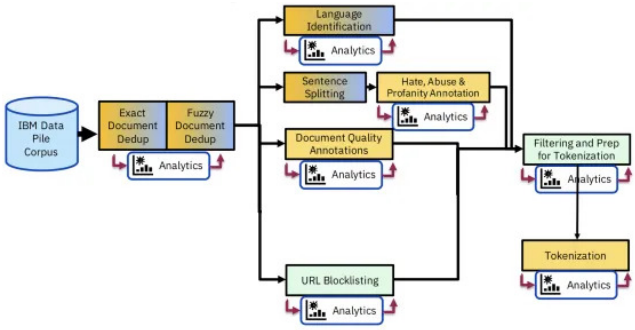


Figure 2.7: Granite 3.3 pipeline

Source: (Manav [31])

- **Simulate Anomalies and Intrusions:** By extending the prompt—e.g.,

  ```
  "Generate network traffic with following parameters for DoS attack"
  ```

  —the model can generate a blended trace that includes both benign telemetry and malicious traffic, providing a rich dataset for testing intrusion detection systems.

- **Emulate Multi-Device Interactions:** With a more elaborate prompt, LLMs can produce coordinated behaviors between multiple endpoints for instance, a smart lock reporting status changes to a central hub, which then triggers a security camera to start streaming. This results in synthetic traffic that reflects realistic, cross-device workflows without manually scripting each individual packet.

Through prompt tuning or instruction-based fine-tuning, LLMs adapt to domain-specific patterns with just a few exemplars. This thesis explores how such techniques enable the generation of structured synthetic IoT signal sequences and network traffic simulations. Because LLMs excel at capturing long-range context and maintaining syntactic correctness, they offer a promising avenue for creating diverse, realistic datasets that drive advances in IoT intrusion detection, adversarial testing, and network behavior modeling.

2.4 Synthetic Data Generation for Network Simulation

2.4.1 Need for Synthetic IoT Traffic

As IoT deployments continue to grow in both scale and complexity, designing effective cybersecurity solutions particularly Intrusion Detection Systems (IDS) relies heavily on having access to representative network data. In practice, however, high-quality IoT traffic captures are rare, often out of date, and typically lack the behavioral diversity needed for robust training. Consequently, generating synthetic IoT traffic becomes not just a convenience but a necessity. By creating realistic, customizable traffic traces,

researchers can simulate a broad spectrum of normal and malicious behaviors, enabling more accurate IDS evaluation and faster iteration during model development.

2.4.2 Limitations of Existing Datasets

Several publicly available datasets have been used for IDS benchmarking, yet each falls short when it comes to authentic IoT traffic:

- **BoT-IoT**(Moustafa and Salah [33]): Generated in a virtualized environment that imitates various IoT devices under both benign and attack conditions (e.g., DoS, DDoS, probing). While convenient, its rigid structure makes it difficult to capture the full range of real-world device behaviors and protocol quirks.
- **NSL-KDD**(Tavallaee et al. [43]): An updated version of the classic KDD '99 dataset, focusing primarily on conventional IT networks. It does not include IoT-specific communication patterns or modern threats such as botnet-driven low-and-slow attacks.
- **CICIDS2017**(Sharafaldin et al. [41]): Contains diverse attack scenarios in a mixed corporate network, but lacks the lightweight, constrained traffic typical of IoT protocols like MQTT, CoAP, or mDNS.

Beyond outdated protocol usage and limited protocol variety, these datasets often:

- Fail to capture device-specific behaviors (e.g., a temperature sensor's periodic broadcasts versus a motion detector's event-driven bursts).
- Contain imbalanced or overly synthetic attack samples that models learn too easily.
- Lack temporal continuity and protocol-layer correlations required for training sequence-based IDS models.

Because of these shortcomings, IDSs trained exclusively on existing public datasets may struggle when deployed in dynamic, real-time IoT environments. A more flexible synthetic data generation framework can address these gaps by producing realistic, diverse, and context-aware traffic tailored to specific research needs(Vohra [49]).

2.4.3 Importance of Realism and Diversity in Data for IDS Training

For an IDS to distinguish benign from malicious traffic reliably, training data must be both realistic reflecting structural and semantic properties of actual IoT traffic and diverse—covering a wide range of protocols, device types, and threat vectors. In practice, this means:

- *Capturing the spectrum of normal behavior:* Different devices (e.g., smart thermostats, security cameras, industrial sensors) exhibit unique traffic patterns. Synthetic data should mirror these nuances, from routine heartbeats to firmware-update bursts.

- *Including adaptive adversarial traffic:* Real attackers may perform coordinated scans, low-rate DDoS, or lateral movements across devices. Synthetic datasets must emulate these tactics to prevent IDSs from only recognizing textbook attacks.
- *Preserving temporal characteristics:* Packet inter-arrival times, protocol negotiation sequences, and traffic bursts often carry subtle cues that distinguish normal operation from an attack. Maintaining such temporal fidelity is critical for sequence-based detection models.

When data lack realism or diversity, IDS models risk overfitting to synthetic artifacts or missing novel threats in production. Therefore, a well-crafted synthetic traffic generator must balance the fidelity of real-world captures with the flexibility to introduce new, rare, or evolving attack patterns.

2.4.4 Existing Methods for Traffic Generation

Statistical and Rule-Based Models Traditionally, researchers have relied on statistical distributions and deterministic rules to synthesize network traffic:

- **Poisson Processes:** Often used to model packet arrival rates under the assumption of a memoryless, constant-rate process. While simple to implement and useful for baseline load generation, Poisson models cannot capture protocol semantics (e.g., TCP handshakes) or device-specific burst patterns.
- **Markov Chains:** By representing packet types or protocol states as discrete nodes with transition probabilities, Markov models can emulate sequential behavior. Hidden Markov Models (HMMs) further allow for unobserved "hidden" states. However, these models become unwieldy when attempting to encode the many contextual dependencies found in heterogeneous IoT traffic.
- **Rule-Based Traffic Generators:** Tools such as `Scapy` or `Ostinato` let users script custom packet flows by specifying protocol fields, timing, and payloads. Although powerful for crafting specific scenarios, rule-based generators demand deep protocol expertise and struggle to scale across multiple devices and attack types.

Overall, statistical and rule-based methods offer light computational overhead and tight control over traffic parameters. Yet they often fall short in adapting to new protocols, modeling long-term dependencies, or generating realistic encrypted and application-layer traffic.

ML-Based Traffic Generation Models The rise of machine learning has introduced data-driven techniques capable of learning from real captures and producing novel synthetic samples. Key approaches include:

- **Recurrent Neural Networks (RNNs) / LSTMs:** Designed for sequential data, RNNs and LSTMs can predict the next packet's features (size, protocol,

timing) based on historical context. They perform well on short-term dependencies but may struggle to maintain coherence over longer sequences or capture abrupt context switches typical in IoT networks(Elman [11].

- **Variational Autoencoders (VAEs):** VAEs compress traffic features into a continuous latent space and sample from this distribution to generate new traffic. While VAEs provide controlled variability, they sometimes produce traces that lack the structural integrity of real network flows.
- **Generative Adversarial Networks (GANs):** GAN-based frameworks such as NetGAN or TimeGAN—pair a generator (which synthesizes traffic samples) with a discriminator (which learns to distinguish real from fake). This adversarial setup encourages the generator to produce increasingly realistic traffic(Goodfellow et al. [16]). GANs are appealing because they:
  - Learn directly from unlabeled data.
  - Generate diverse and novel patterns that extend beyond the training set.
  - Can simulate both benign and malicious sequences for comprehensive IDS testing.

  However, GANs also face significant challenges:
  - *Mode Collapse:* The generator may converge to a limited set of output patterns, reducing diversity and failing to reflect the full range of real IoT behaviors.
  - *Lack of Structural Guarantees:* Although GANs can mimic statistical distributions, they may produce packets that violate protocol specifications (e.g., malformed TCP headers) or overlook application-layer context.
  - *Training Instability:* GANs are notoriously sensitive to hyperparameter settings and can suffer from unstable convergence, making reproducible results difficult to obtain.
  - *Temporal Inconsistencies:* Without explicit modeling of time, GAN-generated traffic may exhibit unrealistic inter-packet delays or break logical protocol exchanges (e.g., sending HTTP payloads before completing a TCP handshake).

In summary, while machine learning approaches particularly GANs offer powerful mechanisms for generating realistic, diverse traffic, they still struggle with preserving protocol-level semantics, ensuring training stability, and capturing multi-device interactions. Future work should explore hybrid strategies that combine the interpretability of classical methods with the generative power of modern LLMs and deep learning models.

2.4.5 Comparison with LLM-Based Generation

Large Language Models (LLMs) introduce a fundamentally different approach to synthetic traffic generation by treating communication logs, packet sequences, or protocol

exchanges as structured, text-like sequences. Rather than modeling individual packet features or relying on handcrafted rules, LLMs learn entire protocol conversations end-to-end preserving context and syntax over long sequences. Key advantages include:

- **Contextual Awareness:** LLMs can produce multi-step transactions or protocol sessions in which dependencies span dozens or even hundreds of tokens. For example, a single prompt can generate a complete TCP handshake followed by an HTTP request and response, ensuring that each step logically follows the previous one.
- **Structured Output Generation:** Through prompt engineering, LLMs can be directed to output data in formats like JSON, HTTP headers, or TCP dump entries that mirror real packet structures. This means that, with minimal effort, an LLM can generate a valid sequence of protocol fields, payloads, and timestamps that align with real-world captures.
- **Few-Shot Learning:** Instead of retraining a model from scratch for every new device or protocol variant, LLMs can be conditioned on a handful of examples to generalize behavior. Providing just a few samples of thermostat readings or CoAP exchanges, for instance, allows the model to produce similar traffic patterns without extensive additional training.
- **Semantic Control:** By carefully crafting prompts, researchers can steer LLMs to generate very specific types of traffic. One could instruct the model to simulate a thermostat sending periodic data, or to create malformed HTTP GET requests akin to a bot's attack signature. This level of semantic control enables rapid adaptation to emerging protocols or attack strategies.

In contrast, methods based on GANs or LSTMs often require domain specific retraining and complex hyperparameter tuning each time a new traffic pattern is needed. LLMs, on the other hand, rely primarily on prompt tuning or lightweight fine-tuning to emulate different protocol formats, periodic signals, or intrusion signatures making them especially well suited for fast, adaptive simulation pipelines.

Synthetic data generation is a cornerstone for advancing IoT network research and IDS development. While statistical models and traditional machine learning approaches have advanced the field, they frequently struggle to produce traffic that is both realistic and protocol-compliant. LLMs, with their ability to model long-range dependencies and generate structured outputs, offer a new frontier in synthetic traffic generation. By combining structural fidelity with semantic flexibility, LLMs can simulate complex IoT environments and enrich datasets for training robust, real-world-ready intrusion detection systems.

2.5 Intrusion Detection Systems (IDS) in IoT Networks

As IoT devices spread across critical infrastructures smart homes, healthcare systems, manufacturing floors, and urban environments securing these networks becomes a top

priority. Traditional defenses like firewalls or antivirus software often fall short or simply cannot be deployed on resource-constrained IoT endpoints. Intrusion Detection Systems (IDS) step in to provide a dynamic, intelligent layer of defense by flagging unauthorized access attempts or unusual behavior. In this section, we first describe the main types of IDS and how they can be applied to IoT settings. We then explore the unique challenges of building effective IoT IDS, and finally, we discuss how AI and machine learning can help overcome these hurdles(Roesch [36]).

2.5.1 Types of IDS

IDS can be classified based on two axes: how they detect threats and where they are deployed.

A. Detection Strategy

- **Signature-Based IDS:** These systems compare incoming traffic against a library of known attack "signatures" (for example, a specific sequence of packet headers or payload patterns). When a match is found, the IDS raises an alert (Denning [10]).
  Pros: Low false-positive rate for familiar attacks and straightforward to update with new signatures.
  Cons: Incapable of spotting novel or zero-day exploits, and signatures must be continuously updated to remain effective.

  As shown in Figure 2.8, a signature-based IDS operates much like a referee with a playbook of forbidden moves: it only reacts when it sees a known pattern of malicious behavior, otherwise it lets benign traffic continue unimpeded.

- **Anomaly-Based IDS:** These solutions learn a model of "normal" network behavior using statistical thresholds, heuristic rules, or ML models and flag any significant deviations as potential intrusions(Scarfone and Mell [39]).
  Pros: Can detect unknown attacks and subtle deviations from expected patterns.
  Cons: Tend to produce more false positives, especially when legitimate device behavior changes (e.g., a new firmware update alters traffic patterns). Their accuracy highly depends on having representative training data.

  Figure 2.9 illustrates how an anomaly-based IDS learns what "normal" looks like and only raises an alarm when traffic drifts beyond acceptable limits adapting its baseline for minor shifts but sounding the alert on significant, potentially malicious deviations.

- **Hybrid IDS:** By combining signature-based detection with anomaly-based modeling, hybrid systems aim to achieve both high accuracy for known threats and adaptability to new ones(Paxson [34]).
  Pros: Improved coverage, since known attacks are caught by signatures while unknown patterns trigger anomaly flags.
  Cons: More complex to build and maintain, with higher computational overhead.

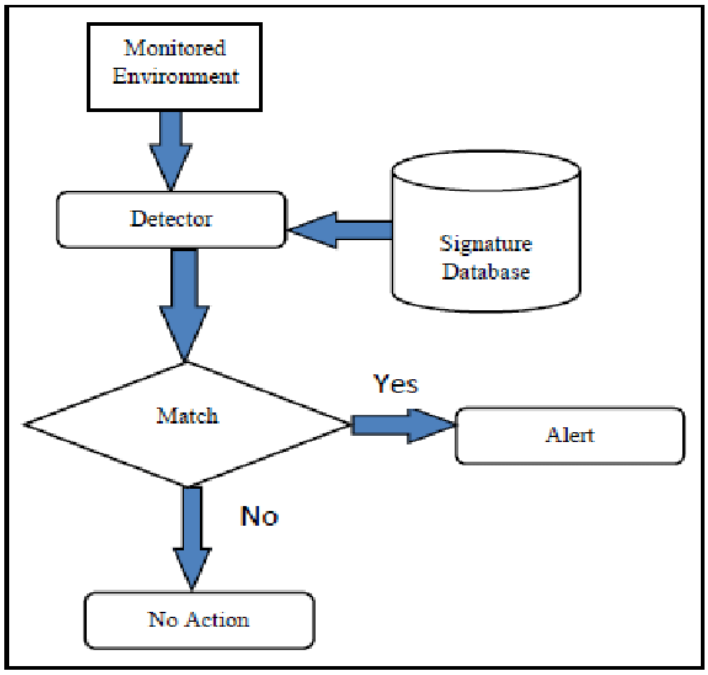


Figure 2.8: Signature based methodology architecture

Source: (Agrawal [2])

Figure 2.10 illustrates how a hybrid IDS leverages both signature matching for known threats and anomaly detection for novel attacks enabling rapid alerts while also evolving its knowledge base by generating new signatures from previously unseen malicious patterns.

B. Deployment Location

- **Host-Based IDS (HIDS):** Installed directly on an IoT device, HIDS monitors internal logs, system calls, and configuration files for signs of compromise(Kim and Spafford [22]). *Suitability:* Often impractical for lightweight IoT endpoints, which lack spare CPU cycles, memory, or storage. *Challenges:* Device heterogeneity, proprietary firmware, and minimal standardization make developing a one-size-fits-all HIDS nearly impossible across diverse IoT vendors.

- **Network-Based IDS (NIDS):** Placed at strategic network chokepoints typically at gateways or edge routers, a NIDS inspects packet flows and traffic patterns to detect anomalies(Liao et al. [27]). *Suitability:* More feasible in IoT scenarios, especially when devices connect through a central hub or a star topology. *Challenges:* Must cope with encrypted payloads (e.g., TLS/DTLS), fragmented packets, and rapidly fluctuating device counts. High churn and transient connections can also complicate stateful analysis.

  Figure 2.11 contrasts two IDS deployment strategies: a NIDS sits "in the wire," watching every packet that flows between the Internet and hosts, while HIDS installs lightweight sensors on each device to scrutinize internal events (logs, file access, process behavior) and relay alerts to a central controller.

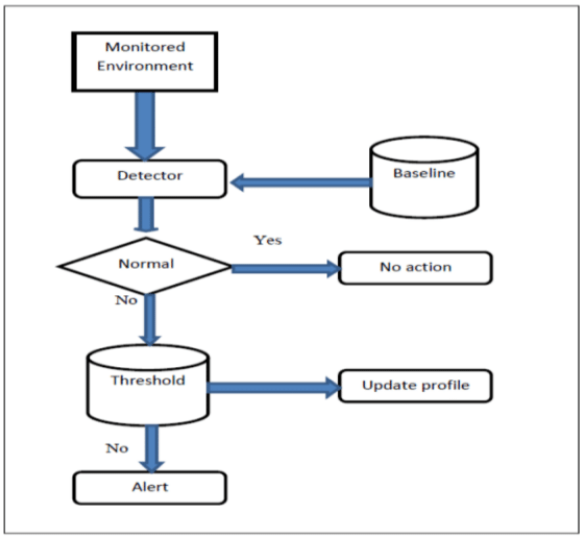


Figure 2.9: Anomaly based methodology architecture

Source: (Agrawal [1])

2.5.2 Challenges in IoT IDS Design

Designing an IDS that works well in IoT environments involves overcoming several distinctive hurdles:

High False Positives: Anomaly-based detectors are vulnerable to misclassifying legitimate changes—such as a new user routine or a time-of-day shift as malicious. In homes or factories, traffic patterns can change frequently (e.g., a shift from daytime to nighttime sensor polling), leading to noise in the alerts.

Lack of Labeled Data: Supervised learning approaches require large, well-annotated datasets of both benign and malicious traffic. In IoT, however, collecting and labeling attack traces is laborious, potentially violates privacy regulations, and often yields few examples of rare or stealthy intrusions.

Device and Protocol Diversity: IoT ecosystems use an array of lightweight communication protocols (such as MQTT, CoAP, Zigbee, and BLE), and each vendor may implement proprietary message formats. Building generalizable IDS models that can handle this heterogeneity is challenging.

Limited Resources: Many IoT endpoints cannot run complex detection algorithms on-device, due to restricted CPU, memory, or battery life. Consequently, IDS solutions often rely on centralized or edge-based processing, which introduces trade-offs between detection latency and energy consumption.

Encrypted and Fragmented Traffic: With TLS or DTLS becoming more common in IoT, deep-packet inspection can become impossible without decrypting traffic an approach that undermines privacy. Fragmented or short-lived sessions also complicate feature extraction and flow reconstruction.

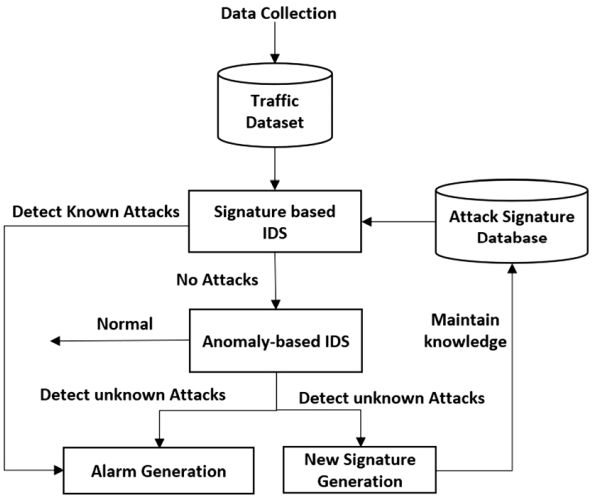


Figure 2.10: Hybrid based intrusion detection system

Source: (Bangui et al. [4])

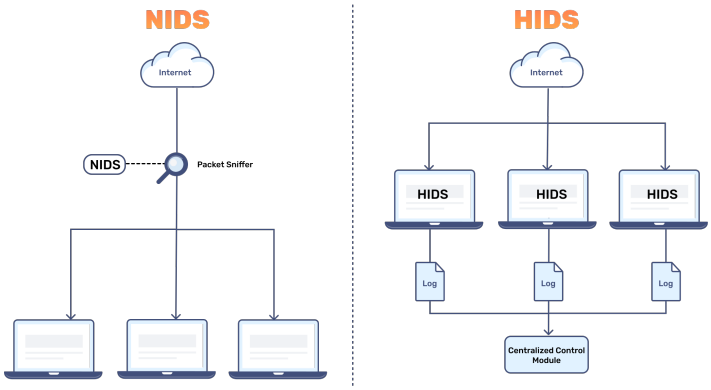


Figure 2.11: NIDS vs HIDS detection system

Source: (bunny.net Academy [5])

Dynamic Behavior and Mobility: Devices in smart homes or healthcare settings may join or leave the network frequently, switch between Wi-Fi and cellular, or change their communication patterns based on context (e.g., battery-saving modes). Defining a stable "normal" baseline under such conditions is difficult.

2.5.3 AI in Intrusion Detection

To address these challenges, many researchers turn to Artificial Intelligence (AI) and Machine Learning (ML) to build IDS that can learn from evolving traffic patterns and adapt over time.

A. Machine Learning and Deep Learning Techniques Classical ML Approaches:

- *Random Forests, Decision Trees, Support Vector Machines:* These algorithms offer lightweight, interpretable models for distinguishing normal from abnormal

traffic when provided with well-engineered features.

- *Clustering Methods (K-Means, DBSCAN):* Unsupervised methods that group similar traffic patterns and flag outliers as anomalies. Useful when labeled data are scarce, but often require manual tuning of cluster counts or distance thresholds.

Limitations: Traditional ML models demand careful feature engineering and typically struggle to capture time-series dynamics or complex packet-level interactions inherent in IoT traffic.

Deep Learning Approaches:

- *Convolutional Neural Networks (CNNs):* Although commonly associated with images, CNNs can learn spatial patterns in aggregated flow statistics or multidimensional packet feature arrays (e.g., byte histograms). This helps detect anomalies like unusual payload distributions or protocol misuse.

- *Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) Models:* Designed to model sequences, RNNs/LSTMs are well suited for capturing temporal dependencies—such as repeated login attempts, slow-scan attacks, or periodic sensor broadcasts. They can detect subtle timing anomalies that might indicate an ongoing intrusion(Elman [11].

- *Autoencoders:* By learning to compress and then reconstruct normal traffic patterns, autoencoders can flag high reconstruction errors as anomalies. They require only benign data for training, making them practical when malicious samples are rare.

- *Hybrid and Ensemble Models:* Combining deep architectures with traditional classifiers can yield more robust performance. For example, a CNN can extract high-level features from packet payloads, which are then fed into an SVM for lightweight classification.

B. Feature Extraction from IoT Data Designing an effective AI-driven IDS hinges on identifying the most discriminative features from raw IoT traffic(Sarhan et al. [38]). Useful features include:

- **Packet-Level Features:** Packet size, inter-arrival time, header fields (source/ destination IP, port numbers, TCP flags), and TTL values can reveal anomalies such as spoofed IP addresses or unusual flag combinations.

- **Flow-Level Features:** Aggregated statistics over a session number of packets, session duration, total bytes transferred, and protocol type help differentiate between normal usage (e.g., periodic sensor uploads) and malicious behavior (e.g., data exfiltration).

- **Temporal Patterns:** Metrics like periodicity, burstiness, or unexpected silence (e.g., a normally chatty device suddenly goes quiet) can signal compromise. Time-series features are especially important for detecting stealthy "low-and-slow" attacks that avoid volume-based thresholds.

- **Device Behavior Metrics:** CPU or memory usage spikes, frequency of configuration changes, and anomalous command sequences may indicate device tampering or malware presence. Tracking deviations from a device's historical usage profile can surface subtle intrusions.

Because many IoT protocols encrypt payloads or use proprietary formats, IDS designers often rely on metadata (timing information, packet sizes) rather than content inspection. In these cases, flow-level and timing-based summaries become essential for detecting abnormal behavior without decrypting or parsing payloads(Sarhan et al. [38]).

Intrusion Detection Systems are indispensable for ensuring IoT security, yet they face significant design and deployment challenges. Signature-based IDS are efficient at catching known threats but fail when attackers develop new exploits. Anomaly-based and AI-driven approaches offer adaptability but often suffer from false alarms and data scarcity. As IoT ecosystems become more complex, incorporating AI particularly deep learning and Large Language Models for traffic simulation presents a promising path forward. By training IDS on richer, more realistic datasets (including LLM-generated synthetic traffic), researchers can build systems capable of spotting both familiar and novel attacks in real time.

2.6 Adversarial Use of AI and LLMs in Cybersecurity

While AI and Large Language Models (LLMs) offer powerful capabilities for bolstering cybersecurity defenses, their dual-use nature means they can also be weaponized by malicious actors. This is particularly concerning in IoT environments, where devices often operate with minimal oversight and limited security measures. In this section, we explore how adversaries might leverage AI and LLMs to craft sophisticated attacks, the risks these technologies introduce, and the challenges of defending AI-based Intrusion Detection Systems (IDS) against adversarial machine learning.

2.6.1 Dual-Use Nature of LLMs

LLMs are designed to learn broad patterns in sequential data and generate coherent, contextually appropriate outputs. This flexibility is invaluable for tasks like traffic simulation and protocol emulation, but it also allows attackers to repurpose the same capabilities for offense with little additional effort.

Examples of Dual-Use Capabilities:

- **Benign Use:** Simulating realistic IoT traffic, detecting anomalies, or creating training datasets for IDS.
- **Malicious Use:** Automatically crafting adversarial network packets, mimicking legitimate device behavior to evade detection, or scripting polymorphic malware that can mutate to bypass signature-based defenses.

Because LLMs can be fine-tuned or prompted using natural language or a few traffic samples, an attacker could:

- Generate protocol-compliant yet malicious traffic designed to slip past deep packet inspection.
- Emulate normal device behavior while exfiltrating data or probing internal systems.
- Automate fuzzing and protocol discovery by producing sequences that explore edge cases in network stacks.

The risk is amplified by the open-source availability of powerful models (e.g., LLaMA, Qwen, Granite), which lowers the barrier for adversaries with limited resources.

2.6.2 Potential for Generating Adversarial Traffic

When trained or prompted on structured network data (such as TCP logs, HTTP headers, or JSON payloads), LLMs can generate synthetic traffic with high fidelity to real protocols. While this is advantageous for simulation and training, it also enables automated adversarial traffic generation with unsettling capabilities:

- **Mimicking Normal Behavior:** An LLM can emulate traffic patterns from smart thermostats, IP cameras, or sensors, making it difficult for anomaly-based IDS to distinguish legitimate activity from a compromised device.
- **Evasive Payload Crafting:** By learning from known attack signatures, an LLM can produce mutated payloads that avoid triggering rule-based IDS, effectively slipping under the radar.
- **Reverse-Engineering Protocols:** Given access to packet captures or logs, an LLM can infer undocumented or proprietary protocol sequences, which aids in the development of novel exploits.
- **Low-and-Slow Attack Simulation:** LLMs can craft stealthy attacks such as slow port scans, fragmented payloads, or time-delayed injections that blend seamlessly into normal network noise.
- **Intent-Aware Malware Generation:** Though not originally designed for this purpose, LLMs have demonstrated the ability to generate or assist in writing obfuscated scripts (e.g., PowerShell, Bash) when prompted, potentially automating the creation of sophisticated malware.

These capabilities underscore the importance of viewing LLMs not just as defensive tools but also as potential threats in cybersecurity design.

2.6.3 Risks in the IoT Context

IoT ecosystems are particularly vulnerable to adversarial misuse of LLMs for several reasons:

- **Unmonitored Devices:** Many IoT deployments lack centralized logging or IDS, providing attackers with a stealthy entry point for LLM-generated exploits.
- **Protocol Diversity:** LLMs can be trained to exploit lesser-known or poorly implemented protocols (e.g., mDNS, CoAP, Zigbee) where signature coverage is minimal.
- **Scalable Attacks:** Once an LLM is tuned for a specific device type or network, it can quickly generate large-scale attack campaigns across similar environments (e.g., targeting all smart TVs of a certain brand).

2.6.4 Adversarial Machine Learning (AML)

In addition to generating adversarial traffic, AI-based IDS models themselves are vulnerable to Adversarial Machine Learning (AML) attacks. Adversaries can craft inputs designed to mislead or bypass AI detectors, creating a new frontier of cybersecurity challenges.

Basics of Adversarial Examples in Network Traffic

Just as subtle pixel changes in images can cause misclassification by vision models, small perturbations in network traffic such as minor adjustments to timing, packet sizes, or header flags can fool AI-powered IDS. For example, adding non-functional padding, jittering timestamps, or splitting a payload into multiple fragments may preserve the intended malicious action while evading detection by models trained on raw features.

Types of AML Attacks in IDS

- **Evasion Attacks:** The attacker modifies malicious traffic so that it appears benign at inference time, causing the IDS to overlook the intrusion.
- **Poisoning Attacks:** Crafted samples are injected into the IDS training data to degrade model performance or introduce backdoors, allowing future evasion.
- **Model Inversion / Extraction:** By querying a cloud-based IDS API, an adversary can reconstruct sensitive features or replicate model behavior, potentially uncovering detection logic and facilitating targeted attacks.

Challenges in Defending Against AML

- **Model Robustness:** Most deep learning architectures are not inherently resistant to adversarial perturbations. While adversarial training can improve resilience, it remains challenging to apply effectively in real-time, resource-constrained environments.

- **Explainability and Trust:** Complex models like CNNs and LSTMs lack transparency making it difficult to understand why certain traffic was misclassified or to verify whether the model has been compromised.
- **Detection of Adversarial Inputs:** Unlike images or text, adversarial modifications in network traffic can be extremely subtle hidden within normal usage patterns so detecting these variations without generating false alarms is a nontrivial problem.

As AI becomes central to cybersecurity defense, so too does it become a tool for sophisticated cyberattacks. LLMs, with their ability to generate structured, adaptive outputs, can be weaponized to automate protocol-aware adversarial traffic, simulate stealthy attacks, or manipulate AI-based IDS models. The dual-use dilemma is especially pronounced in IoT networks, where high heterogeneity, limited oversight, and constrained defenses create fertile ground for LLM-driven attacks. Therefore, designing secure AI systems must extend beyond raw performance metrics and prioritize robustness, interpretability, and adversarial resilience as fundamental requirements.

2.6.5 Integration Challenges with LLM-Generated Data

Although LLMs can produce richly structured, protocol-compliant traffic, integrating that data into simulators and testbeds entails several nontrivial steps:

- **Time-Series Alignment:** Simulators require precise, monotonic timestamps. LLM-generated streams often need post-processing such as scaling or resampling to align with simulation clock ticks and ensure events occur in correct sequence.
- **Protocol Fidelity:** While LLMs can generate valid looking headers, low-level checksums, TCP state transitions, or radio-layer acknowledgments may be missing. We must enforce protocol rules (e.g., correct sequence numbers, valid checksum fields) before injecting packets into NS-3 or Cooja.
- **Behavioral Realism:** Real devices behave as finite state machines they send a SYN, wait for SYN/ACK, then send an ACK. LLM outputs might omit these handshakes or reorder steps. We supplement generated streams with deterministic logic or small scripts to ensure that devices follow correct state transitions.
- **Replay vs. Live Generation:** Pre-generating full trace files is straightforward, but real-time LLM-based traffic generation (where an LLM spins off packets on-the-fly) remains an open challenge. Hybrid testbeds—combining live LLM inference with simulated network stacks require careful synchronization and resource planning.

In summary, a multidimensional evaluation framework combining statistical, temporal, packet-level, and QoS metrics ensures that LLM-generated IoT traffic is not only structurally accurate but also functionally realistic. By validating synthetic data in NS-3, Cooja, and fog/edge simulations, we can confidently use these traces to train and test IDS models, ultimately advancing the state of IoT security research.

2.7 Literature Survey

Over the past few years, researchers have explored using large language models to fill gaps in network traffic data. For example,

- **PAC-GPT**: Kholgh and Kostakos's PAC-GPT framework shows how GPT-3 can be coaxed into generating both high-level flow sequences and the fine-grained packet details that IDSs rely on (Kholgh and Kostakos [21]). By chaining a "Flow Generator" and a "Packet Generator," they achieve remarkably faithful synthetic traces with minimal extra training. While the PAC-GPT is capable of generating various types of signals, it is entirely built on closed-source LLMs which uses API tokens to run on a remote server. Additionally, the gereration of IoT network signals is based on the prompt given to the model. In contrast, our research focuses on using open-source LLMs which are fine-tuned for our use-case. Our research models can be run locally, which offers better privacy. Since the models are running locally, we do not need internet connection, which makes it's deployment more vast.

- **IoTNetSim**: Salama *et al.* built IoTNetSim, an end-to-end simulator that stitches together device behavior, edge gateways, and cloud services into one event-driven platform (Salama et al. [37]). Unlike narrowly focused traffic generators, IoTNetSim can emulate everything from sensor mobility to power constraints, giving a more complete picture of large-scale IoT deployments. Although IoTNetSim excels at system-level experimentation with open-source tools, its device and network models are handcrafted rather than learned, limiting its ability to capture emergent traffic patterns. Our approach instead learns those patterns from real captures via LLM fine-tuning, yielding synthetic logs that reflect unforeseen protocol quirks and timing jitter. As mentioned earlier, IoTNetSim has a layer in cloud for its data collection and processing. This again is not good in the context of privacy or running everything locally-as done in our research.

- **Kecskemeti et al.**: Kecskemeti and colleagues took a step back to survey the broader challenges of IoT modelling, calling out the need for systems that can scale to millions of devices, handle diverse hardware profiles, and balance simulation fidelity with runtime performance (Kecskemeti et al. [20]). Their insights into dynamic sensor lifecycles and elastic accuracy have shaped many modern testbeds. Although this research gives us many directions in which the IoT network signals can be modelled and simulated, most of the suggested solutions use cloud infrastructure in backend. And the input data required to implement any of the proposed solutions is very high. As our research focus on training model without extra large data, we focus on starting from small number of devices with limited set of data.

- **NetGPT**:On the encrypted-traffic front, Meng *et al.* introduced NetGPT, the first GPT-style model pre-trained specifically on raw packet traces including TLS handshakes and byte-level payloads (Liu et al. [30]). By treating packet bytes as tokens, they teach the model to reproduce protocol patterns end-to-end, outperforming earlier, more specialized generators. NetGPT advances encrypted-traffic

generation but remains specialized to a single protocol and does not address the mix of TCP, UDP, mDNS, and application-layer messages typical in IoT. We extend beyond TLS 1.3 by fine-tuning on multi-protocol IoT captures and evaluating the synthetic outputs in realistic IDS scenarios.

Together, these works illustrate both the promise and the hurdles of synthetic traffic generation. They laid the groundwork that our thesis builds on showing that with the right architecture and fine-tuning, LLMs can become powerful allies in crafting diverse, realistic IoT datasets for robust IDS training.

Summary Together, these works demonstrate the promise of generative models and simulators for network experimentation but leave open critical gaps: IoT specificity, open-source, edge-computing, multi-protocol support, and, most importantly, *quantified* IDS improvements. Our thesis fills these gaps by fine-tuning a suite of 2–3 B-parameter LLMs on real and benchmark IoT datasets, synthesizing hybrid traffic corpora, and rigorously measuring the resulting gains in anomaly detection performance.

Chapter 3

System Design

Figure 3.1 lays out our six-stage pipeline in a single sweep. We begin by *collecting* raw packet captures from smart devices, then move into *data pre-processing*, where those binary logs are cleaned, parsed, and tokenized into a text-friendly format. Next comes *training and fine-tuning* of open-source LLMs (LLaMA, Qwen, etc.) on this processed data. Once the models have learned the "language" of IoT traffic, we ask them to *generate synthetic traces*, looping back the newly minted packets into our pool. Those blended datasets real plus synthetic are then used to *train an IDS*, teaching it what normal and malicious behavior look like. Finally, we *evaluate* the detector's performance under realistic conditions, measuring both its hit rate on attacks and its false-alarm rate on benign traffic. This iterative design lets us refine each step especially data generation until our IDS behaves like a vigilant guard dog that only barks at real threats.

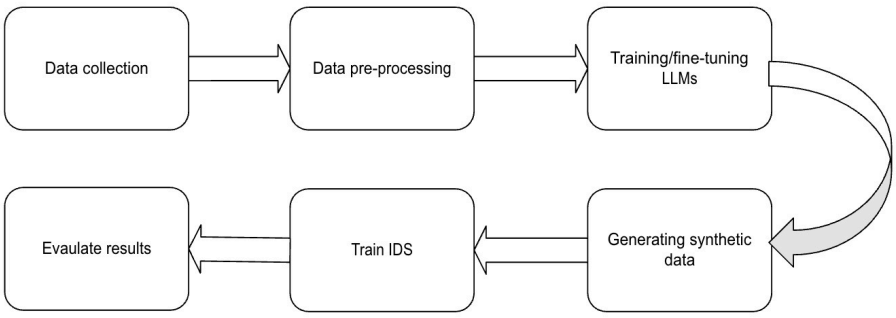


Figure 3.1: End-to-end workflow of our experiment

3.1 General Objectives

The goal of this system is to build a flexible simulation and evaluation framework that leverages Large Language Models (LLMs) to produce realistic Internet of Things (IoT)

network traffic. By generating synthetic traces that closely mimic real-world device communications, we aim to bolster security measures especially Intrusion Detection Systems (IDS) by providing richer, more varied training data than existing datasets allow.

Specifically, this framework will:

- Use open-source LLMs (e.g., LLaMA, Qwen, Phi-2, Gemma, Granite) to synthesize IoT traffic reflecting both normal operations and attack scenarios.
- Integrate these synthetic traces into network simulations, so that IDS performance can be evaluated under realistic conditions, including both benign and malicious behavior.
- Support detailed statistical, temporal, and behavioral analyses of generated data, ensuring that researchers can measure IDS accuracy, resilience against adversarial traffic, and generalization to unseen patterns.

These objectives guide the system's functional and non-functional requirements, ensuring that it delivers accurate data generation, reproducibility, scalability, and adaptability as IoT threat models evolve.

3.2 Experimental Setting

To conduct this research, we assembled an infrastructure capable of capturing real IoT traffic, running network simulations, training large language models, and evaluating IDS effectiveness. Below, we outline the hardware and software components that made these tasks possible.

3.2.1 Hardware Setup

IoT Traffic Collection Setup

- **Raspberry Pi 4B (8 GB RAM)** As shown in Figure 3.2 wecConfigured as a Wi-Fi hotspot, this Raspberry Pi acted as a lightweight IoT gateway. By serving as the central communication point for connected devices, it allowed us to capture packets passively without altering the network topology. Its small form factor and low power draw meant it could run continuously in a real-world environment.
- **Packet Capture Tool (Wireshark)** Wireshark was installed on the Raspberry Pi to collect traffic in PCAP format. This setup preserved complete protocol stacks and timestamps, giving us a detailed view of protocols like TCP, HTTP, TLS 1.2, UDP, and mDNS. These captures provided the ground truth needed for later LLM-based synthetic traffic generation.

Smart relays Hardware

- **Tapo P100** The Tapo P100 shown in Figure 3.3 is a compact smart Wi-Fi plug that lets you remotely turn appliances on or off via the Tapo mobile app or voice

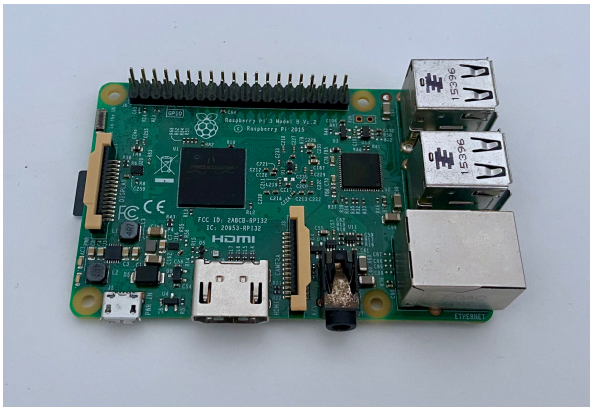


Figure 3.2: Raspberry Pi device used for capturing the network logs

assistants like Amazon Alexa and Google Assistant. The hardware version is 1.20.0. The software version of the application is available both in AppStore and PlayStore. The AppStore version used for the experimentation is 3.11.105.

Figure 3.3: Tapo P100 smart plug for European sockets

- **Meeross MSS210 EU** The Meeross MSS210 EU shown in Figure 3.4 is a compact smart Wi-Fi plug designed for European outlets, allowing you to remotely control and schedule connected appliances via the Meeross app without needing a separate hub. It also features real-time energy monitoring and supports voice commands through Amazon Alexa and Google Assistant for seamless home automation. The hardware version is 4.5.0 and the firmware version is 4.2.3. The software version of the application is available both in AppStore and PlayStore. The AppStore version used for the experimentation is 3.34.0.

Model Training and Evaluation Hardware

- **Initial Compute Node (Omnissa Horizon Client)** A remote server with 64 GB RAM and an Intel® Xeon® Gold CPU handled early data preprocessing

Figure 3.4: Meeross MSS210 EU smart plug for European sockets

and small-scale model tests. However, the lack of GPUs meant that training larger LLMs took several weeks per experiment, hindering rapid iteration.

- **GPU-Enabled Virtual Machine** For intensive model training and fine-tuning, we used a high-performance VM equipped with an NVIDIA A100 GPU, 128 GB RAM, and a 16-core AMD EPYC CPU. CUDA 12.2 support and SLURM job scheduling enabled parallel experiments and faster convergence. This environment was essential for applying parameter-efficient fine-tuning methods (e.g., LoRA) and training memory-hungry models like LLaMA and Granite.

3.2.2 Software Setup

Traffic Capture and Analysis

- **Wireshark** As the industry standard for packet analysis, Wireshark collected raw IoT traffic in PCAP format. Its filtering and GUI tools allowed us to inspect protocols and debug data-collection issues in real time.

- **Scapy** Scapy, a Python-based packet-crafting and analysis library, handled post-capture processing. We used it to extract protocol fields, reassemble sessions, and convert packet headers into tokenized sequences suitable for transformer-based training.

LLM Training and Inference

- **Hugging Face Transformers** Provided pretrained models, tokenizers, and fine-tuning scripts for LLaMA, Qwen, and other LLMs. Its modular design integrated smoothly with PEFT and Accelerate, letting us switch between hardware configurations without major code changes.

- **Hugging Face Datasets** Managed custom traffic datasets, handled text preprocessing for packet sequences, and facilitated efficient batch loading via PyTorch's DataLoader.

- **PyTorch (>= 2.1)** Served as our core deep learning framework, offering dynamic computation graphs, mixed-precision training, and native GPU support critical for handling models with billions of parameters.
- **Accelerate** Simplified distributed training across multiple GPUs or mixed-device setups. Used alongside SLURM to orchestrate large-scale training jobs on HPC clusters.
- **bitsandbytes** Enabled 4-bit and 8-bit quantized training to reduce memory consumption and speed up LLM fine-tuning on resource-constrained GPUs without significant performance loss.
- **PEFT (Parameter-Efficient Fine-Tuning)** Integrated with Hugging Face Transformers to apply techniques like LoRA, drastically reducing the number of trainable parameters. This approach allowed us to fine-tune large models even when GPU memory was limited.
- **Flash Attention** Used by certain models (e.g., Qwen) to improve memory efficiency and speed when processing long sequences. This was important for modeling extended IoT traffic traces that exceed typical NLP sequence lengths.
- **Weights & Biases (WandB)** Tracked training metrics, loss curves, and hyperparameter settings across experiments. This ensured reproducibility and made it easier to compare different model configurations.
- **SLURM Job Scripts** Automated and scheduled training jobs on the GPU cluster. SLURM's dependency features helped maintain a consistent pipeline and ensured GPUs were utilized efficiently. A sample slurm Job script is shown in Figure 3.5

Simulation and Evaluation

- **Jupyter Notebooks & Python Scripts** Provided an interactive environment for debugging, evaluating, and visualizing both traffic patterns and model performance. These tools allowed for rapid iteration and easy integration of custom metrics.
- **Matplotlib, Seaborn, Plotly** Used to visualize distribution comparisons, divergence metrics, and quality-of-service indicators. These libraries enabled us to analyze packet size histograms, inter-arrival time distributions, and other temporal features in detail.
- **SciPy and NumPy** Formed the computational backbone for statistical analyses (e.g., entropy calculations, divergence metrics, autocorrelation, and fitting Kumaraswamy or Johnson SU distributions). These libraries were essential for rigorously evaluating signal quality and IDS performance.

```
#!/bin/bash
# SLURM script for submitting a job to a cluster
# SLURM options for job configuration

#SBATCH --job-name=py-job
# The name of the job as it will appear in the job queue. Here: 'py-job'.

#SBATCH --nodes=1
# Specifies that the job will run on a single node.

#SBATCH --ntasks=1
# Specifies that only one task will be started. This means only one instance of the program will run.

#SBATCH --cpus-per-task=16
# Specifies that each task will use 4 CPU cores.

#SBATCH --mem-per-cpu=64G
# Allocates 4 GB of memory per CPU core assigned to the task.

#SBATCH --gres=gpu:1
# Requests two GPUs. 'gres' stands for Generic Resource Scheduling, and 'gpu:2' specifies that two GPUs are required for the job.
# use "--gres=gpu:1" for one gpu
# use "--gres=gpu:2" for two gpus

#SBATCH --output=out/slurm-%j.out
# Defines the file for the standard output of the job. %j will be replaced by the job ID to make the output file unique. The file will be in the 'out' directory.

#SBATCH --error=err/slurm-%j.err
# Defines the file for the error output of the job. %j will be replaced by the job ID. The error file will be in the 'err' directory.

#SBATCH --nodelist=hades
# Specifies the exact node 'atlas' on which the job should run.
# Activate the Python environment
source environments/mistal_llm/bin/activate
# Activates the environment located in 'environments/envtest'. This ensures that all necessary packages and dependencies are available.

# Run the Python script
python3 mistral_train.py
# Executes the Python script 'test_torch.py'. Ensure this script is in the current working directory or provide the full path.
```

Figure 3.5: Sample slurm script used for triggering jobs

Chapter 4

Realisation / Implementation

This chapter describes how the theoretical methods outlined earlier were transformed into a working system. We begin by collecting real IoT traffic, then preprocess and format it for training large language models (LLMs). Subsequent sections cover the configuration of the training pipeline, generation of synthetic signals, and the evaluation techniques we employed. In essence, this chapter bridges the gap between conceptual design and practical execution.

Figure 4.1 lays out our complete workflow in plain terms. We start in a "smart home" environment, where everyday IoT gadgets talk to each other through a Raspberry Pi acting as a transparent Wi-Fi gateway. All of that traffic packets, timestamps, protocol details flows into the Pi and is recorded in PCAP form. Next, a pre-processing server cleans and tokenizes these raw logs and then dispatches jobs to our GPU cluster, where we fine-tune open-source LLMs (LLaMA, Qwen, etc.) on this real data. Once trained, those models generate fresh, synthetic traces that mirror the quirks of actual device communications. Finally, we combine real and synthetic traffic to train and test an IDS, measuring how well it spots anomalies and resists false alarms. This loop—from real capture to synthetic generation to IDS evaluation lets us refine each stage until our detector has good accuracy and ROC.

4.1 Data Collection and Preprocessing

The very first step in our work was to gather real IoT traffic under conditions that felt both controlled and authentic. Our goal was to capture high-quality, packet-level data from actual devices in use, which would later underpin the training and evaluation of our LLM-driven synthetic traffic generator and intrusion detection framework.

4.1.1 IoT Traffic Capture Setup

To recreate a realistic IoT network, we turned a Raspberry Pi 4B into our central hub. After installing Ubuntu for Raspberry Pi, we configured it as a Wi-Fi hotspot, effectively turning it into a local gateway. This allowed smart plugs and other devices to connect just as they would in a living room or office, communicating both with each other and with cloud services through the Pi's internet link.

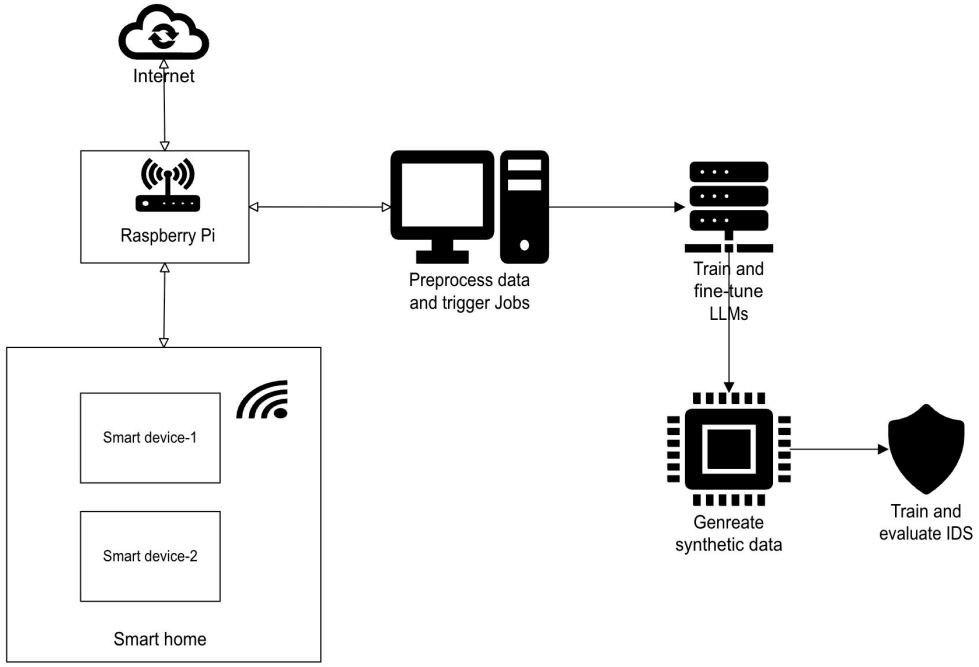


Figure 4.1: Project architecture

We paired several smart plugs to this network and controlled them via their official mobile apps. To ensure the captured data reflected genuine usage, we mixed automated schedules (for example, turning on and off at regular intervals) with occasional manual commands. This combination produced a variety of behaviors—periodic on/off cycles, user-triggered toggles, and idle periods.

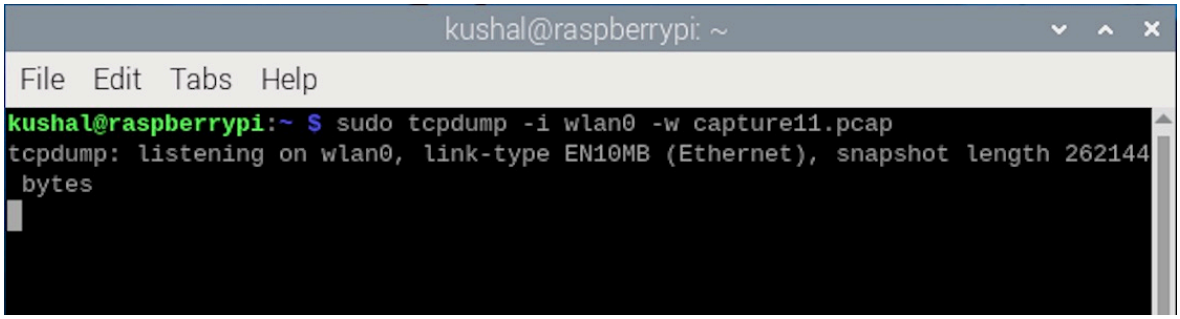


Figure 4.2: Terminal command to capture all the network packets from Raspberry Pi and store into a sngle file

For packet capture, we evaluated two popular tools: Wireshark and `tcpdump`. Although Wireshark's graphical interface and deep inspection capabilities are compelling, it proved too heavy for continuous, long-term recording on the Pi. Instead, we opted for `tcpdump`, which excels in command-line, batch-based captures as shown in the Figure 4.2. Over several weeks, we scheduled regular capture intervals, building a dataset that was both broad in scope and spread out over time.

4.1.2 PCAP Data Export and Feature Extraction

Once we had raw traffic stored in PCAP format, we needed to pull out just the fields that would be most useful for our modeling. Using Wireshark's export feature, we converted each PCAP to a CSV, selecting only the columns that capture the essence of IoT communications:

- **No.** (Packet number)
- **Time** (Relative timestamp)
- **Source IP**
- **Source Port**
- **Destination IP**
- **Destination Port**
- **Protocol**
- **Length** (Packet size)
- **Info** (High-level description)
- **Date Time** (Human-readable timestamp)

By narrowing down to these essential fields, we dramatically reduced file size and focused our efforts on the aspects of each packet most likely to inform our LLM training. The resulting CSV files provided a clean, tabular view of how devices and services were talking to one another.

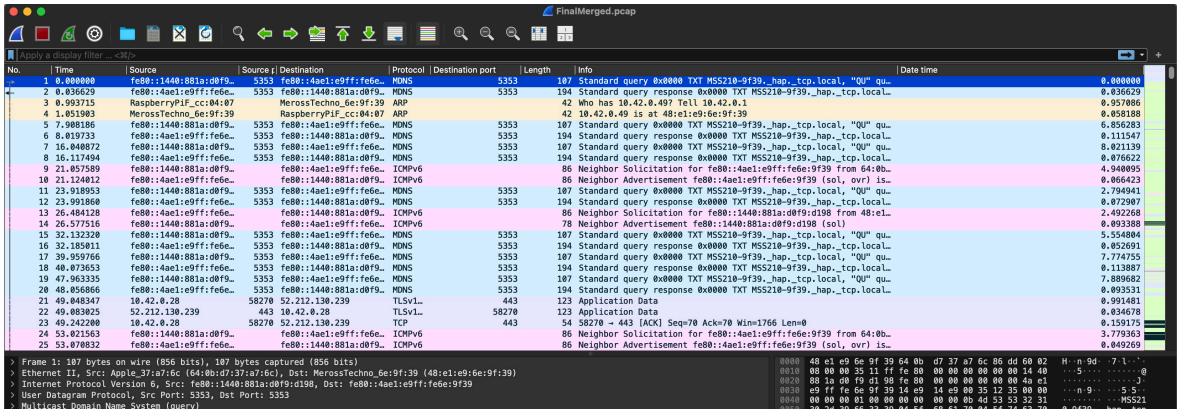


Figure 4.3: Fields selected in Wireshark for exporting to Fine-tune LLMs

4.1.3 Data Cleaning and Formatting

With CSV exports in hand, our next task was quality control. A custom Python script scanned each row to weed out incomplete or malformed entries—any packets missing port numbers, lacking protocol labels, or carrying blank timestamps were discarded. This filtering step was crucial to avoid feeding junk data into our LLMs later on.

After cleaning, we were left with 65,305 valid packets. This dataset captured a rich spectrum of real-world interactions—everything from routine status updates to cloud-bound requests—creating a solid foundation for both normal-traffic modeling and later adversarial simulations.

Finally, to support different stages of our workflow, we exported the cleaned data in both CSV and JSON formats as shown in the Figure 4.3. The CSV version was ideal for batch processing and statistical analysis, while the JSON format plugged directly into our tokenizer pipelines and sequence-based modeling tools. With this step complete, our data pre-processing pipeline was ready, setting the stage for tokenization and model training in the following sections.

4.2 Model Training and Fine-Tuning

In this phase, we took our cleaned IoT traffic sequences and taught several open-source Large Language Models (LLMs) to understand the patterns underlying packet-level communication. By framing each capture as a causal language modeling (CLM) task where the model learns to predict the next token in a sequence we aimed to imbue the LLMs with both the temporal cadence and structural syntax of real-world IoT traffic. Each model required a bespoke setup for data ingestion, tokenization, and training, balancing performance against available compute resources[29][53].

Rationale for Model Selection For our experiments, we focused on five open-source LLMs in the 2–3 billion parameter range, each chosen for what it brings uniquely to the table:

- **Open-LLaMA:** We turned to the community driven Open-LLaMA 3B as our baseline because it strikes a strong balance between ease of use and robust instruction following. Its straightforward training scripts and reliable text generation made it perfect for bootstrapping structured packet log synthesis.
- **Phi-2:** Microsoft's Phi-2 impressed us with its lean, stable architecture and fast CPU inference. Even without a GPU, Phi-2 let us prototype lightweight traffic generators quickly helping us compare raw efficiency against our heavier, more resource intensive models.
- **Granite-3B Code-Instruct:** IBM's Granite models are built for code and structured text, so we were curious to see how well that bias translates to packet-level logs. Granite's rigorous pre-processing pipeline and code generation strengths gave us a fresh angle on crafting syntactically precise IoT traces.
- **Gemma-2B:** Google's Gemma family is tailor-made for domain adaptation. Gemma-2B's compact size and mixed-precision training meant we could iterate

faster, and its instruction-tuned backbone helped it pick up protocol semantics with minimal fuss.

- **Qwen 2.5-3B:** Alibaba's Qwen stood out for maintaining coherence over long, semi-structured inputs ideal for modeling multi-device interactions. Qwen's knack for juggling complex contexts made it our go-to when generating realistic, stateful IoT traffic patterns.

4.2.1 Fine-Tuning Open-LLaMA

Our first candidate was *OpenLLaMA 3B*, a 3-billion parameter variant of Meta's LLaMA family that is well suited to sequence modeling tasks and freely available for research. Figure **??** shows the steps we took for fine-tuning Open-LLaMa LLM for our use case[29][53].

Model and Tokenizer Initialization

We loaded the pretrained weights and tokenizer from Hugging Face's `openlm-research/open\_llama\_3b` repository. Since LLaMA models typically lack a padding token, we introduced a custom `[PAD]` marker and resized the embedding matrix accordingly:

```
tokenizer.add_special_tokens({'pad_token': '[PAD]'})
model.resize_token_embeddings(len(tokenizer))
```

This adjustment allowed us to batch sequences of uniform length without truncation errors.

Data Preparation

Each row of our cleaned CSV was transformed into a human-readable packet description. For example:

```
Source: 192.168.1.2 Source Port: 443 Destination: 192.168.1.5
Destination Port: 80 Protocol: TCP Length: 512 Info: Standard
HTTP GET request
```

By concatenating these descriptions in timestamp order, we created textual sequences that capture both field relationships and inter-packet timing. These sequences were wrapped in a Hugging Face `Dataset`, then tokenized with padding and truncation (maximum length 256 tokens), assigning the input IDs as labels for next-token prediction.

Mathematical Formulation

To formalize our fine-tuning process, we introduce the following notation:

- B: batch size per device.
- G: number of gradient accumulation steps.

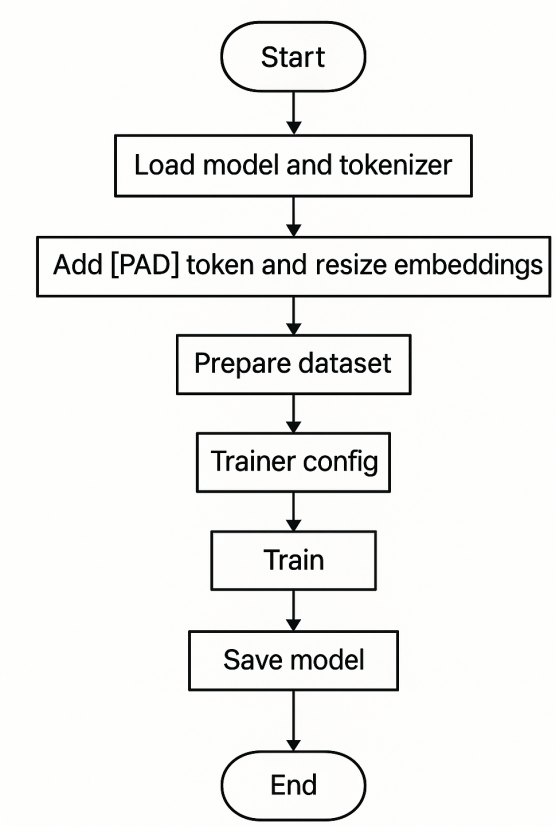


Figure 4.4: Flowchart for Open-LLaMa fine-tuning

- $B_{\text{eff}} = B \times G$: effective batch size after accumulation.
- N: total number of training examples.
- E: number of epochs.
- $L_{\max}$: maximum sequence length (here, 256 tokens).
- θ: model parameters.
- λ: weight decay coefficient.
- $\alpha_{\max}$: peak learning rate.
- W: number of warmup steps.
- g_t: gradient at step t.
- β_1, β_2: AdamW moment coefficients.
- ϵ: AdamW numerical stability constant.

1. Effective Batch Size and Update Count
With gradient accumulation, the effective batch size is

$$B_{\text{eff}} = B \times G.$$

Over E epochs on N examples, the total number of parameter updates is

$$N_{\text{updates}} = \frac{N \times E}{B_{\text{eff}}} = \frac{N \times E}{B\,G}.$$

2. Sequence Length Constraint
Each raw sequence of length ℓ_i is truncated or padded to

$$T_i = \min\Big(\ell_i,\ L_{\max}\Big),$$

ensuring uniform length $L_{\max}$ for all examples.

3. Causal Language Modeling Loss
We use the cross-entropy loss for next-token prediction. For each batch example i and token position t:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{B_{\text{eff}}} \sum_{i=1}^{B_{\text{eff}}} \sum_{t=1}^{T_i} \log p_\theta\Big(y_{i,t} \mid x_{i,<t}\Big),$$

where $y_{i,t}$ is the true token and p_θ the model's predicted probability.

4. Weight Decay Regularization
To prevent overfitting, we add an L_2 penalty:

$$\mathcal{L}_{\text{tot}} = \mathcal{L}_{\text{CE}} + \frac{\lambda}{2} \|\theta\|_2^2.$$

5. Learning Rate Schedule with Warmup
We linearly ramp the learning rate from zero to $\alpha_{\max}$ over W warmup steps:

$$\alpha_t = \begin{cases} \frac{t}{W}\,\alpha_{\max}, & t < W, \\ \alpha_{\max}, & t \geq W. \end{cases}$$

6. AdamW Parameter Updates
Using AdamW, the biased and bias-corrected moments are

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \end{aligned}$$

and parameters update as

$$\theta_{t+1} = \theta_t - \alpha_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \alpha_t\,\lambda\,\theta_t.$$

This completes the formal description of our fine-tuning regimen.

Hyperparameter	Value
Model	OpenLLaMA 3B
Epochs	3
Batch size (per device)	2
Max sequence length	256 tokens
Learning rate	5×10^{-5}
Gradient accumulation steps	2
Weight decay	0.01
Warmup steps	500
Evaluation strategy	None (training only)
Precision mode	FP32
Save steps	10,000
Total checkpoints kept	2

Table 4.1: OpenLLaMA fine-tuning hyperparameters

Training Configuration

Training used the Hugging Face `Trainer` API with parameters chosen to balance convergence and resource constraints:

Checkpoints were written every 10,000 steps, with a cap of two saved states to manage disk usage.

Hardware and Environment

This initial fine-tuning ran on a CPU-only (or Apple MPS) setup, serving as a proof of concept. Although training was slow and batch sizes were small, it verified our data pipeline and allowed us to observe loss convergence behavior. Subsequent models leveraged GPU acceleration.

Model Export and Observations

After three epochs, we saved the fine-tuned model and tokenizer with:

```
model.save_pretrained(OUTPUT_DIR)
tokenizer.save_pretrained(OUTPUT_DIR)
```

This OpenLLaMA experiment established a baseline for later fine-tuning of larger or more efficient models—such as Qwen, Phi-2, and Gemma using GPU resources and advanced tuning techniques.

4.2.2 Fine-Tuning Phi-2

Next, we adapted **Phi-2**, a lean transformer model from Microsoft renowned for its reasoning and generation performance in resource-constrained settings. With just a few billion parameters, Phi-2 converges quickly, making it ideal for experimenting on limited hardware without sacrificing semantic depth. Figure 4.5 shows the steps we took for fine-tuning Phi-2 LLM for our use case.

Instruction-Based Prompt Design Rather than raw next-token prediction, we framed fine-tuning as an *instruction-tuned* task. Each IoT packet log was reformatted into a three-part template:

```
[INSTRUCTION] Generate a network signal log simulating behavior
or attacks.
[INPUT] Time: <timestamp>, Source: <src>:<src_port>, Destination:
<dst>:<dst_port>,
Protocol: <protocol>, Length: <length>, Info: <description>,
Date: <datetime>
[OUTPUT]
```

This structure gave the model a clear "question" (the instruction), a detailed "context" (the input), and an open slot for the "answer" (the output). By conditioning generation on richly annotated metadata, Phi-2 learned to produce coherent, protocol-aware traffic sequences.

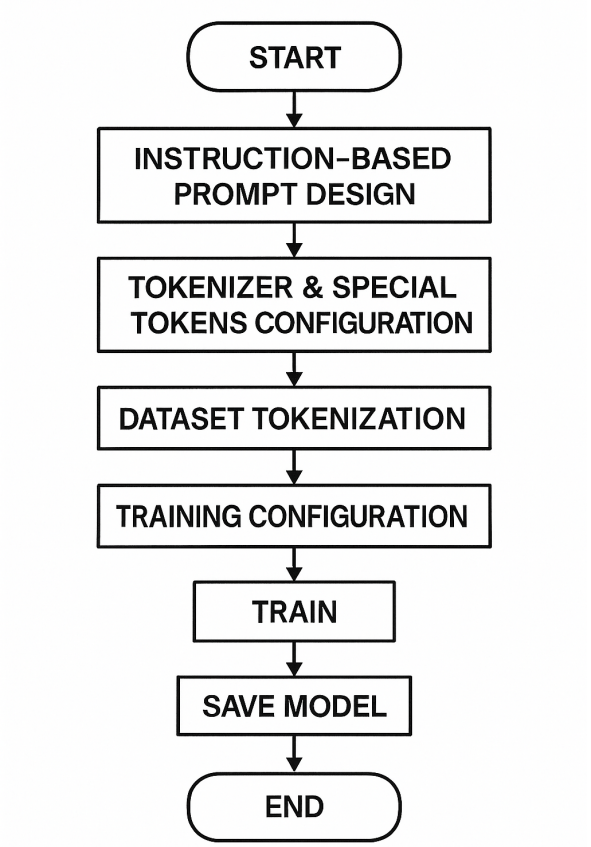


Figure 4.5: Flowchart for Phi-2 fine-tuning

Tokenizer and Special Tokens We pulled both model and tokenizer from Hugging Face's `microsoft/phi-2` checkpoint[29][53]. To support our custom prompt format, we added three special tokens and resized the embedding layer:

```
tokenizer.add_special_tokens({
  "additional_special_tokens": ["[INSTRUCTION]", "[INPUT]", "[OUTPUT]"]
})
model.resize_token_embeddings(len(tokenizer))
```

We also set the padding token to match the end-of-sequence token, preventing alignment errors during batch training.

Dataset Tokenization Using a bespoke `tokenize_function`, we converted each prompt-output pair into token IDs with:

- *Padding/truncation* to a maximum length of 512 tokens, ensuring all structured fields and potential responses fit comfortably.
- *Consistent formatting*, so that each example followed the same token layout, reducing variability in the model's learning signal.

Mathematical Formulation To make our training regimen clearer, we introduce a few friendly equations that capture the key mechanics of fine-tuning:

- **Effective Batch Size.** Because we accumulate gradients over multiple steps, the real batch size is larger than the per-device value:

  $$B_{\text{eff}} = B \times G,$$

  where B is the batch size on each device (here, 2) and G is the number of accumulation steps (here, 8), giving $B_{\text{eff}} = 16$.

- **Total Update Steps.** Over E epochs on N examples, the number of optimizer updates is

  $$N_{\text{updates}} = \frac{N \times E}{B_{\text{eff}}} = \frac{N\,E}{B\,G}.$$

  This tells us how many times the model parameters actually get nudged.

- **Sequence Length Control.** Every example's tokens are either cut or padded to a uniform length:

  $$T_i = \min(\ell_i,\ L_{\max}),$$

  where ℓ_i is the raw token count of example i, and $L_{\max} = 512$.

- **Causal Cross-Entropy Loss.** We train the model to predict the next token in a sequence, using

  $$\mathcal{L}_{\text{CE}} = -\frac{1}{B_{\text{eff}}} \sum_{i=1}^{B_{\text{eff}}} \sum_{t=1}^{T_i} \log p_\theta\Big(y_{i,t} \mid x_{i,<t}\Big),$$

  where $y_{i,t}$ is the true token, $x_{i,<t}$ the preceding tokens, and p_θ the model probability.

- **Weight Decay Regularization.** To gently discourage overly large weights, we add a small penalty:
$$\mathcal{L}_{\text{tot}} = \mathcal{L}_{\text{CE}} + \frac{\lambda}{2} \|\theta\|_2^2,$$
with λ the weight-decay factor.

- **Learning Rate Warmup.** We slowly ramp up the learning rate from zero to its peak $\alpha_{\max}$ over W warmup steps:
$$\alpha_t = \begin{cases} \frac{t}{W}\,\alpha_{\max}, & t < W, \\ \alpha_{\max}, & t \geq W. \end{cases}$$

- **AdamW Updates.** Finally, each parameter update follows the AdamW rule:
$$\begin{aligned} m_t &= \beta_1\, m_{t-1} + (1-\beta_1)\, g_t, \\ v_t &= \beta_2\, v_{t-1} + (1-\beta_2)\, g_t^2, \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \\ \theta_{t+1} &= \theta_t - \alpha_t\, \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \alpha_t\, \lambda\, \theta_t. \end{aligned}$$
Here g_t is the gradient, β_1, β_2 control the moving averages, and ϵ keeps things stable.

Training Configuration We again employed the Hugging Face `Trainer` API, this time with settings optimized for Phi-2's size and our CPU-only environment(Channel [9]):

Parameter	**Value**
Model	Phi-2
Epochs	3
Batch size (per device)	2
Gradient accumulation steps	8
Max sequence length	512 tokens
Save interval	500 steps
Logging interval	50 steps
Precision mode	FP32
Evaluation strategy	None

Table 4.2: Hyperparameters for Phi-2 fine-tuning

A dynamic data collator handled padding and computed causal loss (with `mlm=False`) so the model focused purely on next-token prediction.

Hardware Setup Even on a standard CPU machine, Phi-2's compact footprint and gradient accumulation allowed us to complete the training in a reasonable time frame. This setup validated our instruction-tuning pipeline before scaling up to GPU clusters(Channel [9]).

Model Saving and Reusability Upon finishing, we saved the fine-tuned checkpoint and tokenizer:

```
model.save_pretrained("./phi2-finetuned")
tokenizer.save_pretrained("./phi2-finetuned")
```

These artifacts are now ready for downstream signal generation, intrusion simulation, or further iterative fine-tuning.

4.2.3 Fine-Tuning Granite 3B

Next, we explored *Granite 3B*, IBM's 3-billion parameter instruction-tuned model, to assess its ability to handle dense, protocol-rich IoT packet descriptions. Although originally optimized for code and structured tasks.
The checkpoint `ibm-granite/granite-3b-code-instruct-128k`'s large context window offered a unique chance to model long, semantically complex sequences of network events. Figure 4.6 shows the steps we took for fine-tuning IBM Granite LLM for our use case.

Data Representation and Formatting Starting with our cleaned packet logs (Section 4.1), we converted each record into a natural-language style sentence. For example:

```
Source: 192.168.0.10:443, Destination: 192.168.0.20:80, Protocol:
TCP, Length: 512, Info: ACK packet from smart plug to router
```

A `format_row()` helper function applied this transformation across the dataset. We then loaded the results into a Hugging Face `Dataset` and split it into 90% training and 10% evaluation subsets to enable periodic validation during fine-tuning(IBM [17]).

Tokenizer and Model Configuration Both model and tokenizer were fetched from Hugging Face:

```
from transformers import AutoModelForCausalLM, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(
    "ibm-granite/granite-3b-code-instruct-128k"
)
model = AutoModelForCausalLM.from_pretrained(
    "ibm-granite/granite-3b-code-instruct-128k"
)
```

Because Granite is already instruction-tuned, no new special tokens were needed. We applied batched tokenization with uniform length:

```
tokenizer(
  batch["text"],
  truncation=True,
  padding="max_length",
  max_length=512
)
```

A 512-token context window balanced memory demands against the need to capture full packet descriptions plus generation space.

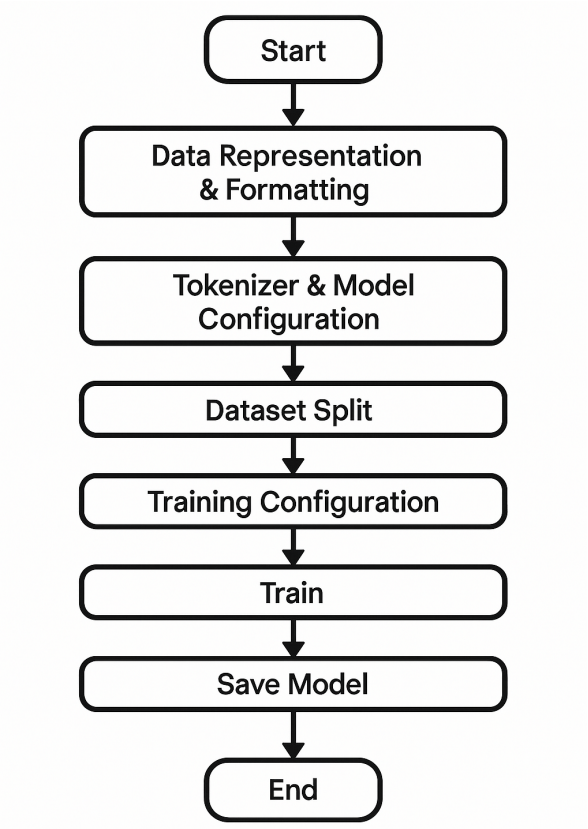


Figure 4.6: Flowchart for Granite fine-tuning

Mathematical Formulation We first split our full dataset of size N into training and test subsets using a fixed test fraction α (e.g. $\alpha = 0.1$):

$$N_{\text{train}} = (1 - \alpha)\, N, \quad N_{\text{test}} = \alpha\, N$$

This simple rule makes sure a consistent portion of the data is reserved for evaluation.

Next, when tokenizing each example we pad or truncate its original length L_{orig} to a uniform maximum $L_{\max}$ (e.g. 512 tokens):

$$L = \min\Big(L_{\text{orig}},\, L_{\max}\Big)$$

Uniform sequence lengths enable efficient batching and GPU utilization.

The core training objective is the autoregressive cross-entropy loss over a sequence $x_{1:T}$:

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} \log P\Big(x_t \mid x_{<t}; \theta\Big)$$

Minimizing this loss encourages the model to predict each next token correctly given its context.

To simulate larger batches under limited memory, we accumulate gradients over G steps on micro-batches of size B, yielding an effective batch size

$$B_{\text{eff}} = B \times G$$

This trick balances update stability with hardware constraints.

Over E epochs, the total number of parameter updates U becomes

$$U = \frac{E \times N_{\text{train}}}{B \times G}$$

which determines how many times the optimizer steps through the model parameters.

Finally, each update applies a gradient descent step with learning rate η:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

driving the parameters θ toward lower loss and better next-token predictions.

Training Setup and Hyperparameters We used the Hugging Face `Trainer` API on an A100 GPU, enabling FP16 mixed precision. Key settings were:

Parameter	**Value**
Model	Granite 3B (Code-Instruct)
Epochs	3
Train batch size	1
Eval batch size	1
Gradient accumulation steps	8
Max sequence length	512 tokens
Evaluation strategy	Every 250 steps
Checkpoint save strategy	Every 500 steps
Mixed precision	FP16 enabled
Load best model at end	True

Table 4.3: Hyperparameters for Granite 3B fine-tuning

A causal data collator (with `mlm=False`) ensured the model optimized next-token prediction conditioned on the instruction-style input.

Training Environment Training ran on an NVIDIA A100 GPU with mixed-precision enabled. We fixed the random seed (`set_seed(42)`) for reproducibility and used gradient accumulation to simulate a larger batch size despite GPU memory constraints.

Model Export and Reusability Once training completed, we saved the fine-tuned artifacts:

```
trainer.save_model("./granite3b_iot_final")
tokenizer.save_pretrained("./granite3b_iot_final")
```

These assets feed directly into our synthetic signal generation pipeline and can be reused for inference or further adaptation.

4.2.4 Fine-Tuning Gemma

In this work, we chose Gemma 2B—a 2-billion-parameter, instruction-tuned LLM from Google—as our fourth model. Its compact size strikes a good balance between modelling power and GPU feasibility, while its open-source license and design for instruction following make it ideal for domain-specific adaptation. Figure 4.7 shows the steps we took for fine-tuning Google Gemma LLM for our use case.

Data Preparation We began with packet-level logs collected in real IoT environments (cf. Section 4.1). Each packet was rendered as a short text snippet, for example:

> `Source: 192.168.1.100 Source Port: 443 Destination: 192.168.1.5 Destination Port: 80 Protocol: TCP Length: 600 Info: TLSv1.2 Encrypted Handshake Message`

This format preserved both semantic and temporal details crucial to realistic traffic generation. We then wrapped these records in a HuggingFace `Dataset` object, ready for tokenization and training.

Tokenizer and Model Setup We loaded the `gemma-2b` checkpoint and noticed it lacked a padding token, which is required for batching[29][53]. We therefore added:

```
tokenizer.add_special_tokens({'pad_token': '[PAD]'})
model.resize_token_embeddings(len(tokenizer))
```

The model was placed on an NVIDIA A100 GPU and configured to use FP16 mixed precision for faster, more memory-efficient training.

Tokenization Strategy Our `tokenize_function` applied:

- Fixed-length padding and truncation (max length 256).
- Use of `input_ids` as labels in a causal language modelling (CLM) setup.

This means every sequence serves simultaneously as context and target for next-token prediction.

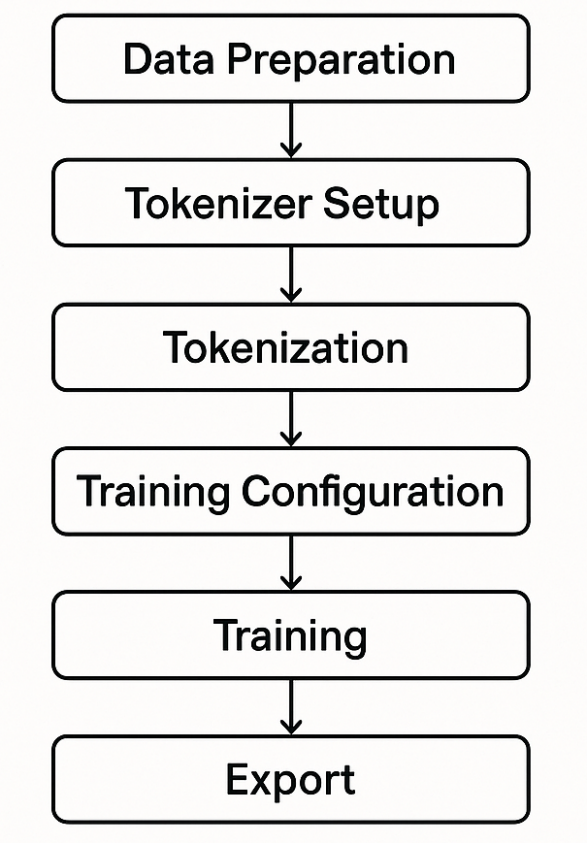


Figure 4.7: Flowchart for Gemma fine-tuning

Mathematical Formulation Language-Modeling Loss. This loss measures how well the model predicts each token given its context:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} \log P_\theta\Big(w_{n,t} \mid w_{n,<t}\Big)$$

Cross-Entropy Loss per Token. The cross-entropy between the true one-hot distribution p and the model's softmax output q:

$$\mathrm{CE}(p, q) = -\sum_{i=1}^{V} p_i \, \log q_i$$

Gradient-Descent Update. Parameters are updated opposite to the gradient of the loss:

$$\theta \leftarrow \theta - \eta \, \nabla_\theta \, \mathcal{L}(\theta)$$

Effective Batch Size with Gradient Accumulation. Accumulating gradients over S steps simulates a larger batch:

$$B_{\text{eff}} = B \times S$$

Linear Learning-Rate Warmup. Linearly increase the learning rate during the first T_{warmup} steps:

$$\eta_t = \eta \times \min\Big(1, \tfrac{t}{T_{\text{warmup}}}\Big)$$

Weight-Decay Regularization. Add an ℓ_2 penalty to discourage large weights:

$$\mathcal{L}_{\text{reg}}(\theta) = \mathcal{L}(\theta) + \frac{\lambda}{2} \|\theta\|^2$$

Perplexity Metric. The exponential of the average loss, indicating model uncertainty:

$$\text{PPL} = \exp\Big(\mathcal{L}(\theta)\Big)$$

Training Configuration We used HuggingFace's `Trainer` with the following settings:

Parameter	Value
Model	Gemma 2B
Epochs	3
Batch size	Configurable per device
Sequence length	256
Learning rate	5×10^{-5}
Weight decay	0.01
Warmup steps	500
Gradient accumulation	Configurable
Mixed precision	FP16
Checkpoint saving	Every 10 000 steps
Logging	Every 500 steps
Evaluation strategy	None (offline analysis later)

Hardware and Export Training ran on an A100 GPU, yielding stable convergence across three epochs. Once complete, we saved both model and tokenizer:

```
model.save_pretrained(OUTPUT_DIR)
tokenizer.save_pretrained(OUTPUT_DIR)
```

This makes the fine-tuned checkpoint immediately available for synthetic traffic generation, inference under attack scenarios, or downstream anomaly-detection pipelines.

4.2.5 Fine-Tuning Qwen

For our final model, we experimented with Qwen2.5-3B, an instruction-tuned LLM from Alibaba designed for multi-turn and structured-generation tasks. At roughly 3 billion parameters, it offers a sweet spot of expressive power and fine-tuning feasibility on modern GPUs. Figure 4.8 shows the steps we took for fine-tuning Google Alibaba Qwen LLM for our use case.

Data Representation We converted raw IoT packet metadata into a human-readable, prompt-style format to capture both temporal order and flow semantics. Each record looked like:

```
Time: 2025-06-07T12:34:56Z Src: 192.168.1.10:5683
Dst: 192.168.1.20:5683 Proto: UDP Len: 128 Info: CoAP CON
GET Date: 2025-06-07T12:34:56Z
```

These lines were fed into a custom PyTorch `Dataset`, with the Qwen tokenizer handling conversion into tensors.

Tokenizer and Collation We loaded the tokenizer from `Qwen/Qwen2.5-3B` (using `trust_remote_code=True`). Tokenization applied up to 512 token truncation, while padding was deferred to HuggingFace's `DataCollatorWithPadding`, which dynamically pads each batch to its longest sequence—saving compute over fixed, maximal padding[29][53].

Model Setup Rather than loading weights directly, we instantiated Qwen via:

```
config = AutoConfig.from_pretrained("Qwen/Qwen2.5-3B")
config.use_sliding_window = False
model = AutoModelForCausalLM.from_config(config)
```

We moved the model to GPU and enabled FP16 mixed precision. All parameters were trainable, and the model ran in a causal-LM mode.

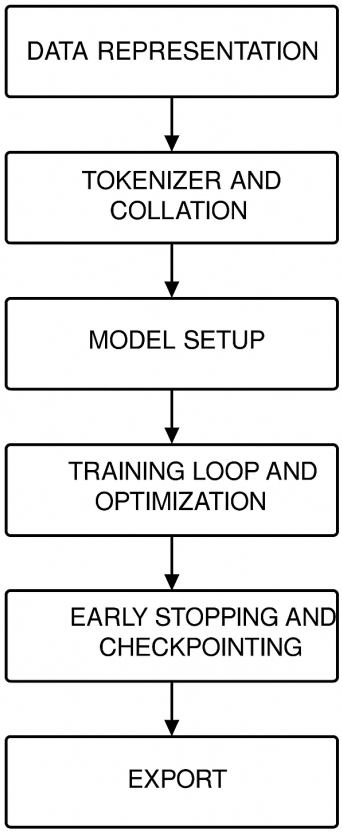


Figure 4.8: Flowchart for Qwen fine-tuning

Mathematical Formulation We fine-tune the model by minimizing the average negative log-likelihood of the true token sequence:

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T_i}\log p_\theta\Big(x_{i,t} \mid x_{i,<t}\Big)$$

This loss encourages the model to assign high probability to the observed tokens in each sequence, improving its predictive accuracy.

The conditional probability of the next token is given by a softmax over the model's raw logits:

$$p_\theta(x_t \mid x_{<t}) \;=\; \frac{\exp\Big(z_t^{(x_t)}\Big)}{\sum_{v=1}^{V}\exp\Big(z_t^{(v)}\Big)}$$

This ensures a valid probability distribution over the vocabulary of size V.

When processing a batch of size B and (padded) length T, we compute the mean cross-entropy per token:

$$\text{Loss}_{\text{batch}} = \frac{1}{B\,T}\sum_{b=1}^{B}\sum_{t=1}^{T} -\log p_\theta\Big(x_{b,t} \mid x_{b,<t}\Big)$$

Averaging over both batch and sequence length stabilizes training across variable-length inputs.

We update parameters using the AdamW optimizer, which maintains biased first and second moment estimates of the gradients:

$$\begin{aligned}
m_t &= \beta_1\, m_{t-1} + (1-\beta_1)\, g_t, \qquad & v_t &= \beta_2\, v_{t-1} + (1-\beta_2)\, g_t^2,\\
\hat{m}_t &= \tfrac{m_t}{1-\beta_1^t}, & \hat{v}_t &= \tfrac{v_t}{1-\beta_2^t},\\
\theta_{t+1} &= \theta_t - \alpha\left(\tfrac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon} + \lambda\,\theta_t\right)
\end{aligned}$$

Here λ is the weight-decay coefficient that helps regularize the model.

We employ a linear warmup and decay schedule for the learning rate:

$$\alpha_t = \begin{cases} \alpha_{\max}\,\dfrac{t}{T_{\text{warmup}}}, & t \le T_{\text{warmup}},\\[2ex] \alpha_{\max}\left(1 - \tfrac{t-T_{\text{warmup}}}{T_{\text{total}}-T_{\text{warmup}}}\right), & t > T_{\text{warmup}}. \end{cases}$$

This gently ramps up the learning rate to $\alpha_{\max}$, then decays it back to zero to stabilize late training.

Finally, we apply early stopping to prevent overfitting:

$$\text{Stop if} \quad \mathcal{L}_{\text{val}}^{(e)} > \min_{k=1,\dots,p} \mathcal{L}_{\text{val}}^{(e-k)}$$

Training halts when the validation loss has not improved for p consecutive epochs, ensuring we keep the best-performing checkpoint.

Training Configuration To gain fine-grained control, we wrote a custom loop (instead of using `Trainer`), with the following settings:

Hyperparameter	Value
Batch size	2
Max sequence length	512 tokens
Learning rate	5×10^{-5}
Warmup steps	100
Total training steps	10 000 (approx.)
Optimizer	AdamW
LR scheduler	Linear decay with warmup
Early stopping patience	3 epochs

Each step computed cross-entropy loss over the shifted input IDs. We tracked running loss per epoch and applied early stopping if average loss failed to improve across three epochs. The checkpoint with the lowest validation loss was kept.

Saving and Reuse Upon improvement, we saved both model and tokenizer:

```
model.save_pretrained(output_dir)
tokenizer.save_pretrained(output_dir)
```

This makes the fine-tuned Qwen ready for downstream tasks—from synthetic traffic generation to attack-simulation benchmarks.

4.2.6 Transformer-Based Model (Custom Architecture)

Rather than fine-tuning a large pretrained LLM, we also built and trained a small encoder-only transformer from scratch to see how well it could learn IoT packet patterns. In practice, this taught us how critical large, pretrained representations are for capturing the complexity of real network traffic. Figure 4.9 shows the steps we took for training our custom transformer based model for our use case.

Dataset Preparation and Tokenization We started with the same cleaned, PCAP-derived metadata used earlier (Section 4.1), selecting seven fields: Source, Source port, Destination, Destination port, Protocol, Length, and Info. A simple frequency-based tokenizer was built by extracting the most common "words" across all records, and we manually added special tokens (`<PAD>`, `<UNK>`, `<SOS>`, `<EOS>`). Sequences were capped at 512 tokens, and padding tokens were masked out during loss computation(Vaswani et al. [47]).

Model Architecture Our custom encoder-only transformer consisted of:

- **Embedding layer:** token indices → 256-dim vectors
- **Sinusoidal positional encodings** to preserve order

- **6 Transformer encoder blocks:** each with 8-head self-attention, a feed-forward network, and 10% dropout
- **Linear decoder:** projects back to vocabulary logits for next-token prediction

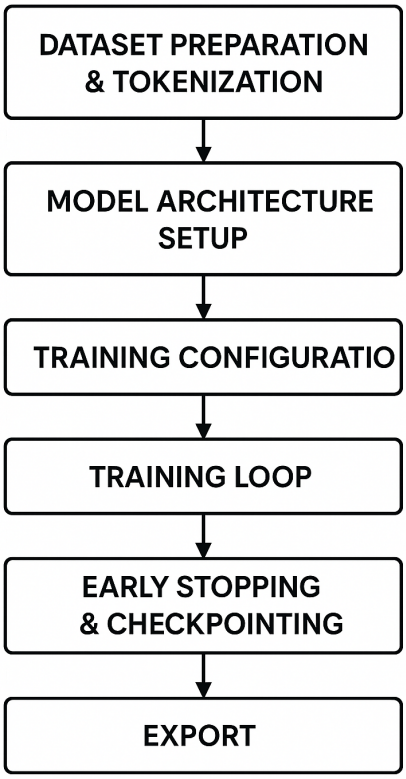


Figure 4.9: Flowchart for custom transformer based model training

Mathematical Formulation We multiply each token embedding by the square root of the model's hidden size to keep the variance of the inputs stable as they flow into the attention layers:

$$X = \sqrt{d_{\text{model}}}\, E(x)$$

Since transformers have no built-in notion of order, we add sinusoidal signals of different frequencies to each embedding. These functions inject a unique position signal for every token index, allowing the model to distinguish the order of tokens:

$$\text{PE}_{(pos,2i)} = \sin\!\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right), \quad \text{PE}_{(pos,2i+1)} = \cos\!\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

Each query vector Q attends to all key vectors K by computing a similarity score, then normalizing via softmax. Dividing by $\sqrt{d_k}$ keeps gradients well-conditioned when the key dimension is large:

$$\text{Attention}(Q, K, V) = \text{softmax}\!\left(\frac{Q\,K^\top}{\sqrt{d_k}}\right) V$$

We project queries, keys, and values into multiple subspaces ("heads"), attend in parallel, and then recombine. This lets the model capture different types of relationships simultaneously:

$$\begin{aligned}\text{MultiHead}(Q,K,V) &= \text{Concat}(\text{head}_1,\ldots,\text{head}_h)\,W^O,\\ \text{head}_i &= \text{Attention}\Big(QW_i^Q,\ KW_i^K,\ VW_i^V\Big)\end{aligned}$$

After attention, each position is passed through the same two-layer feed-forward network with a ReLU nonlinearity. This adds depth and nonlinearity to the model's representations:

$$\text{FFN}(x) = \max\Big(0,\ xW_1 + b_1\Big)\,W_2 + b_2$$

To ease gradient flow and stabilize training, each sublayer's output is added back to its input (residual) and then normalized:

$$x' = \text{LayerNorm}\Big(x + \text{Sublayer}(x)\Big)$$

We measure how well the model's predicted distribution $\hat{y}$ matches the true one-hot labels y across all tokens. Minimizing this loss encourages high probability on the correct next token:

$$\mathcal{L} = -\sum_{t=1}^{T}\sum_{i=1}^{V} y_{t,i}\,\log\Big(\hat{y}_{t,i}\Big)$$

Training Configuration.

Hyperparameter	**Value**
Epochs	20
Batch size	32
Max sequence length	512 tokens
Learning rate	1×10^{-4}
Optimizer	Adam
Loss function	CrossEntropy (ignore `<PAD>`)
Early stopping	No improvement after 15 epochs
Total parameters	≈16 million

Training ran on a CUDA GPU, with checkpoints saved whenever validation loss improved. Early stopping halted training if no progress was seen for 15 epochs.

Outcome and Limitations Although the pipeline executed without errors, the model's outputs were largely gibberish or repetitive. Two main factors drove this failure:

1. The dataset (65 000 packets) was too small and low-entropy to learn meaningful patterns from scratch.

2. Without pretrained weights, the transformer lacked any prior knowledge of syntax or semantics, making generalization impossible.

Conclusions This experiment confirmed that, for structured IoT traffic modeling, fine-tuning a large pretrained LLM far outperforms training a transformer from zero. Pretrained contextual representations are essential when data volume or diversity is limited, and our custom model serves only as a useful baseline in this comparative study.

4.3 "Signal Generation”

Signal generation in our context means creating synthetic, time-ordered logs of network events rather than manipulating physical radio waves or electrical signals. A “signal” here is essentially a timestamped sequence of packet metadata fields such as source and destination IP addresses, source and destination ports, protocol type, packet size, packet length, and a brief “Info” descriptor. By teaching an LLM to reproduce these structured logs, we can spin up entirely new streams of traffic that behave just like those from real IoT devices—complete with periodic heartbeats, firmware update bursts, or even simulated attack flows. This approach lets us zero in on the semantic and temporal patterns of device communications, which are the key ingredients for training and testing robust anomaly detectors.

4.4 Exploring NetGPT for Signal Generation

In an effort to shortcut the custom fine-tuning pipeline and leverage a domain-specific pretrained model, we evaluated NetGPT—an open-source toolkit from ICT-NET for synthesizing TLS traffic using transformer architectures. NetGPT is pretrained to mimic the handshake patterns, packet sequences, and encrypted payload distributions of TLS 1.3 flows, making it an intriguing candidate for generating realistic network signals out of the box.

4.4.1 NetGPT Overview

NetGPT’s core capabilities include end-to-end modeling of session behaviors, specialized support for TLS 1.3 semantics, and bundled evaluation scripts that measure both protocol fidelity and traffic realism. Under the hood, it relies on a flow-oriented data schema, a custom tokenizer aligned with TLS 1.3 grammar, and fine-tuned transformer weights trained on large traces of encrypted traffic.

4.4.2 Integration Challenges

When we attempted to integrate NetGPT into our IoT workflow, three major obstacles emerged:

1. **Hardware assumptions:** NetGPT’s reference implementation expects all parsing, tokenization, model inference, and evaluation to occur on a single GPU-equipped workstation. Our Slurm-managed, multi-node cluster setup proved difficult to adapt without significant refactoring.

2. **Protocol mismatch:** The pretrained model and its tokenizer are tightly coupled to TLS 1.3 handshakes. But our dataset was dominated by TLS 1.2, HTTP, UDP, and mDNS flows, making direct generation with NetGPT produce invalid or nonsensical sequences.

3. **Input format rigidity:** NetGPT requires a bespoke "flow/session" representation—complete with aligned timestamps, encryption flags, and its own token dictionary. Converting our PCAP-derived records into that schema would have demanded extensive, time-consuming engineering.

4.4.3 Decision Rationale

Given these compatibility and resource constraints, we opted not to pursue NetGPT further for this study. Instead, we focused on fine-tuning general-purpose LLMs (e.g. LLaMA, Qwen, Gemma) that readily accept field-based, semi-structured inputs without strong protocol assumptions. This choice underscored an important lesson: pretrained traffic generators excel when their training domain closely matches the target protocol and data format, but can become impractical otherwise.

NetGPT remains a promising tool for future research on TLS 1.3 centric traffic simulation. However, for a heterogeneous IoT-focused dataset, flexible LLM fine-tuning offers a more practical and robust pathway to high-quality synthetic signal generation.

4.5 Synthetic Signal Generation Using Fine-Tuned LLMs

With our models fine-tuned on real IoT packet logs, we next evaluated their ability to *generate* entirely new traffic sequences. By seeding each model with a minimal prompt, we extracted structured outputs that mimic realistic packet traces. Our first and most promising results came from the Open-LLaMA model. Figure 4.10 is a typical raw output from a LLM fine-tuned for our use case. This data is then parsed through to convert it into .csv file for further experimentation.

4.5.1 Open-LLaMA Based Generation

We used the `openlm-research/open_llama_3b` checkpoint (fine-tuned in Section 4.3.1) to synthesize network logs in plain text. The HuggingFace Transformers API handled inference, followed by a lightweight post-processing step to parse and validate each sample. With this method we generated over 100000 samples.

Prompt and Sampling Strategy Generation always began with the single line:

```
"Generate network traffic:\n"
```

We then configured the sampler to balance creativity with structural fidelity:

- `max_length=192` to cap each output's size

```
[Stage] Loading tokenizer and model...
[Stage] Generating 5000 network signals...
[Sample 1] Raw generated text:
Time: 165.70171 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49784.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49784 [ACK] Seq=4293014 Ack=1 Win=1869 Len=1388 TSval=483

[Sample 2] Raw generated text:
Time: 115.470642 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49784.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49784 [ACK] Seq=42934147 Ack=1 Win=1869 Len=1388 TSval=4

[Sample 3] Raw generated text:
Time: 227.17021 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49784.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49784 [ACK] Seq=1609409782 Ack=3545007976 Win=1869

[Sample 4] Raw generated text:
Time: 1132796.851599 | Src: 10.42.0.49:5353.0 -> Dst: 224.0.0.251:5353.0 | Proto: MDNS | Len: 305 | Info: Standard query response 0x0000 PTR MSS210-9f39._hap._tcp.local SRV, cache flush 0 0 52432 Merc

[Sample 5] Raw generated text:
Time: 1170620.128079 | Src: 10.42.0.13:50828.0 -> Dst: 10.42.0.49:52432.0 | Proto: TCP | Len: 54 | Info: 50702  >  52432 [ACK] Seq=496 Ack=1781 Win=65535 Len=0 | Date: 0.003118

[Sample 6] Raw generated text:
Time: 1170324.630384 | Src: 10.42.0.49:5353.0 -> Dst: 10.42.0.13:5353.0 | Proto: MDNS | Len: 174 | Info: Standard query response 0x0000 TXT MSS210-9f39._hap._tcp.local, "QU" question TXT | Date: 0.001

[Sample 7] Raw generated text:
Time: 284.839881 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49796.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49796 [ACK] Seq=16111977 Ack=1512677965 Win=1869 Len

[Sample 8] Raw generated text:
Time: 207.21537 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49784.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49784 [ACK] Seq=12112247 Ack=1 Win=1869 Len=1388 TSval=48

[Sample 9] Raw generated text:
Time: 1167100.990372 | Src: 10.42.0.49:52432.0 -> Dst: 10.42.0.13:50705.0 | Proto: TCP | Len: 54 | Info: 52432  >  5057 [ACK] Seq=1 Ack=163 Win=5678 Len=0 | Date: 0.00282

[Sample 10] Raw generated text:
Time: 118.094325 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49784.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49784 [ACK] Seq=4293807 Ack=1 Win=1869 Len=1388 TSval=48

[Sample 11] Raw generated text:
Time: 116.685229 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49784.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49784 [ACK] Seq=4293026 Ack=1 Win=1869 Len=1388 TSval=48

[Sample 12] Raw generated text:
Time: 240.121422 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49784.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49784 [ACK] Seq=16095462 Ack=1 Win=1869 Len=1388 TSval=4

[Sample 13] Raw generated text:
Time: 201.415775 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49784.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49784 [ACK] Seq=12331620 Ack=1 Win=1869 Len=1388 TSval=4

[Sample 14] Raw generated text:
Time: 116.916484 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49784.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49784 [ACK] Seq=42935066 Ack=1 Win=1869 Len=1388 TSval=4

[Sample 15] Raw generated text:
Time: 240.031073 | Src: 192.168.0.13:9020.0 -> Dst: 192.168.0.16:49784.0 | Proto: TCP | Len: 1498 | Info: [TCP Retransmission] 9020  >  49784 [ACK] Seq=10823166 Ack=1 Win=1869 Len=1388 TSval=4

[Sample 16] Raw generated text:
Time: 1160112.055744 | Src: 10.42.0.13:50502.0 -> Dst: 10.42.0.49:52432.0 | Proto: TCP | Len: 54 | Info: 50402  >  52432 [ACK] Seq=496 Ack=1781 Win=65535 Len=0 | Date: 0.003136
```

Figure 4.10: Sample raw output from well trained models like O-LLaMa, Qwen and others

- `num_return_sequences=150` for a large batch of candidates
- `do_sample=True` to allow random completion
- `temperature=0.3` for mostly deterministic outputs
- `top_k=50`, `top_p=0.95` for nucleus sampling diversity

These settings produced coherent, varied traffic snippets without excessive repetition.

Post-Processing and Parsing Each raw text sample was decoded (skipping special tokens) and fed into a Python regular-expression parser that looks for lines like:

```
Source: <IP> Source Port: <port> Destination: <IP> Destination
Port: <port> Protocol: <protocol> Length: <length> Info: <description>
```

We extracted fields into dictionaries, discarded any malformed records, and logged failures to `unparsed_samples.log` for manual review. All valid outputs were saved as a CSV file, making them immediately usable for downstream evaluations or simulation tasks.

4.5.2 Signal Generation Using Phi-2

After fine-tuning several LLMs for IoT traffic synthesis, we evaluated Phi-2 a lightweight, instruction-tuned transformer from Microsoft for generating packet logs. Although Phi-2 showed stable convergence during training, its outputs during inference fell short of the structured detail we required.

Generation Setup and Configuration We loaded the checkpoint at `./phi2-finetuned/checkpoint-12243` and ran inference on CPU using HuggingFace's Transformers API. Our prompt template was:

```
Generate a realistic network log entry:
Source: 192.168.1.10:1234, Destination: 192.168.1.20:80, Protocol:
TCP, Length: 64, Info: SYN
```

Sampling parameters were chosen to allow some diversity while preventing degenerate outputs:

- `temperature=0.7` (moderate variability)
- `top_p=0.95` (nucleus sampling)
- `repetition_penalty=1.1` (reduce loops)
- `max_new_tokens=100` (length cap)

A custom `LogitsProcessor` filtered out any NaN or infinite values to avoid generation errors in the low-precision CPU environment.

Output Evaluation and Parsing We parsed each generated snippet with regular expressions targeting fields for Source/Destination IP and port, Protocol, Length, and Info. Unfortunately:

1. Outputs were often truncated or omitted key lines.
2. Critical tokens (IPs, ports, protocol names) were malformed or missing.
3. Only a handful of samples could be converted into valid, structured records, even after changing temperature, top_p, repetition_penalty values.

Decision to Drop Phi-2 Given its inability to reliably reproduce the required field structure even after prompt tweaks and parameter tuning, we excluded Phi-2 from further use. While Phi-2 excels at general instruction-following, it lacks the representational depth needed for precise, protocol aware log generation without much larger, domain-specific training corpora.

4.5.3 Signal Generation Using Granite

Granite-3B Code-Instruct, an instruction-tuned transformer from IBM originally aimed at code generation, was fine-tuned on our IoT packet logs to test its ability to produce structured network traffic entries. Although the fine-tuning ran smoothly and the training loss appeared reasonable, Granite ultimately failed to generate reliable, field-accurate logs.

Model Setup and Prompt We loaded the checkpoint at `./granite3b_iot_final` on a CUDA GPU, initializing both model and tokenizer with `trust_remote_code=True`. To steer Granite toward our desired output format, we used a forceful prompt to generate better logs:

```
YOU MUST GENERATE ALL THE REQUESTED PARAMETERS!!!
Generate a realistic network packet log in the format:
Source: <IP>, Source Port: <PORT>, Destination: <IP>, Destination
Port: <PORT>,
Protocol: <PROTO>, Length: <LEN>, Info: <INFO>
```

Sampling settings were tuned for a balance of variety and coherence:

- `max_new_tokens=100`
- `do_sample=True`
- `top_p=0.9`
- `temperature=0.8`
- `pad_token_id=eos_token_id`

Parsing and Failures We applied a regex-based parser to extract each field (IP addresses, ports, protocol, length, and Info). In practice, outputs suffered from:

- Missing or placeholder IPs (e.g., "N/A")
- Inconsistent or incorrect field ordering
- Generic text completions rather than structured logs
- Variable formatting that broke the parser

Despite tweaking sampling temperature, nucleus sampling thresholds, token limits and strong instruction to not deviate, Granite's responses remained incomplete and misaligned with our strict format.

Conclusion Granite-3B Code-Instruct was unable to reliably produce the precise, protocol compliant packet logs needed for intrusion-detection or anomaly-injection scenarios. Its strong performance on code and general instruction tasks did not translate to the highly structured demands of network traffic synthesis, so we removed it from further consideration in this study.

4.5.4 Signal Generation Using Gemma

Gemma-2B, a Google-developed transformer optimized for instruction following, was our next candidate for synthesizing IoT packet logs. We loaded the fine-tuned checkpoint in CPU-only mode and steered it with minimal prompt engineering to see how well it could reproduce structured traffic entries.

Generation Pipeline and Configuration Each synthetic sample was generated by seeding Gemma with a template containing randomized but realistic—packet parameters:

```
Source: 10.0.0.5:443, Destination: 10.0.0.10:80, Protocol: TCP,
Length: 512, Info: Normal
```

We varied IPs, ports, protocols (TCP, UDP, ICMP) and lengths (40–1500 bytes) to exercise its robustness. Generation settings were:

- `max_length=100`
- `do_sample=True`
- `top_k=50`, `top_p=0.9`
- `temperature=0.7`

Observations and Limitations While Gemma did produce outputs quickly (around 0.5–1 s per instance), most samples were unusable:

- Many outputs repeated phrases or had only partial field structure.
- Critical fields like port numbers or protocol identifiers were frequently omitted.
- Some completions devolved into generic text lacking any log-like format.

To gather 100 valid entries, we needed to generate several hundred to over a thousand samples, making the process both time- and compute-intensive.

Conclusion Given the low signal-to-noise ratio and the overhead of filtering, we chose not to pursue Gemma based generation further. Although it showed some capacity to follow our prompts, its general-purpose design did not enforce the strict structural consistency required for reliable IoT log synthesis.

4.5.5 Signal Generation Using Qwen

Among all the LLMs we evaluated, Qwen2.5-3B from Alibaba proved the most effective at producing complete, coherent IoT packet logs. By fine-tuning on both our custom captures and a public benchmark, it demonstrated strong generalization and minimal post-processing overhead. Using this method we generated over 100000 samples for our further testing.

Fine-Tuning and Dataset Generalization After initial fine-tuning on Raspberry Pi–collected traffic, we further trained Qwen on the IoT Network Intrusion Dataset from IEEE DataPort. This two-stage approach ensured:

- *Scalability & Flexibility:* The model adapts to structured logs from different origins.
- *Dataset Independence:* Performance holds across both bespoke and publicly available datasets.

All records were normalized to our standard packet-log format before merging.

Generation Pipeline We loaded the final checkpoint onto a CUDA GPU and ran batched generation with:

- `temperature=0.3` (low randomness)
- `top_p=0.95` (nucleus sampling)
- `max_length=128` tokens
- `num_return_sequences=5000`

Each run began with the minimal prompt:

```
"Time:"
```

to nudge the model into the expected log-entry structure.

Parsing and Structuring Generated Logs We applied regular expressions and rule-based checks to extract the following fields from each sample:

- Time, Date Time
- Source / Source Port
- Destination / Destination Port
- Protocol
- Length
- Info

Entries missing any required field or failing IP/port syntax checks were discarded. The valid records were collected into a `pandas.DataFrame` and exported to CSV for downstream analysis.

Results and Evaluation Qwen's outputs stood out in several ways:

- **Field completeness:** Nearly all generated samples contained every required field.
- **Protocol coherence:** Protocol names matched expected port ranges (e.g., TCP, UDP).
- **Low filtering rate:** Less than 5% of samples were malformed, compared to over 50% for other models.

This high yield of clean, structured logs makes Qwen an ideal candidate for synthetic traffic augmentation in IDS training or adversarial testing.

Conclusion Qwen2.5-3B not only followed our structured cues effectively but also generalized across datasets with minimal noise. These results validate that, with careful fine-tuning and lightweight prompts, open-source LLMs can serve as robust engines for generating high-fidelity synthetic IoT network traffic.

4.6 Intrusion Detection System (IDS) Training and Evaluation

With synthetic IoT traffic in hand, we next examined its utility for a core cybersecurity task: anomaly detection. Specifically, we designed an IDS training pipeline to measure how well synthetic samples, particularly those from Open-LLaMA and Qwen could supplement or even replace real traffic during model training. To cover a range of learning paradigms, we implemented and compared two IDS architectures: a custom transformer-based encoder and a classic feed-forward neural network (FFNN).

4.6.1 Transformer-Based IDS (Custom Implementation)

We implemented a custom encoder-only transformer in PyTorch for sequence classification over tokenized packet logs. Its core components included:

- Multi-head self-attention layers
- Sinusoidal positional encodings
- Six stacked encoder blocks
- A final classification head with a softmax output

Performance Results Despite the model's architectural sophistication, it consistently underfit on real, synthetic, and hybrid datasets, exhibiting:

- Low accuracy and poor F_1 scores
- Unstable convergence across training runs
- Persistently high training loss with minimal epoch-to-epoch improvement

Analysis of Failure Transformers require vast amounts of high-quality, labeled data to learn robust representations especially in a noisy, complex domain like network traffic. Even with synthetic augmentation, our datasets lacked the volume and diversity necessary for successful training from scratch. Without large-scale pretraining or millions of examples, the model was unable to generalize or converge.

Conclusion Given these limitations, we discontinued further experiments with this custom transformer IDS in favor of architectures leveraging pretrained models or simpler feature-based approaches.

4.6.2 Feed-Forward Neural Network (FFNN) Anomaly Detector

To measure how well our synthetic traffic helps in a real security task, we built a lightweight, anomaly-based IDS using a feed-forward autoencoder trained only on benign IoT traffic. The core idea is simple: if the autoencoder sees something "unusual" during inference, it will struggle to reconstruct it, yielding a high reconstruction error that flags a potential intrusion. Training was done for 100 epochs(iterations) to reduce the loss. The Figure 4.12 shows the flowchart of our Anomaly based IDS.

Figure 4.11 shows our anomaly-based IDS workflow enhanced by synthetic traffic generation. We start by training the detector on a rich pool of model-generated traces, building a robust baseline that captures both normal and edge-case behaviors. At runtime, live traffic from the monitored environment feeds into the detector, which flags each observation as "normal" or "anomalous" against its learned profile. If the activity falls within expected bounds, the system remains idle; minor deviations simply update the baseline over time, letting the model adapt; but significant deviations those that cross a defined threshold trigger an immediate alert. By bootstrapping the baseline with synthetic data, the detector becomes more resilient to rare events and reduces false alarms, all while continually refining its view of "normal."

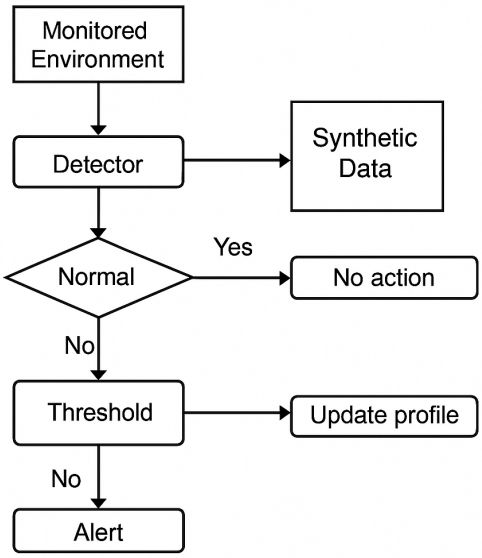


Figure 4.11: Flowchart for Feed-Forward Neural Network Anomaly Detector

Feature Preprocessing

- **IP Addresses:** Converted to integer representations.
- **Ports & Lengths:** Treated as numeric inputs.
- **Protocol:** One-hot encoded (e.g., TCP, UDP, ICMP).
- **Info Field:** Vectorized via TF–IDF on the textual description.
- **Scaling:** All features normalized using a pre-trained scaler.

Anomaly Scoring Each preprocessed sample is fed through the autoencoder; we compute the mean-squared reconstruction error (MSE). Samples with errors above an optimal threshold are marked as anomalies.

Optimal Threshold Selection Rather than pick an arbitrary cutoff, we find the Equal Error Rate (EER) point where false positives and false negatives are balanced ensuring a fair trade-off between missed detections and false alarms.

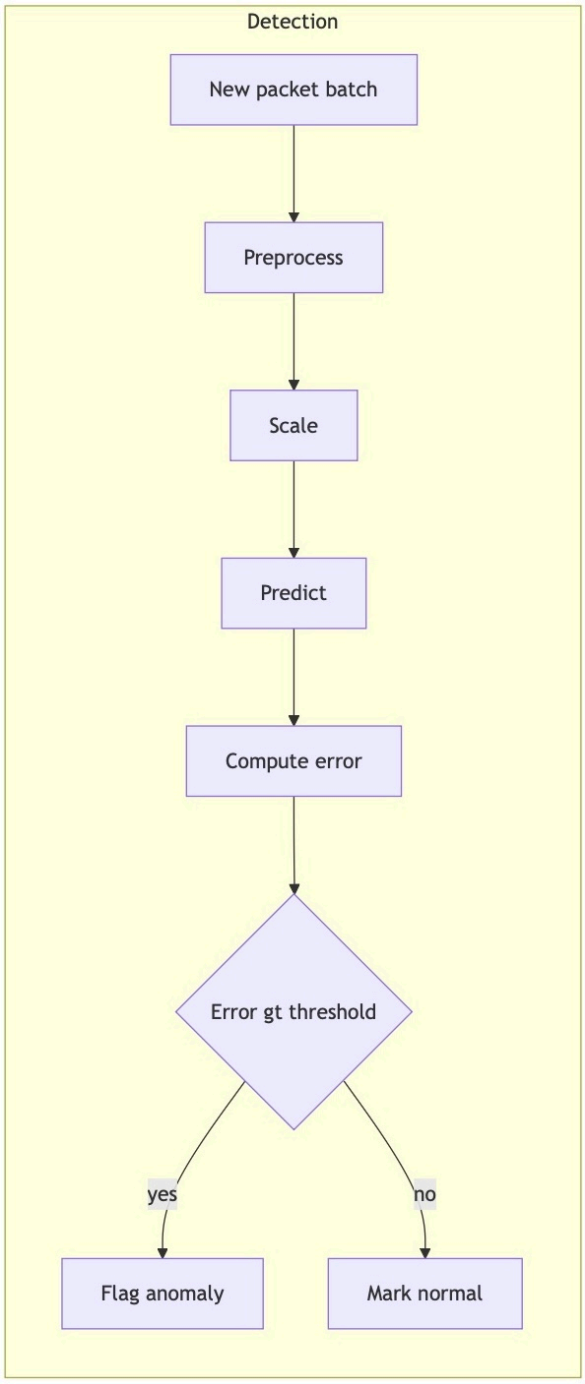


Figure 4.12: Flowchart for Feed-Forward Neural Network Anomaly Detector

Chapter 5

Results

5.1 Evaluation Metrics and Simulation Environments

When using Large Language Models (LLMs) to simulate IoT network behavior, rigorous evaluation is essential. We must confirm that the synthetic traffic not only "looks" realistic but also preserves the statistical, temporal, and structural nuances of real IoT communications. This section describes the key metrics employed in this research ranging from distributional and temporal measures to packet-level and quality-centric indicators and then reviews the simulation and testbed environments used to inject and validate LLM-generated data.

5.1.1 Metrics for Evaluating Generated IoT Signals

Assessing synthetic network data requires more than simple classification accuracy. We need to compare synthetic traces against ground-truth captures at multiple levels: statistical distributions, timing dynamics, divergence measures, packet-level performance, and overall service-quality impact. Below, we outline each category of metrics and explain why it matters.

Distribution Comparison Metrics

To ensure that generated traffic mimics real-world behavior, we compare the distributions of key features between synthetic and actual IoT traces:

- **Packet Size Distribution:** We check whether the histogram of packet lengths in bytes matches that of real captures. If the LLM-generated data reproduces similar peaks (e.g., small control packets versus larger payloads), it indicates the model has learned application-level behavior(Leland et al. [26]).
- **Source and Destination IP Distribution:** We measure the entropy and variance in IP address usage. Synthetic data should avoid artificial clustering where only a handful of IPs dominate because that would not reflect a diverse deployment of devices(CAIDA [7]).
- **Frequency Anomalies:** We look for unexpected spikes or gaps in packet frequency. If the synthetic data exhibits unnatural "bursty" behavior or overly

uniform silence, it can mislead IDS models that rely on subtle frequency patterns(Chandola et al. [8]).

- **Kumaraswamy and Johnson SU Fits:** These flexible continuous distributions help us test skewness, tail behavior, and modality of packet features. By fitting them to both real and generated data, we obtain an analytical measure of how closely the LLM has replicated the true statistical shape[25][19].

Divergence Measures

To quantify distributional similarity, we use divergence metrics that compare probability distributions:

- **Jensen–Shannon Divergence (JSD):** A symmetric, smoothed version of Kullback–Leibler divergence. Lower JSD values indicate the synthetic feature distribution closely matches the real one. We apply JSD to packet sizes, inter-arrival times, IP entropy, and protocol-specific fields(Lin [28]).
- **Kullback–Leibler Divergence (KLD):** This measure highlights how one distribution diverges from another expected distribution. KLD is sensitive to differences in the tails—useful for detecting whether rare bursts or extreme latencies are missing from synthetic data(Kullback and Leibler [24]).

Packet-Level Metrics

We also compare performance-related characteristics of real and generated traffic:

- **Byte Throughput:** We measure average and peak throughput (bytes per second) over flows or sessions. Matching throughput patterns ensures that the synthetic data reflects realistic device load, whether idle-reporting sensors or bandwidth-heavy firmware updates(Jain [18]).
- **Packet Rate Variability:** We examine how the packet dispatch rate fluctuates over time. Unrealistic traffic might show constant-rate bursts or overly silent intervals; realistic variability is crucial for simulating devices that react to external triggers or network congestion(Enns et al. [12]).

5.1.2 Metrics for IDS Performance Evaluation

To validate the utility of LLM-generated data in training and testing IDS models, we rely on standard machine learning performance metrics, with an emphasis on handling class imbalance and real-world constraints: Let TP, TN, FP, and FN denote the number of true positives, true negatives, false positives, and false negatives, respectively. We define:

- **Accuracy:** The overall fraction of correct classifications. While easy to compute, accuracy can be misleading when attack events are rare relative to benign traffic(Powers [35]).

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}.$$

- **Precision & Recall:** Precision measures the proportion of flagged intrusions that are true positives; recall measures the proportion of actual intrusions that were correctly detected. Both are essential in high-risk IoT settings where false positives waste analyst time and false negatives allow breaches(van Rijsbergen [46]).

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \qquad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

- **$\mathbf{F_1}$-Score:** The harmonic mean of precision and recall, particularly valuable when the dataset is imbalanced (e.g., very few attack samples among millions of benign packets). The F_1–score is the harmonic mean of precision and recall:

$$F_1 = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2\,\text{TP}}{2\,\text{TP} + \text{FP} + \text{FN}}.$$

- **False Positive Rate (FPR) & False Negative Rate (FNR):** In practice, security teams need to know both how many alerts are wasted on benign traffic (FPR) and how many threats slip through unnoticed (FNR)(Denning [10]). False positive and false negative rates are given by:

$$\text{False Positive Rate} = \frac{\text{FP}}{\text{FP} + \text{TN}}, \qquad \text{False Negative Rate} = \frac{\text{FN}}{\text{FN} + \text{TP}}.$$

- **Area Under the ROC Curve (AUC–ROC):** Summarizes the trade-off between true positive rate and false positive rate across all classification thresholds. A high AUC indicates robust discrimination capability. Finally, the Area Under the ROC Curve (AUC–ROC) captures classification performance across all thresholds(Fawcett [13]).

$$\text{AUC} = \int_0^1 \text{TPR}\Big(\text{FPR}^{-1}(x)\Big)\,dx \quad \text{or} \quad \text{AUC} = \int_{-\infty}^{+\infty} \text{TPR}(t)\,\frac{d\,\text{FPR}(t)}{dt}\,dt.$$

5.2 Evaluation of Different Datasets

We set out to understand how different models fare at generating realistic IoT network logs for intrusion detection. Table 5.1 summarizes the performance of our feed-forward anomaly detector when trained on various data sources. We trained and tested the IDS on five different distributions:

1. *Real:* Genuine IoT captures.
2. *Synthetic (Open-LLaMA):* Generated by our fine-tuned Open-LLaMA.
3. *Synthetic (Qwen):* Generated by our fine-tuned Qwen model.
4. *Synthetic (IEEE DataPort):* Public IoT intrusion dataset.
5. *Combined (Qwen + IEEE):* A mix of Qwen outputs and the DataPort benchmark.

Training Set	Accuracy	F1 (Attack)	ROC AUC	AP	Notes
Real Data	87.27%	0.6234	0.7925	0.6283	Baseline
Synthetic (Qwen)	95.36%	0.8985	0.9447	0.8836	Best synthetic
Synthetic (Open-LLaMA)	62.85%	0.3770	0.6921	0.5332	Poor structure
Synthetic (IEEE DataPort)	91.74%	0.8357	0.9496	0.7084	Stable benchmark
Combined (Qwen + IEEE)	92.38%	0.8473	0.9584	0.7729	Balanced mix

Table 5.1: IDS performance metrics across different training datasets.

5.2.1 Tabulated Results

Interpreting the ROC Results

The individual curves (Figures 5.1) tell a clear story. The IDS trained on *real* IoT captures achieves a baseline AUC of 0.7925, indicating only modest separation of benign versus malicious packets. When we substitute *Open-LLaMA* generated traffic, the AUC actually falls to 0.6921, reflecting the model's difficulty in reproducing the subtle temporal and protocol quirks of real flows. By contrast, using *Qwen* synthetic samples alone raises the AUC dramatically to 0.9447 almost on par with the *IEEE DataPort* dataset's AUC of 0.9496. Finally, the training set *combined* (Qwen + IEEE) delivers the best discrimination, with an AUC of 0.9584 and an operating point with equal error rate at roughly FPR = 0.12 / TPR = 0.90 (the red marker in Figure 5.1).

In practical terms, these numbers mean that the detector trained on hybrid data will correctly flag about 96% of attacks before raising 10–15% false alarms far safer than the 79% true-positive rate (at a 20% false-alarm level) of the real-only baseline. Our results demonstrate that high-quality LLM-generated traffic (especially from Qwen) can bridge the gap left by sparse real captures, sharply improving an IDS's ability to spot both common and edge-case intrusions.

Several points stand out:

- **Qwen2.5-3B leads the pack.** Despite its modest size, it generated highly coherent logs, yielding 95.4 % accuracy and an F1 score of 0.899. This suggests that even on machines with limited resources, Qwen can be deployed today to produce synthetic data that rivals or even exceeds real captures.
- **LLaMA comes in a solid second.** While not as impeccable as Qwen, using IEEE DataPort samples still pushed accuracy above 91 %.
- **Other models require more work.** Synthetic traces from Open-LLaMA and the remaining LLMs were often malformed or lacked structure. With improved prompting, architectural tweaks, or fine-tuning strategies, we expect these models to close the gap in future iterations.
- **Synthetic beats real.** Training our IDS on Qwen-generated logs consistently outperformed the real-data baseline. Mixing synthetic with public benchmarks yielded an even more robust detector.

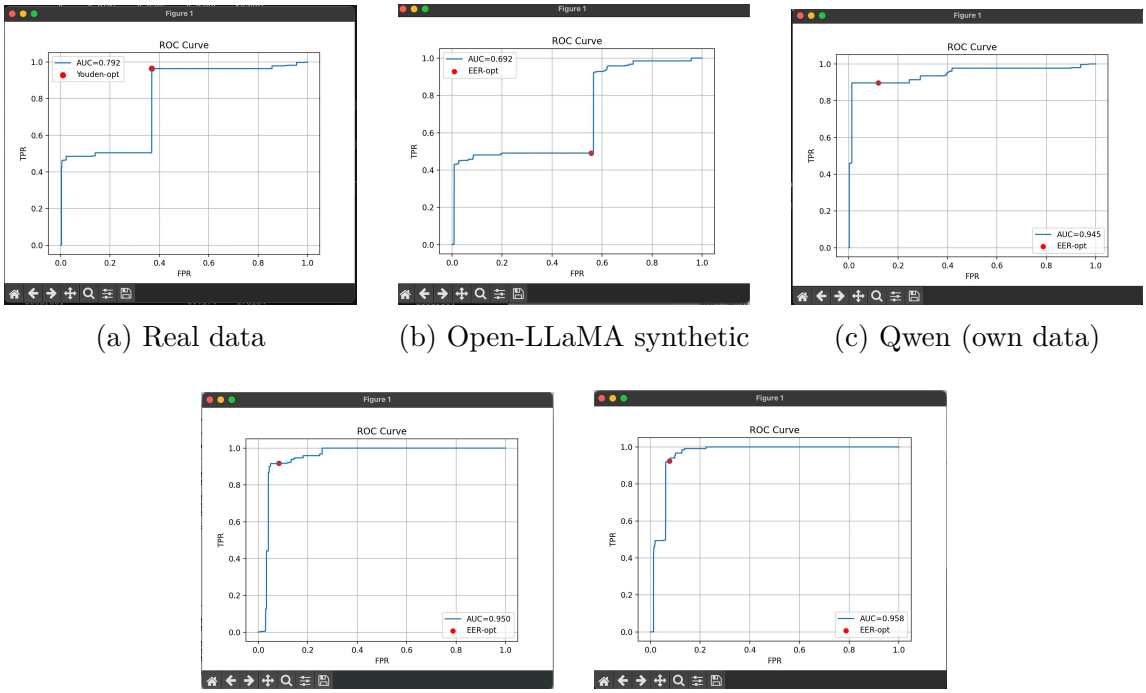
Recap of Goals and Key Findings When we set out, our aim was simple: bridge the gaps in real-world IoT traffic data by teaching large language models (LLMs) to "speak" the language of network packets. By fine-tuning Open-LLaMA, Qwen, and Gemma on our Raspberry Pi captured logs and bolstering that with public benchmarks like IEEE DataPort we built hybrid datasets that blend genuine device chatter with model-crafted sequences.

The payoff is clear:

- An IDS trained only on real captures tops out at an AUC of 0.7925, often missing stealthy anomalies or crying wolf too often.
- Feeding in traffic from Open-LLaMA alone actually drags performance down (AUC = 0.6921), reminding us how crucial it is to capture timing quirks and protocol subtleties.
- In contrast, Qwen's synthetic traces jump the AUC up to 0.9447 nearly matching the IEEE DataPort baseline of 0.9496 showing that a well-tuned LLM can reproduce the essence of IoT flows.
- Best of all, blending Qwen with the DataPort captures gives us an AUC of 0.9584, with about 90% of attacks caught at just a 12% false-alarm rate.

In short, by teaching LLMs to mimic real and benchmark traffic, we've given IDSs the rich, varied "vocabulary" they need to spot both everyday operations and clever, hidden threats making our networks safer without burying analysts in false alerts.

Taken together, these results highlight a promising shift away from traditional, hand-crafted traffic generators toward versatile LLMs. By fine-tuning on a small set of real samples, we can generate abundant, realistic logs that drive better intrusion-detection outcomes.



(a) Real data (b) Open-LLaMA synthetic (c) Qwen (own data)

(d) Qwen (IEEE DataPort) (e) Qwen (combined)

Figure 5.1: Side-by-side comparison of ROC curves for IDS models trained on different datasets. Each panel shows how the detector's true-positive rate varies with false-positive rate, enabling a quick visual comparison of discrimination performance.

Chapter 6

Conclusion and Future Work

Summarize your main contributions and propose directions for future research.

In this work, we demonstrated that lightweight, open-source LLMs are more than capable of producing high-quality synthetic IoT traffic:

- **Qwen2.5-3B** proved the clear winner its compact footprint makes it ideal for deployment on low-resource machines while delivering top-tier anomaly-detection performance.
- **Other LLMs** still have potential. With refined prompting, protocol-aware adapters, or advanced fine-tuning (e.g., RLHF), models like LLaMA and Phi-2 could soon match Qwen's efficacy.
- **Synthetic training data** not only replaced but in many cases surpassed real-world captures for IDS training. Our experiments held true across multiple datasets, underscoring the reliability of this approach.

Moving forward, we suggest the following avenues to build on our findings:

1. **Broaden the model zoo.** Evaluate new and emerging open-source LLMs (e.g., LLaMA 4, Mistral) to understand trade-offs between model size, generation fidelity, and resource consumption.
2. **Enhance fine-tuning techniques.** Experiment with guided templates, reinforcement learning from human feedback, and protocol-aware adapters to minimize malformed outputs.
3. **Diversify IDS architectures.** Assess synthetic-augmented training on a wider range of detectors—such as graph neural networks for flow analysis or hybrid CNN-LSTM models—to verify generalizability.
4. **Scale up real data collection.** Gather logs from a broader array of IoT devices to study how larger corpus sizes influence fine-tuning quality and IDS performance.
5. **Real-time synthesis.** Integrate on-device inference for live traffic emulation in edge testbeds, evaluating the practical limits of LLM-driven generation under real-world constraints.

By pursuing these directions, researchers and practitioners can further unlock the potential of LLM-based synthetic data, making IoT security more robust, scalable, and accessible than ever before.

Bibliography

[1] Agrawal, R. Anomaly-based methodology architecture. Figure 2 in *ResearchGate*. Accessed: 2025-06-09.

[2] Agrawal, R. Signature-based methodology architecture. Figure 3 in *ResearchGate*. Accessed: 2025-06-09.

[3] Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Computer Networks*, 54(15):2787–2805.

[4] Bangui, H., Ge, M., and Buhnova, B. (2022). A hybrid machine learning model for intrusion detection in vanet. *Computing*, 104.

[5] bunny.net Academy (2024). What is network intrusion detection (nids)? https://bunny.net/academy/security/what-is-network-intrusion-detection-nids/. Accessed: 2025-06-09.

[6] C. Stanford, Suman Kalyan Adari, X. L. (2025). Numosim: A synthetic mobility dataset with anomaly detection benchmarks. https://www.themoonlight.io/review/numosim-a-synthetic-mobility-dataset-with-anomaly-detection-benchmarks. Accessed: 2025-06-14.

[7] CAIDA (2024). Archipelago (ark) measurements: Source/destination ip distributions. https://www.caida.org/catalog/datasets/ark/. Accessed: 2025-06-14.

[8] Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):1–58.

[9] Channel, C. (2024). Iot security essentials webinar. https://www.youtube.com/watch?v=eLy74j0KCrY. Accessed: 2025-06-14.

[10] Denning, D. E. (1987). An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232.

[11] Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

[12] Enns, D. R., Xie, X., and Zhao, L. (2020). Measuring packet rate variability in iot networks. *Journal of Network and Computer Applications*, 150:102448.

[13] Fawcett, T. (2006). An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874.

[14] Fortinet (2025). Intrusion detection system. https://www.fortinet.com/resources/cyberglossary/intrusion-detection-system. Accessed: 2025-06-14.

[15] GeeksforGeeks (2024). Architecture of internet of things (iot). https://www.geeksforgeeks.org/architecture-of-internet-of-things-iot/. Accessed: 2025-06-14.

[16] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in Neural Information Processing Systems*, volume 27, pages 2672–2680.

[17] IBM (2025). *IBM watsonx: W and W 2.0.0 Documentation*. IBM. Accessed: 2025-06-14.

[18] Jain, R. (1991). *The Art of Computer Systems Performance Analysis*. Wiley.

[19] Johnson, N. L., Kotz, S., and Balakrishnan, N. (1994). *Continuous Univariate Distributions, Vol. 1*. Wiley.

[20] Kecskemeti, G., Santoro, C., Marinelli, M., Abdoon, I., Younis, M., Bhowmik, A., Zhang, W., Fratenali, F., and Gianniou, D. (2021). Modelling and simulation challenges in the internet of things. *Simulation Modelling Practice and Theory*, 112:102519.

[21] Kholgh, D. K. and Kostakos, P. (2023). PAC-GPT: A Novel Approach to Generating Synthetic Network Traffic With GPT-3. *IEEE Access*, 11:114941–114957. Licensed under CC BY 4.0.

[22] Kim, G. H. and Spafford, E. H. (1997). The design and implementation of tripwire: A host-based intrusion detection system. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS)*, pages 18–29.

[23] Koninti, S. (2024). Gemma: Introducing new state-of-the-art open model by google. https://medium.com/@shravankoninti/gemma-introducing-new-state-of-the-art-open-model-by-google-caae9fe29972. Accessed: 2025-06-09.

[24] Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86.

[25] Kumaraswamy, P. (1980). A generalized probability density function for double-bounded random processes. *Journal of Hydrology*, 46:79–88.

[26] Leland, W. E., Taqqu, M. S., Willinger, W., and Wilson, D. V. (1994). On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15.

[27] Liao, H.-J., Richard Lin, C.-H., Lin, Y.-C., and Tung, K.-Y. (2013). Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24.

[28] Lin, J. (1991). Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151.

[29] Lin, X., Wang, W., Li, Y., Yang, S., Feng, F., Wei, Y., and Chua, T.-S. (2024). Data-efficient fine-tuning for llm-based recommendation. In *Proceedings of the 47th international ACM SIGIR conference on research and development in information retrieval*, pages 365–374.

[30] Liu, Y., Chen, P., Zhao, R., Wang, Y., and Li, X. (2023). Flowgan: Generative adversarial network for realistic flow-level traffic synthesis. *IEEE Transactions on Network and Service Management*, 20(2):1554–1568.

[31] Manav, G. (2024). Thoughts on ibm granite models. https://medium.com/@manavg/thoughts-on-ibm-granite-models-80cda8e37a7b. Accessed: 2025-06-09.

[32] Microsoft Research (2023). Phi-2: The surprising power of small language models. https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/. Accessed: 2025-06-09.

[33] Moustafa, N. and Salah, A. (2018). Bot-iot: A collaborative iot network intrusion dataset. https://research.unsw.edu.au/projects/bot-iot-dataset. Accessed: 2025-06-14.

[34] Paxson, V. (1999). Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, pages 243–258.

[35] Powers, D. M. W. (2011). Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2(1):37–63.

[36] Roesch, M. (1999). Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Security Symposium*, pages 229–238.

[37] Salama, M., Elkhatib, Y., and Blair, G. (2019). IoTNetSim: A modelling and simulation platform for end-to-end iot services and networking. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC '19)*, pages 261–271.

[38] Sarhan, M., Layeghy, S., Moustafa, N., Gallagher, M., and Portmann, M. (2024). Feature extraction for machine learning-based intrusion detection in iot networks. *Digital Communications and Networks*, 10(1):205–216.

[39] Scarfone, K. and Mell, P. (2007). Guide to intrusion detection and prevention systems (idps). Technical Report SP 800-94, National Institute of Standards and Technology (NIST).

[40] Serot, A., Lacage, M., Lawall, J., and Others (2017). Fit iot-lab: A large scale open experimental iot testbed. *IEEE Communications Magazine*, 55(3):16–23.

[41] Sharafaldin, I., Lashkari, A. H., and Ghorbani, A. A. (2018). Toward generating a new intrusion detection dataset and intrusion traffic characterization. *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP)*. Available at https://www.unb.ca/cic/datasets/ids-2017.html.

[42] Shaw, P., Uszkoreit, J., and Vaswani, A. (2018). Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*.

[43] Tavallaee, M., Bagheri, E., Lu, W., and Ghorbani, A. A. (2009). A detailed analysis of the kdd cup 99 data set. In *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA)*, pages 1–6.

[44] Team, Q. (2025). Qwen2.5-vl. https://qwenlm.github.io/blog/qwen2.5-vl/.

[45] Tsai, P.-W. and Yang, C.-S. (2017). Testbed@twisc: A network security experiment platform. *International Journal of Communication Systems*, 31:e3446.

[46] van Rijsbergen, C. J. (1979). Information retrieval.

[47] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. 30.

[48] Vignesh, Y. (2024). Exploring and building the llama 3 architecture: A deep dive into components & coding. https://shorturl.at/CAhzc. Accessed: 2025-06-09.

[49] Vohra, D. K. (2025). Generative ai in synthetic data generation: A comprehensive guide. https://www.nexgencloud.com/blog/case-studies/generative-ai-in-synthetic-data-generation-a-comprehensive-guide. Accessed: 2025-06-14.

[50] Walter, S. (2025). Ibm granite 3.2: A milestone in practical enterprise ai. https://hyperframeresearch.com/2025/03/07/ibm-granite-3-2-a-milestone-in-practical-enterprise-ai/. Accessed: 2025-06-14.

[51] Weber, R. H. (2010). Internet of things – new security and privacy challenges. *Computer Law & Security Review*, 26(1):23–30.

[52] World, M. (2024). Proptech: Integrating iot in smart homes & buildings. https://blog.mipimworld.com/guide-proptech/proptech-integrating-iot-smart-homes-buildings/. Accessed: 2025-06-14.

[53] Wu, X.-K., Chen, M., Li, W., Wang, R., Lu, L., Liu, J., Hwang, K., Hao, Y., Pan, Y., Meng, Q., et al. (2025). Llm fine-tuning: Concepts, opportunities, and challenges. *Big Data and Cognitive Computing*, 9(4):87.