THESIS

DEEP LEARNING FOR IOT FINGERPRINTING

Submitted by

Maxwel A. Bar-on

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2025

Committee:

Advisor: Indrakshi Ray

Bruhadeshwar Bezawada
Indrajit Ray
Anura Jayasumana

Copyright by Maxwel Bar-on 2025

All Rights Reserved

ABSTRACT

DEEP LEARNING FOR IOT FINGERPRINTING

The rapid growth of the Internet-of-Things (IoT) industry has introduced new attack surfaces in home and enterprise networks due to insufficient built-in security measures in many IoT devices. IoT network fingerprinting, the process of identifying IoT devices from their network communication patterns, can be used to select appropriate access controls for vulnerable IoT devices, offering a promising solution to this security problem. Typical state-of-the-art IoT fingerprinting approaches identify devices by applying deep learning models to samples of their network traffic. This work addresses 5 challenges of using deep learning for IoT fingerprinting: (1) traffic collected by a single observer may be insufficient for training an accurate fingerprinting model; (2) variations in communication rates among different IoT devices results in imbalanced datasets, which can lead to biased models; (3) it is difficult for a model to capture the relationships between packets when some packets belong to separate flows; (4) it is inefficient to adapt an existing IoT fingerprinting model to identify new devices; and (5) relying on fixed-length samples of traffic can result in arbitrarily long fingerprinting times. For the first challenge, we propose a federated learning approach for training a fingerprinting model using traffic collected by multiple separate observers. For the second challenge, we propose a hierarchical Mixture-of-Experts that balances training data by grouping devices based on their communication rates before identifying them. For the third challenge, we propose a relative encoding technique for packet endpoints that preserves relationships across flows. For the fourth challenge, we propose a bi-component fingerprinting architecture that efficiently adapts to new devices by reusing a portion of its parameters. Finally, for the fifth challenge, we propose a fixed-time traffic sampling approach. We evaluate our proposed approaches through a series of experiments and demonstrate how each one overcomes its associated challenge in an experimental setting.

ACKNOWLEDGEMENTS

I thank my advisor, Professor Indrakshi Ray, for supporting me, believing in me, and guiding me through my research journey, starting from my sophomore year as an undergraduate, through my Master's and onto my PhD. She has helped me find my way during this time and challenged me to explore new topics, broadening my knowledge and helping me become a better researcher. I thank Professor Bruhadeshwar Bezawada, for pushing me to improve myself and shaping my early career as a researcher by patiently teaching me domain knowledge and skills before I had enough experience to contribute to his work. Professor Bezawada was deeply involved in every aspect of the experiments presented in this thesis. I thank Professor Hossein Shirazi for helping me explore new domains and for helping me refine how I present and communicate my work. I also thank the collaborators of the projects covered in this thesis: Professor Bruhadeshwar Bezawada, Professor Indrakshi Ray, Professor Indrajit Ray, Kati Patterson, Kiley Krosky, Federico Larrieu, and Alanood Alqobaisi. Finally, I would like to thank my friends and colleagues in the DBSec Group.

This work was partially supported by the U.S. National Science Foundation under Grant No. 1822118 and 2335687, Award Numbers DMS 2123761, the member partners of the NSF IUCRC Center for Cyber Security Analytics and Automation – AMI, NewPush, Cyber Risk Research, NIST and ARL – the State of Colorado (grant #SB 18-086) and the authors' institutions. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, or other organizations and agencies.

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iii
LIST OF TABLES vi
LIST OF FIGURES vii

Chapter 1 Introduction 1
1.1 Problem Statement 1
1.2 Challenges of IoT Fingerprinting 2
1.3 Proposed Approach 3
1.4 Research Questions 4
1.5 Organization 5

Chapter 2 Background and System Model 6
2.1 Networking 6
2.1.1 Network Fingerprints 7
2.2 System and Threat Model 7
2.2.1 System Model 7
2.2.2 Threat Model 8
2.3 Deep Learning 9
2.3.1 Loss Functions and Tasks 9
2.3.2 Metrics 10
2.3.3 Multilayer Perceptron 11
2.3.4 Transformer 12

Chapter 3 Related Work 16
3.1 IoT Fingerprinting 16
3.2 Federated Learning 17
3.3 Mixture-of-Experts 17
3.4 Adaptable IoT Fingerprinting 18
3.5 Fixed-Time Traffic Sampling 19

Chapter 4 IoT Fingerprinting With Limited Datasets 20
4.1 Methods 20
4.1.1 Federated Deep Learning 20
4.1.2 Hierarchical Mixture-of-Experts 22
4.2 Evaluation 24
4.2.1 Experimental Setup 24
4.2.2 Federated Learning Evaluation 26
4.2.3 Hierarchical MoE Evaluation 27

Chapter 5 Encoding Relationships in Multi-Flow IoT Traffic 29

5.1 Method . . . 30
5.1.1 Illustrative Example . . . 31
5.1.2 Fingerprinting Architecture . . . 33
5.2 Evaluation . . . 34
5.2.1 Experimental Setup . . . 34
5.2.2 Performance . . . 37

Chapter 6 Adaptable IoT Fingerprinting . . . 38
6.1 Method . . . 40
6.1.1 Behavior Extractor . . . 40
6.1.2 Fingerprint Interpreter . . . 42
6.1.3 Adapting a Model . . . 42
6.2 Evaluation . . . 43
6.2.1 Experimental Setup . . . 43
6.2.2 Efficiency . . . 44
6.2.3 Performance . . . 44

Chapter 7 Fingerprinting with Fixed Time Observations . . . 47
7.1 Methods . . . 48
7.1.1 Traffic Sampling . . . 48
7.1.2 Fingerprinting Architecture . . . 49
7.2 Evaluation . . . 50
7.2.1 Experimental Setup . . . 50
7.2.2 Performance . . . 53

Chapter 8 Conclusion and Future Work . . . 56
8.1 Summary and Outcomes . . . 56
8.2 Future Work . . . 57

References . . . 59

Appendices . . . 66
.1 Appendix A: Hyperparameters . . . 67

LIST OF TABLES

2.1 Classification metrics in terms of true labels $T \in \{1,\ldots,|D|\}^N$ and predicted probabilities $P \in [0,1]^{N\times D}$ for N samples. 11

4.1 Features 24
4.2 Experimental Devices 25
4.3 Average performance of federated and centralized learning. 26
4.4 Average performance of hierarchical vs. standalone MLP. 27

5.1 Attention matrix. 32
5.2 Experimental Devices 35
5.3 Non-relative features. 36
5.4 Performance with vs. without relative features. 37

6.1 Adapting Time in Seconds. 44

7.1 Experimental devices, number of packets and average packets per sample with each time-window. 51
7.2 Percent of total variance in feature value that is accounted for by variance among samples from the same device. 52
7.3 Average performance with fixed-length sampling and fixed-time sampling with different time-window durations. 53

1 Hyperparameter values for chapter 4 experiments. 67
2 Hyperparameters used in chapter 5. 67
3 Hyperparameters of Behavioral Extractor from chapter 6. 68
4 Hyperparameters of MLPs for each task from chapter 6. 68
5 Hyperparameters used in chapter 7 for each sampling approach. 68

LIST OF FIGURES

2.1 Fingerprinting process. 7
2.2 Diagram of Transformer. 12

4.1 Performance of hierarchical MoE vs. standalone MLP on each device. 28

5.1 Example derivation of relative feature values for IP source address for a sample of 4 packets. 30
5.2 Performance with and without relative features. 37

6.1 Diagram of bi-component architecture. 39
6.2 Average recall of bi-component models with different BE training procedures and number of original devices. Static model performance is shown as a horizontal line. . . 45
6.3 Performance of the D bi-component model vs. static model for all devices with 4 and 19 original devices. 46

7.1 Recall of each device with fixed-length sampling and fixed-time sampling with different time-window durations. 54
7.2 Prediction density of NestCam samples. 55

Chapter 1

Introduction

The Internet of Things (IoT) landscape is continuously expanding as manufacturers find new ways to enhance home appliances and industrial equipment by integrating network connectivity. This expansion comes with new security challenges as IoT devices introduce new attack surfaces in home and enterprise networks. Due to weak built-in security mechanisms, these devices are often vulnerable to remote cyberattacks. Remotely compromised IoT devices can be used as entry points into local networks or to launch DDoS-style attacks against critical network infrastructure. Accurate real-time identification of IoT devices is crucial for maintaining the security of modern networks. This can be achieved using deep learning models, deployed at the network edge, which can understand the complex communication patterns of IoT devices. Once a device has been identified, the network can be secured by enforcing appropriate access controls tailored to the requirements of the device. This work addresses several challenges of using deep learning for IoT network fingerprinting, the process of identifying an IoT device based on its network traffic.

1.1 Problem Statement

The problem of IoT fingerprinting is to identify an IoT device using a fingerprint representing a sample of its network communication. Let $\mathcal{T}$ be a collection of network packets sent to or transmitted by a member of the set of known IoT devices D. The goal of IoT fingerprinting is to build a deep learning model $M(\mathcal{T})$ that can match the sample $\mathcal{T}$ to the correct device in D. The model does this by analyzing a network fingerprint consisting of features extracted from $\mathcal{T}$ to predict a probability distribution P over D. The device in D with the highest corresponding probability in P is taken as the most likely match for the sample $\mathcal{T}$.

1.2 Challenges of IoT Fingerprinting

This work addresses challenges in the training, maintenance, and deployment of IoT fingerprinting models. The addressed challenges are as follows:

1. **Limited data availability**. Many IoT devices have low communication rates, which can make it difficult for a single observer to collect sufficient data for training an accurate fingerprinting model. Although it may be possible to cover the range of unique behaviors of such devices by combining data from separate observers, aggregating these datasets into a centralized location raises privacy concerns, as the traffic may contain sensitive information.

2. **Imbalanced datasets**. Differences in communication rates can lead to imbalanced training datasets. This can introduce biases in the fingerprinting model against the underrepresented devices.

3. **Capturing relationships in multi-flow traffic**. Many fingerprinting approaches filter traffic into individual flows to reduce the amount of noise in each sample. However, IoT devices may generate several separate flows to perform a single task. Flow-based filtering discards important relationships between packets from separate flows. Conversely, including packets from multiple flows in a single sample without explicitly defining their relationships can confuse the model.

4. **Efficient and accurate adaptation of existing fingerprinting models**. With the rapid expansion of the IoT industry, new types of devices are regularly introduced and existing fingerprinting models must constantly adapt to identify them. The process of adapting an existing model can be inefficient when the model is not reused. However, reusing an existing model can lead to reduced performance on original devices through catastrophic forgetting, or on new devices when their network behaviors are outside of the distribution learned from the original devices.

5. **Unbounded fingerprinting time**. Deep learning fingerprinting approaches typically require a fixed number of packets to be collected before identifying a device. Therefore, the time

required to fingerprint a device is unbounded, as the system must wait until sufficient traffic has been observed. For IoT devices with low communication rates, this can lead to long fingerprinting times. Additionally, delimiting fingerprint samples based on packet count is not suitable for devices with bursty communication patterns, as packets from separate bursts may not be relevant to each other.

1.3 Proposed Approach

To address these challenges, this work proposes the following methods:

1. **Federated learning**. We propose a federated learning approach for utilizing IoT traffic collected by multiple separate observers to build a single fingerprinting model in a privacy-preserving manner. This addresses challenge 1, limited availability of training data, by combining traffic from separate observers to cover a wider range of possible behaviors of IoT devices. For our federated learning approach, we implement and evaluate two different privacy-preserving mechanisms for aggregating knowledge from multiple observers.

2. **Communication rate-based hierarchical Mixture-of-Experts**. To address challenge 2, imbalanced training datasets caused by differences in communication rates, we propose a hierarchical Mixture-of-Experts (MoE) architecture that groups devices based on their communication rates before applying a deep learning model to differentiate between the devices in a group. Each grouping has a separate "expert" deep learning model trained on a subset of devices with similar communication rates. This way, the smaller data subsets used to train each expert are less imbalanced than the overall dataset.

3. **Relative encoding of communication endpoints**. We propose a technique that encodes the communication endpoints of packets relative to other packets in a traffic sample. This encoding process is used to generate *"relative features"*, which enable the fingerprinting model to understand the relationships between packets, even when they are from separate

flows. We use relative features to address challenge 3, capturing relationships in multi-flow traffic.

4. **Bi-component fingerprinting architecture**. To address challenge 4, adaptability of IoT fingerprinting models, we propose a bi-component architecture that separates the training and inference processes into two stages: (1) extraction of network behavior patterns and (2) device identification. Each stage is associated with a different component of the bi-component architecture. Fingerprinting models can be efficiently adapted by reusing the component responsible for extraction of behavior patterns, which accounts for the majority of the training cost. This component is not modified when the model is adapted, which allows the architecture to effectively integrate knowledge on new devices without forgetting knowledge on the original devices.

5. **Fingerprinting with fixed-time observations**. We propose a time-based sampling technique where each fingerprint sample only contains packets observed within a fixed window of time. This addresses challenge 5, unbounded fingerprinting time, by setting a limit on the amount of time allowed for collecting each traffic sample. This improves quality-of-service (QoS) guarantees and ensures the relevance of packets within a sample by discarding packets after a fixed amount of time has passed.

1.4 Research Questions

Through the evaluation of the approaches outlined in Section 1.3, this work explores the following research questions:

- **RQ1:** How can accurate IoT fingerprinting models be trained on distributed and imbalanced training data?
- **RQ2:** How can IoT fingerprints preserve information about the flows of packets without causing overfitting?

- **RQ3:** How can existing fingerprinting models be reused to accurately identify new devices without reducing performance on the original devices?

1.5 Organization

The remainder of this work is organized as follows. Chapter 2 establishes the threat and system models and provides background information on networking and deep learning in the context of IoT fingerprinting. Chapter 3 summarizes related work on IoT fingerprinting and our approaches. Chapter 4 explores the approaches for addressing challenges related to training data quality, challenges 1 and 2. Chapter 5 explores the encoding of related packets in IoT traffic using relative features and explains how it addresses the challenge of capturing multi-flow relationships. Chapter 6 outlines the bi-component architecture for IoT fingerprinting and shows how it enables efficient and accurate adaptation of existing fingerprinting models. Chapter 7 introduces fingerprinting with fixed-time observations to address the challenge of unbounded fingerprinting time. Chapter 8 concludes this work by summarizing the outcomes of each approach and introduces future directions for research on IoT fingerprinting. Finally, the appendix lists the deep learning hyperparameters used in each experiment.

Chapter 2

Background and System Model

This chapter defines the system and threat model and provides essential background information on deep learning and networking.

2.1 Networking

Our fingerprinting approach works by analyzing internet traffic from IoT devices. This traffic consists of packets of data that are transmitted or received by network-connected devices. These packets are formatted according to protocols that facilitate communication over the network. Data packets have two components: a header containing protocol information, and a payload containing the application data that is being transmitted. We primarily use the network and transport layers of the header for constructing fingerprints. The network layer contains information for routing packets between different devices on the network. This includes the source and destination IP addresses and the type of transport protocol in use, such as TCP or UDP. The transport layer contains information for communication between specific processes running on devices. This includes the source and destination port numbers, which identify sockets on the sender and receiver devices. These sockets are owned by specific processes on a device and are used to transmit and receive application data. The stream of packets sent back and forth between two specific sockets is referred to as a bi-directional network flow. The flow of a packet is identified by its IP addresses, port numbers, and transport protocol type.

When a device receives a data packet, its operating system uses the IP addresses, port numbers, and transport protocol type to locate the appropriate socket and writes the data to the socket. The owner of the socket can then read the application data from the socket. When a process writes data into a socket file, the device's operating system encloses the data in a packet and transmits it over the network. Network routers use the destination IP address to forward the packet until it reaches the destination device.

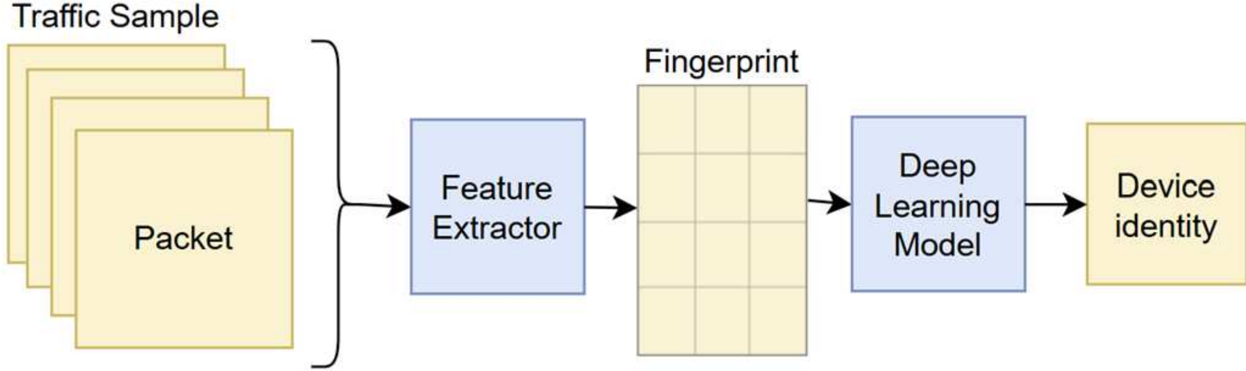

Figure 2.1: Fingerprinting process.

2.1.1 Network Fingerprints

In this work, we represent IoT network traffic using feature vectors containing essential information extracted from network packets. An IoT network fingerprint is made up of one or more feature vectors representing a sample of the device's traffic. The fingerprint is the input for a deep learning model, which predicts the corresponding IoT device. Depending on the deep learning architecture, the fingerprint can be a single feature vector, a feature matrix, or a tuple of feature vectors/matrices. In all cases, the features in each fingerprint are extracted from a set of contiguous packets captured from the device.

We construct our experimental datasets by parsing .pcap files and extracting features from traffic samples using a sliding-window approach. Let $\{\rho^{(1)}, \rho^{(2)}, \ldots, \rho^{(n)}\}$ be a traffic capture of n packets from a particular device. In our experimental datasets, the ith fingerprint will contain features extracted from the sample of packets $\{\rho^{(i)}, \rho^{(i+1)}, \ldots, \rho^{(i+w)}\}$ where w is the window size, which determines the number of packets per sample.

2.2 System and Threat Model

2.2.1 System Model

Our system model consists of a collection of IoT devices connected to a home or enterprise network. The local network administrator is capable of monitoring packets sent to or from the

devices and enforcing appropriate access control policies, which are tailored to individual devices based on their security requirements. We assume that information from the network and transport layers of each packet will be available in plain text, while the packet payloads may be encrypted. This is a reasonable assumption, as the packet headers are needed for delivering packets. To select appropriate access control policies, the administrator identifies the types of IoT devices connected to the network. This is done by collecting traffic from devices, extracting feature vectors from the collected traffic, and applying a deep fingerprinting model to the extracted features as shown in Figure 2.1. For each traffic sample, the fingerprinting model predicts the best match out of the set of known IoT devices.

We assume that the fingerprinting model runs at the network edge rather than in a data center. This protects the privacy of the devices by keeping the recorded traffic inside of the local network. With this constraint in mind, we design our deep learning models to run on consumer hardware.

Our experimental datasets include private IoT traffic collected by researchers at Colorado State University and traffic from two public datasets: the Canadian Institute of Cybersecurity [10] and University of New South Wales [38].

2.2.2 Threat Model

Our threat model considers a remote adversary capable of gaining control of vulnerable IoT devices through the internet. The adversary gains access by exploiting software vulnerabilities or weak authentication mechanisms. We assume the adversary can monitor local network traffic and exchange packets with any internal or external service through the compromised device. We also assume that the adversary is not capable of decrypting wireless traffic in a secure network, but does have access to MAC header fields in plain-text. Compromised devices can be used to launch DDoS attacks [21] or exfiltrate private information [37].

By enforcing device-specific access control policies, the network administrator can protect the network [36] by restricting remote adversaries from contacting exposed ports on devices, limiting lateral movement of malware between devices, or preventing devices from communicating with

command-and-control (C&C) servers. Additionally, live threat intelligence feeds can be integrated with existing access control policies to dynamically respond to threats or isolate devices with reported vulnerabilities until they can be patched [18].

2.3 Deep Learning

Deep learning is a field of machine learning that involves training and applying artificial neural networks (deep learning models) to specific tasks. After training, deep learning models can be applied to new samples of data during inference. Deep learning models are trained by iteratively adjusting their learnable parameters to minimize a loss/error function evaluated on some training data. These loss functions typically have two inputs: the expected output of the model, and the true output of the model given some input data. In this way, training a deep learning model is a form of supervised machine learning. This work uses traditional supervised learning, where input samples are explicitly annotated with labels that are used as the expected output, and self-supervised learning, where the expected outputs are derived from the inputs.

Parameters are optimized by taking the gradient of the loss function evaluated on some training data with respect to the learnable parameters of the model. The parameters are then adjusted in the direction opposite of the gradient in order to reduce the loss. We use the *Adam* [23] optimization algorithm, which maintains the momentum of parameter updates to stabilize gradient descent. A learning rate is used to control the rate of descent and ensure that the model's parameters gradually converge to an optimal setting. Our learning algorithms perform multiple training epochs, where each epoch is defined as one pass over the training dataset. After each epoch, the model is evaluated on a separate validation dataset.

2.3.1 Loss Functions and Tasks

The tasks covered in this work fall under two broad categories: regression and classification. In regression, a model predicts a set of continuous variables; whereas, in classification, the model predicts a set of categorical variables. We approach IoT fingerprinting as a multi-class classification

task, where the deep learning model predicts one categorical variable that represents the type of device that generated an input sample of traffic out of the set of known IoT devices. For classification tasks, we apply the $softmax$ operation to the final output of the model to convert it to a probability distribution $P \in [0,1]^{|D|}$ over the set of devices D where $P_i = P(D_i|X)$ and X is a fingerprint. Let $Y \in \mathbb{R}^{|D|}$ be the output of the model for X, the softmax operation is defined as follows:

$$softmax(Y)_i = \frac{\exp(Y_i)}{\sum_{j=1}^{|D|} \exp(Y_j)}$$

During inference, the element of P with the highest probability is taken as the predicted class. For the task of IoT Fingerprinting, if P_i has the highest probability, then D_i will be the predicted device.

For regression tasks, we train models to minimize the mean-squared-error loss function defined as follows:

$$\frac{1}{Nwm} \sum_{i=1}^{N} \sum_{j=1}^{w} \sum_{k=1}^{m} (Y_{i,j,k} - T_{i,j,k})^2$$

where $Y \in \mathbb{R}^{N \times w \times m}$ is the output of the model for a batch of N samples with a window size of w and m features, and $T \in \mathbb{R}^{N \times w \times m}$ is the expected output of the model. For classification tasks, we minimize the negative-log-likelihood loss function defined as follows:

$$-\frac{1}{N|D|} \sum_{i=1}^{N} \log(P_{i,T_i})$$

where $P \in [0,1]^{N \times |D|}$ are the device probabilities for a batch of N samples and $T \in \{1, \ldots, |D|\}^N$ are the labels for the batch where T_i is the index of the correct device in D for the ith sample.

2.3.2 Metrics

We evaluate our fingerprinting models using three standard classification metrics defined in Table 2.1. These metrics are calculated using the model's predicted labels for the testing dataset and their true labels.

Metric	Definition
Recall	$\frac{1}{\lvert D\rvert}\sum_{j=1}^{\lvert D\rvert}\frac{\lvert\{i \mid T_i=D_j \wedge \operatorname*{arg\,max}_{k\in\{1,\ldots,\lvert D\rvert\}} P_{i,k}=j\}\rvert}{\lvert\{i \mid T_i=D_j\}\rvert}$
Precision	$\frac{1}{\lvert D\rvert}\sum_{j=1}^{\lvert D\rvert}\frac{\lvert\{i \mid T_i=D_j \wedge \operatorname*{arg\,max}_{k\in\{1,\ldots,\lvert D\rvert\}} P_{i,k}=j\}\rvert}{\lvert\{i \mid \operatorname*{arg\,max}_{k\in\{1,\ldots,\lvert D\rvert\}} P_{i,k}=j\}\rvert}$
F1	$\frac{2*\text{Precision}*\text{Recall}}{\text{Precision}+\text{Recall}}$

Table 2.1: Classification metrics in terms of true labels $T \in \{1,\ldots,|D|\}^N$ and predicted probabilities $P \in [0,1]^{N\times D}$ for N samples.

We note that we do not measure Accuracy because it is not a suitable metric for imbalanced classification tasks and tends to exaggerate performance when the model is biased towards majority classes.

2.3.3 Multilayer Perceptron

Multilayer Perceptrons (MLPs) are the simplest deep learning architecture. We use MLP-based fingerprinting models in chapter 4, and use them as a building block for more complex architectures in chapters 5, 6, and 7. The MLP is made up of one or more hidden layers and a linear output layer. Each hidden layer consists of a linear layer followed by a non-linear activation function. By stacking these linear and non-linear layers, MLPs are able to learn complex non-linear functions.

Linear layers in the MLP perform an affine transformation on the layer input using their learnable parameters, which includes a matrix of weights and a vector of biases. Let $X \in \mathbb{R}^{N\times m}$ be the input to a linear layer with weights $\mathcal{W} \in \mathbb{R}^{m\times d}$ and biases $\beta \in \mathbb{R}^d$. The output of the layer $Z \in \mathbb{R}^{N\times d}$ is defined as follows:

$$Z_{i,j} = \beta_j + \sum_{k=1}^{m} X_{i,k}\mathcal{W}_{k,j}$$

For hidden layers, this is followed by an activation function to introduce non-linearity. We use two types of activation functions in this work, *ReLU* and *Tanh*, defined as follows:

$$ReLU(Z)_{i,j} = max(0, Z_{i,j})$$
$$Tanh(Z)_{i,j} = \frac{exp(Z_{i,j}) - exp(-Z_{i,j})}{exp(Z_{i,j}) + exp(-Z_{i,j})}$$

For classification problems such as IoT fingerprinting, the dimensionality of the output layer will be equal to the number of classes.

Input rows to the MLP are processed independently, the MLP only captures dependencies between the last axis of an input tensor. If the input to the MLP is an $[w \times m]$-dimensional matrix, the output will be an $[w \times d]$-dimensional matrix where d is the dimensionality of the output layer. Therefore, when we use an MLP to classify a sequence of packets, we either manipulate the packets into a single input feature vector or apply a sequence-wise average pooling layer to the output prior to the $softmax$ operation. Let $Y' \in \mathbb{R}^{w \times |D|}$ be the output of the MLP's output layer for a sequence of w packets where D is the set of devices. The pooling layer will produce the final output of the MLP, $Y \in \mathbb{R}^{|D|}$ as follows:

$$Y_i = \frac{1}{w} \sum_{j=1}^{w} Y'_{j,i}$$

2.3.4 Transformer

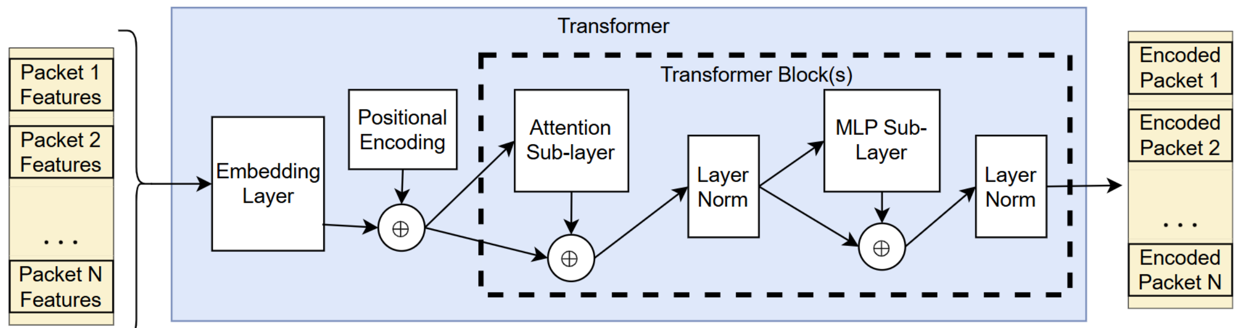

Figure 2.2: Diagram of Transformer.

The Transformer is a deep learning architecture for handling sequential data introduced by Vaswani *et al.* [40]. Unlike the MLP, the Transformer is capable of capturing dependencies in the last two axes of input tensors. We use the second-to-last axis as the sequence dimension and last axis as the feature dimension. Let $X \in \mathbb{R}^{w \times m}$ be an input to the Transformer, each row of X is a vector of m features representing a different packet in a sample of w packets.

As shown in Figure 2.2, the Transformer consists of three basic components: the embedding layer, a positional encoding, and a stack of Transformer blocks. The embedding layer homogenizes the input features and projects them onto the embedding dimensionality of the model. The positional encoding introduces a signal that allows the Transformer to determine the position of each packet in the input sequence. The Transformer blocks capture dependencies between different packets and features using attention mechanisms and MLPs.

Let d be the embedding dimensionality of the Transformer. The embedding layer projects the input X onto an embedding matrix $E \in \mathbb{R}^{w \times d}$ using its learnable parameters. This layer can be implemented using lookup-tables for categorical features or a simple matrix multiplication for continuous features. The positional encoding $PE \in \mathbb{R}^{w \times d}$ is added to the embeddings through element-wise addition to produce the input to the Transformer stack. We use a fixed trigonometric positional encoding defined as follows:

$$PE_{i,j} = \begin{cases} sin(\frac{i}{10^{j/d}}) & \text{if } \ j\%2 = 0 \\ cos(\frac{i}{10^{j/d}}) & \text{else} \end{cases}$$

We denote the input to the ith Transformer block as $E^{(i)}$ and $E^{(1)} = E \oplus PE$ is the input to the first block on the stack.

Transformer blocks consist of a multi-head self-attention sub-layer and an MLP sub-layer. Each sub-layer is surrounded by a residual connection and followed by a layer normalization. The attention sub-layers capture temporal relationships between packets, MLP sub-layers help maintain the rank of embedding matrices [13], residual connections increase the strength of gradient signals to deeper layers, and normalization ensures the stability of gradients during training [44].

The input to the attention sub-layer is projected onto matrices $Q, K, V \in \mathbb{R}^{w \times d}$, known as the Query, Key, and Value respectively, using 3 sets of learnable $[d \times d]$-dimensional weights and d-dimensional biases. The attention mechanism is performed over H different $[h = \frac{d}{H}]$-dimensional subspaces of the Query, Key, And Value matrices where H is the number of heads. Let $\alpha^{(i)} \in \mathbb{R}^{w \times h}$ be the output of the ith attention head and let $q^{(i)}, k^{(i)}, v^{(i)} \in \mathbb{R}^{w \times h}$ be all rows and columns $1 + (i-1)h$ through $1 + ih$ of Q, K, and V respectively. $\alpha^{(i)}$ is defined as follows:

$$\alpha^{(i)} = softmax\left(\frac{q^{(i)} k^{(i)^T}}{\sqrt{d}}\right) v^{(i)}$$

The output of the attention sub-layer $\mathcal{A} \in \mathbb{R}^{w \times d}$ combines the outputs from each head using an affine transformation with learnable weights $\mathcal{W} \in \mathbb{R}^{d \times d}$ and bias $\beta \in \mathbb{R}^{d}$ as follows:

$$\mathcal{A}_{k,j} = \beta_j + \sum_{i=1}^{H} \sum_{r=1}^{h} \alpha_{k,r}^{(i)} \mathcal{W}_{(i-1)h+r,j}$$

The MLP sub-layer consists of a standard Multilayer Perceptron as described in Section 2.3.3. We denote the operation of the attention sub-layer in the ith block as $MHA(E^{(i)}; \theta^{(i)})$ where $\theta^{(i)}$ are the parameters of the ith Transformer block. Likewise, we denote the operation of the *MLP* sub-layer as $MLP(\mathcal{A}^{(i)}; \theta^{(i)})$ where $\mathcal{A}^{(i)}$ is the output of the attention sub-layer. The operation of the ith Transformer block is defined as follows:

$$\mathcal{A}^{(i)} = LayerNorm\left(MHA(E^{(i)}; \theta^{(i)}) \oplus E^{(i)}\right)$$
$$E^{(i+1)} = LayerNorm\left(MLP(\mathcal{A}^{(i)}; \theta^{(i)}) \oplus \mathcal{A}^{(i)}\right)$$

where $\oplus$ denotes element-wise addition and represents a residual connection, $E^{(i+1)} \in \mathbb{R}^{w \times d}$ is the input to the next block, and *LayerNorm* normalizes the feature values. The *LayerNorm* operation is defined as follows:

$$LayerNorm(X)_{i,j} = \frac{X_{i,j} - \mu(X_{*,j})}{\sigma(X_{*,j})}$$

where $\mu(X_{*,j})$ and $\sigma(X_{*,j})$ return the mean and standard deviation of the jth feature of X respectively.

During training, we apply dropout to the outputs of each sub-layer as a form of regularization to reduce overfitting. The dropout layers work by randomly setting a portion of activations to zero during forward passes.

Chapter 3

Related Work

3.1 IoT Fingerprinting

Early works [31, 7, 16, 32] on IoT fingerprinting use traditional machine learning algorithms for identifying IoT devices. Miettinen *et al.* [31] propose IoT SENTINEL, a binary random forest based IoT fingerprinting architecture that identifies devices using the traffic generated when they connect to a network. IoT SENTINEL [31] uses binary classifiers to address the issue of scalability; however, Miettinen *et al.* do not evaluate how the performance of their architecture is affected when new devices are introduced. Bezawada *et al.* [7] evaluate a variety of shallow machine learning approaches for IoT fingerprinting including KNNs, Random Forest, and Gradient Boost Regression Trees (GBRT). They found that GBRT results in the best fingerprinting performance, achieving more consistent results compared to previous shallow learning approaches [7]. Feng *et al.* [16] evaluated several shallow learning approaches and showed that deep learning can result in better performance.

Various deep learning approaches [12, 24, 33, 25, 42, 6, 15, 41, 3, 4, 5] have been proposed for IoT fingerprinting. Lopez-Martin *et al.*[24] apply a deep learning architecture consisting of convolutional and LSTM layers for classifying traffic in IoT networks. The use of recurrent LSTM layers allows their model to interpret temporal patterns in sequences of packets. Dong *et al.* [12] and Ortiz *et al.* [33] also use LSTM-based IoT fingerprinting architectures.

Following its success in Natural Language Processing, several works [25, 42, 6, 4, 5] have adapted the Transformer [40] architecture for IoT fingerprinting due to its powerful time-series modeling capabilities. Luo *et al.* [25] use Transformer-based models to separate IoT traffic into normal and abnormal groups, then use an additional Transformer to identify the devices in the normal traffic group. Bazaluk *et al.* [6] fine-tune a pre-trained tabular Transformer for identifying IoT traffic.

3.2 Federated Learning

Federated learning is an approach proposed by McMahan *et al.* [29] for training a deep learning model on sensitive distributed data. It enables a federation of clients with separate datasets to collectively train a global fingerprinting model by sharing updates to their local models with a server, which aggregates them. The clients do not share their local datasets, making it challenging for an adversary to extract private information. The authors refer to their approach for aggregating updates as *FedAvg*, where the server averages the values of clients' gradients or parameters after each mini-batch. Federated learning has since been applied to various domains [11, 35, 43, 34].

Wang *et al.* [41] apply federated learning with FedAvg-style aggregation for training IoT fingerprinting models in a privacy-preserving manner. Bar-on *et al.* [3] also use federated learning for IoT fingerprinting, but use an aggregation method based on the selective swapping of parameters between client and server models. In this work, we implement FedAvg-based and parameter swapping-based aggregation approaches and evaluate them for federated IoT fingerprinting.

3.3 Mixture-of-Experts

Mixture-of-Experts (MoE) is a deep learning approach that divides the problem space into subspaces and trains individual expert models for each subspace. Masoudnia and Ebrahimpour [28] outline two categories of MoE architecture: implicitly localized and explicitly localized. In implicit localization [20, 19], the problem space is partitioned automatically during the training process through a gating network. The experts and gating network are trained jointly. As the experts become specialized for different subspaces, the gating network learns to select appropriate experts for each sample. In explicit localization [17, 39], the problem space is partitioned prior to training the experts. Each subspace is assigned to a different expert, and the experts are trained on their assigned subspaces. In chapter 4, we implement an implicitly localized MoE architecture for IoT fingerprinting, where data is partitioned based on the communication rate of devices. To the best of our knowledge, this is the first work that applies a MoE architecture to IoT fingerprinting.

3.4 Adaptable IoT Fingerprinting

Several architectures have been proposed for adapting IoT fingerprinting models to new devices.

IoT-Portrait is an IoT fingerprinting architecture, proposed by Wang *et al.* [42], consisting of a Transformer encoder and a softmax classifier. They propose an approach to efficiently adapt their model to identify new devices based on the Fine Tuning with Distillation Loss (FTDL) method, which retrains a new classifier for new devices while retaining knowledge from the old classifier through knowledge-distillation. We improve upon this approach by training the encoder as a separate component, which allows us to adapt our model without updating the majority of its parameters. Additionally, the accuracy of this approach degrades as the number of new devices increases; whereas, our approach can adapt to increasingly large numbers of new devices while maintaining high performance.

ScaNeF-IoT, proposed by Alyaha *et al.* [1], achieves similar performance to IoT-Portrait using an Online Stream Learning classifier. They use an Aggregated Mondrian Forest classifier, which identifies devices by taking the weighted average over the predictions of an ensemble of decision trees. This approach can adapt a trained model for an unseen device by adjusting the existing nodes of the model's decision trees using traffic samples from the new device without retaining previously used training data. The drawback of this approach is that it is susceptible to unpredictable fluctuations in accuracy when new devices are added to the model.

AutoIoT [15] is a CNN-based IoT fingerprinting architecture that identifies devices using the characteristics of packets captured over a fixed 30-minute time window. Their approach includes a mechanism for adapting fingerprinting models on-the-fly by automatically assigning labels to traffic from unknown devices using k-means clustering, then updating their classifier to identify the new labels through transfer learning. However, the learned labels may not represent real IoT devices because they are discovered by the k-means algorithm. In addition, AutoIoT does not include a method for determining when to apply clustering to the collected traffic from new devices. If the clustering is applied before their model has observed sufficient behaviors from a new device, it risks splitting its behaviors into two separate labels for the same device.

3.5 Fixed-Time Traffic Sampling

Several IoT fingerprinting works [33, 14, 38, 15] have used fixed-time traffic sampling, where devices are identified using traffic captured over a fixed interval of time. Sivanathan *et al.* [38] identify IoT devices using various attributes extracted from hour long samples of network activity. Marchal *et al.* [27] and Fan *et al.* use a shorter 30-minute capture time. In chapter 7, we explore fixed-time sampling with much shorter capture durations in the range of $0.1ms$ to $0.1s$ and demonstrate that many IoT devices exhibit sufficient variation to identify them within those intervals.

Chapter 4

IoT Fingerprinting With Limited Datasets [1]

Building a reliable fingerprinting model requires a large and diverse training dataset that covers the various unique behaviors of the devices included in the system. Otherwise, during inference time, unseen fingerprint samples may be outside of the data distribution that the model learned during training. Training datasets are assembled by collecting traffic from IoT devices when the types of the devices are known and assigning ground truth labels. Collecting this data is a time consuming task, especially for devices with low communication rates. A single observer may not have enough time to collect sufficient traffic from such devices to cover their different behaviors. Additionally, differences in communication rates can result in imbalanced datasets, which can introduce biases in the resulting fingerprinting model. This chapter explores methods for training accurate IoT fingerprinting models when the quality of the available training dataset is limited.

Through this exploration, this chapter addresses research question **RQ1**, how can accurate IoT fingerprinting models be trained on distributed and imbalanced training data? This chapter proposes a federated learning approach to handle distributed training data, and a hierarchical Mixture-of-Experts (MoE) architecture to handle imbalanced training data. Our results demonstrate that these methods can be combined to address both aspects of **RQ1**.

4.1 Methods

4.1.1 Federated Deep Learning

Federated deep learning allows multiple participants to collectively train a single deep learning model without sharing their individual datasets. This preserves the privacy of their data while yielding a stronger and more generalizable model. Federated learning can improve the coverage

---

[1] Adapted from: [3] Bar-on, M., Bezawada, B., Ray, I., Ray, I. (2024). A Small World–Privacy Preserving IoT Device-Type Fingerprinting with Small Datasets. In: Mosbah, M., Sèdes, F., Tawbi, N., Ahmed, T., Boulahia-Cuppens, N., Garcia-Alfaro, J. (eds) Foundations and Practice of Security. FPS 2023. Lecture Notes in Computer Science, vol 14551. Springer, Cham. https://doi.org/10.1007/978-3-031-57537-2_7

of unique network behaviors of IoT devices by utilizing traffic samples collected by multiple observers from separate networks. It achieves this without aggregating the datasets in a single location, which is important because the collected traffic may contain sensitive information such as IP address ranges, passwords or audio recordings. Instead of aggregating data, federated learning works by aggregating the participants' updates to a deep learning model throughout the training process.

We implement federated learning using a client-server architecture, where the clients maintain and update local deep learning models and the server maintains a global fingerprinting model. The global model is updated by aggregating updates from the clients. Each client and the server have separate datasets that remain on their individual machines throughout the training process.

Before training, the clients randomly initialize their local models and divide their datasets into mini-batches. The clients use their datasets to train their local models and send updates to the server after each mini-batch. The server aggregates these updates and applies them to the global model, then sends the updated global model to the clients. The clients wait to receive the global model from the server and replace their local models with the global model before continuing to the next mini-batch. Once the clients have used all of their mini-batches, the server uses its dataset to evaluate the current global model by calculating a validation error, which is saved in a trace. If the latest validation error is lower than the rest of the errors in the trace, the server creates a checkpoint by storing the current global model. This process is repeated for 800 epochs, then the checkpoint that resulted in the lowest validation error is used as the final IoT fingerprinting model.

We implement two approaches for aggregation: *parameter swapping* and *FedAvg*.

Parameter Swapping

This approach works by aggregating the learnable parameters of the client models. After updating their local models on a mini-batch, the clients send their new parameters to the server. The server then swaps a subset of the global model parameters for the values at the corresponding positions in the parameter vectors sent by each client. The positions that are swapped have the greatest difference between the client parameters and global parameters.

Let $\theta^{(i)} \in \mathbb{R}^d$ be the parameter update sent by client i where d is the number of parameters, and let $\theta^{(g)} \in \mathbb{R}^d$ be the global model parameters. The server updates the global model parameters as follows:

$$\theta_j^{(g)} = \begin{cases} \theta_j^{(i)} & \text{if } \sum_{k=1}^{d} \mathbb{1}\{(\theta_j^{(i)} - \theta_j^{(g)})^2 > (\theta_k^{(i)} - \theta_k^{(g)})^2\} \geq \frac{d}{C} \\ \theta_j^{(g)} & \text{else} \end{cases}$$

where C is the number of clients and $\mathbb{1}\{condition\}$ is the indicator-function defined as follows:

$$\mathbb{1}\{condition\} = \begin{cases} 1 & \text{if } condition \\ 0 & \text{else} \end{cases} \tag{4.1}$$

In other words, the server replaces the $\frac{d}{C}$ positions with the greatest difference between $\theta^{(i)}$ and $\theta^{(g)}$.

FedAvg

This approach aggregates loss gradients and uses the aggregated gradient to update the global model. After computing the loss of their local models on a mini-batch of samples, the clients share the gradient of the loss with respect to their models' parameters with the server. Then, the server aggregates the clients' gradients by taking the average and applies the aggregated gradient to the global model. The aggregated gradient $\Delta^{(a)} \in \mathbb{R}^d$ is calculated as:

$$\Delta_j^{(a)} = \frac{1}{C} \sum_{i=1}^{C} \Delta_j^{(i)}$$

where $\Delta^{(i)} \in \mathbb{R}^d$ is the gradient sent by client i.

4.1.2 Hierarchical Mixture-of-Experts

We observed that IoT traffic datasets are highly imbalanced due to the significant variation in communication rates among different IoT devices. Devices with low communication rates produce fewer samples of traffic during data collection and, therefore, make up a smaller portion of the

available training data compared to devices that communicate frequently. Imbalanced training data can lead to biased fingerprinting models, which are less likely to correctly identify samples of traffic from devices with low communication rates. Furthermore, removing training samples for devices with high communication rates to balance the dataset can hurt the performance of the fingerprinting model, as the removed samples may cover behaviors that aren't covered elsewhere in the training data. We address this challenge using a hierarchical Mixture-of-Experts (MoE), which exploits the differences in communication rates to reduce biases against underrepresented devices and improve performance.

Architecture

The hierarchical MoE has two levels. The first level separates traffic samples into one of three different groups based on communication rate and the second level applies a deep learning model to identify the device. Each group is associated with a different expert deep learning model and includes a subset of the devices in the dataset. Each expert only learns to identify the devices in its associated group and is trained only on data from these devices. This way, the architecture is an explicitly localized MoE [28] where each expert learns a different subset of devices.

Since devices are grouped based on communication rate, the amount of available samples will be similar for devices in the same group. This improves the balance of the training data, as the datasets used for each expert are less imbalanced compared to the dataset as a whole.

The first level uses a predefined policy to assign devices to groups based on the amount of packets they generated during the collection period. We use this number as an approximation of the communication rate. The policy includes two boundaries, which separate devices into three groups: low communication rate, moderate communication rate, and high communication rate. The low communication rate group includes devices with fewer than $2,000$ packets. The moderate group includes devices with between $2,000$ and $12,000$ packets. Finally, the high communication rate group includes devices with more than $12,000$ packets.

This architecture is designed to overcome the imbalanced dataset challenge with minimal additional computational overhead. Although it requires more parameters than a standalone model, only a portion of parameters are utilized for each sample during inference.

4.2 Evaluation

	Feature	Explanation
Packet	TCP	1 if transport protocol is TCP else 0
	UDP	1 if transport protocol is UDP else 0
	HTTP	1 if src or dst port is 80 else 0
	TLS	1 if src or dst port is 443 else 0
	Header Length	Length of packet header in bytes
	Payload Length	Length of payload in bytes
	Payload Entropy	Byte entropy of payload
	Common Src	1 if src port is $\leq$ 1023 else 0
	Common Dst	1 if dst port is $\leq$ 1023 else 0
Window	Max flow	Packet count of largest flow in window
	Unique IPs	Count of unique IP addresses in window
	Local IPs	Count of unique IP addresses from local subnet

Table 4.1: Features

4.2.1 Experimental Setup

For these experiments, we use a Multilayer Perceptron (MLP) classifier with ReLU activation function as our deep learning architecture. We partition our dataset using a train:validation:test ratio of 7:1:2. In each experiment, we run 5 trials with different dataset partitions and report the average of performance on the test sets over all trials. We simulate a distributed dataset by randomly partitioning the available training data evenly among a set of “observers”, implemented as separate OS processes, at the beginning of each experimental trial. In the federated learning environment, these observer processes are clients that connect to a federated learning server, where we keep the test and validation data. When training is complete, the server uses the test data to

evaluate the global model. Before training, we standardize the training, testing, and validation data using the averages and standard deviations of the training data.

We train our models using the *Adam* optimizer with a constant learning rate. Additional hyperparameters are listed in Appendix .1 Table 1.

Features

	Device	Packet Count
High Rate	Arlo Camera	25000
	Somfy	25000
	Home Eye Camera	25000
	Philips Hue Bridge	17840
	Taxi Cab WiFi AP	17514
	Amazon Echo	12195
Moderate Rate	Eufy Homebase	11431
	Netatmo Camera	10625
	NestCam	7490
	Sonos One Speaker	6567
	Borun Camera	4970
	Google NestMini	3269
	Roomba Vacuum	2292
Low Rate	LG Smart TV	1785
	Ring	1460
	Globe Lamp	1253
	Standard WiFi APs	1203
	Amcrest Camera	682
	Yutron Plug	473
	SimCam	452
	Atomi Coffee Maker	413
	D-Link Omna	321

Table 4.2: Experimental Devices

Traffic samples are represented as feature vectors of fixed dimensionality determined by the window size used during sampling. These feature vectors are the input samples for the MLP.

Feature vectors include 9 features extracted from each packet in the corresponding sample, along with 3 aggregated window features, which are statistics describing all of the packets in the sample. The packet features are arranged in sequential order and flattened into a vector which is appended to the window features. With a window size of 20, each input sample will be a [183]-dimensional vector since it has $20*9 = 180$ packet features plus 3 window features. Table 4.1 shows the features included in the feature vector.

Devices

We selected a set of 22 devices with a wide variety of communication rates. In Table 4.2, we show the devices included in these experiments, their packet counts, and their communication-rate-based group assignments.

4.2.2 Federated Learning Evaluation

Learning Mode	# Observers	Recall	Precision	F1
Centralized	2	93.9%	94.3%	94.1%
	4	90.1%	90.8%	90.4%
FedAvg	2	98.6%	98.5%	98.6%
	4	98.1%	98.2%	98.2%
Parameter	2	96.6%	97%	96.8%
	4	97.7%	97.6%	97.6%

Table 4.3: Average performance of federated and centralized learning.

Here, we compare the performance of the two federated learning approaches introduced in Section 4.1.1. We use centralized learning as a baseline, where each observer trains a local fingerprinting model using only their assigned portion of the training data. In this environment, we report the average performance across all observers' models on the test data. Like with federated learning, our centralized learning implementation trains a fingerprinting model for 800 epochs and uses the parameters with the lowest validation error as the final model. In this experiment, we use a hierarchical MoE as our fingerprinting architecture in both learning environments.

As shown in Table 4.3, both federated learning approaches perform better than centralized learning, demonstrating that fingerprinting models can be effectively trained on distributed datasets. Increasing the number of observers results in much worse performance for centralized learning, but has minimal impact on the federated learning performance. FedAvg performs better than parameter swapping on average because the parameter swapping approach introduces additional noise into the learning process.

4.2.3 Hierarchical MoE Evaluation

We evaluate the performance of our hierarchical MoE against a standalone MLP trained to identify all 22 devices. For both models, we use the FedAvg federated learning approach with 4 clients. Table 4.4 shows the average performance of the hierarchical and standalone models across

Model	**Recall**	**Precision**	**F1**	**Balance**
Hierarchical MoE	98.1%	98.2%	98.2%	94.8%
Standalone	83.7%	84.7%	83.8%	82%

Table 4.4: Average performance of hierarchical vs. standalone MLP.

all devices, along with the balance level of the training data. We measure the balance of the training data using the Shannon entropy of device labels, defined as follows:

$$\frac{-1}{\ln(|D|)} \sum_{d \in D} P(d) \ln(P(d))$$

where D is the set of devices and $P(d)$ is the frequency of device d in the training dataset. For the hierarchical MoE, we report the average entropy across the three training subsets used to train the expert models. As shown in Table 4.4, the hierarchical MoE performs significantly better. Using a MoE improves the recall score by 17.2%, which aligns with the 15.6% increase in the balance of the dataset caused by grouping devices based on communication rate.

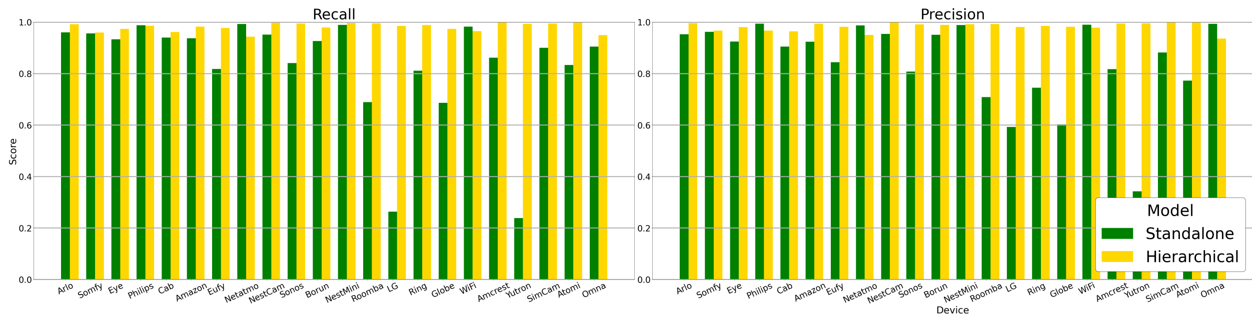

Figure 4.1: Performance of hierarchical MoE vs. standalone MLP on each device.

Figure 4.1 shows that the hierarchical MoE performs better for the majority of devices. The hierarchical MoE correctly identifies at least 90% of samples from each device. This demonstrates its overall reliability compared to the standalone model, which has low correct identification rates in the range of $20 - 80\%$ for several devices. The challenge of imbalanced datasets can also be addressed without a MoE by using more complex deep learning architectures, such as Transformers, which are less sensitive to the distributions of individual features and positioning of packets in a sample compared to MLPs. However, these architectures have higher computational costs. The limitation of the hierarchical MoE compared to this alternative approach is that it depends on accurate estimation of communication rates for unknown traffic samples so that the correct expert can be selected. This can be difficult in an open-world setting, as many devices may have inconsistent communication rates.

Chapter 5

Encoding Relationships in Multi-Flow IoT Traffic[2]

IoT traffic samples may contain packets associated with unrelated tasks or background chatter, which can introduce noise that makes it difficult for a fingerprinting model to understand a device's behavior, as it is unclear which packets are related to each other. The naive solution to this problem, used by many state-of-the-art fingerprinting approaches [30, 8, 9, 26], is to filter traffic into separate flows before applying the model so that all packets in a sample will be from the same flow. However, this approach ignores relationships between packets from separate flows, which may be important for understanding the behavior of more complex tasks that use multiple traffic streams. In this chapter, we propose an approach for encoding the relationships between packets without filtering traffic into separate flows. We refer to these encodings as "*relative features*".

The approach works by encoding the flow identifier of packets in a vector space where the relative orientation of vectors provides information about the relationships between packets in a sample. The model can use this information to better contextualize packets. This encoding space has a far smaller dimensionality than would be required to uniquely represent all possible flow identifiers, which avoids the overfitting issues caused by including raw flow identifiers directly in the input to deep learning models. Using relative features, this chapter addresses research question **RQ2**, how can IoT fingerprints preserve information about the flows of packets without causing overfitting?

This chapter uses a Transformer-based fingerprinting architecture because its self-attention layers provide a mechanism for directly comparing individual packets in a sample to capture relationships, which is not done in MLPs.

[2] Adapted from: [4] Bar-on, M., Krosky, K., Larrieu, F., Bezawada, B., Ray, I., & Ray, I. (2025, June). Jibber-Jabber!: Encoding the (Un-) Natural Language of Network Devices and Applications. In IFIP Annual Conference on Data and Applications Security and Privacy (pp. 3-22). Cham: Springer Nature Switzerland.

5.1 Method

Relative features are encoded representations of the endpoint identifiers of packets. A packet's endpoints are defined by a four-tuple of the following fields:

$<$*source IP address*, *destination IP address*, *source port*, *destination port*$>$.

When combined with the type of transport-layer protocol used, this tuple identifies the uni-directional flow of a packet. The transport-layer protocol is included in the feature vector as a simple categorical variable rather than a relative feature because the set of unique values for transport protocol is much smaller, making it unlikely to cause overfitting.

Relative features are obtained by mapping each field from a global scale to a local scale. The global scale contains all possible unique values for an endpoint field. For example, for an IPv4 address, the global scale is the set of all 32 bit integers. On the other hand, the local scale only contains the unique values in a given sample of traffic.

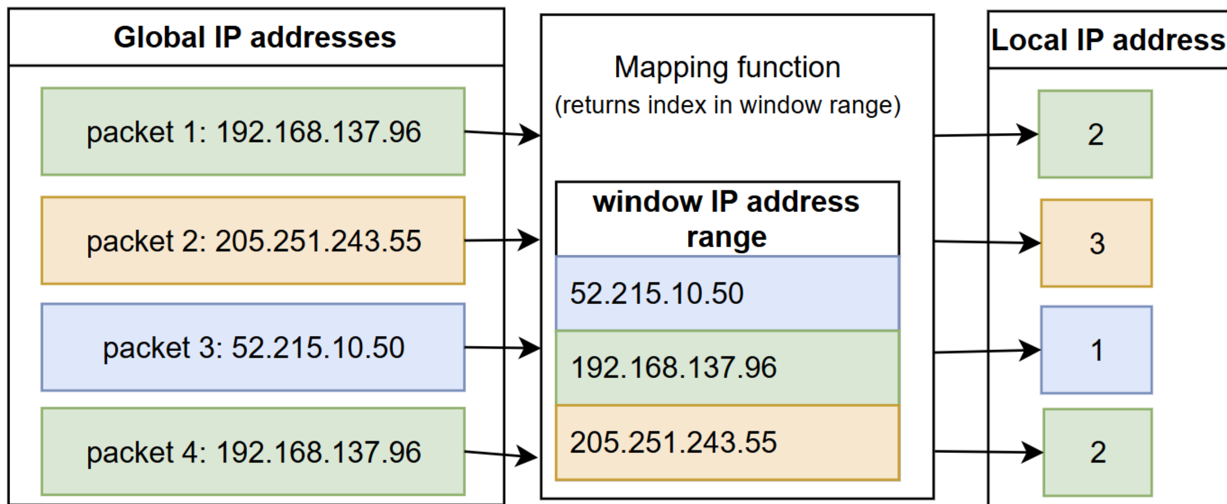

Figure 5.1: Example derivation of relative feature values for IP source address for a sample of 4 packets.

As shown in Figure 5.1, we use a mapping function to convert global/raw endpoint identifier values to a local scale. Let $S = \{\rho^{(1)}, \rho^{(2)}, \ldots, \rho^{(n)}\}$ be a window of n packets where each packet $\rho^{(j)} : (\textit{src-IP,dst-IP,src-prt,dst-prt})$ is defined as a four-tuple containing the packet's endpoint identifiers. To obtain the relative features for a packet, we use the mapping function $R(\rho^{(j)}, S) \rightarrow \mathbb{Z}^4$. We define

the mapping function for endpoint identifier field i of packet $\rho^{(j)}$ as follows:

$$R(\rho^{(j)}, S)_i = \sum_{k \in \pi_i(S)} \mathbb{1}\{k \leq \rho_i^{(j)}\}$$

where $\rho_i^{(j)}$ is the raw value of the ith endpoint identifier field for packet $\rho^{(j)}$, $\pi_i(S)$ is the set of unique values for field i found in window S, and $\mathbb{1}\{condition\}$ is the indicator-function defined in Equation 4.1. Essentially, the mapping function counts the number of unique values for field i in window S that are less than or equal to $\rho_i^{(j)}$.

Relative encoding preserves semantic properties that help the deep learning model identify relationships between packets in a sequence. These relationships enhance the model's understanding of multi-packet behaviors contained in a traffic trace, which improves its decision making. Due to the size of the global scales for endpoint identifiers, there is minimal overlap between the unique sets of values associated with different devices' traffic in the available training data. As a result, using raw values can cause the model to overfit by memorizing device-specific endpoint identifiers rather than learning generalizable patterns, This is undesirable because unseen samples of traffic won't necessarily use the exact same raw endpoint identifiers observed in the training data, causing an overfit model to misclassify them. By mapping values to a local scale, we ensure that devices in the training data cannot be uniquely separated by their endpoint identifiers. This encourages the model to use flow identifiers for capturing relationships between packets instead of identifying devices.

5.1.1 Illustrative Example

To illustrate how relative features can reveal relationships, we use an example window S of five packets ($|S| = 5$) with the following endpoint identifiers and relative features:

- $R\left(\rho^{(1)} : (192.168.137.96,\ 205.251.243.55,\ 42000,\ 443),\ S\right) = [1,2,3,2]$
- $R\left(\rho^{(2)} : (205.251.243.55,\ 192.168.137.96,\ 443,\ 42000),\ S\right) = [2,1,1,3]$

	$\rho^{(1)}$	$\rho^{(2)}$	$\rho^{(3)}$	$\rho^{(4)}$	$\rho^{(5)}$
$\rho^{(1)}$	0.4538	0.0083	0.0614	0.4538	0.0226
$\rho^{(2)}$	0.0171	0.9317	0.0171	0.0171	0.0171
$\rho^{(3)}$	0.0950	0.0129	0.7021	0.0950	0.0950
$\rho^{(4)}$	0.4538	0.0083	0.0614	0.4538	0.0226
$\rho^{(5)}$	0.0397	0.0146	0.1080	0.0397	0.7979

Table 5.1: Attention matrix.

- $R\left(\rho^{(3)} : (192.168.137.96,\ 205.251.243.55,\ 45000,\ 137),\ S\right) = [1,2,4,1]$
- $R\left(\rho^{(4)} : (192.168.137.96,\ 205.251.243.55,\ 42000,\ 443),\ S\right) = [1,2,3,2]$
- $R\left(\rho^{(5)} : (192.168.137.96,\ 205.255.4.123,\ 40000,\ 137),\ S\right) = [1,3,2,1]$

Based on these features, we can see that $\rho^{(1)}$ and $\rho^{(4)}$ belong to the same uni-directional network flow, $\rho^{(2)}$ is the only incoming packet, $\rho^{(3)}$ is communicating with the same host machine as $\rho^{(1)}$ and $\rho^{(4)}$ but is part of a separate flow, and $\rho^{(5)}$ is using the same type of service as $\rho^{(3)}$ (audio/video streaming) but on a different host machine. To illustrate how the Transformer uses relative features to identify relationships, we calculate attention scores between indicator vector representations of these features for the packets in the example. In our Transformer architecture, relative features are implemented as indices in an embedding lookup-table. This is equivalent to using indicator vectors, but with reduced storage and computational overhead.

Let $\delta(\rho^{(j)}, S) \in \{0,1\}^{4*|S|}$ be a vector containing the indicator representations for each relative feature of a packet $\rho^{(j)} \in S$ defined as:

$$\delta(\rho^{(j)}, S)_{k+(i-1)|S|} = \mathbb{1}\left\{R(\rho^{(j)}, S)_i = k\right\}, \forall i \in \{1,\ldots,4\}\, k \in \{1,\ldots,|S|\}$$

Using these representations, we calculate an attention matrix M for S using the attention operation [40] as follows:

$$M_{i,j} = \frac{\exp\left(\sum_k^{4|S|} \delta(\rho^{(i)}, S)_k \delta(\rho^{(j)}, S)_k\right)}{\sum_h^{|S|} \exp\left(\sum_k^{4|S|} \delta(\rho^{(i)}, S)_k \delta(\rho^{(h)}, S)_k\right)}$$

We show the attention matrix M for this example in Table 5.1. This matrix shows that the attention scores between packets is higher when the packets are from related flows. $\rho^{(2)}$ has a high attention score with itself because it is the only incoming packet. The attention values along the diagonal are smaller for outgoing packets because there are more of them in the window. The attention values between $\rho^{(1)}$ and $\rho^{(4)}$ are relatively high, reflecting the strong relationship between packets in the same uni-directional flow.

5.1.2 Fingerprinting Architecture

Our architecture consists of an embedding layer, a stack of Transformer encoder blocks, and an MLP classifier with a sequence-wise average pooling layer. Each input samples is a feature matrix representing a sequence of packets with one row per packet. Each row is a feature vector including the four relative features, along with other important non-relative network features.

Embedding Layer

The embedding layer homogenizes the input data by projecting the features onto a continuous vector space. The embedding layer handles relative features using a set of 4 lookup-tables implemented as a tensor $\mathcal{W} \in \mathbb{R}^{4 \times w \times d}$ with one weight matrix per relative feature, where w is the window size and d is the embedding dimensionality of the model. Lookup-tables are used by returning the row corresponding to a relative feature value. The selected row vectors from each table are combined through element-wise addition. We set the number of rows to the window size w for all lookup-tables since this is the maximum possible number of unique values for a particular endpoint identifier that may be observed in a window of w packets. Therefore, each relative feature will be in the range $[1, w]$. Let $X \in \mathbb{R}^{w \times 4}$ be the relative features for the packets in an input sequence. The embedding layer will produce a matrix $E^{(r)} \in \mathbb{R}^{w \times d}$ as follows:

$$E^{(r)}_{i,j} = \sum_{k=1}^{4} \mathcal{W}_{k, X_{i,k}, j}$$

$E^{(r)}$ is then combined with the embeddings from the non-relative features $E^{(s)} \in \mathbb{R}^{w \times d}$ through element-wise addition before adding the positional encoding, as defined in Section 2.3.4, to produce the output of the embedding layer $E \in \mathbb{R}^{w \times d}$ as follows:

$$E = PE \oplus E^{(r)} \oplus E^{(s)}$$

Representing relative features as lookup-table indices will produce the same result as using indicator vectors, but with lower computation and storage. Let v be a relative feature with corresponding lookup-table $\mathcal{W} \in \mathbb{R}^{w \times d}$. Our lookup-table approach will represent v as the row-vector $\mathcal{W}_v \in \mathbb{R}^d$, which is equivalent to taking the dot-product of $\mathcal{W}$ and the indicator vector of v, $\mathbb{1}\{v = j | j \in \{1, \cdots, w\}\}$:

$$\mathcal{W}_{v,i} = \sum_{j=1}^{w} \mathcal{W}_{j,i} * \mathbb{1}\{v = j\}$$

Transformer and Classifier

The input to the Transformer stack is the matrix of packet embeddings E. The Transformer identifies and encodes relationships between rows of E and outputs a new matrix $Z \in \mathbb{R}^{w \times d}$. The MLP classifier uses Z to predict the device.

5.2 Evaluation

5.2.1 Experimental Setup

As in Chapter 4, we partition our dataset using a train:validation:test ratio of 7:1:2 and run 5 trials. Before training, we standardize the non-relative features of the training, testing, and validation data using the averages and standard deviations of the training data. It is not necessary to standardize relative features, as they are categorical variables. We train our model using *Adam* on the training data with an initial learning rate of 0.01 for a maximum of $10,000$ epochs and evaluate the validation data loss after every epoch, saving the model's parameters if the loss is lower than the previous lowest validation loss. If the validation loss does not improve for 15 epochs, we decay the

Device	Packet Count
Somfy	25000
WthSleep	25000
DropCam	25000
WthBaby	25000
Belkin	25000
SamsungTab	25000
HP	25000
PIXSTAR	25000
Insteon	25000
Triby	25000
Arlo	25000
Eye	25000
LiFX	24823
Cab	17513
Philips	16920
iHome	13872
TPLink	12465
Eufy	11207
Netatmo	10576
Amazon	8982
NestCam	7484
Sonos	6382

Table 5.2: Experimental Devices

learning rate by a factor of $3\times$ and halt the training if the learning rate decays below the threshold 0.0001. We evaluate our model on the testing data using the parameters that resulted in the lowest validation loss.

For these experiments, we include the 22 devices that had at least $5,000$ packets in the available datasets as shown in Table 5.2.

Additional hyperparameters are listed in Appendix .1 Table 2.

Features

In addition to the 4 relative features, we included 13 non-relative network features per packet. These features encode basic information about the protocol and payload of the packet and include a mixture of binary and continuous features as shown in Table 5.3. Payload Length and Payload Entropy are statistics describing the payload of a packet, while the remaining features are extracted from header fields.

Feature	Explanation
Subnet Src	1 if source IP is in subnet else 0
Subnet Dst	1 if destination IP is in subnet else 0
Broadcast Dst	1 if destination IP is broadcast else 0
TLS Handshake	1 if packet includes TLS handshake header else 0
TCP	1 if packet includes TCP header else 0
UDP	1 if packet includes UDP header else 0
HTTP	1 if packet uses HTTP port else 0
SSL/TLS	1 if packet uses SSL port else 0
Common Src	1 if packet uses common source port else 0
Common Dst	1 if packet uses common dst port else 0
Header Length	length of transport-layer header (bytes)
Payload Length	length of payload (bytes)
Payload Entropy	information entropy of payload

Table 5.3: Non-relative features.

5.2.2 Performance

Features	Recall	Precision	F1
With Relative Features	99.0%	98.9%	98.9%
Without Relative Features	97.7%	97.2%	97.4%

Table 5.4: Performance with vs. without relative features.

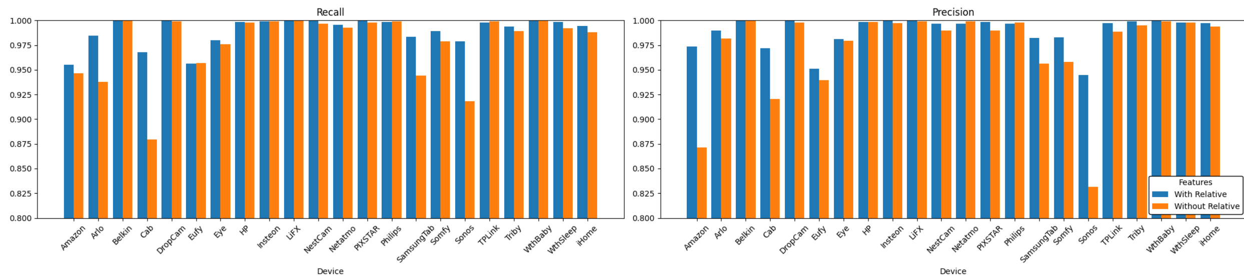

Figure 5.2: Performance with and without relative features.

As shown in Table 5.4, using relative features results in a 1.3% improvement in recall on average and a 1.7% improvement in precision on average. The performance improvement from using relative features is particularly evident in complex devices such as "Cab" and "Sonos" as shown in Figure 5.2. With relative features, the model is able to understand the complex network behaviors of these devices. These results demonstrate the potential of relative features for improving the performance and reliability of IoT fingerprinting models.

Chapter 6

Adaptable IoT Fingerprinting[3]

The global IoT industry is rapidly expanding and new IoT devices are being introduced regularly. When new devices are introduced into a network, existing fingerprinting models are unable to identify them, as they were not part of the data used to train the model. When a new device is added, the network becomes vulnerable to attacks until the fingerprinting model can be updated to integrate the new device.

With traditional fingerprinting methods, the process of adapting a fingerprinting model to accommodate new devices is inefficient, as the existing model is not reused. Instead, a new model is trained from scratch to identify the new and original devices. Reusing the original model can result in reduced performance on the original devices due to catastrophic forgetting [22]. It can also lead to poor performance on new devices when their network behaviors are outside of the distribution learned from the original devices.

To address these challenges, this chapter proposes a bi-component architecture that separates the training process into two stages: (1) extraction of network behavior patterns and (2) device identification from extracted patterns. The first component of the architecture is a Transformer-based *behavior extractor (BE)* responsible for extracting and encoding the patterns of multi-packet network behaviors in traffic samples. The BE produces an encoding, which acts like an enhanced fingerprint where packet features are enriched with context from other packets in the sample based on their roles in the device's behavior. This component is trained during the first training stage. The second component, referred to as a *fingerprint interpreter*, is an MLP classifier responsible for identifying devices using enhanced fingerprints produced by the BE. This component is trained in the second stage. This approach improves efficiency by enabling the reuse of the BE when a model is adapted

[3] Adapted from: [5] Bar-on, M., Patterson, K., Bezawada, B., Ray, I., & Ray, I. (2025, July). " Bring your own device!": Adaptive IoT Device-type Fingerprinting using Automatic Behavior Extraction [Work In Progress Paper]. In Proceedings of the 30th ACM Symposium on Access Control Models and Technologies (pp. 201-206).

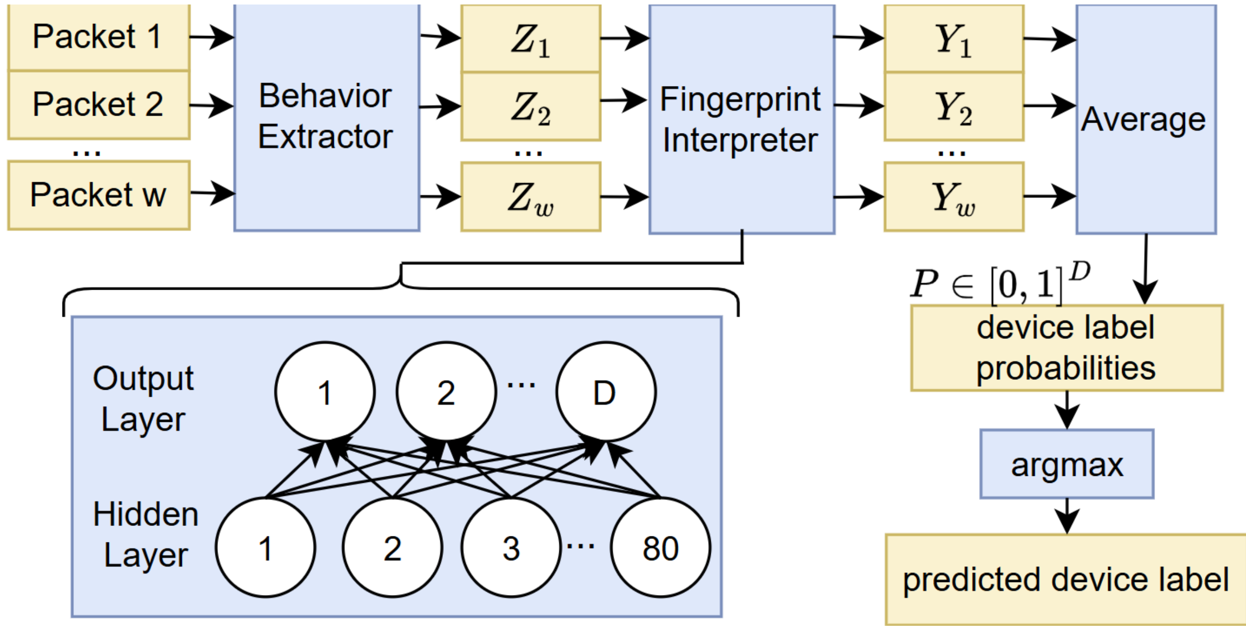

Figure 6.1: Diagram of bi-component architecture.

to identify new devices. This is efficient because the BE accounts for the majority of the cost of training the bi-component architecture. When new devices are introduced, a fingerprinting model can be adapted by creating a new fingerprint interpreter and repeating the second training stage.

The first training stage is not repeated, which avoids the issue of catastrophic forgetting and allows the model to maintain high performance on original devices. To achieve high performance on new devices, our approach leverages a generalized training technique for the BE based on self-supervised learning. This allows the BE to extract behavioral information from devices, even if they were not part of the initial set of devices available during the first training stage. Our BE training technique formulates behavior extraction as a separate task from device identification. Instead of learning to separate devices, the BE learns to extract behaviors by considering relationships between packets.

Through this two-stage training, our bi-component architecture addresses research question **RQ3**, how can existing fingerprinting models be reused to accurately identify new devices without reducing performance on the original devices?

6.1 Method

As shown in Figure 6.1, the bi-component architecture consists of two deep learning components: the BE and the interpreter. These components are trained in separate stages and each perform a different stage of the fingerprinting process during inference.

6.1.1 Behavior Extractor

The BE consists of a Transformer encoder and a linear layer. The output of the Transformer is projected down to 15-dimensional space using the linear layer. This encourages generalizability and reduces the size of the interpreter's input layer. The Transformer extracts behavioral patterns by identifying and encoding relationships between packets in a sample using its attention mechanisms. Given an input sample $X \in \mathbb{R}^{w \times m}$ of w packets where m is the number of features, the BE outputs an enhanced fingerprint $Z \in \mathbb{R}^{w \times 15}$, which summarizes the temporal and feature dependencies in the input.

BE Training

We train the BE using a training *procedure* consisting of one or more self-supervised learning tasks. Our self-supervised tasks include *anomaly-locating*, *denoising*, and *masked-autoencoding*. Training on self-supervised tasks encourages the BE to learn general behavioral patterns, allowing it to effectively extract behaviors from new devices. For each task, we train a self-supervised model $g(X) = f(BE(X))$ consisting of the BE followed by a task-specific MLP f. Each task also has an associated loss function that quantifies the performance of $g(X)$. We update the BE by back-propagating losses through the task's MLP to calculate the gradient of the loss with respect to the BE's learnable parameters.

In the *masked-autoencoding* task, we apply a binary mask to the input features and train the model to reconstruct the missing values. For each input sample, we randomly generate a mask $M \in \{0,1\}^{w \times m}$ from a binomial distribution where each value has a 60% probability of having the value 1. The output of the model $Y \in \mathbb{R}^{w \times m}$ is defined as $Y = g(M \odot X)$ where $\odot$ denotes

element-wise multiplication. We train the model to minimize the reconstruction loss, defined as the mean-squared-error (MSE) between X and Y.

The *denoising* task is the same as the *masked-autoencoding* task; however, instead of applying a binary mask, we add a noise vector $\mathcal{U} \in (-0.15, 0.15)^{w \times m}$ sampled from a uniform distribution to the input. The output of the model $Y \in \mathbb{R}^{w \times 15}$ is defined as $Y = g(X \oplus \mathcal{U})$ where $\oplus$ denotes element-wise addition.

In *anomaly-locating*, we randomly insert an anomalous packet into each sample and train the model to predict its position relative to each packet. For each input sample $X \in \mathbb{R}^{w \times m}$, we randomly select one row a, and replace it with a random vector $\mathcal{A} \in [-1, 1]^m$. We implement this as a 3-class classification problem where, for each packet X_i, the model predicts the most likely option from the set of classes $C = \{i < a, i = a, i > a\}$. The MLP includes a $softmax$ operation following the output layer, and produces a probability matrix $P \in [0, 1]^{w \times 3}$ where $P_i = [P(i < a|X), P(i = a|X), P(i > a|X)]$. We train the model to minimize the negative-log-likelihood loss defined for a sample output P as follows:

$$-\frac{1}{3w} \sum_{i=1}^{w} \begin{cases} \log(P_{i,1}) & \text{if } \ i < a \\ \log(P_{i,2}) & \text{if } \ i = a \\ \log(P_{i,3}) & \text{if } \ i > a \end{cases}$$

To apply a procedure, we start by randomly initializing the learnable parameters of a single BE, along with individual MLPs for each of the tasks in the procedure. We then combine the BE and the MLPs to create a single model. In this model, each task has a designated path that spans the BE and its associated MLP so that the BE is shared among all of the paths. We also assign a learning rate with an initial value of 0.005 to each path. During the procedure, we iteratively optimize the parameters along each path to minimize the loss function for its associated task on the training data. In each epoch, we calculate the gradients of the loss functions *w.r.t.* each path using the traffic from the initial set of devices, then use the gradients to adjust the parameters along the path. We then evaluate the loss function of each path on the validation data. If the validation loss for a path does not improve for 15 epochs, we reduce its learning rate by a factor of $3\times$. If the

learning rate for a path decays below 0.0001, we remove the path from the combined model and discard its MLP. Once all paths have been removed, the procedure terminates.

6.1.2 Fingerprint Interpreter

The fingerprint interpreter is an MLP classifier with a sequence-wise average pooling layer. Given an enhanced fingerprint $Z \in \mathbb{R}^{w \times 15}$ from the BE, the interpreter produces an output sequence $Y \in \mathbb{R}^{w \times |D|}$, where D is the set of devices. The sequence-wise average pooling layer converts this to a probability distribution $P \in [0,1]^{|D|}$ over D by aggregating the outputs from each packet.

Interpreter Training

Compared to the BE, the fingerprint interpreter is small and inexpensive to train. To train a fingerprint interpreter $f(Z)$, we use the BE to encode samples of IoT traffic. Then, we train f to identify the devices that generated the enhanced fingerprint samples. Specifically, let $Z \in \mathbb{R}^{N \times w \times 15}$ be a batch of N enhanced fingerprints with labels $T \in [0,1]^{N \times |D|}$. We train f to minimize the negative-log-likelihood loss between $f(Z)$ and T. We train the fingerprint interpreter using the *Adam* optimizer with an initial learning rate of 0.03, a decay rate of $3\times$, and decay threshold of 0.0001. After training, we use f to create a fingerprinting model $g = f(BE(X))$.

6.1.3 Adapting a Model

An existing model can be adapted by training a new interpreter to identify an updated set of devices. Let $D' = D \cup D^{new}$ be the updated set of devices, where D is the set of original devices and D^{new} is the set of newly introduced devices. The new fingerprint interpreter $f'(Z) \rightarrow [0,1]^{|D'|}$ has an expanded output dimensionality of $|D'|$. To train f', we collect traffic from the devices in D^{new}, then apply the BE to the collected traffic to generate new enhanced fingerprints. We combine the new enhanced fingerprints from D^{new} with the previously used enhanced fingerprints from D and use them to train f'. We then save a new fingerprinting model, $g' = f'(BE(X))$.

Adapting does not require adjusting the parameters of the BE since it has already learned how to extract behaviors when it was trained on the initial set of devices. This approach to

adapting is efficient because the process of training an interpreter as a separate component is significantly less expensive in terms of memory and computational cost compared to training an equivalent static fingerprinting model, where the two components are merged and trained in a single stage. This is due to the small size of the fingerprint interpreter, making up less than 4% of the architecture's learnable parameters. Removing the BE from the training process allows for more efficient hardware utilization because it requires less GPU buffer space for storing intermediate values during back-propagation. Additionally, training a static model requires more operations per parameter because of the high computational complexity of the attention mechanism in the BE.

6.2 Evaluation

6.2.1 Experimental Setup

Like in previous chapters, we use 5-fold cross validation with a train:validation:test ratio of 7:1:2 and standardize the features using the training data. For these experiments, we use a "static" fingerprinting model as a baseline for comparing the performance and efficiency of our approach. The static model has the same size as the adaptable bi-component model; however, the BE and the fingerprint interpreter are merged and trained jointly for device identification. We evaluate the following procedures for training the BE component:

- **L+M:** anomaly locating and masked autoencoding
- **D:** only denoising
- **L+D:** anomaly locating and denoising
- **M+D:** masked autoencoding and denoising
- **M:** only masked autoencoding

We use the same set of devices as chapter 4; however, we do not divide the dataset into groups based on communication rate. We train each model to identify all 22 devices shown in Table 4.2. Input features include the standard (non-relative) and relative features discussed in chapter 5.

Additional hyperparameters are listed in Appendix .1 in Table 3 for the BE and Table 4 for the task-specific MLPs and the MLP used for the fingerprint interpreter.

6.2.2 Efficiency

We evaluate the efficiency of our approach in terms of the time required to adapt a model when new devices are introduced to a network and explore how the procedure used to train a BE impacts the adapt time. For each procedure, we measure the minimum amount of time required to train a new interpreter to achieve over 98% average recall across all devices. To reflect networks without access to internal GPUs, we measure the adapting time on CPUs as well as GPUs.

	Bi-component models					
	L+M	**D**	**L+D**	**M+D**	**M**	**Static Model**
GPU	384	60	67	303	391	4,698
CPU	22,921	3,619	4,021	18,095	23,323	78,513

Table 6.1: Adapting Time in Seconds.

In Table 6.1, we compare the adaptation efficiency of the bi-component model with each procedure against the static model. In both computational environments, the bi-component model achieves a faster adapting time than the static model for all BE training procedures. Of the five procedures, “D” results in the fastest adapting time, followed closely by “L+D”. On the GPU, this model can be adapted 78.3× faster than the static model, while on the CPU, it is 21.7× faster. The improvement is greater on the GPU because the bi-component model reduces the number of synchronization steps required to prevent overflowing the GPU memory during gradient calculation.

6.2.3 Performance

In these experiments, we withhold some devices from the dataset before training the BE, then re-introduce them and adapt the model to identify them. The devices that are used to train the BE represent the set of original devices, while the devices that are withheld represent new devices that

can only be identified after adapting the model. We run many trials with different sets of original and new devices to thoroughly explore the performance of our bi-component model. The goal of this exploration is to compare how different BE-training procedures affect a model's ability to adapt to changing network requirements.

We evaluate our architecture with varying numbers of original devices to demonstrate its ability to adapt in various scenarios. Cases with fewer original devices represent more challenging scenarios since there are fewer examples that the BE can pull from to learn how to extract behaviors. We evaluate our model with 4, 7, 10, 13, 16, and 19 original devices. For each amount, we repeatedly evaluate our model with different combinations of original devices until we have used each device as an original device, and a new device, at least three times.

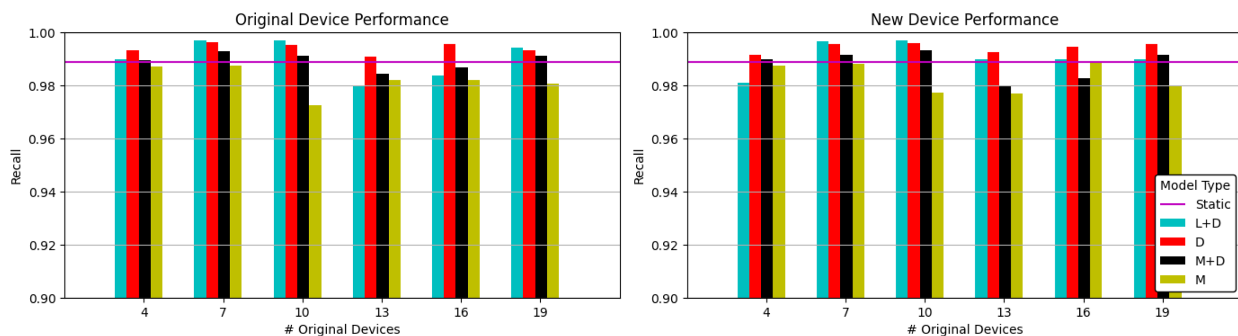

Figure 6.2: Average recall of bi-component models with different BE training procedures and number of original devices. Static model performance is shown as a horizontal line.

In Figure 6.2, we compare how changing the number of original devices affects the performance of bi-component models with different BE-training procedures. In all scenarios, the bi-component models achieve a high recall, above 97%, for the original devices as well as new devices. With all procedures, the model can reliably identify new devices with very few original devices for training the BE. This shows that the BE is capable of learning to extract behaviors from new devices, even if it is only exposed to the specific behaviors of a small number of original devices during BE training. While they still achieve a high identification rate for new and original devices, the "M" and "M+D" models perform worse in some scenarios compared to the other models.

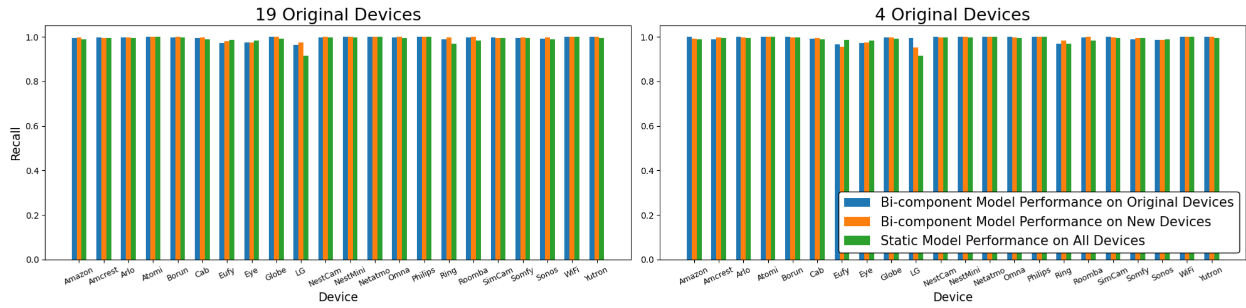

Figure 6.3: Performance of the D bi-component model vs. static model for all devices with 4 and 19 original devices.

This suggests that the masked-autoencoding task is less reliable than the denoising task, as procedures that include masking result in less stable performance across scenarios. Of the four, the "D" model is the only bi-component model that consistently outperforms the static model, both on original and new devices. This demonstrates that it is robust to catastrophic forgetting, while being generalizable to new devices. Although the "L+D" model performs better with 7 or 10 original devices, its performance is less consistent across all scenarios.

Figure 6.3 shows the performance of the bi-component model with a BE trained using the D procedure on each device when it is included in the set of original devices and when it is re-introduced when the model is adapted. For all devices, the bi-component model correctly identifies samples at a rate on-par with the static model.

Chapter 7

Fingerprinting with Fixed Time Observations[4]

IoT fingerprinting approaches that use packet-level representations typically require fixed-length samples of traffic for identifying devices. In a closed-world setting, this constraint is trivial to satisfy, as all of the experimental data is collected in advance. However, when a fingerprinting model is deployed in an open-world setting, it may take arbitrarily long to collect sufficient packets to identify a device. Although this can be mitigated by maintaining a buffer of packets from each device, older packets in the buffer may become irrelevant for understanding the most recent behavior of a device. This is especially true for devices that generate short bursts of traffic. To address these issues, this chapter proposes a fixed-time approach for IoT traffic sampling. Instead of having a fixed number of packets, traffic samples contain all packets observed within a fixed window of time. This ensures that the packets in a sample will be relevant to the most recent device behavior and enables better QoS guarantees because the amount of time required to collect traffic is constant despite differences in communication rates among different devices.

In this chapter, we evaluate fixed-time sampling and introduce an architecture that combines aggregated window features with packet-level representations of traffic. The window features capture the frequency of important activities in a device's communication within fixed time intervals. These features exploit the variation of activity with respect to time to provide additional signals for understanding traffic samples.

[4]Adapted from: [2] Bar-on, M., Alqobaisi, A., Bezawada, B., Ray, I., Ray, I.: It's about time!: Exploiting timing variance for iot device-type fingerprinting. In: 2025 IEEE 7th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA) (2025), to appear

7.1 Methods

7.1.1 Traffic Sampling

Fingerprints are extracted from samples of traffic where each sample includes the set of incoming and outgoing packets observed from a particular device within a fixed window of time. Let $t(\rho^{(i)})$ be the timestamp for packet $\rho^{(i)}$, the ith packet observed from an IoT device. The traffic sample $\mathcal{T}^{(i)}$ starting from packet $\rho^{(i)}$ will contain the following set of packets:

$$\mathcal{T}^{(i)} = \left\{ \rho^{(j)} \mid t(\rho^{(i)}) + l < t(\rho^{(j)}),\ j \in \{i, i+1, \ldots, i+w-1\} \right\}$$

where l is the duration of the capture time-window and w is the maximum number of packets per sample. We put a limit on the maximum number of packets per sample to enable efficient vectorized training. In our experiments, we set $w = 50$.

We use two types of features to represent traffic samples: *"packet features"* and *"window features"*. Packet features represent the attributes of individual packets in the sample, while window features represent the frequency of important activities in the sample. In the model input, packet features are represented as a feature matrix. For each sample of traffic, we generate an input matrix $X \in \mathbb{R}^{w \times m}$ where w is the maximum number of packets and m is the number of packet features. For traffic samples with fewer than w packets, we pad the matrix with zeros to ensure it has w row vectors. Let $\mathcal{T}^{(i)}$ be a sample of traffic, and let $f(\rho^{(i)}) \rightarrow \mathbb{R}^m$ be a function that returns a packet feature vector from packet $\rho^{(i)}$. The input feature matrix $X^{(i)} \in \mathbb{R}^{w \times m}$ for $\mathcal{T}^{(i)}$ will be defined as follows:

$$X_j^{(i)} = \begin{cases} f\left(\rho^{(i+j-1)}\right) & \text{if } j \leq |\mathcal{T}^{(i)}| \\ \{0\}^m & \text{else} \end{cases}$$

where $\{0\}^m$ is an $[m]$-dimensional vector of zeros. On the other hand, since window features are aggregated over all packets in a sample, they are represented in the model input as an individual vector $G \in \mathbb{R}^g$, where g is the number of window features. A device fingerprint is defined as a tuple $< X \in \mathbb{R}^{w \times m}, G \in \mathbb{R}^g >$.

7.1.2 Fingerprinting Architecture

Our architecture consists of a Transformer encoder and an MLP classifier with sequence-wise average pooling. The encoder processes the packet features and presents an encoded representation to the MLP. The MLP uses the encoded representation and the window features to predict the device.

Transformer Encoder

We use a standard Transformer encoder with a modification to the attention mechanism to enable vectorized training. The modified attention mechanism includes a mask that allows the encoder to ignore padding rows in the input feature matrix. Let $X \in \mathbb{R}^{w \times d}$ be the input to an attention layer where d is the embedding dimensionality, and let s be the number of packets in the sample. The mask $M \in \{-\infty, 0\}^{w \times w}$ is defined as follows:

$$M_{i,j} = \begin{cases} -\infty & \text{if } j > s \\ 0 & \text{else} \end{cases}$$

The modified attention operation is defined as follows:

$$softmax\left(\frac{qk^T}{\sqrt{u}} \oplus M\right) v$$

where $q, k, v \in \mathbb{R}^{w \times u}$ are $[u]$-dimensional subspaces of the Query, Key, and Value projections of X.

Classifier

The MLP classifier includes modifications to the input and pooling layers. The input layer is a hidden layer that integrates information from the window features with the output of the encoder. The input layer includes learnable bias $\beta \in \mathbb{R}^h$ and two sets of learnable weights: $\mathcal{W}^{(E)} \in \mathbb{R}^{d \times h}$ for the output of the encoder and $\mathcal{W}^{(G)} \in \mathbb{R}^{g \times h}$ for window features, where h is the hidden dimensionality of the layer. Let $E \in \mathbb{R}^{w \times d}$ be the output of the encoder and let $G \in \mathbb{R}^g$ be the

vector of window features. The output $H \in \mathbb{R}^{w \times h}$ of the modified input layer is defined as:

$$H_{i,j} = \sigma\left(\beta_j + \sum_{k=1}^{g} G_k \mathcal{W}_{k,j}^{(G)} + \sum_{l=1}^{d} E_{i,l} \mathcal{W}_{l,j}^{(E)}\right)$$

Where σ is the activation function.

Like the attention layer, the pooling layer is modified to ignore padding during vectorized training. Let $Z \in \mathbb{R}^{w \times |D|}$ be the input to the pooling layer where D is the set of devices. The output of the pooling layer $Z' \in \mathbb{R}^{|D|}$ is defined as follows:

$$Z'_i = \frac{1}{s} \sum_{j=1}^{s} Z_{j,i}$$

7.2 Evaluation

7.2.1 Experimental Setup

We use the same dataset partitioning, cross-validation, and data standardization as previous chapters. We evaluate our approach with four different durations for the sampling time-window: 0.1s, 10ms, 1ms, and 0.1ms. Shorter durations correspond to faster fingerprinting times, as less time is required to collect traffic before identifying a device. As a baseline, we use fixed-length sampling with a window size of 50 packets and an unlimited time-window for packet capture. We train our models using *Adam* with the same initial learning rate and decay settings as Chapter 5. Additional hyperparameters are listed in Appendix .1 in Table 5.

Device	# Packets	Average Packets Per Sample Time-Window			
		0.1s	10ms	1ms	0.1ms
PIXSTAR	25000	3.059	2.56	1.397	1.095
Insteon	25000	47.785	10.867	1.678	1.01
WthBaby	25000	1.391	1.333	1.274	1.005
Triby	25000	1.401	1.146	1.105	1.003
DropCam	25000	1.046	1.021	1.0	1.0
Arlo	25000	39.568	36.335	35.4	34.028
Somfy	25000	17.321	6.376	2.888	1.981
SamsungTab	25000	4.559	2.938	1.441	1.045
Eye	25000	27.264	10.482	7.831	6.091
Belkin	25000	25.073	6.755	1.806	1.271
HP	25000	2.864	1.638	1.096	1.023
WthSleep	25000	1.559	1.428	1.023	1.001
LiFX	24823	1.737	1.326	1.059	1.0
Cab	17513	22.355	12.474	4.402	3.327
Philips	16920	2.515	1.304	1.137	1.028
iHome	13872	1.245	1.098	1.047	1.001
TPLink	12465	1.604	1.464	1.106	1.015
Eufy	11207	42.521	39.113	34.399	32.173
Netatmo	10576	44.97	24.054	18.157	17.336
Amazon	8982	49.577	49.387	49.361	49.355
NestCam	7484	28.736	26.488	25.696	25.546
Sonos	6382	36.661	36.481	32.903	31.897

Table 7.1: Experimental devices, number of packets and average packets per sample with each time-window.

Table 7.1 shows the 22 devices that we use for these experiments and their average number of packets per sample for each time-window.

Features

Feature	Duration			
	0.1s	10ms	1ms	0.1ms
DNS Queries	30.76%	52.51%	76.60%	81.48%
DNS Responses	38.95%	62.92%	86.11%	89.59%
Num Packets	32.48%	46.25%	49.89%	54.05%
Unique Source IPs	8.63%	39.67%	59.21%	67.23%
Unique Destination IPs	10.76%	43.82%	60.57%	60.82%
Unique Destination Ports	26.09%	51.10%	67.67%	74.07%
Unique Source Ports	31.12%	50.70%	66.65%	71.09%

Table 7.2: Percent of total variance in feature value that is accounted for by variance among samples from the same device.

For our packet features, we use the 13 non-relative features from Chapter 5 shown in Table 5.3. We use the following 7 window features:

- Total number of packets in a sample
- Total number of DNS queries in a sample
- Total number of DNS responses in a sample
- Total number of unique source IP addresses in a sample
- Total number of unique destination IP addresses in a sample
- Total number unique source ports in a sample
- Total number of unique destination ports in a sample

Table 7.2 shows the proportion of overall variance for each window feature that is accounted for by the average intra-device variance, or variance among samples from the same device.

For a feature j, this proportion is defined as:

$$\frac{\frac{1}{|D|}\sum_i^{i \in D} Var(\{G_{k,j} \mid T_k = i\})}{Var(G_{*,j})}$$

where D is the set of devices, $G_{k,j}$ is the value for feature j at sample k, T_k is the device for sample k, and $Var(x)$ returns the variance of a vector x. This proportion is smaller when there is more variation in feature values between different devices. The amount variance that is accounted for by intra-device variance increases as the time-window duration decreases for all window features. However, even with small durations, there are still observable differences in feature distributions among different devices.

7.2.2 Performance

	Sampling	Recall	Precision	F1
	Fixed-length	98.9%	98.8%	98.8%
Fixed-time	0.1s	99.2%	99.5%	99.4%
	10ms	98.5%	99.3%	98.9%
	1ms	97.6%	99.0%	98.3%
	0.1ms	97.4%	99.0%	98.2%

Table 7.3: Average performance with fixed-length sampling and fixed-time sampling with different time-window durations.

As shown in Table 7.3, our fixed-time fingerprinting approach is capable of reliably identifying devices. With a time-window of 0.1 seconds, our approach performs better on all metrics than fixed-length sampling. Fixed-time sampling achieves higher precision scores than fixed-length with all time-windows. In terms of recall, our approach performs better with longer time-windows because there are more available packets in each window for understanding the devices' behaviors. However, the performance is only $\approx 1.5\%$ worse with 0.1ms than with 100ms despite requiring $1{,}000\times$ less time for capturing packets to identify a device. This demonstrates that IoT devices can be identified accurately without requiring long intervals for collecting traffic. We recommend tuning the time-window duration based on the specific requirements of the network. A shorter time-window duration can satisfy stricter QoS requirements while still achieving strong performance.

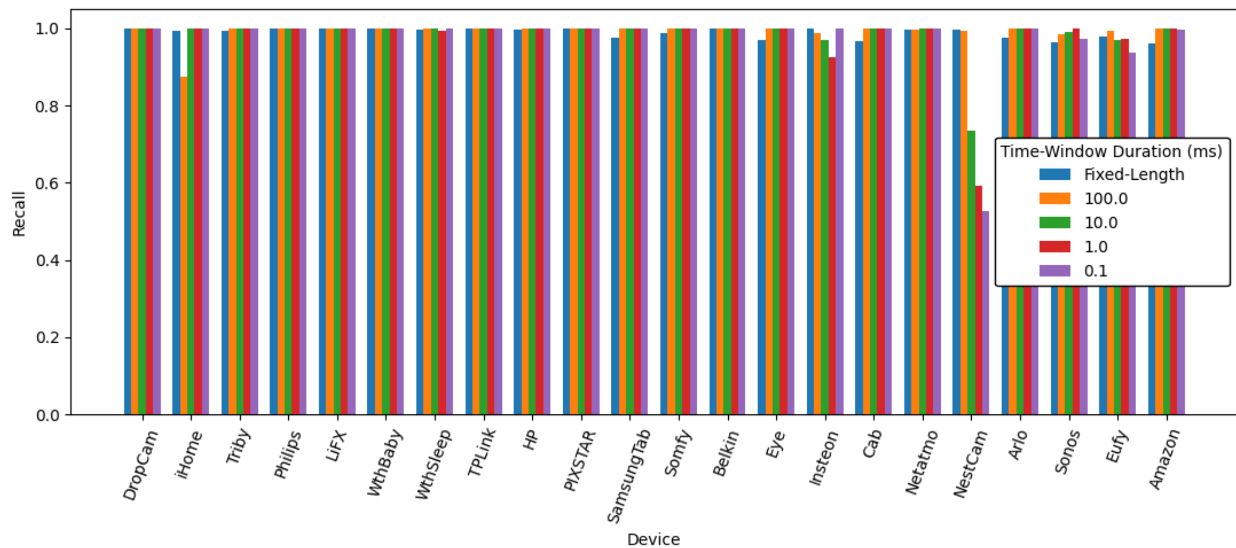

Figure 7.1: Recall of each device with fixed-length sampling and fixed-time sampling with different time-window durations.

As shown in Figure 7.1, our fixed-time approach performs on-par with fixed-length sampling across nearly all devices. With short time-windows, our approach has a low identification rate for NestCam samples. This can be explained by Figure 7.2, which shows the distribution of predictions for NestCam samples. With fixed-length sampling and fixed-time with a duration of 0.1s, the model correctly identifies all NestCam samples. On the other hand, with shorter time-windows,

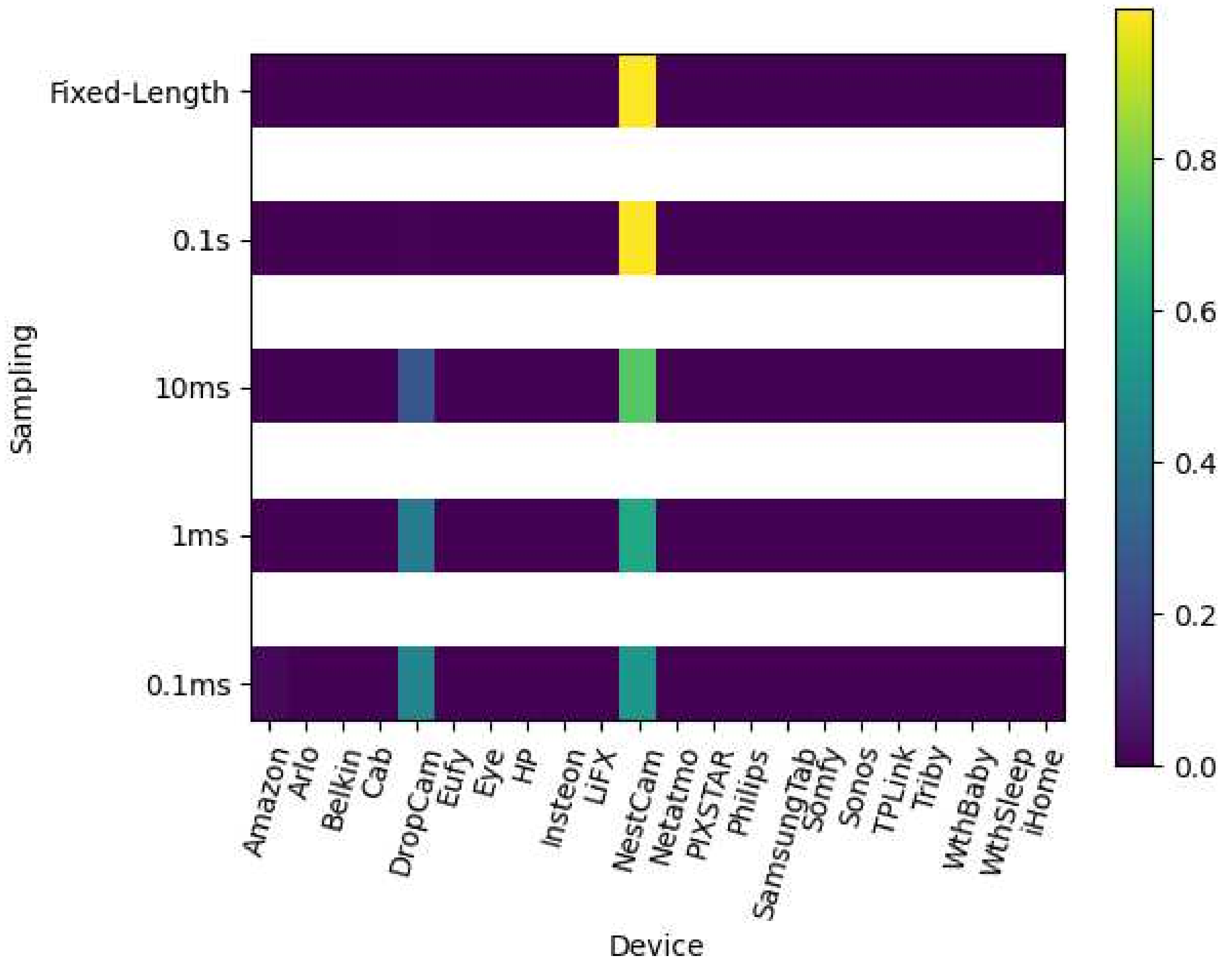

Figure 7.2: Prediction density of NestCam samples.

the model incorrectly assigns the DropCam label to a portion of NestCam samples. These devices serve a similar function and it is likely that they don't exhibit sufficient difference in behavior within shorter time-windows.

Chapter 8

Conclusion and Future Work

8.1 Summary and Outcomes

This work addresses 5 challenges of using deep learning for IoT fingerprinting. To address the challenge of limited training data availability from individual observers, we propose a federated learning approach to improve the coverage of unique IoT device behaviors during training by utilizing data collected by separate observers in a privacy-preserving manner. To address the challenge of imbalanced datasets introducing biases into the learning process, we design a hierarchical MoE architecture that separates traffic samples based on communication rate before identifying their corresponding devices. To address the challenge of capturing relationships between packets when some packets are not part of the same network flow, we introduce relative features, which encode the communication endpoints of packets relative to other packets in a traffic sample. To address the challenge of adapting an existing fingerprinting model when new devices are introduced to a network, we design a bi-component fingerprinting architecture that enables part of the model to be re-used when it is adapted. Finally, to address the challenge of unbounded packet capture times for identifying a device, we propose a fixed-time approach for sampling traffic that ensures a consistent capture time among devices with varying communication rates.

In our experimental evaluation, our federated learning approach results in better fingerprinting performance than the centralized learning baseline, demonstrating that distributed training data can be utilized to build a more reliable model. Of the two federated learning aggregation approaches, FedAvg results in slightly better performance than parameter swapping. In an open-world setting, federated learning can be used to increase the quantity and diversity of training samples by utilizing distributed datasets, leading to stronger fingerprinting models than centralized learning.

Our hierarchical MoE results in significant improvements in performance compared to using a standalone MLP. The hierarchical MoE achieves 17.2% higher recall on average. Additionally, it improves the identification rate for samples from several small devices by a factor of over $4\times$.

Our relative features also result in improved performance. Relative features increase recall and precision by over 1%, achieving 99% average recall across 22 devices.

For our adaptable architecture, we evaluate 5 different procedures for training the behavioral extractor component and demonstrate that all 5 reduce the time to adapt a model, both on GPUs and CPUs. The D procedure results in the fastest adaptation time, reducing the time by a factor of $78.3\times$. The D procedure also results in the best performance, both on new and original devices. It consistently performs better than the baseline in our evaluation, which covers scenarios where the number of new devices ranges from 5 to 18. Our results demonstrate that existing fingerprinting models can be adapted without compromising performance on new or original devices.

Our fixed-time sampling approach demonstrates that traffic capture time can be constrained without sacrificing the performance of fingerprinting models. Our approach performs the best with the longest capture time duration of 0.1 seconds and achieves a higher recall and precision than the fixed-length sampling baseline. Reducing the capture time by $1,000\times$ only reduces the recall by less than 2%.

8.2 Future Work

Future work will evaluate these approaches on larger datasets containing more unique IoT devices, as well as in open-world settings. Additionally, future work will evaluate alternative deep learning architectures for IoT fingerprinting, such as LSTM or MAMBA. Approaches from different chapters will also be combined and evaluated together. For example, the Transformer architecture with relative features from chapter 5 can be trained using the federated learning approach from chapter 4.

Other future work will extend individual approaches to improve results or better address their associated challenges. For chapter 4, future work will enhance the federated learning algorithm

to address non-independent and identically distributed (non-i.i.d.) IoT training data. For chapter 5, future work will explore alternative techniques for encoding packet endpoints to better capture bi-directional packet relationships. Finally, for chapter 6, future work will use knowledge-retrieval-based approaches in-place of the MLP-based fingerprint interpreter for more efficient adaptability.

References

[1] Alyahya, T.N., Aniello, L., Sassone, V.: Scanef-iot: Scalable network fingerprinting for iot device. In: Proceedings of the 19th International Conference on Availability, Reliability and Security. ARES '24, Association for Computing Machinery, New York, NY, USA (2024)

[2] Bar-on, M., Alqobaisi, A., Bezawada, B., Ray, I., Ray, I.: It's about time!: Exploiting timing variance for iot device-type fingerprinting. In: 2025 IEEE 7th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA) (2025), to appear

[3] Bar-on, M., Bezawada, B., Ray, I., Ray, I.: A small world–privacy preserving iot device-type fingerprinting with small datasets. In: International Symposium on Foundations and Practice of Security. pp. 104–122. Springer (2023)

[4] Bar-on, M., Krosky, K., Larrieu, F., Bezawada, B., Ray, I., Ray, I.: Jibber-jabber!: Encoding the (un-) natural language of network devices and applications. In: IFIP Annual Conference on Data and Applications Security and Privacy. pp. 3–22. Springer (2025)

[5] Bar-on, M., Patterson, K., Bezawada, B., Ray, I., Ray, I.: " bring your own device!": Adaptive iot device-type fingerprinting using automatic behavior extraction [work in progress paper]. In: Proceedings of the 30th ACM Symposium on Access Control Models and Technologies. pp. 201–206 (2025)

[6] Bazaluk, B., Hamdan, M., Ghaleb, M., Gismalla, M.S.M., Correa da Silva, F.S., Batista, D.M.: Towards a transformer-based pre-trained model for iot traffic classification. In: NOMS 2024-2024 IEEE Network Operations and Management Symposium. pp. 1–7 (2024). https://doi.org/10.1109/NOMS59830.2024.10575448

[7] Bezawada, B., Bachani, M., Peterson, J., Shirazi, H., Ray, I., Ray, I.: Behavioral fingerprinting of iot devices. In: Proceedings of the 2018 Workshop on Attacks and Solutions

in Hardware Security, ASHES@CCS 2018, Toronto, ON, Canada, October 19, 2018. pp. 41–50. ACM (2018)

[8] Charyyev, B., Gunes, M.H.: Iot traffic flow identification using locality sensitive hashes. In: ICC 2020-2020 IEEE International Conference on Communications (ICC). pp. 1–6. IEEE (2020)

[9] Chen, Q., Song, Y., Jennings, B., Zhang, F., Xiao, B., Gao, S.: Iot-id: robust iot device identification based on feature drift adaptation. In: 2021 IEEE Global Communications Conference (GLOBECOM). pp. 1–6. IEEE (2021)

[10] Dadkhah, S., Mahdikhani, H., Danso, P.K., Zohourian, A., Truong, K.A., Ghorbani, A.A.: Towards the development of a realistic multidimensional IoT profiling dataset. Submitted to the 19th Annual International Conference on Privacy, Security & Trust (PST 2022) (August 22–24 2022)

[11] Deval, S.K., Tripathi, M., Bezawada, B., Ray, I.: "x-phish: Days of future past": Adaptive & privacy preserving phishing detection. In: 2021 IEEE Conference on Communications and Network Security (CNS). pp. 227–235 (2021). https://doi.org/10.1109/CNS53000.2021.9705052

[12] Dong, S., Li, Z., Tang, D., Chen, J., Sun, M., Zhang, K.: Your smart home can't keep a secret: Towards automated fingerprinting of iot traffic. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security. p. 47–59. ASIA CCS '20, Association for Computing Machinery, New York, NY, USA (2020)

[13] Dong, Y., Cordonnier, J.B., Loukas, A.: Attention is not all you need: Pure attention loses rank doubly exponentially with depth. In: International conference on machine learning. pp. 2793–2803. PMLR (2021)

[14] Duan, C., Gao, H., Song, G., Yang, J., Wang, Z.: Byteiot: A practical iot device identification system based on packet length distribution. IEEE Transactions on Network and Service Management **19**(2), 1717–1728 (2021)

[15] Fan, L., He, L., Wu, Y., Zhang, S., Wang, Z., Li, J., Yang, J., Xiang, C., Ma, X.: Autoiot: Automatically updated iot device identification with semi-supervised learning. IEEE Transactions on Mobile Computing **22**(10), 5769–5786 (2023)

[16] Feng, J., Zhao, T., Sarkar, S., Konrad, D., Jacques, T., Cabric, D., Sehatbakhsh, N.: Fingerprinting iot devices using latent physical side-channels. Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. **7**(2) (Jun 2023)

[17] Goodband, J., Haas, O., Mills, J.: A mixture of experts committee machine to design compensators for intensity modulated radiation therapy. Pattern Recognition **39**(9), 1704–1714 (2006). https://doi.org/https://doi.org/10.1016/j.patcog.2006.03.018, https://www.sciencedirect.com/science/article/pii/S0031320306001294

[18] Iacovazzi, A., Wang, H., Butun, I., Raza, S.: Towards Cyber Threat Intelligence for the IoT . In: 2023 19th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT). pp. 483–490. IEEE Computer Society, Los Alamitos, CA, USA (Jun 2023). https://doi.org/10.1109/DCOSS-IoT58021.2023.00081, https://doi.ieeecomputersociety.org/10.1109/DCOSS-IoT58021.2023.00081

[19] Jacobs, R.A., Jordan, M.I., Nowlan, S.J., Hinton, G.E.: Adaptive mixtures of local experts. Neural Computation **3**(1), 79–87 (1991). https://doi.org/10.1162/neco.1991.3.1.79

[20] Jordan, M., Jacobs, R.: Hierarchical mixtures of experts and the em algorithm. In: Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan). vol. 2, pp. 1339–1344 vol.2 (1993). https://doi.org/10.1109/IJCNN.1993.716791

[21] Kambourakis, G., Kolias, C., Stavrou, A.: The mirai botnet and the iot zombie armies. In: MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM). pp. 267–272 (2017)

[22] Kemker, R., McClure, M., Abitino, A., Hayes, T., Kanan, C.: Measuring catastrophic forgetting in neural networks. In: Proceedings of the AAAI conference on artificial intelligence. vol. 32 (2018)

[23] Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)

[24] Lopez-Martin, M., Carro, B., Sanchez-Esguevillas, A., Lloret, J.: Network traffic classifier with convolutional and recurrent neural networks for internet of things. IEEE Access **PP**, 1–1 (09 2017)

[25] Luo, Y., Chen, X., Ge, N., Feng, W., Lu, J.: Transformer-based device-type identification in heterogeneous iot traffic. IEEE Internet of Things Journal **10**(6), 5050–5062 (2023)

[26] Mainuddin, M., Duan, Z., Dong, Y., Salman, S., Taami, T.: Iot device identification based on network traffic characteristics. In: GLOBECOM 2022-2022 IEEE Global Communications Conference. pp. 6067–6072. IEEE (2022)

[27] Marchal, S., Miettinen, M., Nguyen, T.D., Sadeghi, A.R., Asokan, N.: Audi: Toward autonomous iot device-type identification using periodic communication. IEEE Journal on Selected Areas in Communications **37**(6), 1402–1412 (2019). https://doi.org/10.1109/JSAC.2019.2904364

[28] Masoudnia, S., Ebrahimpour, R.: Mixture of experts: a literature survey. Artificial Intelligence Review **42**(2), 275–293 (2014)

[29] McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, B.A.: Communication-efficient learning of deep networks from decentralized data. In: Proceedings

of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA. vol. 54, pp. 1273–1282. PMLR (2017)

[30] Meidan, Y., Bohadana, M., Shabtai, A., Guarnizo, J.D., Ochoa, M., Tippenhauer, N.O., Elovici, Y.: Profiliot: A machine learning approach for iot device identification based on network traffic analysis. In: Proceedings of the symposium on applied computing. pp. 506–509 (2017)

[31] Miettinen, M., Marchal, S., Hafeez, I., Asokan, N., Sadeghi, A.R., Tarkoma, S.: Iot sentinel: Automated device-type identification for security enforcement in iot. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). pp. 2177–2184 (2017)

[32] Msadek, N., Soua, R., Engel, T.: Iot device fingerprinting: Machine learning based encrypted traffic analysis. In: 2019 IEEE Wireless Communications and Networking Conference (WCNC). pp. 1–8 (2019)

[33] Ortiz, J., Crawford, C., Le, F.: Devicemien: network device behavior modeling for identifying unknown iot devices. In: Proceedings of the International Conference on Internet of Things Design and Implementation. p. 106–117. IoTDI '19, Association for Computing Machinery, New York, NY, USA (2019)

[34] Pfitzner, B., Steckhan, N., Arnrich, B.: Federated learning in a medical context: A systematic literature review. ACM Trans. Internet Technol. **21**(2) (Jun 2021). https://doi.org/10.1145/3412357, https://doi.org/10.1145/3412357

[35] Phong, L.T., Phuong, T.T.: Privacy-preserving deep learning via weight transmission. IEEE Transactions on Information Forensics and Security **14**(11), 3003–3015 (2019). https://doi.org/10.1109/TIFS.2019.2911169

[36] Ravidas, S., Lekidis, A., Paci, F., Zannone, N.: Access control in internet-of-things: A survey. Journal of Network and Computer Applications **144**, 79–101 (2019).

https://doi.org/https://doi.org/10.1016/j.jnca.2019.06.017, https://www.sciencedirect.com/science/article/pii/S108480451930222X

[37] Sikder, A.K., Petracca, G., Aksu, H., Jaeger, T., Uluagac, A.S.: A survey on sensor-based threats to internet-of-things (iot) devices and applications. ArXiv **abs/1802.02041** (2018), https://api.semanticscholar.org/CorpusID:3638888

[38] Sivanathan, A., Gharakheili, H.H., Loi, F., Radford, A., Wijenayake, C., Vishwanath, A., Sivaraman, V.: Classifying iot devices in smart environments using network traffic characteristics. IEEE Transactions on Mobile Computing **18**(8), 1745–1759 (Aug 2019)

[39] Tang, B., Heywood, M., Shepherd, M.: Input partitioning to mixture of experts. In: Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No.02CH37290). vol. 1, pp. 227–232 vol.1 (2002). https://doi.org/10.1109/IJCNN.2002.1005474

[40] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. p. 6000–6010. NIPS'17, Curran Associates Inc., Red Hook, NY, USA (2017)

[41] Wang, H., Eklund, D., Oprea, A., Raza, S.: Fl4iot: Iot device fingerprinting and identification using federated learning. ACM Trans. Internet Things (jun 2023), https://doi.org/10.1145/3603257

[42] Wang, J., Zhong, J., Li, J.: Iot-portrait: Automatically identifying iot devices via transformer with incremental learning. Future Internet **15**(3) (2023)

[43] Wang, X., Wang, Y., Javaheri, Z., Almutairi, L., Moghadamnejad, N., Younes, O.S.: Federated deep learning for anomaly detection in the internet of things. Computers and Electrical Engineering **108**, 108651 (2023).

https://doi.org/https://doi.org/10.1016/j.compeleceng.2023.108651, https://www.sciencedirect.com/science/article/pii/S0045790623000769

[44] Xu, J., Sun, X., Zhang, Z., Zhao, G., Lin, J.: Understanding and improving layer normalization. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 32. Curran Associates, Inc. (2019)

Appendices

.1 Appendix A: Hyperparameters

In this section, we list the deep learning hyperparameters used in each experiment.

Hyperparameter	**Value**
Window Size	20
# Epochs	800
Mini-Batch Size	$1,000$
Learning Rate	0.005
Activation	*ReLU*
Hidden Layer Sizes	$80, 60, 40$

Table 1: Hyperparameter values for chapter 4 experiments.

Table 1 shows the baseline hyperparameters used in chapter 4. We made a few adjustments to these baselines depending on the experimental settings. For federated learning, we divided the mini-batch size by the number of clients. We also removed the final hidden layer from the MLP for devices with low communication rates in the hierarchical classifier. In Table 2, we show

Hyperparameter	**Value**
# Transformer Blocks	3
MLP Hidden Layer Sizes	80
Embedding Dimension	40
# Attention Heads	6
Activation	*ReLU*
Window Size	50
Dropout Rate	10%

Table 2: Hyperparameters used in chapter 5.

the hyperparameters used in chapter 5. Table 3 shows the hyperparameters of the BE and Table 4 shows the hyperparameters of the interpreters used in chapter 6. Table 5 shows the hyperparameters used in chapter 7 with fixed-length and fixed-time sampling.

Hyperparameter	**Value**
# Transformer Blocks	3
MLP Hidden Layer Sizes	$60, 40$
Embedding Dimension	40
# Attention Heads	6
Activation	*ReLU*
Window Size	32
Dropout Rate	10%

Table 3: Hyperparameters of Behavioral Extractor from chapter 6.

	Task			
Hyperparameter	**Fingerprinting**	**Anomaly**	**Denoising**	**Masked AE**
Activation	*ReLU*	*ReLU*	*Tanh*	*Tanh*
Hidden Layer Sizes	80	80	$60, 40$	$60, 40$

Table 4: Hyperparameters of MLPs for each task from chapter 6.

	Sampling				
Hyperparameter	**Fixed-Length**	**0.1s**	**10ms**	**1ms**	**0.1ms**
# Transformer Blocks	3	3	3	3	3
MLP Hidden Layer Sizes	80	80	80	160	80
Embedding Dimension	80	80	80	40	40
# Attention Heads	6	6	6	6	6
Activation	*ReLU*	*ReLU*	*ReLU*	*ReLU*	*ReLU*
Window Size	50	50	50	50	50
Dropout Rate	10%	10%	10%	0%	10%

Table 5: Hyperparameters used in chapter 7 for each sampling approach.