An Input-Agnostic Hierarchical Deep Learning Framework for Traffic Fingerprinting

Jian Qu
Xi'an Jiaotong University

Xiaobo Ma[*]
Xi'an Jiaotong University

Jianfeng Li
Xi'an Jiaotong University

Xiapu Luo
The Hong Kong Polytechnic University

Lei Xue
Sun Yat-sen University

Junjie Zhang
Wright State University

Zhenhua Li
Tsinghua University

Li Feng
Southwest Jiaotong University

Xiaohong Guan
Xi'an Jiaotong University

Abstract

Deep learning has proven to be promising for traffic fingerprinting that explores features of packet timing and sizes. Although well-known for automatic feature extraction, it is faced with a gap between the *heterogeneousness* of the traffic (i.e., raw packet timing and sizes) and the *homogeneousness* of the required input (i.e., *input-specific*). To address this gap, we design an *input-agnostic* hierarchical deep learning framework for traffic fingerprinting that can *hierarchically* abstract comprehensive *heterogeneous* traffic features into *homogeneous* vectors seamlessly digestible by existing neural networks for further classification. The extensive evaluation demonstrates that our framework, with just one paradigm, not only supports heterogeneous traffic input but also achieves better or comparable performance compared to state-of-the-art methods across a wide range of traffic fingerprinting tasks.

1 Introduction

Due to its independence of handcrafted features [1–4], deep learning has proven to be promising for traffic fingerprinting that explores *features of packet timing and sizes* in the past years [5–7]. As the amount of encrypted network traffic grows, it offers a fundamental technique facilitating broad security-domain scenarios, such as website fingerprinting [8, 9], application fingerprinting [10, 11], Internet of Things (IoT) device identification [12–14], and intrusion detection [15].

Although deep learning is promising and well-known for automatic feature extraction, it has inherent limitations that keep it from being seamlessly applied to traffic fingerprinting.

Traffic Customization. A significant limitation is the requirement of traffic customization (e.g., *traffic tailoring*) in diverse traffic fingerprinting tasks because existing deep learning methods typically require traffic customization to adapt to the *homogeneous* input structure of neural networks (i.e., *input-specific*). Through traffic tailoring, each traffic sample (i.e., a trace with a label) can be easily aligned and represented by one fixed-dimension vector of the same structure. For example, previous studies like [16] truncate a traffic trace and use only a part (e.g., the foremost part) of a traffic trace as the input of neural networks. The reason for this is that most neural networks, such as Stacked Autoencoder (SAE) [15] and Capsule Neural Networks (CapsNet) [17], take vectors with fixed-dimension and the same structure as input. In contrast, the sizes of traffic samples generated by different network activities may vary significantly.

Traffic customization is *labor-intensive* and *ineffective* since it introduces significant human intervention into traffic samples while bottlenecking deep learning's expected feature-learning ability. It also makes deep learning-based traffic fingerprinting sensitive to specific customization methods because of the potential loss of features. Although Recurrent Neural Networks (RNN) may elude traffic tailoring by allowing the input of a variable-length sequence of vectors to represent a traffic sample [18], it requires the vectors to be of fixed dimension and the same structure. Consequently, traffic customization (e.g., fixed-dimension vectorization) remains to meet the input requirement of essentially homogeneous fixed-dimension vectors. Moreover, some studies using RNN models may also use a fixed-length sequence of vectors to represent a traffic sample [19] because variable-length input may comprise long input that harms RNN optimization due to the vanishing gradient problem. The same situation exists in Long Short-Term Memory (LSTM) networks, as confirmed by Rimmer et al. in [2].

Hierarchy Unawareness. More importantly, for a given network activity, its traffic features may hierarchically exist in individual *packets*, *flows* (typical sequences of TCP or UDP packets sharing the same communicating IPs and ports) that reflect the correlation between packets, and the *trace* (i.e., a mixture of flows) involving the correlation between flows. Nevertheless, existing deep learning methods have no specialized built-in mechanisms to characterize such a feature hierarchy (i.e., *hierarchy unawareness*). As a matter of fact, traffic fingerprinting tasks with feature hierarchy are common in the real world, making existing methods lack sufficient

[*]Corresponding author (MOE KLINNS Lab)

sophistication in traffic fingerprinting.

For example, when one visits a website, multiple HTTP(S) connections are established, resulting in multiple variable-length TCP flows, each consisting of a sequence of variable-size packets. To adapt to the required input of existing deep learning methods in this example, a straightforward approach is to unfold the traffic into a sequence of (homogeneous) vectors according to the original packet arrivals (i.e., vectorization of all packets). However, this approach would fail to capture the upper-layer correlation between packets. Take one particular case of correlation "which packets belong to the same flow" as an instance. Even if one manually labels the originating flow of each packet in the vector, it is hard, if not impossible, for existing neural network structures to be effectively aware of which packets belong to the same flow.

To overcome the limitations above, we design an *input-agnostic* hierarchical framework for landing deep learning onto traffic fingerprinting. The design objectives are twofold. The first is to support the heterogeneousness traffic input with reduced task-by-task traffic customization when applying deep learning in traffic fingerprinting. The second is to design specialized built-in mechanisms for deep learning to enable feature hierarchy characterization in traffic fingerprinting.

Fulfilling such two objectives is not easy due to the heterogeneousness-traffic and homogeneousness-input contradiction. The basic idea is to design a *cross-layer* (i.e., packet-flow-trace) deep learning framework that supports traffic feature learning conditioned on *heterogeneous* input (i.e., raw packet timing and sizes). The framework, with reduced traffic customization, can *hierarchically* abstract comprehensive *heterogeneous* traffic features into *homogeneous* vectors seamlessly digestible by existing neural networks for further classification. Such homogeneous vectors involve spatial-temporal inter-packet correlation of all packets in a flow and inter-flow correlation of all flows in a trace.

Due to the input-agnostic nature, the proposed framework reduces the need to perform task-by-task traffic customization and model design across traffic fingerprinting tasks, thereby seamlessly lands deep learning onto traffic fingerprinting. Because of the *built-in* capability of hierarchically abstracting heterogeneous features, the framework also accomplishes better or comparable performance compared to state-of-the-art methods across a wide range of traffic fingerprinting tasks.

To our best knowledge, we are the *first* to enable deep learning to be input-agnostic and hierarchy-aware in traffic fingerprinting. We release datasets and code at `https://github.com/shashadehuajiang/trace_classifier`. Our contributions include the following:

- We design an input-agnostic hierarchical deep learning framework for traffic fingerprinting. The framework supports automatic traffic feature learning conditioned on heterogeneous input (i.e., raw packet timing and sizes). It reduces the need to perform task-by-task traffic customization and model design across fingerprinting tasks. Moreover, its modular design enables the *replaceability* of neural network components for flexible task-specific settings.
- Under the framework, we propose four neural network structures representative of mainstream categories (i.e., chain-structured, tree-structured, attention-structured, and hybrid) to perform hierarchical bottom-up abstraction of heterogeneous cross-layer features. The structures can extract spatial-temporal inter-packet correlation of all packets in a flow and inter-flow correlation of all flows in a trace. We also propose techniques to avoid overfitting and analyze real-world factors (e.g., dataset size and noise) that affect the performance.
- Using both public and proprietary datasets, we evaluate our framework in five typical fingerprinting tasks, such as website fingerprinting, intrusion detection, and keyword searching fingerprinting. The results demonstrate our framework's capacity to handle heterogeneous inputs and better or comparable performance compared to state-of-the-art methods ubiquitously across various tasks with just one paradigm.

2 Background and Problem Description

It has been commonly observed that the vast majority of network traffic is encrypted, and the adoption of network encryption is still rapidly growing [20]. The pervasive use of encryption necessitates traffic fingerprinting, exploiting the encrypted traffic's metadata without decrypting it to infer the underlying plaintext meanings. Traffic fingerprinting offers a fundamental technique facilitating a broad range of security-domain scenarios based on the encrypted traffic analysis, such as website fingerprinting [8, 9], application fingerprinting [10, 11], Internet of Things (IoT) device identification [12–14], and intrusion detection [15].

Regardless of security-domain scenarios, the common objective of traffic fingerprinting is to classify traffic traces into meaningful labels. Figure 2 depicts the typical traffic fingerprinting problem where endpoints communicate with servers and generate encrypted traffic traces. The analyzer sniffs the traffic traces and performs (typically supervised) classification to support network management actions.

Take the website fingerprinting scenario as an example. Suppose in Figure 2 the endpoints access websites via a server that acts as a proxy. An attacker, who can monitor traffic between the endpoints and the server, wants to figure out the specific websites that the endpoints are accessing through the server. To this end, the attacker will select a list of monitored websites, use automated scripts to collect traffic traces of accessing these websites through the same type of proxy, train a classifier, and deploy it to classify the traffic of the endpoints from/to the server. Once an endpoint visits a website that falls into the list of websites, the attacker would identify that visit.

Due to its feature learning capability, deep learning is a

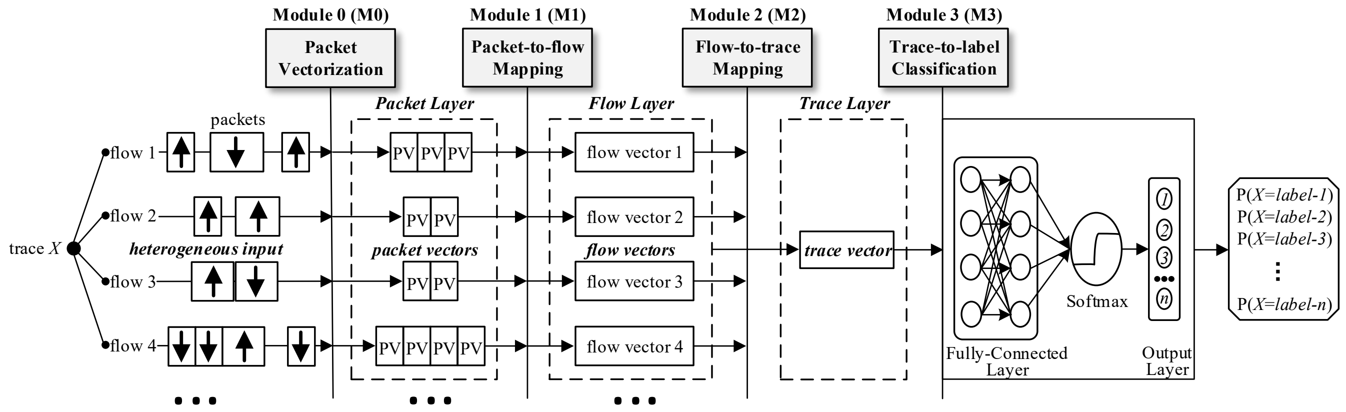


Figure 1: The input-agnostic hierarchical deep learning framework supporting heterogeneous input for traffic fingerprinting.

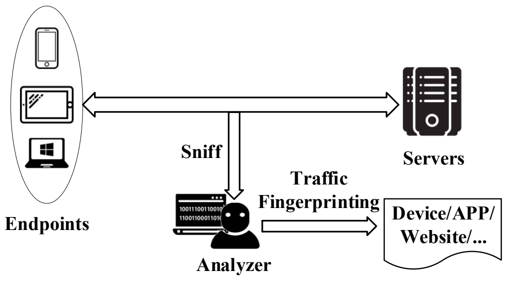


Figure 2: The traffic fingerprinting problem.

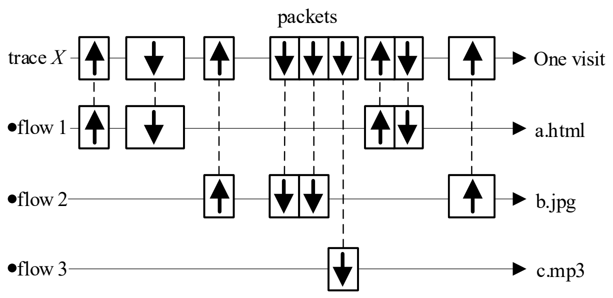


Figure 3: An example of the heterogeneousness of a trace.

promising approach for traffic fingerprinting. However, there is a gap between its required input's homogeneousness and the traffic's heterogeneousness. Figure 3 exemplifies the trace-flow-packet heterogeneousness of a traffic trace, wherein a box with an arrow denotes a (width-sized) packet with its direction. Precisely, the traffic trace results from accessing a website and consists of a sequence of inbound/outbound packets of varying sizes. These packets, despite belonging to the traffic trace of the same network activity (i.e., visiting a website), actually originate from different flows uniquely identified by IPs and ports. Each flow is in charge of transmitting different resource files (e.g., `a.html`, `b.jpg`, `c.mp3`), thereby differing in features, such as the number of packets, packet sizes/directions, and packet arrival times.

As mentioned earlier, the heterogeneousness-traffic and homogeneousness-input contradiction leads to the limitations of traffic customization and hierarchy unawareness, restricting the competence of existing traffic fingerprinting techniques. To overcome these limitations, we aim to design an *input-agnostic* hierarchical framework for landing deep learning onto traffic fingerprinting. The input-agnostic objective is to make our framework receive traffic data in any heterogeneous formats, and still process that data effectively. The hierarchical objective is to design specialized built-in mechanisms for deep learning to characterize traffic hierarchy to perform in-depth fingerprinting of high performance.

3 Framework Design

Figure 1 shows the overall design of the input-agnostic hierarchical deep learning framework for traffic fingerprinting. The proposed framework, fed with a trace X, which is a mixture of multiple flows (e.g., sequences of TCP/UDP packets sharing the same communicating IPs and ports), aims to predict the probabilities of X belonging to all possible labels.

Intuition. To support the heterogeneous input and perform final prediction, the framework is designed hierarchically to work at three layers, i.e., packet, flow, and trace layers, in a bottom-up manner. This design not only provides the coarse-to-fine grained cross-layer feature learning mechanism but also enables the bottom-up gradual feature abstraction of the lower layer into the upper layer. Moreover, the lowest layer (i.e., the packet layer) straightforwardly docks all individual packets into the framework without traffic tailoring (i.e., no packets added or removed), and the highest layer (i.e., the trace layer) is seamlessly connected with existing neural networks for classification. Compared to directly constructing a sequence of homogeneous vectors to represent multi-layer features, our design captures the original shapes of the heterogeneous traces and hence eliminates traffic customization.

Our framework consists of four modules: packet vectorization, packet-to-flow mapping, flow-to-trace mapping, and trace-to-label classification. The first two modules work at the packet and flow layers, respectively, and the last two are at the trace layer. Next, we detail the design of the modules in Figure 1.

3.1 Packet Vectorization (M0)

Packet vectorization is to convert each packet into a packet vector (PV). It is typically performed *straightforwardly* using the metadata of a packet. For example, packet timing and sizes could be incorporated into the packet vector since they are the most commonly available information when confronted with encrypted traffic. After being processed by packet vectorization, the sequence of packets of the same flow becomes the sequence of packet vectors at the packet layer.

As the starting point of original traffic processing, packet vectorization is typically performed manually using task-specific metadata of a packet. Packet metadata from informative fields of packets, such as the flags in the packet header, could be extracted according to the specific tasks. Note that packet vectorization is by no means equivalent to feature extraction. In fact, it is much simpler. The reason is that it is just a formal representation of a packet using as much (potentially informative) packet metadata as possible in the form of a vector. In contrast, feature extraction is the process of crafting distinguishing features for raw traffic traces, including but not limited to attributes, characteristics, and temporal-spatial correlation, using domain knowledge and experiences.

In the traffic fingerprinting task under our investigation, we incorporate packet timing and sizes into the packet vector since they are the most *commonly* available information when confronted with encrypted traffic. Let $p_1(t_1,s_1), p_1(t_2,s_2), \ldots, p_k(t_k,s_k)$ be the sequence of all k packets belonging to a flow, where p_i denotes the ith packet with a timestamp of t_i and a size of s_i ($i = 1,2,\ldots$). Formally, a packet p_i is represented as a packet vector PV_i as follows:

$$PV_i = < s_i, (t_i - t_0), (t_i - t_{i-1}), i, i/k >, \qquad (1)$$

where $(t_i - t_0)$ and $(t_i - t_{i-1})$ calculate the time interval between packets p_i and p_0, and the time interval between packets p_i and p_{i-1}, respectively. We define $t_0 = t_1$ so that the first packet and its preceding packet have no time interval and i/k is the relative position of p_i in the flow.

3.2 Packet-to-flow Mapping (M1)

To abstract packet-layer information into the flow layer, packet-to-flow mapping takes the sequence of packet vectors (of a flow) as an input and then transforms the input into a flow vector at the flow layer. The sequence of packet vectors could be variable length, while the flow vector is of fixed length. To boost the mapping performance, we first perform packet vector sequence compression. Then, we design four neural network structures to achieve packet-to-flow mapping based on the compressed packet vector sequences.

3.2.1 Packet Vector Sequence Compression

Because it is computationally expensive for the neural networks responsible for packet-to-flow mapping in § 3.2.2 (e.g., the chain-structured recurrent neural network) to handle long sequences of packet vectors, we use Convolutional Neural Network (CNN) to compress a long sequence of packet vectors into a short sequence of *long* vectors. We term such long vectors as *Compressed Packet Vectors (CPVs)*. One benefit of the compression is the linear reduction in computational overhead since calculation in packet-to-flow mapping is proportional to the number of packet vectors. In addition, adding multi-convolution layers could increase the fitting ability of the neural network model and may reduce training epochs.

The extent to which a long sequence of packet vectors can be converted into CPVs depends on the multiplication of all pooling layers' stride sizes. For example, given a flow containing 48 packets, if no compression is used, the input to the neural network structure is a sequence of 48 packet vectors; on the contrary, if three convolutional layers, each followed by a pooling layer with stride size 2, are added before the neural network structure to allow compression, the input becomes a sequence of six CPVs (i.e., only 1/8 of 48). If the given flow contains less than eight packets, zero paddings can reshape the flow into one with eight packets. We want to point out that packet vector sequence compression lowers computation overhead and also inherits CNN's ability to capture local features.

3.2.2 Neural Network Structure Construction

To transform a sequence of packet vectors into a flow vector, we construct four neural network structures of mainstream artifact categories, including the *chain-structured* recurrent neural network, the *tree-structured* recurrent neural network using a balanced binary tree, the scaled dot-product *attention-structured* neural network, and the hybrid neural network. The major differences among the first three structures are their iterative, parallel, and selective abstraction mechanisms (i.e., the order of hierarchically abstracting a sequence of packet vectors), respectively. The hybrid neural network combines neural networks with different abstraction mechanisms.

Chain-structured. The chain-structured recurrent neural network processes the input sequentially [21], thereby achieving the *iterative* abstraction mechanism. Figure 4(a) shows an example. In this example, the inputs are a sequence of CPVs (CPV_1, CPV_2, CPV_3) with an initial state h_0, and h_3 is the final fixed-length output flow vector. The goal of the chain-structured recurrent neural network is to capture the sequential dependency of the packet vectors. There are three typical neural networks to fulfill such a goal.

First, classical RNNs can learn any long-term dependency in the input sequence. However, the backpropagation algorithm has the vanishing gradient problem. Second, LSTM and Gated Recurrent Unit (GRU) partially mitigate the vanishing gradient problem [22], but the problem fundamentally remains. For example, in Figure 4(a), the influence of PV_1 on the gradient is usually less than PV_3 when using the backpropagation algorithm to optimize the parameters. Worse still, their relative performance competence is controversial. Specifically, studies like [23] show that the performance of GRU outperforms LSTM, while others like [24] are the opposite. Third, BiLSTM, a bi-directional improvement on LSTM [25], works in both directions, i.e., from CPV_1 to CPV_3 and from CPV_3 to CPV_1 in Figure 4(a), while LSTM only works in one direction, i.e., from CPV_1 to CPV_3. The bi-directional improvement can largely alleviate the gradient vanishing problem because the average length of the propagation path is halved. Therefore, we will use BiLSTM to represent the chain-structured recurrent neural network, outputting the sum of the bidirectional hidden states.

Tree-structured. The tree-structured recursive neural network takes leaf nodes (i.e., CPVs) as the input and recursively uses the child node to calculate the parent node value [26]. Because all packet vectors are processed as leaf nodes simultaneously, the tree-structured network can be considered the *parallel* abstraction mechanism.

Figure 4(b) exemplifies the tree-structured neural network, where the sequence of packet vectors (CPV_1, CPV_2, CPV_3) is the (variable-length) input, and the flow vector h_2 is the (fixed-length) output. We design a tree-structured neural network named Balanced Binary Recursive Neural Network (BBRNN), as shown in Figure 5. Since the height difference between the left and right subtrees of a balanced binary tree is no more than one and both the left and right subtrees are balanced binary trees, the distance from the root node to any leaf node is approximate $log(N)$, where N is the length of the input sequence of packet vectors. This structure can solve the vanishing gradient problem since the balanced tree makes the distance between input and output equal to $O(log(N))$, significantly smaller than that of the chain structure (i.e., $O(N)$).

Attention-structured. Unlike the chain and tree structures that predefine the order of processing CPVs, the attention mechanism can selectively extract essential features from a group of CPVs while ignoring unimportant information. It augments importance by mimicking human attention and adding attention weights to a model as trainable parameters. To use the attention mechanism for traffic fingerprinting, we fine-tune Vaswani's scaled dot-product self-attention [27]. The details can be found in Appendix A.1.

Hybrid Structure. The hybrid neural network connects multiple neural network structures in series or parallel. Neural networks connected in series are connected along a single path, and the input of one neural network is the output of another [28]. Neural networks connected in parallel compute their outputs separately and then combine them [29].

We instantiate the hybrid neural network by connecting the chain-structured and attention-structured neural networks in series for experiments. Specifically, we use BiLSTM and the attention method in (5) to build the hybrid neural network. The input (i.e., CPVs) first passes through the chain-structured neural network, and $y_i (i = 1, 2, \ldots)$ is obtained, as shown in Figure 4(a). Then, the obtained $y_i (i = 1, 2, \ldots)$ is then used as the input (i.e., CPVs) of the attention-structured neural network to calculate the final output.

3.3 Flow-to-trace Mapping (M2)

Having a sequence of flow vectors (of a trace), flow-to-trace mapping serves to derive the trace vector at the trace layer since it essentially abstracts flow-layer information into the trace layer. Again, the sequence of flow vectors could be variable length, while the trace vector is of fixed length. As such, the flow-to-trace mapping module and the packet-to-flow mapping module have similar functions and can use the same neural network structures in § 3.2.2.

3.4 Trace-to-label Classification (M3)

Finally, a trace that is a mixture of a variable number of flows, each consisting of a sequence of a variable number of packets (of variable size), is mapped into a trace vector (of fixed length). Given a trace vector, the trace-to-label classification uses multiple linear layers and the softmax function to predict the probabilities of the trace belonging to different labels. The nodes within the output layer are associated with the labels.

4 Evaluation

We evaluate our framework in five typical traffic fingerprinting tasks. Our evaluation aims to answer three research questions:

RQ1. Can our framework be competent in typical fingerprinting tasks compared to state-of-the-art methods that are based on deep learning or handcrafted features?

RQ2. What benefits can one get from no traffic customization (*RQ2.1*) and being hierarchy awareness (*RQ2.2*)?

RQ3. When our framework is deployed, are there any real-world factors affecting its fingerprinting performance?

4.1 Datasets and Traffic Fingerprinting Tasks

Five datasets, corresponding to five traffic fingerprinting tasks, are prepared. Detailed data characteristics are in Appendix A.3. Below are three public datasets used for user activity fingerprinting, Internet of Things (IoT) device identification, and intrusion detection, respectively.

User Activities (UAV). The dataset, published by Labayen et al. in 2020 [4], contains five different activities (Video, Bulk, Idle, Web, and Interactive), each lasting for 1 to 3 hours. Following Labayen's dataset-splitting method, the traces are divided according to a time window of 5 seconds.

IoT Device Identification (IDI). The dataset, also known as the IoT SENTINEL dataset, was released by Miettinen et al. in 2017 [30]. It has 27 different types of IoT devices, and each device type has about 20 traces. All traces are generated during the device setup procedure.

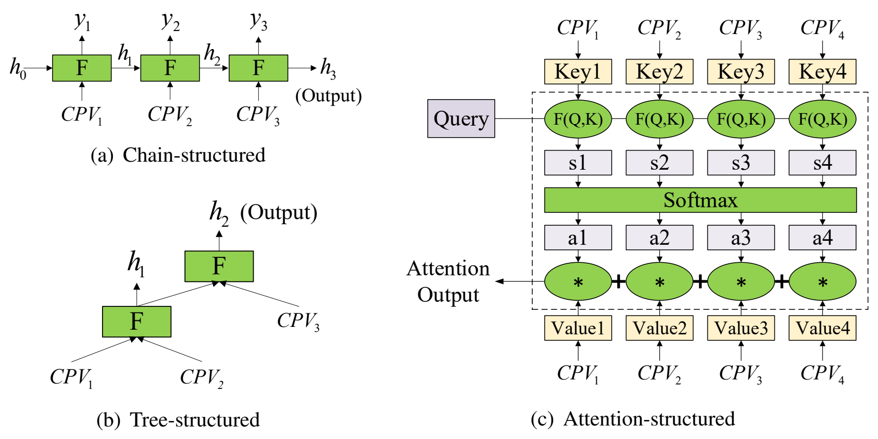


Figure 4: Three neural network structures for packet-to-flow mapping.

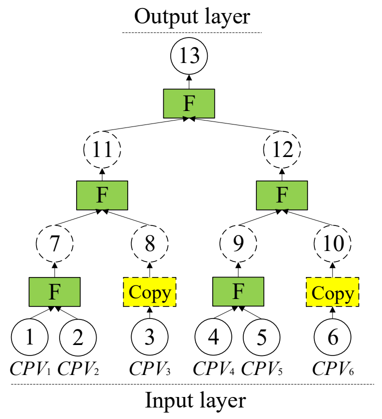


Figure 5: The balanced binary recursive neural network (BBRNN).

Intrusion Detection (ISD). The dataset, also known as the ETF IoT Botnet dataset, released by Jovanovic et al. in 2021 [31], includes both normal and malware traffic. The normal traffic is recorded on a personal computer for several hours, and the malware traffic is captured on Raspberry Pi devices.

Two proprietary datasets below are also collected for the tasks that pose threats to user privacy: Shadowsocks website fingerprinting and keyword searching fingerprinting,

Shadowsocks Website Fingerprinting (SWF). Shadowsocks is a Socks5 protocol-based proxy for Internet surfing. It is popular due to its *lightweight* positioning, high transmission rate (no extra bytes needed in all packets for communication tunnels), and flexibility to define which websites are accessed through the proxy [32]. The dataset, involving 23 monitored websites and more than 3,300 open-world websites from Alexa's top websites, was generated during users accessing websites via the Shadowsocks-proxied Chrome browser from Oct. 26 to Nov. 2, 2020. Each monitored website was accessed 120 times and resulted in 120 traces. The cache was cleaned before each access, and no protection like private browsing or tracking prevention was enabled. Our task is to identify which website the browser is accessing via the Shadowsocks proxy.

Keyword Searching (KWS). The dataset was generated by user searching keywords via the Baidu search app on an Android simulator from Apr. 13 to Apr. 15, 2021. There are 50 keywords, and each is searched 100 times, resulting in 100 HTTPS-encrypted traces. The cache was kept for each access, and no protection was enabled.

4.2 Framework Settings

The framework settings consist of training parameters, neural network structures, CNN compression, and methods for handling overfitting, which should be configured appropriately.

4.2.1 Default Settings

We use the AdamW algorithm to optimize training parameters and follow the default settings in Pytorch v1.7.1, except for the learning rate. To stabilize the training process, we use a warm-up strategy to schedule the learning rate [33]. For all datasets, we use 10-fold cross-validation for training and testing and the Macro F1-score to measure the performance because of its suitability in small and unbalanced samples.

By default, our framework uses CNN compression, a hybrid neural network structure (i.e., Chain-structured and Attention-structured in series) for both packet-to-flow mapping and flow-to-trace mapping modules, and early stop, weight decay, and batch normalization for handling overfitting. Appendix A.5 details the parameter settings of neural network structures.

4.2.2 Speeding up Training

Although the model training could finally converge, the convergence speeds may be highly dispersed across datasets, primarily due to different classification complexities. Figure 6 shows that the training converges at varying speeds for different datasets. Notably, the IDI and KWS datasets converge significantly slower than others.

We employ batch normalization to speed up training. The basic idea is to normalize the hidden layers' inputs by bringing the feature distribution closer to the normal distribution. Batch normalization, though commonly used in deep neural networks [34], is hard to implement on neural networks supporting variable-length input. Therefore, we only add batch normalization to the fully connected layer at the end of the model. We use layer normalization instead to normalize flow vectors [35] (detailed in Appendix A.4). Figure 7 shows that the convergence speed increases by about 2.5 times using batch normalization on the KWS dataset.

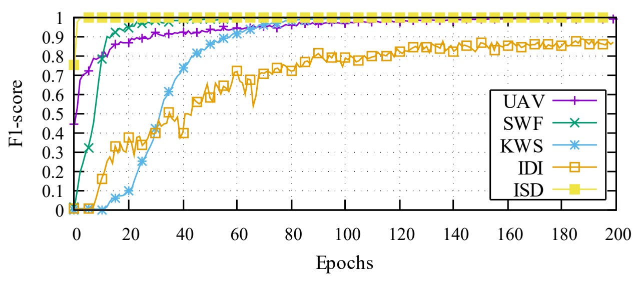


Figure 6: The training set Macro F1-scores over epochs.

4.2.3 Neural Network Structures

Structures in the M1 module. We test four neural network structures in M1. The mean and standard deviation of the Macro F1-scores of different structures are listed in Table 1. We note that all structures have no significant differences.
Structures in the M2 module. We test four neural network structures in M2, and the results are in Table 2. We see that the tree structure performs poorly compared to others. Although the attention structure or the chain structure achieves the highest Macro F1-scores on different datasets, the hybrid structure has the highest average Macro F1-score across datasets.
CNN compression. In § 3.2, we introduce CNN compression to compress packet vectors. To verify the effectiveness, we conduct experiments with and without CNN compression. As shown in Figure 3, the Macro F1-score with CNN compression is similar to that without CNN compression. Consider that CNN compression can significantly save computing resources (reducing 80% to 90% of the time and roughly 50% of memory). Constantly enabling CNN compression would be beneficial.

4.2.4 Handling Overfitting

Directly using the framework may lead to the overfitting problem: good performance on the training set and poor performance on the test set. As demonstrated in Figure 12 (Appendix A.2), the *Macro F1-score* of the training set keeps rising without handling overfitting. Finally, it approaches 0.999 with the increased training epochs, whereas the score of the validation set rises very slowly after about 100 epochs.

We compare six handling overfitting methods under our framework, among which four methods are off-the-shelf, and two methods are adapted from existing ones. The four off-the-shelf methods, including Early Stopping (ES) [36], Weight Decay (WD) [37], Dropout (DO) [38], and Batch Normalization (BN) [39], are detailed in Appendix A.6. Auxiliary Loss (AL) and Data Enhancement (DE) presented below are the two methods that we adapt to fit our framework.
Auxiliary Loss (AL). AL prevents overfitting by providing the model with prior knowledge that helps classification [40]. In our problem, in addition to information on trace training, we also provide the model with hints regarding flow training.

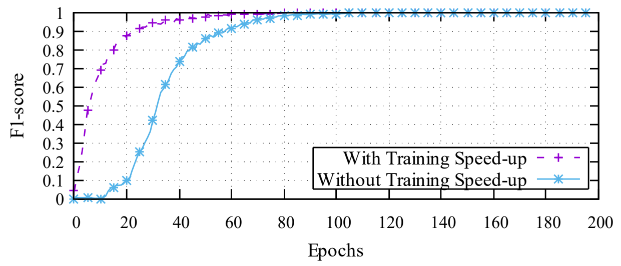


Figure 7: The training set Macro F1-scores over epochs of the KWS dataset with vs. without training speed-up.

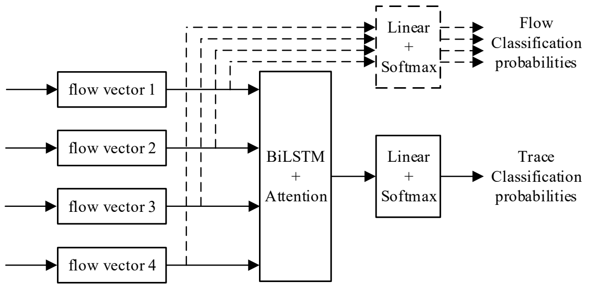


Figure 8: The flow-trace auxiliary loss forward paths.

Such hints are helpful because accurately classifying flows is beneficial for classifying traces. To implement AL, we add new forward paths to the original framework, as shown by the dashed lines in Figure 8. After obtaining flow vectors, we assign a *trace label* to each of them, indicating from which trace the flow originates. Each flow in a trace makes a classification and calculates its loss using the same trace label to which it belongs. Since the number of flows in a trace is uncertain, the loss needs to be normalized to ensure gradient stability. The auxiliary loss for each trace is

$$\text{Loss}=\text{CrossEntropy}(P,\mathcal{L})+\frac{1}{N}\sum_{i=1}^{N}\text{CrossEntropy}(P'_i,\mathcal{L}), \quad (2)$$

where N counts flows, P is the predicted trace label, P'_i is the predicted label of flow i, and $\mathcal{L}$ is the real trace label. It consists of trace classification loss (i.e., $\text{CrossEntropy}(P,\mathcal{L})$) and flow classification loss (i.e, $\text{CrossEntropy}(P'_i,\mathcal{L})$).
Data Enhancement (DE). DE increases the size of a dataset by adding slightly modified dataset copies [41]. Three data enhancement methods are suitable for traffic fingerprinting: cropping, dropping, and noising. Cropping randomly cuts a sub-trace from the original trace, dropping randomly drops packets with a certain probability, and noising randomly adds noises to the timing of packets.

Besides being used separately as pure solutions, the handling overfitting methods can be used in combination, leading to various hybrid solutions. All the methods, except DO, could be mixed as the hybrid solution denoted as "H". DO is excluded because combining DO and BN harms performance due to the influence of variance shift [42]. Based on "H", we remove the method "*" (i.e., the H-* solution) to test

Table 1: The mean and standard deviation of Macro F1-scores using different neural network structures in the M1 module.

M1 module	UAV [4]	SWF	KWS	IDI [30]	ISD [31]
Attention	**0.925** (±0.019)	**0.995** (±0.004)	0.977 (±0.010)	**0.968** (±0.024)	0.990 (±0.010)
Chain	0.922 (±0.029)	0.992 (±0.006)	**0.979** (±0.014)	0.954 (±0.033)	0.995 (±0.008)
Tree	0.923 (±0.023)	0.993 (±0.001)	0.976 (±0.010)	0.937 (±0.037)	0.986 (±0.014)
Hybrid	0.920 (±0.025)	0.993 (±0.003)	0.974 (±0.020)	0.940 (±0.040)	**0.997** (±0.005)

Table 2: The mean and standard deviation of Macro F1-scores using different neural network structures in the M2 module.

M2 module	UAV [4]	SWF	KWS	IDI [30]	ISD [31]
Attention	0.912 (±0.031)	0.991 (±0.003)	**0.979** (± 0.009)	0.924 (±0.055)	0.992 (±0.008)
Chain	**0.920** (±0.027)	**0.994** (±0.004)	0.950 (±0.014)	**0.963** (±0.030)	0.982 (±0.022)
Tree	0.907 (±0.027)	0.987 (±0.006)	0.806 (±0.039)	0.848 (±0.041)	0.992 (±0.012)
Hybrid	0.920 (±0.025)	0.993 (±0.003)	0.974 (±0.020)	0.940 (±0.040)	**0.997** (±0.005)

whether "*" contributes. Table 12 shows that, among all pure solutions, "BN" performs the best. Overall, hybrid solutions achieve better and more stable performance than pure ones, and on average, "H" outperforms other hybrid solutions with one method removed.

Framework Setting Summary. Speeding up training is essential to achieve fast convergence, and constantly enabling CNN compression would be beneficial to both performance and computing resources. For both M1/M2 module design and overfitting handling, hybrid structures and solutions should be adopted for stable and high Macro F1-scores.

4.3 Experiment Results

4.3.1 Our Framework *vs.* State-of-The-Art

To answer RQ1, we conduct fingerprinting tasks using our framework compared to state-of-the-art (SOTA) methods. For all the datasets, the SOTA methods are based on handcrafted features [4, 43–46] (detailed in Appendix A.9). The reason why deep learning is not one of the SOTA methods is that neural networks like CNN and RNN, to their best flexibility, can be fed with a sequence of vectors and cannot handle the heterogeneous input of such datasets. As shown in Figure 1, one trace of the KWF dataset may comprise multiple flows, and each flow has multiple packets.

Table 5 shows the Macro F1-scores across five task-specific datasets using our framework compared to SOTA methods. We see that our method achieves the highest Macro F1-scores for most of the datasets (SWF, KWF, IDI, and ISD), and the top Macro F1-scores are 0.996, 0.977, 0.940, and 0.984 (in bold), respectively. The results demonstrate that our framework not only enables deep learning to be applicable for these datasets (where existing deep learning methods are not) but also is slightly outperforming (or at least comparable to) SOTA methods. Although our framework does not achieve the best performance for the UAV dataset, it remarkably eases the applicability of deep learning in diverse traffic fingerprinting tasks with just one paradigm, reducing the need to perform task-by-task traffic customization. More importantly, it just uses *small-sized* datasets (not in favor of deep learning) to reach the Macro F1-scores of SOTA methods. For example, the KWS dataset only has 5,000 traces belonging to 50 classes, while our framework could still achieve a high Macro F1-score.

It is worth noting that our framework just uses packet timing and sizes to vectorize packets as the input, which by no means indicates that our framework cannot incorporate other types of input. Instead, our framework considers the ubiquitous availability of packet timing and sizes across all (encrypted) traffic fingerprinting tasks and assumes a difficult traffic fingerprinting scenario with minimal information. In practice, one can even use the raw packet information as the input of our framework. For instance, in the packet vectorization process of the IDI dataset, we directly embed the application layer data of each packet into the packet vector. For each byte, we map it to a number from 0 to 255. Before embedding the application layer data, the Macro F1-score is 0.844 and becomes 0.940 after the embedding operation.

Answer to RQ1. Our framework is competent. For the tasks where SOTA methods rely on handcrafted features and deep learning cannot be easily applied, our framework can be directly applicable and accomplish better (or at least comparable) performance only using small-sized datasets that are not in favor of deep learning.

4.3.2 Feature Visualization

To see if our framework creates a meaningful representation, we visualize the datasets using the t-SNE method [47], a nonlinear dimensionality reduction technique that embeds high-dimensional features into a low-dimensional space. We embed the feature vectors of the last layer of our framework into a 2-dimensional space. As shown in Figure 9, each point and its color represent a trace vector and a class, respectively. We see that the points of different classes are scattered, and those of the same class exhibit cohesion, indicating our framework's strong feature representation capability. To observe the information essential for classification, we visualize flow and packet importance using attention values in Appendix A.7, revealing that the neural network tends to pay attention to long flows and the first few packets in a flow.

Table 3: The mean and standard deviation of Macro F1-scores with/without CNN compression.

M1 module	UAV [4]	SWF	KWS	IDI [30]	ISD [31]
+CNN	**0.920** (±0.025)	0.993 (±0.003)	**0.974** (±0.020)	0.940 (±0.040)	**0.997** (±0.005)
-CNN	0.909 (±0.024)	**0.995** (±0.005)	0.972 (±0.012)	**0.976** (±0.024)	0.979 (±0.015)

Table 4: Macro F1-scores using different solutions to handle overfitting. H-* removes method * from the hybrid solution.

Solutions		UAV [4]	SWF	KWS	IDI [30]	ISD [31]
No Handling		0.918 (±0.027)	0.956 (±0.025)	0.204 (±0.275)	0.721 (±0.137)	0.998 (±0.005)
Pure	ES	0.913 (±0.034)	0.953 (±0.024)	0.203 (±0.275)	0.723 (±0.134)	0.995 (±0.010)
	WD	0.922 (±0.022)	0.945 (±0.027)	0.391 (±0.391)	0.720 (±0.066)	0.998 (±0.005)
	DO	0.922 (±0.024)	0.736 (±0.161)	0.052 (±0.132)	0.277 (±0.105)	0.994 (±0.008)
	BN	0.922 (±0.021)	0.994 (±0.003)	0.970 (±0.011)	0.859 (±0.211)	0.992 (±0.011)
	AL	0.919 (±0.023)	0.990 (±0.005)	0.869 (±0.010)	0.833 (±0.070)	0.992 (±0.008)
	DE	0.921 (±0.022)	0.980 (±0.016)	0.140 (±0.261)	0.789 (±0.071)	0.995 (±0.010)
Hy-brid	H-ES	**0.932** (±0.023)	0.996 (±0.002)	0.820 (±0.076)	0.940 (±0.049)	0.998 (±0.005)
	H-WD	0.918 (±0.026)	0.995 (±0.004)	0.974 (±0.007)	0.933 (±0.037)	0.989 (±0.016)
	H-BN	0.917 (±0.024)	0.991 (±0.008)	0.872 (±0.033)	0.848 (±0.080)	0.994 (±0.008)
	H-AL	0.916 (±0.021)	0.993 (±0.003)	0.970 (±0.015)	0.944 (±0.040)	0.995 (±0.007)
	H-DE	0.924 (±0.024)	0.996 (±0.004)	0.973 (±0.005)	**0.958** (±0.026)	0.990 (±0.011)
	H	0.928 (±0.023)	**0.996** (±0.003)	**0.977** (±0.009)	0.935 (±0.041)	**0.998** (±0.004)

Table 5: Performance comparison with the SOTA methods.

Dataset	Our F1-score	SOTA method	SOTA F1-score
UAV [4]	0.928 (±0.023)	K-Means +RF [4]	**0.957**
SWF	**0.996** (±0.003)	RF+LCS [43]	0.982
KWF	**0.977** (±0.009)	PSC+ET [44]	0.974
IDI [30]	**0.940** (±0.040)	RF [45]	0.91
ISD [31]	**0.984** (±0.022)	CCR-ELM [46]	0.961

4.3.3 The Benefits of No Traffic Customization (TC)

To answer RQ2.1, we conduct four experiments: (TC-1) tailoring flows into fixed flow length and using our (original) framework; (TC-2) tailoring flows into fixed flow length and using our framework with the M1 module replaced by 1-dimensional CNN; (TC-3) no tailoring (i.e., unlimited flow length), and using our framework with the M1 module replaced by 1-dimensional traffic scaling and 1-dimensional CNN. (TC-4) no tailoring, and using our framework with the M1 module replaced by 2-dimensional Hilbert curve scaling and 2-dimensional CNN [48].

When tailoring a flow, we choose a flow length, say n, as the input size of the neural network. For the flow with a length exceeding n, we tailor the flow, and only the first n packets are left. After flow tailoring, the packets are vectorized to generate packet vectors. For the flow with less than n packets, we pad zero packet vectors to reach up to n packet vectors. Tailoring flows into fixed-length enables the M1 module (i.e., packet-to-flow mapping) to be replaced by fixed-length input neural networks such as CNN. In the case of no flow tailoring, M1 could be replaced by traffic scaling methods capable of mapping variable-length vector sequences into fixed-length vectors. However, for all the above methods, the layers above M1, such as M2 (flow-to-trace mapping), are reserved to enable hierarchy awareness so that the impact of traffic customization can be measured.

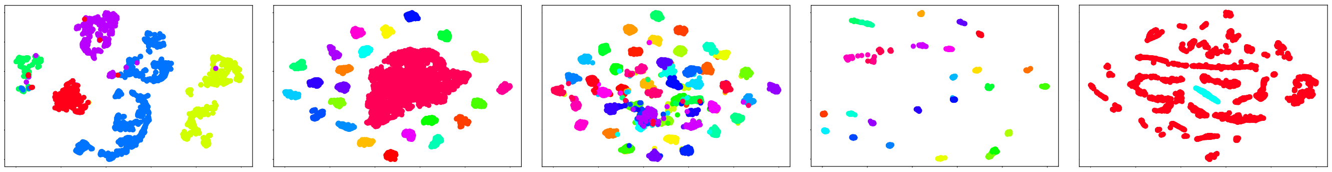
Specifically, TC-1 and TC-2 compare the performance discrepancy between our (original) framework and our framework with M1 replaced by CNN, as the flow length varies. TC-3 and TC-4 use traffic scaling for converting a variable-length packet vector sequence into a fixed-length flow vector. To perform traffic scaling, we employ both 1-dimensional resampling and 2-dimensional Hilbert space-filling curve scaling [48]. The Hilbert space-filling curve can map 1-dimensional data into high-dimensional space, maintain good locality, and transform the sequence of packet vectors of one flow into an image. All the flows are then scaled to be equal-sized (256 vectors). The scaled vectors will be sent to a 1-dimensional/2-dimensional CNN for generating the corresponding flow vector, which will be fed into the flow-to-trace mapping and trace-to-label modules. Note that TC-3 and TC-4 can also be conducted after flow tailoring.

Table 6 shows the Macro F1-scores under different traffic customization methods. For each dataset, we determine its flow tailoring size by computing flow length quantiles. We use the 25th (Q_{25}), the 50th (Q_{50}), the 75th (Q_{75}), the 90th (Q_{90}), and the 95th (Q_{95}) percentile to tailor flows. Below we present observations and insights into all experiments.

TC-1. As the flow length increases, the Macro F1-scores tend to increase for most of the datasets but decrease with the growing variance until the "Unlimited" flow length (i.e., no flow tailoring) for the SWF dataset. We speculate that the decrease in performance may be caused by a large amount of invalid zero padding, which makes the model training unstable.

TC-2. As the flow length increases, the Macro F1-scores tend to increase for the UAV dataset but are unstable for the remaining datasets. The same reason may cause unstable performance as in TC-1. In the absence of traffic customization, TC-2 is no longer applicable. Overall, TC-2 has a lower performance than TC-1.

TC-3. TC-3 can work with and without traffic customization. With traffic customization, TC-3 can achieve comparable or even better performance than TC-1. The results indicate that when traffic customization is enabled, our framework allows the replaceability of the M1 module as needed to achieve



(a) UAV Dataset [4] (b) SWF Dataset (c) KWS Dataset (d) IDI Dataset [30] (e) ISD Dataset [31]

Figure 9: T-SNE visualization using the last layer in our framework. Each color represents one class.

Table 6: Macro F1-scores under different traffic customization methods TC-1, TC-2, TC-3, and TC-4.

Dataset	Flow length (Quantile)	Our original framework (TC-1)	Our framework with M1 replaced (TC-2)	(TC-3)	(TC-4)
UAV	2 (Q_{25})	0.732 (±0.124)	0.826	0.804	0.749
	4 (Q_{50})	0.749 (±0.117)	0.819	0.779	0.815
	23 (Q_{75})	0.840 (±0.075)	0.888	0.838	0.835
	150 (Q_{90})	0.903 (±0.024)	0.848	0.886	0.868
	2283 (Q_{95})	0.908 (±0.028)	0.904	0.879	0.850
	Unlimited	**0.928** (±0.023)	N/A	0.917	0.888
SWF	10 (Q_{25})	0.882 (±0.204)	0.992	0.995	0.995
	14 (Q_{50})	0.858 (±0.200)	0.996	0.996	0.995
	25 (Q_{75})	0.867 (±0.206)	0.993	0.996	0.992
	87 (Q_{90})	0.783 (±0.254)	0.987	**0.997**	0.994
	179 (Q_{95})	0.778 (±0.257)	0.986	0.991	0.995
	Unlimited	0.996 (±0.003)	N/A	0.995	0.992
KWS	4 (Q_{25})	0.839 (±0.055)	0.899	0.909	0.789
	6 (Q_{50})	0.870 (±0.033)	0.834	0.763	0.815
	19 (Q_{75})	0.884 (±0.023)	0.536	0.715	0.931
	49 (Q_{90})	0.903 (±0.050)	0.607	0.805	0.898
	167 (Q_{95})	0.944 (±0.040)	0.570	0.832	0.960
	Unlimited	**0.977** (±0.009)	N/A	0.795	0.937
IDI	2 (Q_{25})	0.943 (±0.044)	0.965	**0.992**	0.842
	10 (Q_{50})	0.899 (±0.068)	0.942	0.940	0.810
	12 (Q_{75})	0.851 (±0.180)	0.925	0.839	0.842
	21 (Q_{90})	0.879 (±0.176)	0.940	0.951	0.857
	37 (Q_{95})	0.928 (±0.047)	0.954	0.900	0.857
	Unlimited	0.940 (±0.040)	N/A	0.964	0.866
ISD	2 (Q_{25})	0.994 (±0.008)	0.994	0.997	0.994
	2 (Q_{50})	0.992 (±0.010)	0.998	0.995	0.998
	4 (Q_{75})	0.992 (±0.010)	0.997	0.994	0.997
	20 (Q_{90})	0.988 (±0.022)	0.995	0.995	0.995
	26 (Q_{95})	0.994 (±0.008)	0.996	0.994	1.000
	Unlimited	0.997 (±0.005)	N/A	**1.000**	0.996

comparable or even better performance.

TC-4. TC-4 can also work with and without traffic customization. Compared with TC-3, TC-4 has a similar performance.

We also find that just using the initial two packets of a flow can achieve a high Macro F1-score, e.g., up to 0.992 for the IDI dataset (attributed to flow correlation of the M2 module since individual flows are incompetent in identifying traces). After observing the original packets, we observe that in the IDI dataset, the first two packets in a flow may contain the plaintext of the device type identification information (the IDI dataset uses TCP payloads for training). Consequently, the importance of the first two packets is highlighted after tailoring, thus improving performance. This observation implies that enabling flow tailoring could be a choice for some tasks.

Answer to RQ2.1. Our framework minimizes tricky traffic customization and retains the flexibility for high-accuracy demanding tasks to achieve the best overall performance with heterogeneous input. Moreover, when input customization is enabled, it could still work and allows the replaceability of the M1/M2 module as needed in specific tasks to achieve comparable or even better performance.

4.3.4 The Benefits of Hierarchy Awareness (HA)

To answer RQ2.2, we conduct two categories of experiments: (HA-1) trace-layer classification, i.e., treating a trace consisting of multiple flows as a whole, without (*HA-1.1*) or with (*HA-1.2*) distinguishing between flows, and classifying the trace into different trace labels; (HA-2) flow-layer classification, i.e., treating each flow of a trace as a sample, and classifying it into different trace labels.

Since existing deep learning methods cannot perform fingerprinting across multiple layers, HA-1 and HA-2 perform fingerprinting at the trace and flow layers. Particularly, HA-1 comprises HA-1.1 and HA-1.1, where the former does not distinguish between flows, and the latter distinguishes by embedding flow IDs and trace labels in packet vectors (i.e., a naive approach of hierarchy awareness). Below, we detail observations and insights into the experiments.

HA-1. The first row of Table 7 shows the Macro F1-scores of HA-1.1. HA-1.1 achieves much poorer performance than our original framework, indicating the importance of distinguishing between flows in traffic fingerprinting. The second row of Table 7 shows the Macro F1-scores of HA-1.2. HA-1.2 significantly outperforms HA-1.1, implying that embedding a priori traffic hierarchy helps greatly. However, its performance is still (much) lower than that of our original framework. Therefore, HA-1.2, as a naive approach, can realize hierarchy awareness to some extent, but a gap exists (i.e., roughly 0.02∼0.05 lower Macro F1-scores).

HA-2. The third row of Table 7 shows Macro F1-scores of HA-2. We observe that HA-2 achieves poor performance. The Macro F1-score ranges between 0.2 and 0.8 across datasets. In particular, the lowest Macro F1-score is achieved on the KWS dataset. The reason is that the traces in this dataset are generated by accessing the same search engine, and flows in different traces are similar. The poor performance of HA-2 verifies that individual flows are incompetent in identifying traces, and flow correlation is vital.

Table 7: Macro F1-scores when confronted with hierarchy unawareness methods HA-1.1, HA-1.2, and HA-2.

Method	UAV [4]	SWF	KWS	IDI [30]	ISD [31]
HA-1.1	0.835 (± 0.024)	0.964 (± 0.012)	0.388 (± 0.365)	0.816 (± 0.072)	0.811 (± 0.057)
HA-1.2	0.906 (± 0.022)	0.966 (± 0.013)	0.927 (± 0.021)	0.840 (± 0.112)	0.914 (± 0.063)
HA-2	0.556 (± 0.025)	0.800 (± 0.022)	0.204 (± 0.012)	0.749 (± 0.018)	0.872 (± 0.029)
Ours	**0.928** (± 0.023)	**0.996** (± 0.003)	**0.977** (± 0.009)	**0.940** (±0.040)	**0.997** (±0.005)

Answer to RQ2.2. Without hierarchy awareness, the performance degradation of deep learning is significant in tasks where traffic hierarchy exists. Experiments using individual flows to identify traces achieve poor performance, indicating the importance of hierarchy awareness.

4.4 Sensitivity to Real-world Factors

Real-world factors, such as dataset size and background noise, may also affect fingerprinting performance.

4.4.1 Dataset Size

Analyzing the impact of dataset size, especially training sample number per class, helps to understand the relationship between dataset size and fingerprinting performance. To demonstrate such a relationship, we select the KWS dataset (the largest number of samples among all balanced datasets and thus easy to tune the training sample number per class) and conduct experiments under different training sample numbers.

We increase the training sample number per class from 2 to 90 and conduct experiments. Figure 10 shows the Macro F1-scores over the training sample number per class. We see that the Macro F1-score increases as the number grows. Our framework eventually approaches a high Macro F1-score of 0.977. Because such performance is achieved with just a small number of samples per class (i.e., 90) based on deep learning, we conclude that there is much room for performance improvement as more training samples are available.

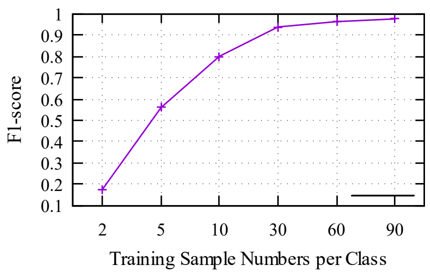


Figure 10: Macro F1-score over training sample number.

4.4.2 Background Noise

Each sample in the dataset may not only contain the pure traffic trace corresponding to a class label but also may be mixed with background traffic noises irrelevant to the class label. To demonstrate the impact of background noises, we collect traffic of 3,300 irrelevant websites as background noises, in addition to the traffic of 23 target websites in the SWF dataset (different traces share the same client and server IPs and server port, and hard to be filtered out using heuristic rules). Each irrelevant website has one traffic trace. Then, we randomly add background noises to training samples. Specifically, we choose a random number n falling within [0,2] and randomly select n irrelevant websites. The traces of the selected n websites will be randomly added to training samples.

Table 8: Macro F1-scores with background noises.

Condition	Our framework	PSC+ET [44]
+ Background Noise	0.943 (± 0.015)	0.880

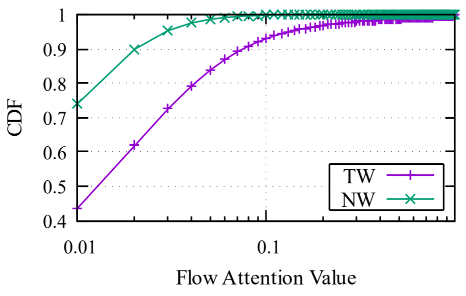


Figure 11: The Cumulative Distribution Function (CDF) of the attention values of TW (target websites) flows and NW (noise websites) flows.

Table 8 presents the Macro F1-score accomplished by our framework using the trace with background noises. We also use the same trace to evaluate another existing method that relies on handcrafted features. The results demonstrate that our framework outperforms this existing method due to a higher Macro F1-score (i.e., 0.943 vs. 0.880), indicating our framework's robustness against background noises.

Our framework can automatically filter out noises because it is expected to set the attention value of noisy flows close to 0. To verify this expectation, we perform a test. Specifically, we divide the flows into two categories. One is the flows belonging to the 23 target websites (TW), and the other is the flows belonging to the noisy websites (NW). We traverse all the traces in the training set and calculate the attention distribution, i.e., the output of the softmax function, as shown in Figure 11. The x-axis uses a log scale for clarity. It can be seen that the attention values in NW are lower than that in TW, which verifies our expectations.

Answer to RQ3. Real-world factors like dataset size and background noise would affect the fingerprinting performance. Although our framework can achieve high performance with a small dataset size, increasing the training sample number is the key to improving performance. Meanwhile, our framework can filter out background noises by assigning low attention values to noisy flows.

5 Discussion

Deep learning would become indispensable in many traffic fingerprinting tasks because the encrypted traffic grows in volume and hides human-observable features for stealthiness, thereby making it too complicated for an expert to handcraft features using traditional machine learning methods. The experiment of framework settings is actually to search hyperparameters over intermediate dimensions, which differs from the choice of input dimension search (i.e., feature design). Specifically, the search over input dimensions is experience-driven, i.e., depending on the expert's understanding of the task. Expert knowledge has a huge impact on performance stability. In contrast, the hyperparameter search is data-driven and independent of the expert's experiences [49].

Like all the other deep learning models, our framework has a bunch of parameters to be configured and fine-tuned. However, all major configurations, like learning rate and neural network depth, can be conducted automatically through hyperparameter search. In addition, as a deep learning-based solution, our framework will also inevitably bring additional computation overhead. In our experience, compared with standard classifiers such as random forests, neural networks usually require hundreds of times more training time. According to different configurations, the training time of our framework on a single GPU varies from several hours to several days. For handling overfitting methods during training, the additional computational time of ES, WD, DO, and BN is almost negligible (less than 3%). According to our experiments, the AL and DE methods require approximately 55% and 78% extra time. Compared with the task-by-task labor cost of manually crafting features, handling different tasks with just one paradigm at the cost of additional training time is acceptable.

As to the neural network structure design, a major criterion is their capability to convert an input sequence into a fixed-length vector. This capability is essential in support of heterogeneous input. Other neural networks with this capability are also applicable to our framework. However, to our best knowledge, there are few such neural networks except those we use. A transformer can be considered a special case of the attention mechanism. We do not use a transformer because it cannot directly convert variable-length input to a fixed-length vector. Another reason is that the original transformer is computationally expensive when there are many packets in one flow because the time complexity of its encoder is $O(n^2)$, where n is the input sequence length. Moreover, we do not compare our framework with other deep learning methods, such as [2], because the various fingerprinting problems under our investigation fundamentally differ from the problem faced by [2], and the models in [2] are not suitable for datasets where one trace contains multiple flows.

6 Limitations

Tailored approaches. The tailored approaches may achieve better results in a few cases, such as in the IDI dataset. For example, when tailoring the flows in § 4.3.3, using the initial two packets for the IDI dataset can achieve a high Macro F1-score up to 0.992, while using all packets may result in a Macro F1-score of 0.964, which is slightly lower. This can be attributed to the fact that the initial few packets of a flow may be much more important than the rest, making most packets noisy to our framework. Specifically, in the IDI dataset, the payload of the initial two packets of a flow may contain plaintext of the device types. However, plaintext features will become less available due to the growing volume of encrypted traffic [50]. Note that, in our experiments, tailoring flows generally leads to performance degradation for encrypted traffic datasets where a packet vector only includes timestamps and packet sizes (i.e., UAV, KWS, and ISD).

Besides tailoring flows, the training pipeline or the individual modules may need to be tailored to adapt to specific tasks. For one thing, for those tasks where each trace contains only one flow, the flow-to-trace module can be removed from the training pipeline. For another, for those tasks where additional packet information, such as payload and metadata in packet headers (e.g., TCP flags, window size, and options [12]), can be exploited, the packet vectorization module can have extended packet vectors beyond timing and sizes.

Inapplicable scenarios. The framework assumes a packet-flow-trace hierarchical traffic structure following the TCP/IP model. Therefore, it may not apply to traffic without this property, such as Zigbee and Bluetooth. If the traffic follows the TCP/IP model, but the hierarchy disappears, the framework still works with diminished advantages. For example, multiple logical flows are transmitted over a single flow in TCP or UDP multiplexing (e.g., QUIC, SSH, Tor, and IPsec). In this example, our framework can still be used as an input-agnostic approach since it can handle variable-length flow input, but the flow-to-trace module no longer contributes. As more concurrent logical flows are multiplexed over a single flow [51], the framework tends to be less effective due to the mutual interference between concurrently multiplexed flows.

In addition, the framework may be less effective for small datasets due to deep learning's data-hungry nature, which is a general challenge. For example, the experiment on the UAV dataset in Table 5 has demonstrated that manually engineered features may outperform the framework for small datasets.

Susceptibility to perturbations. The framework's performance degrades when traffic data is perturbed to deviate from the characteristics of the data on which the framework is trained. For example, packets may be lost, or dummy packets may be added to achieve the deviation [52]. We evaluate the susceptibility of the framework to five types of perturbations, namely, packet loss, flow loss, packet padding, dummy packets, and dummy flows, on the WFP dataset. The definitions and the experimental results are in Appendix A.8.

As shown in Figure 15, when we increase the perturbed rate (i.e., the rate of randomly performing loss, padding, and dummy perturbations) from 0.1 to 0.5, the Macro F1-scores of our framework without retraining (i.e., testing data deviates from training data) drop under all perturbation types. The Macro F1-scores under packet loss and dummy packets drop the most drastically, and those under packet padding decrease the slowest. When the perturbed rate reaches 0.3, the Macro F1-scores of the above perturbations drop to 0.17, 0.70, 0.91, 0.25, and 0.81, respectively. This observation shows that perturbations like packet loss and dummy packets change traffic characteristics significantly and thus lead to drastically decreased performance, while perturbations like packet padding change traffic characteristics and hurt performance the least. It also indicates that randomly losing or adding packets changes traffic characteristics more significantly than just randomly changing packets that already exist (i.e., packet padding). In terms of flow perturbations, randomly losing flows affects traffic characteristics more than randomly adding flows.

Nevertheless, the performance degradation due to perturbations can be mitigated by training the framework with perturbed traffic data. Training the classifiers with perturbed (or defended) data is a commonly adopted paradigm in the literature [3, 53]. This paradigm is based on the availability of perturbed data to the attacker. After retraining the framework with 90% perturbed data and testing on the remaining 10%, the Macro F1-scores under the above perturbations increase to 0.96, 0.90, 0.98, 0.97, and 0.97 when the perturbed rate is 0.3, respectively. We see that the Macro F1-scores are substantially improved under all perturbations, decreasing slowly as the perturbed rate increases. The results imply that the framework, given the availability of perturbed data, exhibits robustness against random perturbations. In particular, dummy packets and flows lead to a relatively slighter decrease in Macro F1-scores than packet loss and flow loss. This is because the framework's attention mechanism can filter out randomly added (rather than lost) packets and flows.

7 Related Work

Feature-based traffic fingerprinting. Leveraging machine learning to classify network traffic has become popular since about 2005. The pioneering work manually crafted features and designed classifiers, such as support vector machines and random forests [44,54,55]. The classifier's performance relies on the quality of crafted features, and extensive efforts have been made to craft distinguishing features [54].

For example, Hayes et al. proposed a system for website fingerprinting on Tor using random forests to extract fingerprints in 2016 [55]. The system has a complex feature set, including packet number statistics, packet ordering statistics, transmission time statistics, etc. Yan et al. consider the scenario of keyword searching fingerprinting by crafting a feature set. [44]. They analyzed factors affecting fingerprinting performance, including client platforms, search engines, feature sets, etc. Ma et al. designed a context-aware system that can fingerprint access to websites using the Shadowsocks proxy [43]. Labayen et al. built a model for classifying user activities using both supervised and unsupervised learning [4].

Despite the effectiveness, crafting features are both time and domain dependent. Manually-crafted features may be invalidated as the data evolves, while they work well in their respective domains, but may perform poorly in other domains.

Deep learning-based traffic fingerprinting. Deep learning has gained growing popularity due to its capability of automatic feature extraction [1–4, 56, 57]. Salient deep learning models suitable for traffic fingerprinting include CNN [1, 3, 10, 16], RNN [2, 10, 58], SAE [2, 16], CapsNet [56], tree structural RNN [59], attention mechanism [58], and so forth.

Many studies have applied deep learning in traffic fingerprinting tasks. For example, Rimmer et al. proposed a website fingerprinting attack over Tor by comparing SDAE, CNN, and LSTM [2]. Liu et al. tested the attention-based bidirectional gated recurrent unit (BiGRU) neural network to identify web services using HTTPS accurately [58]. Cui et al. applied CapsNet in traffic classification and obtained better performance than SAE and CNN [56]. Existing deep learning-based methods, however, do not consider multiple flow correlation [2, 56, 58]. Studies like [60] performed traffic classification considering multiple flow correlation. Nevertheless, these studies rely on manually crafting flow features, thereby not taking full advantage of deep learning. Lu et al. used a graph neural network to classify encrypted traffic [61], but graph creation still requires human involvement.

8 Conclusion

To seamlessly land deep learning onto traffic fingerprinting, we take the first step to designing an input-agnostic hierarchical deep learning framework aware of feature hierarchy. Furthermore, we proposed techniques to handle overfitting and analyzed real-world factors that affect performance. Our framework successfully applies in various fingerprinting tasks where state-of-the-art methods rely on handcrafted features and deep learning is not easily applicable. It achieves better (or at least comparable) performance with just one paradigm.

Acknowledgment

This work was supported in part by the National Natural Science Foundation (62272381, 61972313, 62202405), Natural Science Basic Research Program of Shaanxi (2023-JC-JQ-50), Program of 62602010207, Postdoctoral Science Foundation (2019M663725, 2021T140543), Hong Kong ITF Project (GHP/052/19SZ), Hong Kong RGC Project (No. PolyU15224121), and Fundamental Research Funds for the Central Universities of China. Xiaobo Ma is a Tang Scholar and Siyuan Scholar.

References

[1] S. Bhat, D. Lu, A. Kwon, and S. Devadas, "Var-cnn: A data-efficient website fingerprinting attack based on deep learning," *Proc. PETS*, 2019.

[2] V. Rimmer, D. Preuveneers, M. Juarez, T. Van Goethem, and W. Joosen, "Automated website fingerprinting through deep learning," *arXiv preprint arXiv:1708.06376*, 2017.

[3] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep fingerprinting: Undermining website fingerprinting defenses with deep learning," in *Proc. ACM CCS*, 2018.

[4] V. Labayen, E. Magaña, D. Morató, and M. Izal, "Online classification of user activities using machine learning on network traffic," *Computer Networks*, 2020.

[5] Z. M. Fadlullah, F. Tang, B. Mao, N. Kato, O. Akashi, T. Inoue, and K. Mizutani, "State-of-the-art deep learning: Evolving machine intelligence toward tomorrow's intelligent network traffic control systems," *IEEE Communications Surveys & Tutorials*, 2017.

[6] H. Tahaei, F. Afifi, A. Asemi, F. Zaki, and N. B. Anuar, "The rise of traffic classification in IoT networks: A survey," *Journal of Network and Computer Applications*, 2020.

[7] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Toward effective mobile encrypted traffic classification through deep learning," *Neurocomputing*, 2020.

[8] M. Shen, Y. Liu, S. Chen, L. Zhu, and Y. Zhang, "Webpage fingerprinting using only packet length information," in *Proc. IEEE ICC*, 2019.

[9] M. Di Martino, P. Quax, and W. Lamotte, "Realistically fingerprinting social media webpages in HTTPS traffic," in *Proc. ACM ARES*, 2019.

[10] X. Wang, S. Chen, and J. Su, "App-Net: A Hybrid Neural Network for Encrypted Mobile Traffic Classification," in *Proc. IEEE INFOCOM WKSHPS*, 2020.

[11] J. Li, H. Zhou, S. Wu, X. Luo, T. Wang, X. Zhan, and X. Ma, "FOAP: Fine-Grained Open-World Android App Fingerprinting," in *Proc. USENIX Security*, 2022.

[12] X. Ma, J. Qu, J. Li, J. C. Lui, Z. Li, and X. Guan, "Pinpointing Hidden IoT Devices via Spatial-temporal Traffic Fingerprinting," in *Proc. IEEE INFOCOM*, 2020.

[13] X. Ma, J. Qu, J. Li, J. C. S. Lui, Z. Li, W. Liu, and X. Guan, "Inferring hidden iot devices and user interactions via spatial-temporal traffic fingerprinting," *IEEE/ACM Transactions on Networking*, 2021.

[14] R. Perdisci, T. Papastergiou, O. Alrawi, and M. Antonakakis, "IoTFinder: Efficient Large-Scale Identification of IoT Devices via Passive DNS Traffic Analysis," in *IEEE Proc. EuroS&P*, 2020.

[15] Y. Zeng, H. Gu, W. Wei, and Y. Guo, "$Deep-full-range$: A deep learning based network encrypted traffic classification and intrusion detection framework," *IEEE Access*, 2019.

[16] M. Lotfollahi, M. J. Siavoshani, R. S. H. Zade, and M. Saberian, "Deep packet: A novel approach for encrypted traffic classification using deep learning," *Soft Computing*, 2020.

[17] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," *arXiv preprint arXiv:1710.09829*, 2017.

[18] R. Girshick, "Fast r-cnn," in *Proc. IEEE ICCV*, 2015.

[19] L. Vu, H. V. Thuy, Q. U. Nguyen, T. N. Ngoc, D. N. Nguyen, D. T. Hoang, and E. Dutkiewicz, "Time series analysis for encrypted traffic classification: A deep learning approach," in *Proc. IEEE ISCIT*, 2018.

[20] E. Papadogiannaki and S. Ioannidis, "A survey on encrypted network traffic analysis applications, techniques, and countermeasures," *ACM Computing Surveys*, vol. 54, no. 6, 2021.

[21] C. Liu, L. He, G. Xiong, Z. Cao, and Z. Li, "Fs-net: A flow sequence network for encrypted traffic classification," in *Proc. IEEE INFOCOM*, 2019.

[22] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[23] R. Fu, Z. Zhang, and L. Li, "Using LSTM and GRU neural network methods for traffic flow prediction," in *Proc. IEEE YAC*, 2016.

[24] K. Irie, Z. Tüske, T. Alkhouli, R. Schlüter, H. Ney *et al.*, "LSTM, GRU, highway and a bit of attention: an empirical overview for language modeling in speech recognition," in *Proc. INTERSPEECH*, 2016.

[25] T. Chen, R. Xu, Y. He, and X. Wang, "Improving sentiment analysis via sentence type classification using BiLSTM-CRF and CNN," *Expert Systems with Applications*, 2017.

[26] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint*

arXiv:1706.03762, 2017.

[28] P. Fu, C. Liu, Q. Yang, Z. Li, G. Gou, G. Xiong, and Z. Li, "NSA-Net: A NetFlow Sequence Attention Network for Virtual Private Network Traffic Detection," in *Proc. Springer WISE*, 2020.

[29] L. Wu, C. Kong, X. Hao, and W. Chen, "A short-term load forecasting method based on GRU-CNN hybrid neural network model," *Mathematical Problems in Engineering*, 2020.

[30] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A.-R. Sadeghi, and S. Tarkoma, "IoT sentinel: Automated device-type identification for security enforcement in iot," in *Proc. IEEE ICDCS*, 2017.

[31] D. Jovanovic and P. Vuletic, "ETF IoT Botnet Dataset," Mendeley Data, V1, 2021, doi: 10.17632/nbs66kvx6n.1.

[32] "Shadowsocks or vpns – which is best for you and why," https://www.wizcase.com/blog/pros-cons-of-shadowsocks-and-vpns/, accessed Feb., 2023.

[33] J. Ma and D. Yarats, "On the adequacy of untuned warmup for adaptive optimization," in *Proc. AAAI*, 2021.

[34] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. ICML*, 2015.

[35] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[36] L. Prechelt, "Automatic early stopping using cross validation: quantifying the criteria," *Neural networks*, 1998.

[37] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[38] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, 2014.

[39] J. Bjorck, C. Gomes, B. Selman, and K. Q. Weinberger, "Understanding batch normalization," *arXiv preprint arXiv:1806.02375*, 2018.

[40] S. Ruder, "An overview of multi-task learning in deep neural networks," *arXiv preprint arXiv:1706.05098*, 2017.

[41] Y. Zhang, X. Sun, X. Qin, C. Li, S. Wang, and Y. Xie, "Tripod: Use Data Augmentation to Enhance Website Fingerprinting," in *Proc. IEEE ISCC*, 2021.

[42] X. Li, S. Chen, X. Hu, and J. Yang, "Understanding the disharmony between dropout and batch normalization by variance shift," in *Proc. IEEE/CVF CVPR*, 2019.

[43] X. Ma, M. Shi, B. An, J. Li, D. X. Luo, J. Zhang, and X. Guan, "Context-aware Website Fingerprinting over Encrypted Proxies," in *Proc. IEEE INFOCOM*, 2021.

[44] J. Yan, H. F. Alan, and J. Kaur, "Fingerprinting Search Keywords over HTTPS at Scale," *arXiv preprint arXiv:2008.08161*, 2020.

[45] S. A. Hamad, W. E. Zhang, Q. Z. Sheng, and S. Nepal, "IoT device identification via network-flow based fingerprinting and learning," in *Proc. IEEE TrustCom/BigDataSE*, 2019.

[46] N. Hasan, Z. Chen, C. Zhao, Y. Zhu, and C. Liu, "IoT Botnet Detection framework from Network Behavior based on Extreme Learning Machine," in *Proc. IEEE INFOCOM WKSHPS*, 2022.

[47] L. Van der Maaten and G. Hinton, "Visualizing data using t-SNE." *Journal of Machine Learning Research*, 2008.

[48] G. Bendiab, S. Shiaeles, A. Alruban, and N. Kolokotronis, "IoT Malware Network Traffic Classification using Visual Representation and Deep Learning," in *Proc. IEEE NetSoft*, 2020.

[49] A. Zela, A. Klein, S. Falkner, and F. Hutter, "Towards automated deep learning: Efficient joint neural architecture and hyperparameter search," *arXiv preprint arXiv:1807.06906*, 2018.

[50] E. Papadogiannaki and S. Ioannidis, "A survey on encrypted network traffic analysis applications, techniques, and countermeasures," *ACM Computing Surveys (CSUR)*, 2021.

[51] Q. Yin, Z. Liu, Q. Li, T. Wang, Q. Wang, C. Shen, and Y. Xu, "Automated Multi-Tab Website Fingerprinting Attack," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[52] T. Wang, I. Goldberg *et al.*, "Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks." in *Proc. USENIX Security*, 2017.

[53] P. Sirinam, N. Mathews, M. S. Rahman, and M. Wright, "Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning," in *Proc. ACM CCS*, 2019.

[54] M. Shafiq, X. Yu, A. K. Bashir, H. N. Chaudhry, and D. Wang, "A machine learning approach for feature selection traffic classification using security analysis," *The Journal of Supercomputing*, 2018.

[55] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique," in *Proc. USENIX Security*, 2016.

[56] S. Cui, B. Jiang, Z. Cai, Z. Lu, S. Liu, and J. Liu, "A Session-Packets-Based encrypted traffic classification using capsule neural networks," in *Proc. IEEE HPCC/SmartCity/DSS*, 2019.

[57] J. Hyland, C. Schneggenburger, N. Lim, J. Ruud, N. Mathews, and M. Wright, "What a SHAME: Smart Assistant Voice Command Fingerprinting Utilizing Deep Learning," in *Proc. ACM WPES*, 2021.

[58] X. Liu, J. You, Y. Wu, T. Li, L. Li, Z. Zhang, and J. Ge, "Attention-based bidirectional GRU networks for efficient HTTPS traffic classification," *Information Sciences*, 2020.

[59] X. Ren, H. Gu, and W. Wei, "Tree-RNN: Tree structural recurrent neural network for network traffic classification," *Expert Systems with Applications*, 2021.

[60] M. Zhu, K. Ye, Y. Wang, and C.-Z. Xu, "A deep learning approach for network anomaly detection based on AMF-LSTM," in *Proc. Springer NPC*, 2018.

[61] J. Lu, G. Gou, M. Su, D. Song, C. Liu, C. Yang, and Y. Guan, "GAP-WF: Graph Attention Pooling Network for Fine-grained SSL/TLS Website Fingerprinting," in *Proc. IEEE IJCNN*, 2021.

[62] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine Learning Springer*, 2006.

A Appendix

A.1 Attention Mechanism in Our Framework

Figure 4(c) shows the principle of the attention mechanism, which has three types of inputs: query, key, and value. Query and key are used to calculate attention weights. Then, a weighted sum of the values, i.e., the attention value, is derived:

$$\text{Attention}(Query, Key, Value) = \sum_{i=1}^{Lx} a_i * Value_i, \tag{3}$$

where Lx is the number of values, and a_i is the similarity between the query (i.e., $Query$) and the ith key (i.e., Key_i).

$$a_i = \text{Similarity}(Query, Key_i). \tag{4}$$

To use the attention mechanism for traffic fingerprinting, we fine-tune Vaswani's scaled dot-product self-attention [27]:

$$\begin{cases} \text{Attention} = \text{Softmax}(\frac{QK^T}{\sqrt{d_k}})V, \\ Q = W^Q, \\ K = XW^K, \\ V = XW^V. \end{cases} \tag{5}$$

X is a matrix containing CPVs. Assuming there are 10 CPVs after compression, and the dimension of each vector is 100, then X is a matrix of size 10×100. W^Q, W^K, and W^V are three parameter matrices with fixed sizes for training. Q, K, and V correspond to $Query$, Key, and $Value$ in (3). The scaling factor d_k stabilizes the gradient and equals the length of K. The difference between Vaswani's self-attention and our method is that the size of Q in our method is a constant. This significantly reduces the computational complexity from $O(n^2)$ to $O(n)$, where n is the number of rows of X.

A.2 Without handling overfitting

Figure 12 shows a case without handling overfitting.

A.3 Dataset Characteristics

Table 9 details dataset characteristics, including the number of classes, dataset sizes (i.e., number of traces), etc.

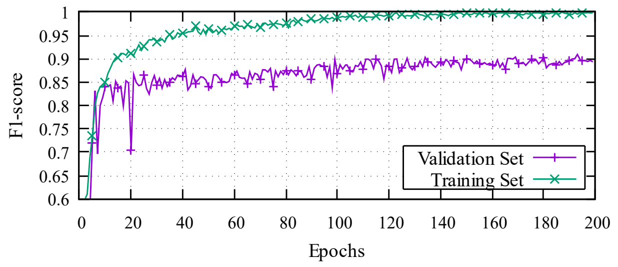


Figure 12: The Macro F1-scores of training/validation sets over epochs on the UAV dataset without handling overfitting.

A.4 Batch Normalization in Our Framework

The formal expression of batch normalization [34] is:

$$\begin{aligned} \mu_B &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i, & \sigma_B^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2, \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}, & y_i &\leftarrow \gamma\hat{x}_i + \beta. \end{aligned} \tag{6}$$

It first calculates the mean and variance of input variable x_i, $i \in [1, m]$, where m represents the number of input variables. Then, it normalizes x_i to $\hat{x}_i$ by subtracting the mean and dividing by the variance. Finally, it re-centers and re-scales $\hat{x}_i$ using learnable parameters γ and β, which enable the model to determine whether to use normalization. Unfortunately, batch normalization is hard for neural networks supporting variable-length input. Therefore, we only add batch normalization to the fully connected layer at the end of the model and use layer normalization instead to normalize flow vectors [35].

A.5 Module Parameter Settings

We set parameters for each module of our framework. The packet vectorization module is performed straightforwardly and has no parameters for setting. The parameters of the packet-to-flow mapping, flow-to-trace mapping, and trace-to-label classification modules are detailed in Table 10.

Packet-to-flow Mapping. This module is set up with five neural networks. The first one indexed by ⓪ is a classical convolutional neural network responsible for packet vector sequence compression. It has three convolutional layers, each followed by a max-pooling function. The remaining four ones correspond to the attention-based, chain-structured, tree-structured, and hybrid neural networks, indexed by ①, ②, ③, and ④, respectively, for different ways to perform packet-to-flow mapping and extract flow vectors.

Specifically, ① contains one attention and two convolutional layers. The two convolutional layers compress the attention output. ② uses a three-layer BiLSTM neural network and adds the bidirectional hidden vectors of the third layer as the output. ③ uses three linear layers as a node merging function of BBRNN and outputs the root node value as the flow vector. ④ contains a BiLSTM layer, an attention layer, and two convolutional layers.

Table 9: The characteristics of the five datasets in our experiments.

Dataset	Classes	Traces	Flow number of one trace				Flow length			
			Average	Minimum	Maximum	Std. Deviation	Average	Minimum	Maximum	Std. Deviation
UAV	4	3,705	5.02	0	266	16.57	554.13	1	50,332	2984.48
SWF	24	6,088	43.51	1	891	69.81	66.66	1	57,261	443.49
KWS	50	5,000	31.30	6	68	8.29	36.39	1	8,378	180.74
IDI	27	550	27.34	1	215	44.58	12.20	1	297	25.75
ISD	2	4402	10.47	0	578	22.7	166.42	1	2,252,416	12,831.47

Table 10: Parameter settings of neural networks of different modules in our framework.

Module ID	Function	Neural Networks	Network Layers	Output Size
M1&M2	Packet Vector Sequence Compression	⓪ CNN	Conv1d(in_channels=3,out_channels=10,kernel=5)	$n \times 10$
			Maxpool1d(kernel=3, stride=3,padding=1)	$\approx \text{int}(n/3) \times 10$
			Conv1d(in_c=10,out_c=50,kernel=5)	$\approx \text{int}(n/3) \times 50$
			Maxpool1d(kernel=3, stride=3,padding=1)	$\approx \text{int}(n/9) \times 50$
			Conv1d(in_channels=50,out_channels=100,kernel=5)	$\approx \text{int}(n/9) \times 100$
			Maxpool1d(kernel=3, stride=3,padding=1)	$\approx \text{int}(n/27) \times 100$
	Packet-to-flow Mapping or Flow-to-trace Mapping	① Attention	Eqn. (5): size(Q) $= 50 \times 100$	50×100
			Conv1d(in_channels=50,out_channels=10,kernel=1)	10×100
			Conv1d(in_channels=10,out_channels=1,kernel=1)	100
		② BiLSTM (Chain-structured)	BiLSTM(input_size=100,hidden_size=100,num_layers=3)	100
		③ BBRNN (Tree-structured)	Linear(in=200,out=175)	175
			Linear(in=175,out=150)	150
			Linear(in=150,out=100)	100
		④ BiLSTM + Attention	BiLSTM(input_size=100,hidden_size=100,num_layers=1)	Len(flows)$\times$100
			Eqn. (5): size(Q) $= 50 \times 100$	50×100
			Conv1d(in_channels=50,out_channels=50,kernel=1)	50×100
			Conv1d(in_channels=50,out_channels=10,kernel=1)	10×100
M3	Trace-to-label Classification	⑤ Fully Connection	Linear(in=1000,out=100)	100
			Linear(in=100,out=50)	50
			Linear(in=50,out=50)	50
			Linear(in=50,out=class number)	class number

Flow-to-trace Mapping. This module shares the same neural network structures with packet-to-flow mapping.
Trace-to-label Classification. This module uses a fully connected neural network indexed by ⑤ . It has four linear layers with the leaky relu activation function. The last linear layer does not use an activation function but a softmax function to perform classification.

A.6 Handling Overfitting Methods

Early Stopping (ES). This is a simple yet effective way to prevent overfitting. The basic idea is to establish a validation set, and perform testing based on the validation set to select a suitable epoch to stop training when overfitting occurs [36]. Our stopping criterion is to keep the model with the highest F1 score in the validation set.
Weight Decay (WD). It aims to reduce model complexity by forcing weights to be small and thus reduce overfitting. Researches show that the neural networks with smaller weights are observed to generalize better. Specifically, we will use Pytorch's AdamW algorithm to implement this method [37].
Dropout (DO). It randomly drops neuron units from the neural network during training [38]. This forces the network to learn robust features in random subsets of neurons. If the network makes a prediction, it should not be too sensitive to some specific cues. Even if a specific cue is lost, DO can learn the common features from other cues.

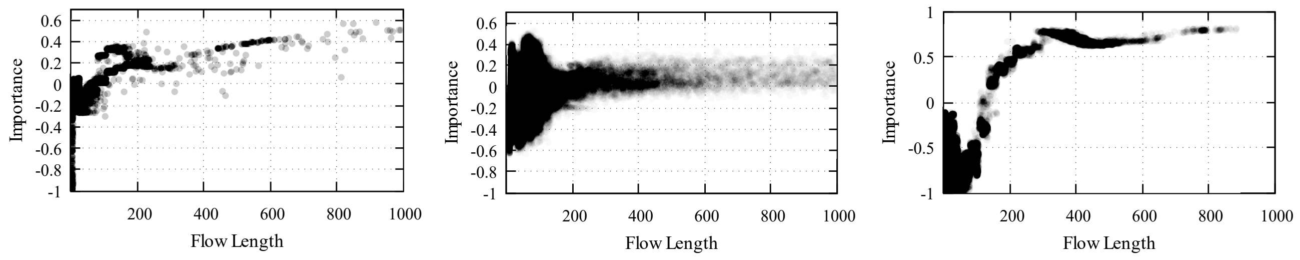
Batch Normalization (BN). It can not only accelerate the training speed but also achieve better generalization [39]. The rationale is that it acts as a regularizer to some extent because the output of one sample is limited by other samples in the mini-batch. It is formally shown in Appendix A.4.

A.7 Visualization of Importance

Figure 13 and Figure 14 visualize flow importance and packet importance using the attention values, respectively. To obtain the pure attention output, we only use the attention neural network in flow-to-trace mapping and packet-to-flow mapping.

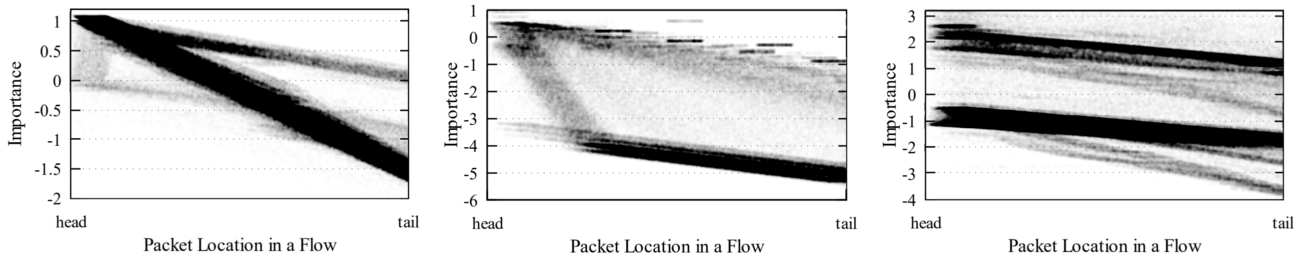
A.8 Susceptibility to Perturbations

Table 11 details each type of perturbation. Figure 15 shows the Macro F1-scores under these perturbations.



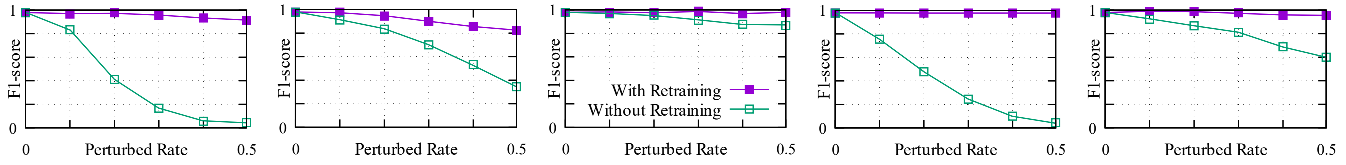
(a) UAV Dataset [4] (b) SWF Dataset (c) KWS Dataset

Figure 13: Scatter plots showing the relationship between the flow length and its importance. The importance value is equal to the flow attention values before the softmax function. Each point represents a flow. The darker the place, the more points.



(a) UAV Dataset [4] (b) SWF Dataset (c) KWS Dataset

Figure 14: Scatter plots showing the relationship between the packet position and its importance. The importance value is equal to the packet attention values before the softmax function. Each point represents a packet.



(a) Packet Loss (b) Flow Loss (c) Packet Padding (d) Dummy Packets (e) Dummy Flows

Figure 15: Macro F1-scores over perturbed rate under different perturbations on the WFP dataset. "Without Retraining" indicates susceptibility to perturbations, while "With Retraining" implies susceptibility can be mitigated if retrained with perturbed data.

Table 11: The description of different types of perturbations.

Perturbation	Description
Packet Loss	randomly dropping packets with a perturbed rate.
Flow Loss	randomly dropping flows with a perturbed rate.
Packet Padding	randomly choosing packets with a perturbed rate and padding their packet sizes to the same.
Dummy Packets	randomly choosing packets with a perturbed rate and adding dummy packets after them. The dummy packets are copies randomly chosen from the dataset.
Dummy Flows	randomly choosing flows with a perturbed rate and adding dummy flows after them. The dummy flows are copies randomly chosen from the dataset.

A.9 Methods in [4, 43–46]

K-Means+RF [4]. Labayen et al. built a three-layer system that combines unsupervised learning and supervised learning for classifying user activities by manually crafting features. The first two layers use K-Means for unsupervised trace clustering, while the last layer uses Random Forest (RF) for classification.

RF+LCS [43]. Ma et al. designed a website fingerprinting method especially for proxy software like Shadowsocks. They hierarchically extracted features from traces, used RF to classify each flow, and then used the Longest Common Subsequence (LCS) algorithm to correlate flows.

PSC+ET [44]. Yan et al. tested five feature sets using Extra-Trees (ET) [62]. We use one of the best feature sets, Packet Size Count (PSC), to make comparisons. PSC counts the frequency of packet size, which has proven to be one of the most informative features for keyword fingerprinting.

RF [45]. Hamad et al. used handcrafted features and different machine learning algorithms. They extracted behavioral and flow-based features from the header and payload of (Ethernet, IP, TCP, and UDP). Random Forest (RF) has the best performance among all the algorithms.

CCR-ELM [46]. Hasan et al. proposed an IoT botnet detection method by combining the state transition matrix and Class-specific Cost Regulation Extreme Learning Machine (CCR-ELM). First, the method extracts features and builds a state transition matrix, and each flow has a state transition matrix. Next, CCR-ELM is used to make a classification.