FINN: Fingerprinting Network Flows using Neural Networks

Fatemeh Rezaei
fatemehr1992@gmail.com
University of Massachusetts Amherst
USA

Amir Houmansadr
amir@cs.umass.edu
University of Massachusetts Amherst
USA

ABSTRACT

Traffic analysis is essential to network security by enabling the correlation of encrypted network flows; in particular, traffic analysis has been used to detect stepping stone attackers and de-anonymize anonymous connections. A modern type of traffic analysis is flow fingerprinting, which works by slightly perturbing network flows to embed secret information into the flows that later can be used for traffic analysis. It is shown that flow fingerprinting enables the use of traffic analysis in a wide range of applications. In this paper, we introduce an effective flow fingerprinting technique by leveraging neural networks. Specifically, our system uses a fully connected network to generate slight perturbations that are then added to the live flows to fingerprint them. We show that our fingerprinting system offers reliable performance in the different network settings, outperforming the state-of-the-art. We also enforce an invisibility constraint in generating our flow fingerprints and use GAN to generate fingerprinting delays with Laplacian distribution to make it similar to natural network jitter. Therefore, we show that our fingerprinted flows are highly indistinguishable from benign network flows.

CCS CONCEPTS

• **Security and privacy → Network security**.

KEYWORDS

Traffic Analysis, Flow Fingerprinting, Neural Networks

ACM Reference Format:
Fatemeh Rezaei and Amir Houmansadr. 2021. FINN: Fingerprinting Network Flows using Neural Networks. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3485832.3488010

1 INTRODUCTION

Linking network flows is an important problem in network security. In particular, network intruders can relay their attack traffic through intermediate devices (e.g., public proxies) in order to evade detection and hide their identities, a problem known as stepping stones [13, 15, 23, 33]. In such settings, network defenders need to be able to link the intruder's ingress and egress network flows (i.e., going into and leaving the stepping stone node) in order to trace back to the intruder. Linking network flows can also be used as a technique to compromise anonymous communication systems like Tor [7] by correlating the traffic patterns of the flows entering and leaving such anonymity systems [6, 13–15, 18, 27, 28, 32].

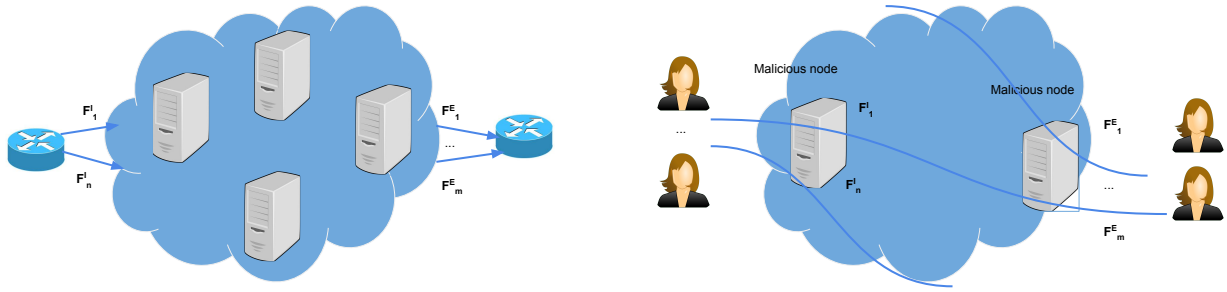
The naive, traditional approach for linking network flows is matching packet contents; however, with the increasing use of encryption on network traffic, packet contents are becoming unusable in linking network flows. Instead, the literature has developed techniques that link network flows based on traffic characteristics [14, 15], such as packet timings and packet sizes, as such features are mainly left intact by encryption; such approaches are broadly referred to as *traffic analysis*. The work on traffic analysis is divided into two categories. Some works focus on analyzing network flows *passively* [6, 8, 10, 20, 26, 30, 31], which requires relatively long flows to perform well. Alternatively, other methods perturb some characteristics of network flows to embed one or more bits of information. This approach is called *active traffic analysis* [13–15, 23, 27, 28, 32]. Active traffic analysis is more efficient and requires shorter flows to link the flows. However, they might introduce large delays to the packets, which may impact the activity of benign users and also potentially give them away [16, 23]. In this work, we focus on active traffic analysis.

The main body of work on active traffic analysis is known as watermarking [14, 15, 24, 27, 29, 32], which attempts to embed a *single* bit of information into a flow. In watermarking, two entities are involved: watermarker and detector. Watermarker receives the flow and inserts a single bit of data, which conveys the information that *whether the flow is watermarked or not.* The detector receives the message on another side of the network and attempts to decode this message. A more recently proposed (and less studied) variant of active traffic analysis is called *flow fingerprinting* [9, 13, 25]. Fingerprinting is used to embed *multiple bits* of information into a flow to convey more complex data such as the origin of the flow. In this work, we present a novel flow fingerprinting technique, as motivated below.

Our flow fingerprinting technique: Previous fingerprinting schemes are *non-blind* [9, 13], which requires the two involved entities to build a secret channel to convey information about the received flows to perform the fingerprinting. However, in practical scenarios, blind schemes are preferable since they reduce communication and computation overhead. In a fingerprinting scenario with n ingress and m egress flows, the blind technique has a computation overhead of $O(m)$ as opposed to $O(nm)$ in a non-blind scenario, which is imposed by correlation of every ingress and egress flows. Moreover, the blind scenario imposes a communication overhead of only $O(1)$ compared to $O(n)$ in non-blind situations. However, designing blind fingerprinting techniques is significantly more difficult as they do not have access to the side-channel available to non-blind schemes.

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3488010



(a) Stepping stone scenario **(b) Anonymity scenario**

Figure 1: Example application scenarios of network flow fingerprints. (F^I_* and F^E_* represent ingress and egress flows).

In this work, we take the first steps towards designing practical *blind* flow fingerprinting techniques by proposing a novel system called FINN. Our work is motivated by DeepCorr [20], which leverages neural networks to perform passive correlation of network flows. FINN inserts fingerprints into flows by slightly delaying the packets while keeping the fingerprints *invisible* (in order to evade detection and avoid interfering benign traffic). FINN uses a DNN, trained on a large corpora of network flows, to decide on how to generate such fingerprinting delays for each target connection. We have simulated FINN to evaluate its performance and we have used statistical measures to evaluate its invisibility. Additionally, we have implemented FINN in real network environments (wireless and cellular networks) in order to validate our simulation results.

In summary, the main contributions of our paper are as follows:

- We design a novel flow fingerprinting system based on neural networks. To the best of our knowledge, we are the first to use deep neural networks for flow fingerprinting. We show that the use of DNNs improves the performance of our fingerprinting substantially compared to prior works.
- We have performed thorough experiments, using both simulations and real-world implementations, to evaluate the performance of our fingerprinting system in various network settings, e.g., different noise environments. We find that our system can embed an average of 0.96 bits of information in every ten packets, which is almost twice the state of the art. We also measure the extraction rate and bit error rate of our system and show that it can reach more than 96% extraction rate and around 2% bit error rate using flows as short as 10 packets to embed 1 bit of data. This is close to twice the state of the art blind fingerprinting, TagIt [25].
- To ensure the invisibility of our fingerprint delays, we design a novel GAN-based technique that generates fingerprinting delays with Laplacian distribution in order to be indistinguishable from natural network noise [21]. We demonstrate the strong invisibility if FINN through extensive evaluations.

The rest of this paper is organized as follows. We review some background on traffic analysis on Section 2 and discuss the threat model. In Section 3, we describe the components of our neural network model. In Section 4, we present the dataset that we use for training our model. We evaluate the performance of our system through simulations and experiments in Section 5 and 6. Finally, we discuss the invisibility of our system in Section 7 and conclude in Section 8.

2 BACKGROUND

In this section, we discuss the threat model of our problem. Also, we talk about passive and active traffic analysis, and mention the pros and cons of each one.

2.1 Threat Model

The problem studied in this paper is flow fingerprinting. Flow fingerprinting has two main applications: deanonymizing Tor and stepping stone detection. A flow fingerprinting system consists of two entities: fingerprinter (encoder) and extractor (decoder). Fingerprinter inserts a message to the network flow, and the decoder tries to extract the inserted message from the flow.

Detecting stepping stone attackers is our main attack scenario. The stepping stone attacker relays her traffic through compromised machines in an enterprise in order to hide her identity. Figure 1a shows this scenario in which the fingerprinter sits on the border routers of an enterprise and inserts a fingerprint to the incoming flows. Then, it attempts to extract the inserted fingerprint from the outgoing flows of the enterprise. Two compromised entities are involved in deploying the attack.

Moreover, Tor [7] tunnels the network traffic through three nodes (entry, middle, and exit node) to enable anonymous communication. In the Tor scenario, the attacker controls some entry and exit nodes. Then, it inserts a message to the incoming flows of the entry node and aims to extract the inserted message in the exit node. Doing this, it links the flows in the entry and exit nodes, and therefore, compromise the anonymity of Tor. For instance, the attacker in Figure 1b can de-anonymize a Tor connection using two compromised guard and exit relays.

2.2 Passive Traffic Analysis

Passive traffic analysis techniques use intrinsic characteristics in the flows such as packet timings, sizes, ordering, direction, etc. to correlate flows. For example, in [33], they divide the flow to on/off

periods and use the timing of these periods to correlate two connections. In [30], authors use the inter-packet-delays to correlate the connections using four different correlation functions. In [5], authors use random walks and computational learning theory to give an upper bound on the number of packets needed to detect stepping stone connections. Also, they study the scenario in which the attacker uses chaffing to avoid detection, and give bounds on amount of needed chaff. Moreover, in [12], authors introduce methods to detect stepping stone when the attacker uses chaffing and delaying packets to avoid detection. They show that the number of chaff packets that their algorithm can tolerate depends on the size of the attacking traffic. Note that they use synthetic and real Internet traces to evaluate their theoretical results. The most recent work in passive traffic analysis is [20], which uses neural networks to link Tor[7] connections.

The biggest advantage of passive traffic analysis is that adversary cannot learn that somebody is listening to the flows to correlate them (invisibility).

2.3 Active Traffic Analysis

The significant disadvantage of the passive approach is that it needs long flows to correlate them. This is not always possible since some connections do not have adequate packets to correlate. For example, an SSH connection in an stepping stone scenario might be too short for this method. Another disadvantage of this technique is its scalability. For instance, in a network with m incoming flows and n outgoing flows, we need $O(m)$ and $O(mn)$ communication and computation overhead, respectively. The active analysis is used to address the weaknesses of the passive approach. Using active traffic analysis, this overhead is reduced to $O(1)$ and $O(m)$ communication and computation, respectively, by inserting hidden messages to the flows. Note that we also require much shorter flows in this method to link the flows. Modifying the packets' timings in the flow by delaying each packet is the main method that is used for active traffic analysis [13–15, 25]. We study the previous work in traffic analysis in two subcategories of watermarking and fingerprinting.

2.3.1 Watermarking. In this technique, a traffic analysis party embeds a single bit of information on the network flow by perturbing some of its characteristics. This bit of data conveys that if a tag exists on the flow or not. As we mentioned earlier, the main body of work on active traffic analysis utilizes the packet timings to embed a secret message to the flow. There are two ways to employ watermarks on the timings: inter-packet-delay based (IPD-based) and interval-based approach.

In the IPD-based approach, the watermark is embedded into the IPDs, which is introduced in [30] for the first time. In RAINBOW [15], authors use IPDs to embed watermarks to the flows in a *non-blind* architecture in which they share the timing information of packets -through a side-channel- with the watermarking parties.

The alternative approach (interval-based) delays the packets into specific time intervals to embed the watermark. Most of the recent work on watermarking uses the latter approach [24, 27, 32] since this method is more resilient to packet-loss and re-ordering compared to the IPD-based approach. Most of the previous interval-based methods are susceptible to multi-flow attacks [16] in which an attacker accumulates multiple watermarked flows and use them to recover the secret key and remove the watermark from the flows. Swirl [14]was designed to be resistant to this attack.

2.3.2 Fingerprinting. As discussed in part 2.3.1, we only add a single bit of information to the flow in watermarking, which conveys the information that the flow is tagged or not. On the other hand, we can embed multiple bits of information to the flows using fingerprinting, which can convey more complex data such as the origin of the flow and identify the fingerprinter. Fingerprinting was introduced in Fancy [13] in which authors use a non-blind IPD-based fingerprinting system using a similar architecture as RAINBOW [15]. In a non-blind technique, the fingerprinter and extractor establish a secret channel to share information about the intercepted flows making the system less scalable. TagIt [25] was introduced as the first blind interval-based fingerprinting system. It used a similar architecture as Swirl [14] for fingerprinting, and employed randomization to avoid multi-flow attacks.

In this paper, we design the first blind IPD-based technique, which uses a much less number of packets to reach the same performance as previously introduced fingerprinting systems. There are two main things to consider when watermarking/fingerprinting a flow:

- Watermark/fingerprint needs to be invisible to the adversary. It should be tough for an adversary to detect whether a flow is watermarked/fingerprinted. Otherwise, it would be trivial to remove the watermark from the flow. To achieve invisibility, such systems should not introduce large delays to packets in the flow.
- Watermark/fingerprint should be robust to network jitters. As we know, jitters in the network would change the timings of the packets, and these systems should be robust to those jitters and detect/extract the watermark/fingerprint in the presence of such noises. The watermark/fingerprint that we embed to the flow should be big enough not to be removed by the network jitters to achieve robustness.

It is evident from the above explanation that robustness and invisibility are the two sides of the same coin. A system cannot have perfect robustness and be entirely invisible simultaneously. We need to play with system parameters to reach an acceptable invisibly and robustness.

3 DESIGN OF FINN FINGERPRINTING SYSTEM

In this section, we design a flow fingerprinting technique using neural networks. Flow fingerprinting is a fundamental problem for enterprises that want to detect compromised machines used as stepping stones to relay cybercriminal's traffic. Our system works by delaying packets in the flow to embed secret fingerprints. Previous methods use statistical approaches [14, 15, 25] for fingerprinting, which requires careful selection of features to manipulate for embedding the fingerprints. We leverage neural networks in our design to avoid the limitation of using a manual process for embedding and extracting fingerprints. Neural networks learn the traffic and extract the complex features from it instead of using carefully engineered features. In designing our system, we follow three main goals:

- **Invisibility.** Introducing small delays to the packets makes the system invisible to the adversary, which has access to the fingerprinted flows and attempts to see any difference in the traffic compared to the regular traffic. We use small fingerprinting delays to the packets. Also, we use a Generative Adversarial Network to generate fingerprinting delays that follow Laplace distribution, which is known to be the distribution of network jitter [22].
- **Robustness.** A robust system can extract the fingerprint from the flows even in the presence of large network jitter. We train our model with network jitter. Our model learns network noise and can de-noise the flows and extract the fingerprints.
- **Scalability.** We use a blind approach to lower the cost of computation and storage to provide scalability. FINN uses a *blind* approach to fingerprint flows. In a blind approach, we do not need to store or share any information about the flow between the fingerprinting entities, which imposes no communication and storage overhead. To extract the fingerprint from flows, we do not need to compare it with all incoming flows (compared to non-blind approaches), which reduces the computation overhead significantly.
  FINN's low cost allows it to be implemented in a real-world setting.
- **Speed.** We want to use a small number of packets for fingerprinting since many flows in real-word do not contain a few packets. Also, this helps in finding the attacker early.

Next, we present the design of FINN, which consists of two main components: a fingerprinter (encoder) and an extractor (decoder). The encoder embeds the secret fingerprint into the flows. The decoder extracts the fingerprint from the flows.

Figure 2 shows the architecture of these two components.

We show the input of the model as following:

$$Input = [F_i, \&_i] \tag{1}$$

Where F_i is the fingerprint that we intend to embed to the flow i, and $\&_i$ is the network noise on the flow i. Note the T_i and $\&_i$ has the size of N, and the F_i has the length of ℓ. F_i is an all zero vector with a single one.

Encoder. is a fully connected network. It takes the F_i and $\&_i$, and passes it to a fully connected network with four hidden layers to generate the fingerprinting delays. This fingerprinting delay is used to delay the packets in the flow i to embed the secret fingerprint of F_i. The fully connected network has layers of size 1000, 2000, 2000, and 1000, and has a output layer of N, which is our fingerprinting delays. The detail description of the layers is represented in Table 2. The fingerprinting delays are added to the vector of inter-packet-delays (IPD),T_i, and $\&_i$ to create the noisy fingerprinted IPD, which would be the input for the decoder.

Decoder. The decoder receives the flow (IPDs) when it passes the network and accumulates the network noise.

Our decoder consists of two parts: convolutional and fully connected. The convolutional part is responsible for de-noising the flow and removing the extra network noise added to the flow. The fully connected part is responsible for decoding the embedded fingerprint.

Table 1: FINN Parameters

Parameter	Definition
N	Number of packets in each flow
ℓ	Length of inserted fingerprint
α	Amplitude of the fingerprint
σ	Standard deviation of noise
η	Ratio of α to σ
K_w	Weight of the decoder-loss to ensure robustness
I_w	Weight of the encoder-loss to ensure invisibility

The convolution layers have a kernel size of 10 and filter sizes of 50 and 10, respectively. The output of the convolution part flattens (flatten layer) to feed the fully connected network. The first fully connected layer's size is 256, and the size of the second fully connected layer is fingerprint length. We use a Softmax function to normalize the output of the decoder. Softmax scales the output between zero and one. Each element of the output vector is a probability that the corresponding element is 1. To get the extracted fingerprint, which is in on One-hot format, we make the largest element one and the rest to be zero. This output is the F_i', which is the extracted fingerprint.

3.1 Training

As we mentioned earlier, FINN has two main components: encoder and decoder. The encoder is responsible for generating the fingerprinting delays for each flow, and the decoder is responsible for extracting the fingerprints from the fingerprinted flows. We define two loss functions: decoder-loss and encoder-loss to control how well each of these tasks work. For the first task, we use mean-absolute-error (MAE) that tries to reduce the error in the fingerprint generation. For the second task, we use categorical-cross-entropy to minimize the error in decoding the embedded fingerprints. The encoder-loss is an MAE loss. Note that we tune the weights of these loss functions (K_w and I_w) to reach required robustness and invisibility. As we increase the K_w, we are giving more weight to reducing the fingerprint extraction error.

In the loss function, n is the size of the training data, K is the number of possible fingerprints, y is a binary indicator that the observation o is of class c (1 or 0), and p is the probability that the observation is of class c. We use an Adam optimizer [17] to minimize the loss.

$$\mathcal{L} = \frac{I_w}{n} |\sum_{i=1}^{n} \hat{I}_i| + \frac{K_w}{n} \sum_{i=1}^{n} \sum_{c=1}^{K} -y_{o,c}^{i} \log p_{o,c} \tag{2}$$

4 EXPERIMENTAL SETUP

In this section, we discuss our dataset, hyperparameter selection, and evaluation metrics.

4.1 Dataset

As we explained in Section 3, FINN consists of two main entities: encoder (fingerprinter) and decoder (extractor). The encoder takes

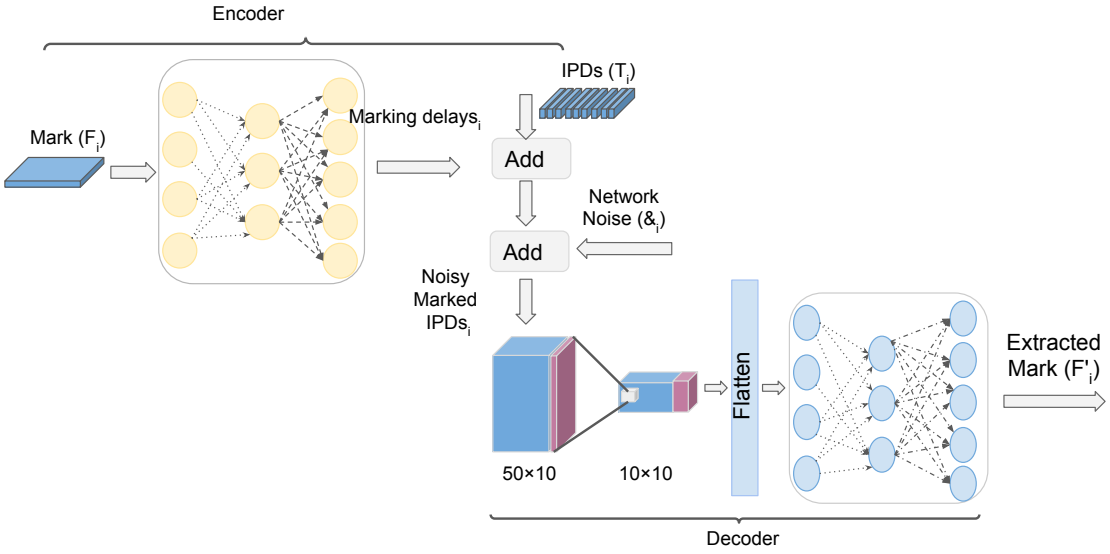


Figure 2: The network architecture of the FINN.

IPDs and fingerprints to generate fingerprinting delays. The decoder receives the fingerprinted IPD, which is generated by adding fingerprinting delays to the IPDs and extract the embedded fingerprint. Note that we have an additional input of network noise to make the robust extraction of fingerprints from the fingerprinted flows possible. To train our model, we need to feed our network with quintuples (IPD, fingerprint, fingerprinting delay, network noise). In the following, we explain the dataset that we use for each of these quintuple components.

IPDs. We use CAIDA's 2016 and 2018 anonymized traces [19] to build our IPD dataset. We use CAIDA because we want to have IPDs of real network traces to simulate the actual network traffic. We extract the flows in this database based on the IP addresses, port numbers, and protocol types of the end-hosts, which is enough to separate the network connections. Note that we build each flow by considering only one side of the traffic between two end-hosts because, in fingerprinting, we only have access to one side of a connection.

Fingerprints. Fingerprints are the messages that we embed in each flow. There are two options to consider for fingerprints: binary or One-hot representation. We employ the one-hot encoding to build our fingerprint dataset. One-hot is a group of K bits that only have a single one. Assume that we want to embed 2 bits of information in each flow. Using a binary representation, we have following options as secret message: 01, 00, 11, 10. For the one-hot representation, we have the following options for the one-hot encoding: $0001, 0010, 0100, 1000$. We choose the second format for our fingerprints since it gives us better performance. It is because we are using categorical-loss as the decoder loss, which works better when its data has a one-hot representation. Note that in our above example for one-hot encoding, the fingerprint length (K) is 4, enabling us to embed $\log_2 K$ bits in each flow.

Fingerprinting Delays. As discussed earlier, FINN is an IPD-based approach, which means it embeds the secret fingerprints into the IPDs. FINN modifies the timings of the packets in the flow in a way to embed the fingerprint into the IPDs. In order to train our model, we need to build fingerprinting delays for many pairs of (flow, fingerprint). Here, we describe how we generate these fingerprinting delays.

Suppose that we have a network flow with the following timings: $f_i = \{t_0, t_1, ...t_n\}$. We compute the inter-packet-delays as : $ipd_i = \{t_1 - t_0, ..., t_n - t_{n-1}\}$ and delays the packets such that the jth element of the ipd$_i$ changes to $ipd_{ij} = ipd_{ij} + \alpha_{ij}$. The fingerprinting delay components are as following:

$$\text{fingerprinting delay}_i = \{\alpha_{i0}, \alpha_{i1}, ..., \alpha_{i(n-1)}\} \tag{3}$$

in which, α_is are chosen from a Laplacian distribution with a standard deviation of α_i. Note that we choose α_is according to the standard deviation of noise in each network connection. To embed the fingerprint into the flow, the fingerprinter delays the jth packet in the ith flow using the following formula:

$$\sum_{n=0}^{n=(j-1)} \alpha_{in} \tag{4}$$

We need to choose α_{i0} large enough to avoid having negative delays on packets. Also, we have to choose α_i as small as possible to avoid fingerprint detection by an adversary. Note that we generate fingerprinting delays for every pair of (flow, fingerprint), as mentioned above. We expect to achieve high invisibility since we are generating all of the α_i randomly for every pair.

Network Noise. Network noise is one of the main inputs of our fingerprinting model. Since network jitter delays the packets and might eventually remove the embedded message, we need to feed it to our model to de-noise it. Also, it is essential to know the jitter since we need to generate the fingerprints according to it. If a

link has high jitter, we need to embed fingerprints with a higher amplitude to resist the network jitter and vice versa. We estimate this jitter by sending several packets through the link and use the standard deviation of jitter as the amplitude of our fingerprint.

4.2 Evaluation Metrics

We use the following metrics to analyze FINN:

- **Extraction Rate (*ER*)**: The ratio of fingerprints that we successfully extract from fingerprinted flows to the number of all fingerprinted flows. This metric shows how many of the fingerprinted flows lost their embedded fingerprint due to the network noise.
- **Bit Error Rate (*BER*)**: Bits of error that occurs for each wrongly extracted fingerprint. We convert each fingerprint to its binary representation to compute bit rate error. This metric shows the number of bits that have been altered due to the fluctuations caused by the network noise.

BER is important when using error correction codes to recover the lost bits. Since we are not using any error correction codes, the BER is the main metric here.

4.3 Choosing FINN's Hyperparameters

Table 1 shows the main parameters of FINN. Here, we select our system's hyperparameters, including I_w, and K_w. I_w is the weight of our encoder loss, and selecting larger I_ws enhances the invisibility of FINN. K_w is the weight of decoder loss, and selecting larger numbers for this parameter improves the performance of FINN. Also, we select our model's learning rate, which controls how quickly a neural network model updates its weight, and therefore learns the problem. Note that we train our model with a standard deviation of noise (σ) in the range of (2, 10) msec. It is because the nodes that we used in our experiments showed a standard deviation of noise in this range. Also, in our experiments, for the value of α, we choose values in this range.

Model parameters. As we discussed in Section 3, our encoder consists of a fully-connected network with three hidden layers. We tried [32, 64, 128, 256] for the size of these layers and learned that [128, 32, 64] works the best on our data. The decoder consists of two convolutional layers and one fully-connected layer. For the window sizes, we tried values in [5, 10, 20, 40, 50, 80, 100], and we learned that [50, 10] works the best. We tried values in [5, 10, 20, 50, 100, 200] for the kernel size and learned that [10, 10] gives us the best performance. Table 2 shows the structure of the encoder and decoder in detail.

Optimum learning rate: This is one of the hyperparameters of each neural network model. It represents the rate at which weights are updated in each iteration. Setting a large number for the learning rate may cause an unstable training process. On the other hand, setting a very small number may result in a long training process. To choose the optimum value for this hyperparameter, we try the values in $[1e-2, 5e-3, 2e-3, 1e-3, 1e-4, 1e-5]$ while fixing the other parameters: number of training data = $500K$, N = 100, and $\ell = 2^{12}$. Through this experiment, we find that $1e-3$ works the best for our model. Therefore, in the following experiments, we use $1e-3$ as the learning rate. Note that we use 5000 flows to evaluate our model in all of the experiments in this section.

Table 2: FINN fingerprint model hyperparameters.

	Layer	Details
Encoder	Fully Connected 1	Size: 1000, Activation: Relu
	Fully Connected 2	Size: 2000, Activation: Relu
	Fully Connected 3	Size: 2000, Activation: Relu
	Fully Connected 4	Size: 500, Activation: Relu
Decoder	Convolution Layer 1	Kernel number: 50 Kernel size: 10 Stride: (1, 1) Activation: Relu
	Convolution Layer 2	Kernel number: 10 Kernel size: 10 Stride: (1, 1) Activation: Relu
	Fully Connected 1	Size: 128, Activation: Relu

Selecting I_w and K_w: Two main things to consider when fingerprinting is robustness and invisibility, and we define two loss functions to control them. We use a mean absolute error for encoder-loss to force the system to avoid generating fingerprinting delays that are too large, resulting in low invisibility. The decoder-loss is a categorical-cross-entropy that minimizes the difference between the extracted and inserted fingerprint to ensure robustness. I_w and K_w are the weights that control the impact of encoder-loss and decoder-loss on the total loss, respectively. We need to choose these two values in a way to achieve optimum invisibility and extraction rate. We try the values in $[1, 5, 50]$ for I_w and values in $[1, 5, 10, 50]$ for K_w while fixing the $\ell = 2^{12}$, $N = 100$. We get the optimum result when we set the pair of (1, 5) for (I_w, K_w). Note that choosing larger values for K_w improves the extraction rate, but at the same time lowers the invisibility. We use the pair $(I_w = 1, K_w = 5)$ for the rest of the experiments.

5 SIMULATIONS

In this section, we run simulations to evaluate the performance of FINN offline. We show the impact of fingerprint length (ℓ) and flow length (N) on the performance. Note that we train our model using the 2018 CAIDA anonymized traces of **equinix-nyc** link and test it using the CAIDA anonymized traces of 2016 **equinix-chicago** link. This ensures that we do not tailor the model for a specific link, and it is transferable to different network conditions.

5.1 Impact of Fingerprint Length (ℓ) on Performance

One of the main parameters of FINN is the fingerprint length (ℓ). To evaluate this parameter's impact on the performance, we fix the other parameters ($N = 100$) and increase ℓ until the extraction rate drops significantly. Figure 4 shows the impact of increasing ℓ and size of training data on the performance of FINN. The figure shows our model's performance for $\ell = 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}$, and 2^{14}. As the figure shows, when the size of training data is $500K$, we have a

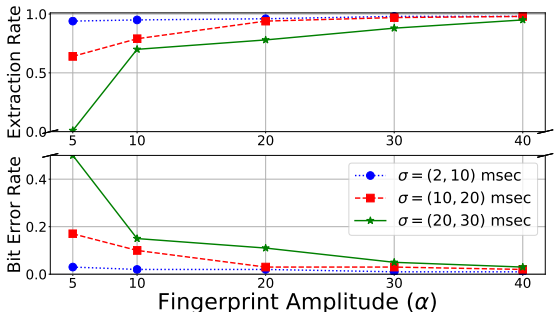


Figure 3: Result of increasing α on performance of FINN fingerprint in different network conditions (flow length = 100, $\ell = 2^{10}$).

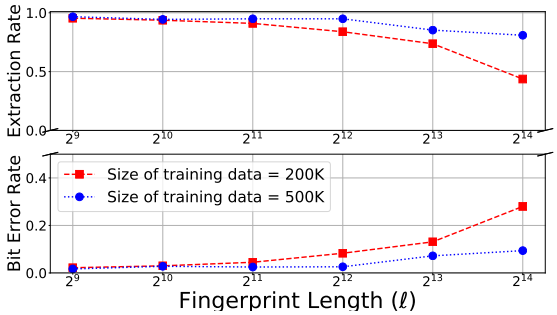


Figure 4: Result of increasing ℓ and number of training data on performance of FINN fingerprint(flow length = 100).

96.6% extraction rate and 1.6% bit error rate when $\ell = 2^9$, and this result degrades to 94.9% and 2.8% when we increase it to 2^{12}.

Also, we get similar results for ℓ of 2^{12}, and the performance of FINN drops when we increase ℓ to 2^{14} (14 bits). Note that although $\ell = 2^{12}$ gives better performance than 2^{12}, we choose 2^{12} since we can send more data on average when using $\ell = 2^{12}$. The following formula computes the number of bits that we can embed on average when the $N = 100$ for $\ell = 2^9$ and 2^{12}. We multiply the extraction rate by the number of bits embedded in the flow to find the number of bits that are correctly extracted from the flows on average.

$$\log_2(2^{12}) \times 0.949 = 11.38$$
$$\log_2(2^9) \times 0.966 = 8.69 \quad (5)$$

Also, as is expected, increasing the size of the training data improves our performance. This is because having more samples in training results in having a more generalized model, and therefore, better performance.

5.2 Impact of Fingerprint Amplitude (α) on Performance

In this experiment, we want to see how α impacts the performance of our model. We fix the other parameters as: $N = 100, \ell = 2^{12}$.

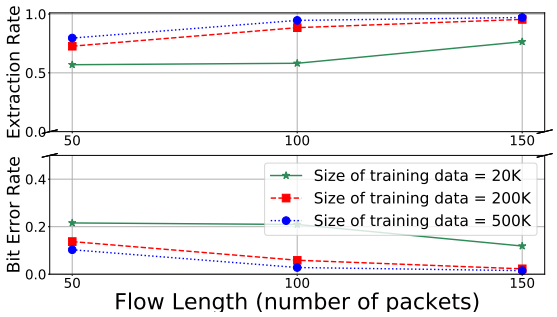


Figure 5: Result of increasing flow length and number of training data on performance of FINN fingerprint (fingerprint length = 2^{10}).

Figure 3 shows the result of this experiment. We train our model with α in the range [5, 10, 20, 30, 40] for three ranges of the standard deviation of the noise. For the $\sigma = (2, 10)$ msec, we have more than 94% extraction rate for all amplitudes. However, as the σ increases, we need to use a fingerprint with a higher amplitude to get the same results.

5.3 Impact of Increasing the Flow Length

Flow length is another main parameters of FINN. As expected, increasing it improves the performance of our system. Our goal is to find the optimum number of packets for fingerprinting when fixing other parameters as following: fingerprint length = 1024, $\sigma \in (2, 10)$. Figure 5 shows the result of our experiment. Note that the number of epochs is 100. As the figure shows, as we increase the flow length, our results improve. For example, when we have $500K$ of training data, the extraction rate for flow length of 50, 100, and 150 are : 79.5%, 94.9%, and 97%, respectively. Also, the bit error rate is 10.2%, 2.8%, and 1.5% for the flow length of 50, 100, and 150, respectively. Moreover, it is evident from the figure that as we increase the size of training data, our results improve. For example, for the flow length of 100, we improve from 88.4% to 94.9% as we increase the size of training data from $200K$ to $500K$.

Additionally, we increase the number of epochs to 200 to evaluate the model's performance with higher epochs and learn that the performance enhances as following: 85.3%, 96.1%, 97.4%, and the bit rate error enhances as following: 7.3%, 1.8% and 1.2% for the flow length of 50, 100, and 150, respectively. Since the model's performance does not change much in higher epochs when we increase the flow length (with the fix parameters) from 100 to 150 (96.1% to 97.4%), we conclude that the flow length of 100 is enough to embed 10 bits (fingerprint length of 1024) of information and achieve good performance.

5.4 FINN as a Watermark

In this experiment, we want to evaluate our algorithm's performance when used as a watermark. Due to the space limitation, we move the results of its simulation and implementation to Appendix 9.1 and 9.3, respectively.

6 REAL-WORLD IMPLEMENTATION

In this section, we implement FINN in real-time to evaluate it on actual network flows. We implement FINN on Ubuntu Linux (version 5.4.0-58-generic) using `iptables` (version 1.8.4), and using `libnetfilter_queue` library (version 1.0.4) [3]. We add rules to the `iptables`'s OUTPUT chain to keep the packets, and our program obtains them for fingerprinting using the `libnetfilter_queue` library. To evaluate the performance of FINN, we set up an encoder on campus and a decoder, which is a digital ocean node [2] located in Bangalore. The goal is to see if FINN works in a real-time setup. Note that the network flows that we use are replayed SSH connections extracted from the CAIDA dataset. To evaluate our system in different network conditions, we replace the Bangalore node with seven Amazon EC2 nodes worldwide, and perform the same experiments. Also, we test FINN on a cellular network to ensure that it works on different network settings.

6.1 Impact of Fingerprint Length on Performance

As we explained in Section 4, we can embed $\log_2(\ell)$ bits in each flow. As expected, increasing the fingerprint length (ℓ) worsens the performance of the system. To evaluate the impact of this parameter on performance, we set the $N = 100$ and σ in range of $(2, 10)$ msec, and choose ℓ from 2^9, 2^{10}, 2^{12} and 2^{14}. Moreover, we use $500K$ as the training size since Section 5 showed that this training size was enough to have a generalized model. Figure 7 shows the result of this experiment for eight different nodes located worldwide. The encoder (fingerprinter) is on a PC on campus, while the decoder locates in seven Amazon EC2 nodes and one digital ocean node in Bangalore. Note that our decoder nodes are in South and North America, Australia, Europe, and Asia. As shown in the figure, we can extract the fingerprints with a high extraction rate for all eight links. Using N of 100, we see that we have more than 96% extraction rate when the fingerprint length is 2^{12} and lower, which degrades to around 75% as we increase the fingerprint length to 2^{14} (14 bits). This result is better than what we got from the simulations. We believe it is because the noise that we faced in real-world experiments was lower than the noise that we introduced in our simulations, which improved the results. we compute the number of bits embedded in a flow as:

$$0.962 \times \log 2^{12} = 11.54 \tag{6}$$

6.2 Experiments on Cellular Network

To evaluate the performance of FINN in a different network setting, we implement it on the cellular network. To do so, we hotspot a cellular phone network with 2.4GHz bandwidth and connect the PC to it (encoder). The decoder is set up on the Bangalore node. Figure 8 shows the result of our experiment for different ℓ for two nodes located in Bangalore and Frankfurt. As the figure shows, we achieve lower extraction rates on these experiments compared to the wireless network in Figure 7. More specifically, for the Bangalore link, we get a 93% extraction rate and a 0.05 bit error rate when the ℓ is 2^{10}. For the Frankfurt link, we gt 87% extraction rate, and 0.1 bit error rate when the $\ell = 2^9$. The reason for having lower performance is that the σ in our links when we connect to the

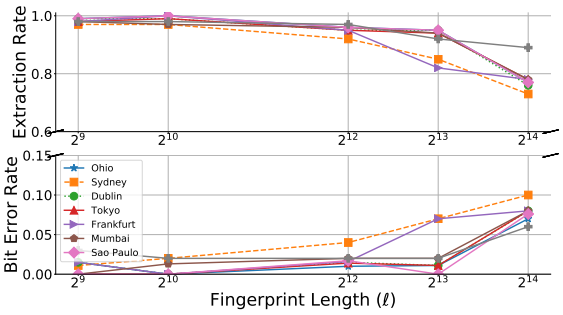


Figure 6: Performance of the real-time FINN fingerprint experiment from campus to various nodes (Wireless, flow length = 100).

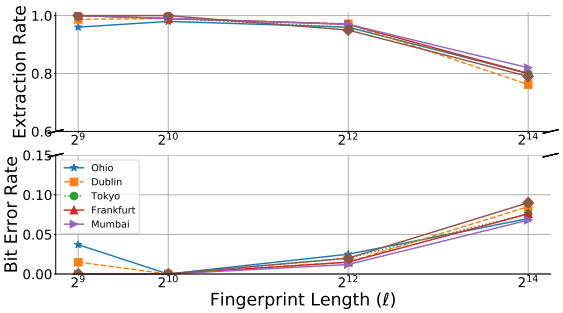


Figure 7: Performance of the real-time FINN fingerprint experiment from Bangalore to various nodes (Wireless, flow length = 100).

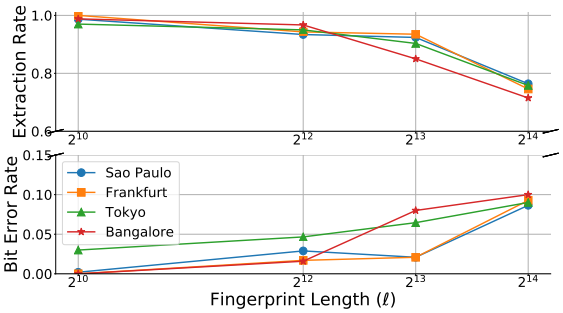


Figure 8: Performance of the real-time FINN fingerprint experiment from campus to various nodes (Cellular, flow length = 100).

cellular network is (7.17, 64.65) msec, which is much larger than the wireless network, which was in the range of (2, 10) msec. Note that the model that we used here was trained on σ in the range of (2, 10) msec. To improve the results, we should train our model with a larger range of σ , which as Figure 5.2 shows, would improve the results.

Table 3: Performance comparison of TagIt and FINN

Method	ER	BER	Average # of bits	# of flows
FINN	96.2	2.1%	11.54	
TagIt (T=1620)	90.1	2.2%	5.32	190
TagIt (T=2160)	96.8	1.4%	4.13	

Table 4: Discriminator model hyperparameters.

Layer	Details
Fully Connected 1	Size: 100, Activation: Relu
Fully Connected 2	Size: 1, Activation: Sigmoid

6.3 Comparing FINN with Previous Methods

We want to compare the performance of FINN with previous fingerprinting systems. TagIt [25] is the most recent blind fingerprinting system that uses an interval-based scheme. It embeds fingerprints on the flows by sending the packets to specific intervals. Each interval, T, is used to embed a single bit of information to the flows. To compare our system's performance to TagIt, we implement it in real-time on our Bangalore link to evaluate its performance in the same condition. The main parameter of TagIt is the interval length, and the packet rate is the main factor to consider when choosing this parameter. We choose 190 flows with packet rates in the range of [5, 25] and select the interval length accordingly. Table 3 shows the result of TagIt for two different interval length. As the table suggests, we can embed 5.3 bits per-flow using TagIt, while this number is 11.54 for FINN. We compute the average number of bits embedded in each flow and learn that FINN can embed 11.54 bits per flow, which is 2.18 of TagIt (5.3 bits per flow). Moreover, fixing the ER for FINN and TagIt at around 96%, TagIt has better Bit Error Rate (BER), which is important in the case we are using error correction codes to recover the lost bits. ER and average number of embedded bits are more important metrics for us since we are not using error correction codes. Comparing the average number of bits embedded in each flow, FINN wins with a large margin (11.54 vs 4.13 in TagIt, which is reaching near 3 times more capacity). Moreover, Fancy [13] is a non-blind fingerprinting approach, therefore, it is not fair to compare it our work. Since non-blind techniques have access to more information, they intuitively offer better performances, but they are hard to deploy in practice.

6.4 FINN's Scalability

We compute the resources required for a stepping stone detection scenario using FINN for the CICS building. The number of graduate employees and staff in CICS is around 450. The only thing that we store is the trained model, which has around 3 MB size. We compute the time it takes to decode the fingerprint from the flows in the worst-case scenario that every person has one connection. Performing the FINN's decoding for N=100 and $\ell = 2^{12}$ for 450 flows on a 3.5 GHz Ubuntu machine with a 32 GB of RAM, takes 1.57 seconds. Note that this time linearly increases as the organization becomes larger. For a larger organization, we might need a Commodity PC. For a vast organization, every sub-network can run its fingerprinting system. Our system runs on a gateway, and the router connects to the gateway to send the IPDs. Then, the gateway runs the encoder model and returns its output, the fingerprinting delay, to the router. The router uses the fingerprinting delay that receives to delay the current packet. The average traffic size and time required for our fingerprinting system is $155KB$ and 6.2 seconds, respectively. Therefore, if we deploy our system on a 10Gbps link, the link will consist 50K number of flows with size 166KB each, which will takes around 43 seconds to correlate the flows using FINN. The time can significantly reduce by performing the computations in parallel.

7 FINGERPRINT INVISIBILITY

A useful fingerprint needs to be invisible to prevent being detected by an adversary and eventually removed. Also, a fingerprint needs to be invisible to not interfere with the activities of benign users. As we discussed earlier, we insert fingerprints by changing the IPDs of the flows. We have to be careful not to add a significant delay, which makes the fingerprint visible and easily removable. To study the invisibility of our system, we use the Kolmogorov–Smirnov test, which has been used in [15, 25] to detect watermarks added to IPDs in a flow.

7.1 Generative Adversarial Network

Generative Adversarial Network is a class of machine learning introduced by Ian Goodfellow et al. [11]. In a GAN framework, two models contest to win a zero-sum game. At the end of the learning, the model learns to generate new data with the same statistics as the training data. A GAN framework consists of two contestants: discriminator and generator. The discriminator gets trained with real and fake data. Real data is the data that we are interested in generating. The fake data is a set of randomly generated data. The discriminator attempts to distinguish between real and fake data,. The generator attempts to generate data similar to the real data to fool the discriminator. GANs have been used to generate real-looking images, human faces,image-to-image translation, text-to-image translation, etc. Here, we use GANs to create Laplace fingerprinting delays. This improves the invisibility of the system since the fingerprinting delays added to the flow get lost in the network jitter imposed on the flows, which also has Laplace distribution [21, 22]. This method has been used in [21] to generate Laplace distribution. The generator and extractor are borrowed from our original model. The discriminator is a fully-connected network with one hidden layer with size of 100. The discriminator uses a binary-crossentropy as its loss function. Figure 5 shows the architecture of FINN while using GAN.

The detail of the architecture is as follows:

- Train the discriminator with Laplace and uniform data.
- Freeze the discriminator and train generator (fingerprinter) to generate Laplace fingerprinting delays.
- Freeze Generator and discriminator. Feed the extractor with the noise and IPDs. Have an Add layer to add IPDs, noise, and the output of generator. Train the extractor.

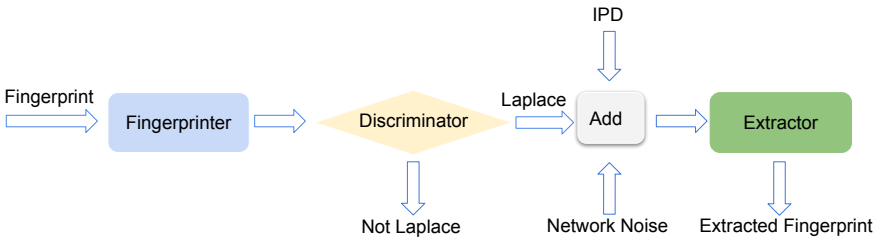


Figure 9: Using GAN to improve invisibility.

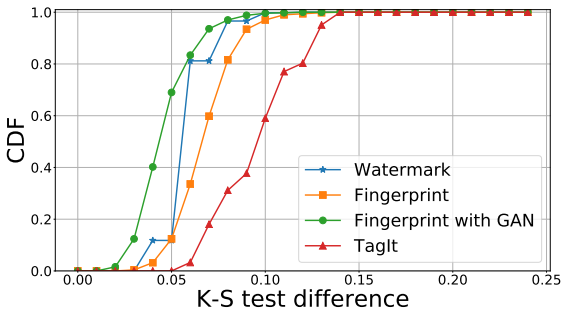


Figure 10: K-S Test Difference for different methods (Fingerprint: flow length = 100, fingerprint length = 2^{10}, Watermark: flow length = 50).

7.2 Kolmogorov–Smirnov Similarity Test

Kolmogorov–Smirnov (**K-S** test) is used to determine if a flow is from a certain distribution, or if two flows belong to the same distribution by measuring the maximum distance between the flows. In the second case, K-S statistics is:

$$D_{n,m} = \sup_x |F_{1,n}(x) - F_{1,m}(x)| \tag{7}$$

$F_{1,n}$ and $F_{1,m}$ are the empirical distribution of the first and second flows. The null hypothesis (two flows are from the same distribution) is rejected at level of α if

$$D_{n,m} > c(\alpha)\sqrt{\frac{n+m}{nm}} \tag{8}$$

In which, n and m are the sizes of two flows and $c(\alpha)$ can be computed as following:

$$c(\alpha) = \sqrt{-0.5 \ln \alpha} \tag{9}$$

We fix our parameters as flow length = 100 and fingerprint length = 1024, and use 500 fingerprinted and non-fingerprinted flows to evaluate the invisibility of our fingerprinting system. We do our investigation on the Bangalore node, where the standard deviation of the noise is 2-10 milliseconds. Figure 10 shows the result of the Kolmogorov-Smirnov test on our data. Using (8), we compute the KS threshold for 0.95 confidence interval, which is 0.19205. As the Figure 10 shows, only 1 of our flows (0.2%) fails to pass the K-S test.

Table 5: Result of clustering the fingerprinted and non-fingerprinted samples

Clustering Algorithm	True Positive	False Positive
GMM	0.63	0.6
K-Means	0.0002	0.0002
DBSCAN	0.90	0.91

7.3 Clustering

A clustering algorithm groups the data so that the samples in the same group are more similar than those in other groups. We use clustering algorithms to see if the fingerprinted and non-fingerprinted samples can be clustered in two groups.We use three prominent clustering algorithms from Python Scikit-learn packages: Distribution-based, Centroid-based, and Density-based clustering.

The distribution-based clustering, samples most likely to belong to the same distribution would be clustered in the same group. This type of clustering generates complex models that captures the correlation and dependence of attributes. The downside of this technique is that there might not exist a concise mathematical model for many real datasets. We use a prominent method that is known as Gaussian mixture models to cluster our dataset. GMM assumes that data consists of a certain number of gaussian distributions. For the centroid-based clustering, we use the K-Means algorithm. K-Means represents each cluster with a single mean vector and has some interesting theoretical properties: it partitions the data space into a structure known as the Voronoi diagram. It can be considered a variation of distribution-based clustering.

DBSCAN is a density-based Spatial clustering that defines the clusters as connected dense regions [1]. In this method, the data in sparse space are considered noise and outlier, therefore, ignored. This clustering algorithm is suitable for the discovery of clusters with arbitrary shapes. We use the inter-packet-delays as the feature vector for the clustering task. The length of the feature vector is 100, which was used in our experiments for fingerprinting. To represent the features in two dimensions, we use the principal component analysis to reduce the dimensionality of the feature space. Figure 11 shows the fingerprinted and non-fingerprinted samples. As the figure shows, fingerprinted and non-fingerprinted flows are not easily separable. However, we use the three mentioned clustering algorithms to see if it is possible to group the fingerprinted and non-fingerprinted samples separately.

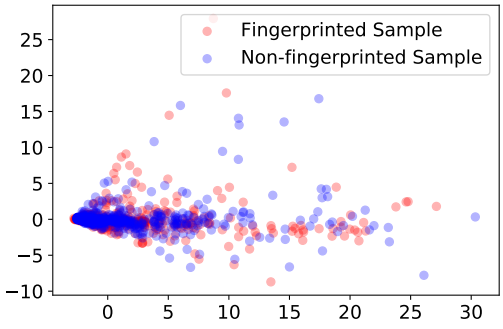


Figure 11: We reduce the dimensionality of the samples using principal component analysis (PCA) to represent it in two dimensions.

Note that we run the GMM clustering algorithm with the number of components as 2, which is the number of groups we expect to have. The random state is initialized as 0 to make the results reproducible. For the K-Means, we set the number of clusters to 2, and initialize the random state to 0. The DBSCAN algorithm takes two main arguments: the number of samples and epsilon. The number of samples is the required number of samples in the neighborhood of a point to consider it a core point. Epsilon is the maximum distance between two points to be considered neighbors. Table 5 shows the result of these clustering algorithms. All of these three clustering algorithms fail to group the data correctly and offer a random grouping. In other words, they could not find a clear pattern separating the fingerprinted flows from the non-fingerprinted ones.

8 CONCLUSIONS

In this paper, we introduced the first blind flow fingerprinting system using neural networks,FINN, which is robust to network noise. FINN learns the network noise through training, and thus, it is able to de-noise the noisy fingerprinted flows to decode the embedded message. We evaluated the performance of our system thorough simulations and experiments on live network connections. Our experiments evaluate the impact of different parameters, and find the optimum values for different settings. We compute the capacity of FINN to be 0.96 bits in every ten packets, which is nearly twice the state of the art. Moreover, we show that FINN performs well in conditions where real-time noise is different from what it is trained on. This is important as it prevents us from needing to re-train the model for every connection with different network jitter. Finally, we measure the invisibility of our system using Kolmogorov–Smirnov test and show that it is extremely hard for an attacker to detect the presence of our fingerprint.

An important avenue for future work is to investigate effective countermeasures against DNN-based fingerprinting systems like FINN. Note that traditional countermeasures against traffic analysis work by adding substantial delays to the packets or by inserting dummy packets [22]. Such countermeasures are not practical for the underlying scenarios of flow fingerprinting, as these countermeasures will likely cause intolerable perturbations (like excessive delays) on benign network connections. Therefore, we suggest future work to investigate the use of DNN-specific countermeasures, like adversarial perturbations [4, 21], which work by applying significantly smaller amplitudes of traffic perturbations.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers and our shepherd, Bimal Viswanath, for their invaluable feedback. This work was supported in part by the NSF CAREER grant 1553301 and the NSF grant 1953786.

REFERENCES

[1] [n. d.]. Cluster analysis. https://en.wikipedia.org/wiki/Cluster_analysis.
[2] [n. d.]. DigitalOcean. https://www.digitalocean.com/.
[3] [n. d.]. Libnetfilter queue. http://www.netfilter.org/projects/libnetfilter_queue.
[4] Alireza Bahramali, Milad Nasr, Amir Houmansadr, Dennis Goeckel, and Don Towsley. 2021. Robust Adversarial Attacks Against DNN-Based Wireless Communication Systems. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[5] A. Blum, D. Song, and S. Venkataraman. 2004. Detection of Interactive Stepping Stones: Algorithms and Confidence Bounds. In *RAID*.
[6] George Danezis. 2004. The Traffic Analysis of Continuous-Time Mixes. In *PETS*.
[7] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX*.
[8] David L. Donoho, Ana Georgina Flesia, Umesh Shankar, Vern Paxson, Jason Coit, and Stuart Staniford. 2002. Multiscale Stepping-Stone Detection: Detecting Pairs of Jittered Interactive Streams by Exploiting Maximum Tolerable Delay. In *RAID*.
[9] Juan A. Elices and Fernando Pérez-González. [n. d.]. The flow fingerprinting game. In *2013 IEEE International Workshop on Information Forensics and Security, WIFS 2013, Guangzhou, China, November 18-21, 2013*.
[10] Juan A. Elices and Fernando Pérez-González. 2013. A highly optimized flow-correlation attack. *CoRR* (2013).
[11] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Networks. arXiv:1406.2661 [stat.ML]
[12] Ting He and Lang Tong. 2007. Detecting Encrypted Stepping-Stone Connections. *IEEE Trans. Signal Processing* (2007).
[13] Amir Houmansadr and Nikita Borisov. [n. d.]. The Need for Flow Fingerprints to Link Correlated Network Flows. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*.
[14] Amir Houmansadr and Nikita Borisov. 2011. SWIRL: A Scalable Watermark to Detect Correlated Network Flows. In *NDSS*.
[15] Amir Houmansadr, Negar Kiyavash, and Nikita Borisov. [n. d.]. RAINBOW: A Robust And Invisible Non-Blind Watermark for Network Flows. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*.
[16] Amir Houmansadr, Negar Kiyavash, and Nikita Borisov. 2009. Multi-flow attack resistant watermarks for network flows. In *ICASSP*.
[17] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* (2014).
[18] Brian Neil Levine, Michael K. Reiter, Chenxi Wang, and Matthew K. Wright. [n. d.]. Timing Attacks in Low-Latency Mix Systems (Extended Abstract). In *Financial Cryptography, 8th International Conference, FC 2004, Key West, FL, USA, February 9-12, 2004. Revised Papers*.
[19] meek [n. d.]. Caida Dataset. https://www.caida.org/data/monitors/passive-equinix-nyc.xml.
[20] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. [n. d.]. DeepCorr: Strong Flow Correlation Attacks on Tor Using Deep Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*.
[21] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2021. Defeating DNN-Based Traffic Analysis Systems in Real-Time With Blind Adversarial Perturbations. In *USENIX Security*.
[22] Milad Nasr, Amir Houmansadr, and Arya Mazumdar. [n. d.]. Compressive Traffic Analysis: A New Paradigm for Scalable Traffic Analysis. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*.
[23] Pai Peng, Peng Ning, and Douglas S. Reeves. [n. d.]. On the Secrecy of Timing-Based Active Watermarking Trace-Back Techniques. In *S&P, year = 2006*.
[24] Young June Pyun, Young Hee Park, Xinyuan Wang, Douglas S. Reeves, and Peng Ning. 2007. Tracing Traffic through Intermediate Hosts that Repacketize Flows. In

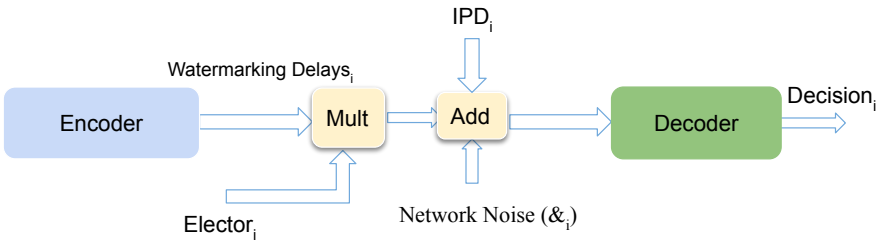


Figure 12: FINN watermarking system.

Table 6: Hyperparameters of FINN watermarking model.

	Layer	Details
Encoder	Fully Connected 1	Size: 128, Activation: Relu
	Fully Connected 2	Size: 256, Activation: Relu
	Fully Connected 3	Size: 512, Activation: Relu
Decoder	Refer to Table 2	

INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies.

[25] Fatemeh Rezaei and Amir Houmansadr. 2017. TagIt: Tagging Network Flows using Blind Fingerprints. *PoPETs* (2017).

[26] Stuart Staniford-Chen and Todd L. Heberlein. 1995. Holding intruders accountable on the Internet. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995.*

[27] Xinyuan Wang, Shiping Chen, and Sushil Jajodia. [n. d.]. Network Flow Watermarking Attack on Low-Latency Anonymous Communication Systems. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA.*

[28] Xinyuan Wang, Shiping Chen, and Sushil Jajodia. [n. d.]. Tracking anonymous peer-to-peer VoIP calls on the internet. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005.*

[29] Xinyuan Wang and Douglas S. Reeves. [n. d.]. Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays. In *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, Washington, DC, USA, October 27-30, 2003.*

[30] Xinyuan Wang, Douglas S. Reeves, and Shyhtsun Felix Wu. [n. d.]. Inter-Packet Delay Based Correlation for Tracing Encrypted Connections through Stepping Stones. In *Computer Security - ESORICS 2002, 7th European Symposium on Research in Computer Security, Zurich, Switzerland, October 14-16, 2002, Proceedings.*

[31] Kunikazu Yoda and Hiroaki Etoh. [n. d.]. Finding a Connection Chain for Tracing Intruders. In *Computer Security - ESORICS 2000, 6th European Symposium on Research in Computer Security, Toulouse, France, October 4-6, 2000, Proceedings.*

[32] Wei Yu, Xinwen Fu, Steve Graham, Dong Xuan, and Wei Zhao. [n. d.]. DSSS-Based Flow Marking Technique for Invisible Traceback. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA.*

[33] Yin Zhang and Vern Paxson. [n. d.]. Detecting Stepping Stones. In *9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000.*

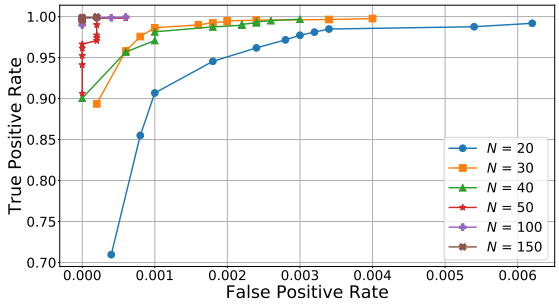


Figure 13: Performance of the FINN watermark for different flow length.

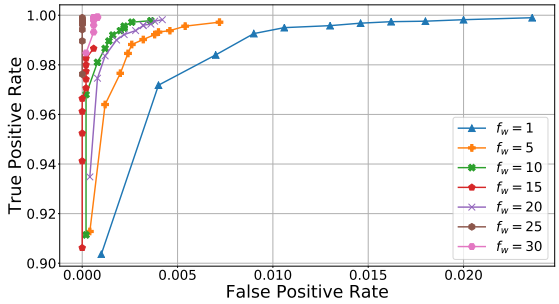


Figure 14: Selecting weight of false class by fixing the weight of class of true ($t_w = 1$ and flow length = 50, and f_w is increasing).

9 APPENDICES

9.1 Simulations

As we explained before, a watermark carries a single bit of information, which is if the flow is marked or not. Therefore, it is a more straightforward task compared to fingerprinting, which conveys multiple bits.

Here, we train our model with two keys (0 and 1) to distinguish the watermarked (true flow) and non-watermarked flows (false flow). When the key is 0, we do not watermark the flow, and when the key is 1, we add a watermark to the flow. Figure 12 shows our watermarking system. As the figure shows, the system consists of two components: encoder and decoder, borrowed from the FINN fingerprinting system. We introduce three parameters here: elector and f_w and t_w. The elector divides the model into two parts: watermarking and non-watermarking. The Elector is a vector of all ones or zeros depending on the type of corresponding flow. It would multiply into the watermarking delays, and its result adds to the IPDs. It ensures that the true flow adds to the watermarking delays to generate the watermarked IPDs by multiplying it in an array of all ones, and the false flow adds to 0 to remain intact.

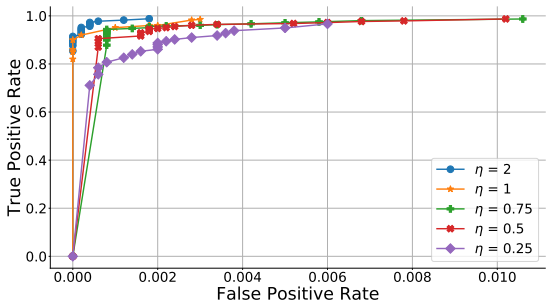


Figure 15: Performance of the watermark for different η. η is the ratio of watermark amplitude to network noise (flow length=50).

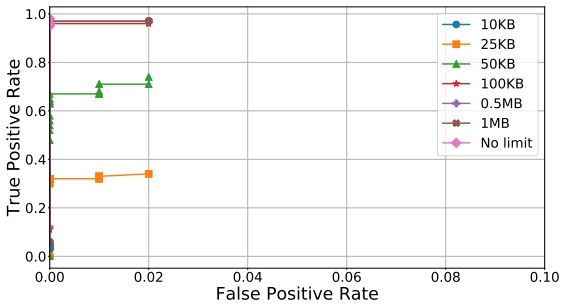


Figure 16: Performance of the real-time FINN watermark experiment on Frankfurt-Bangalore link with different bandwidths (flow length = 50).

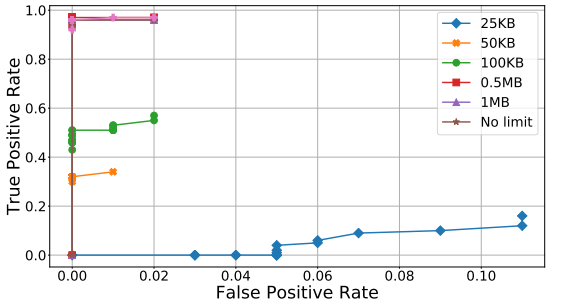


Figure 17: Performance of the real-time FINN watermark experiment on São Paulo-Bangalore link with different bandwidths (flow length = 50).

Two metrics that we use to evaluate the system are TP and FP, which is defined in below. According to each application, one of these metrics might be more critical. Therefore, we use two weights to specify the importance of each class. We weigh the FP class with f_w and the FP class with t_w. Increasing each of these weights implies that we care about the specific metric more.

Metrics: To evaluate the performance of the watermark, we use FP and TP:

- False Positive: fraction of unwatermarked flows that we wrongly flag as watermarked.
- True Positive: fraction of the watermarked flows that we flag as watermarked.

9.2 Discussion

For a specific link, the error rate is a function of watermark amplitude (α) and the flow length (ℓ). Choosing an appropriate watermark amplitude depends on the jitter of the link. We need to use a larger amplitude for a link with a high network jitter and a smaller amplitude for a lower network jitter link. We define η as the ratio of watermark amplitude to the SD of jitter on the link. We expect to get better TP and FP by increasing this parameter. When choosing this parameter, we need to keep in mind that increasing this parameter decreases the system's invisibility. Figure 15 shows the result of having η in $[2, 1, 0.75, 0.5, 0.25]$. As the figure shows, we get better results as we increase this parameter. In particular, we get near 100% true positive and lower than 10^-3 FP when setting the η to 1. Houmansadr et al. show in the Rainbow [15]; this value ensures that our system is invisible to the adversary.

In our experiments, we choose larger f_w to decrease the false positive and try it with the values in [1, 5, 10, 15, 20, 25, 30] while fixing the t_w at 1. Figure 14 shows the tradeoff of FP and TP as we increase this f_p, which shows that as we increase the f_w, the false positive improves, and the true positive degrades.

9.3 Implementations

We implement FINN as a watermark, embedding one bit of data in the flows. We use six different links to evaluate its performance on cellular and wireless network. The links are from the Bangalore or campus node to 6 Amazon EC2 located worldwide (Sao Paulo, Dublin, Ohio, Frankfurt, Mumbi, Tokyo). We send more than 100 watermarked and non-watermarked flows in each link. Note that we use 50 packets in the watermarking flow. Table 7 and 8 show results of our experiment for the wireless and cellular experiments, respectively. Our results show that we get a false positive of 0 and a true positive larger than 90% for all links in both network conditions. Note that the jitter's SD in these links is in the range (0.05, 32.02) msec. The Bangalore-Ohio link has a lower detection rate because the SD of noise was higher compared to the other links with a comparatively smaller SD of noise. Remember, we train our model to assume that the SD of noise is in ranger(2, 10 msec. We can improve the results by training our model with a larger SD of noise.

Different bandwidths. To further investigate the FINN watermark's performance, we implement it on different bandwidths starting from 10KB to more than 10MB per second. Figures 16 and 17 show the result of our experiment on two links (Frankfurt-Bangalore and Sao Paulo-Bangalore). We notice that the watermark detection rate degrades in very low bandwidth. To be more specific, our system shows a low TP when the bandwidth is lower than 50KB. A bandwidth of 500KB/sec is enough to get the TP arbitrarily close

Table 7: Performance of FINN watermark experiment on wireless connection for different links (False positive is fixed at 0)

Link	True Positive	False Positive
Campus - (Dublin, Ohio)	0.98	
Campus - (Tokyo, São Paulo)	0.96	
Campus - (Frankfurt)	0.96	
Campus - (Mumbai)	0.94	0
Bangalore - (Dublin)	0.98	
Bangalore - (Tokyo)	0.97	
Bangalore - (São Paulo, Frankfurt)	0.96	
Bangalore - (Ohio)	0.94	

Table 8: Performance of watermark on cellular connection for different links (False positive is fixed at 0)

Link	True Positive	False Positive
Campus - (Tokyo)	0.98	
Campus - (São Paulo)	0.97	0
Campus - (Bangalore, Frankfurt)	0.93	

to 1 and FP arbitrarily close to 0. We do not have similarly good results on low bandwidth because we train our model with higher bandwidth. Therefore, our performance degrades for the conditions that we have not trained our model for. To improve our results, we need to train the model using samples for different bandwidths.

Comparing to previous work. We use FINN for the application of stepping stone detection. For our scenario, we use network jitter with Laplacian distribution, and for packet loss, we use Bernoulli distribution. The SD of jitter is between $[2, 10]$ msec, and we use the amplitude of 0.75σ and σ, and the loss rate of 2%. This is similar to the setting of DeepCorr [20] in its stepping stone scenario. Our results show that we get slightly better results than the DeepCorr by using much fewer packets (50 packets compare to 300 packets in DeepCorr). Speed of correlation is one of the main principles in designing our model and comparing it to the previous work, and we reduce the number of packets needed to $1/6$ of state of the art.