October 2021

Design and Implementation of Algorithms for Traffic Classification

Fatemeh Rezaei
University of Massachusetts Amherst

Design and Implementation of Algorithms for Traffic Classification

A Dissertation Presented

by

Fatemeh Rezaei

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2021

College of Information and Computer Sciences

Design and Implementation of Algorithms for Traffic Classification

A Dissertation Presented

by

Fatemeh Rezaei

Approved as to style and content by:

Associate Professor Amir Houmansadr, Chair

Professor Donald F. Towsley, Member

Associate Professor Phillipa Gill, Member

Professor Dennis L. Goeckel, Member

James Allan, Chair
College of Information and Computer Sciences

Design and Implementation of Algorithms for Traffic Classification

A Dissertation Presented

by

Fatemeh Rezaei

Approved as to style and content by:

Associate Professor Amir Houmansadr, Chair

Professor Donald F. Towsley, Member

Associate Professor Phillipa Gill, Member

Professor Dennis L. Goeckel, Member

James Allan, Chair
College of Information and Computer Sciences

ABSTRACT

Design and Implementation of Algorithms for Traffic Classification

SEPTEMBER 2021

Fatemeh Rezaei

B.Sc., SHARIF UNIVERSITY OF TECHNOLOGY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Amir Houmansadr

Traffic analysis is the practice of using inherent characteristics of a network flow such as timings, sizes, and orderings of the packets to derive sensitive information about it. Traffic analysis techniques are used because of the extensive adoption of encryption and content-obfuscation mechanisms, making it impossible to infer any information about the flows by analyzing their content. In this thesis, we use traffic analysis to infer sensitive information for different objectives and different applications. Specifically, we investigate various applications: p2p cryptocurrencies, flow correlation, and messaging applications. Our goal is to tailor specific traffic analysis algorithms that best capture network traffic's intrinsic characteristics in those applications for each of these applications. Also, the objective of traffic analysis is different for each of these applications. Specifically, in Bitcoin, our goal is to evaluate Bitcoin traffic's resilience to blocking by powerful entities such as governments and ISPs. Bitcoin and similar cryptocurrencies play an important role in electronic commerce and other trust-based distributed systems because of their significant advantage over traditional currencies, including open access to global e-commerce. Therefore, it is essential to

the consumers and the industry to have reliable access to their Bitcoin assets. We also examine stepping stone [**?**] attacks for flow correlation. A stepping stone is a host that an attacker uses to relay her traffic to hide her identity. We introduce two fingerprinting systems, TagIt and FINN. TagIt embeds a secret fingerprint into the flows by moving the packets to specific time intervals. However, FINN utilizes DNNs to embed the fingerprint by changing the inter-packet delays (IPDs) in the flow. In messaging applications, we analyze the WhatsApp messaging service to determine if traffic leaks any sensitive information such as members' identity in a particular conversation to the adversaries who watch their encrypted traffic. These messaging applications' privacy is essential because these services provide an environment to discuss politically sensitive subjects, making them a target to government surveillance and censorship in totalitarian countries. We take two technical approaches to design our traffic analysis techniques. The increasing use of DNN-based classifiers inspires our first direction: we train DNN classifiers to perform some specific traffic analysis task. Our second approach is to inspect and model the shape of traffic in the target application and design a statistical classifier for the expected shape of traffic. DNN-based methods are useful when the network is complex, and the traffic's underlying noise is not linear. Also, these models do not need a meticulous analysis to extract the features. However, deep learning techniques need a vast amount of training data to work well. Therefore, they are not beneficial when there is insufficient data available to train a generalized model. On the other hand, statistical methods have the advantage that they do not have training overhead.

ACKNOWLEDGMENTS

My sincere gratitude goes first to my advisor, Prof. Amir Houmansadr, for his outstanding support, valuable mentorship, and patience at every stage of my Ph.D. program. I would also like to thank my committee members, Prof. Donald Towsley, Prof. Dennis Goeckel, and Prof. Phillipa Gill, for their valuable feedback on my thesis. This work has been improved immensely by their invaluable remarks.

Table of Contents

1 INTRODUCTION **1**
- 1.1 Applications of Traffic Analysis . . . 2
- 1.2 Approaches . . . 4
- 1.3 Thesis Overview . . . 6

2 BACKGROUND AND RELATED WORK **8**
- 2.1 Flow Correlation . . . 8
- 2.2 Bitcoin Identification . . . 12
- 2.3 Traffic Analysis Attacks on WhatsApp . . . 15

3 TagIt FLOW FINGERPRINTING **17**
- 3.1 Design of TagIt Fingerprinting System . . . 17
- 3.2 System Analysis . . . 25
- 3.3 Simulations . . . 28
- 3.4 Real-World Implementation . . . 31
- 3.5 Fingerprint Invisibility . . . 32
- 3.6 Conclusion . . . 34

4 FINN Flow Fingerprinting **35**
- 4.1 Design of FINN Fingerprinting System . . . 35
- 4.2 Experimental Setup . . . 38
- 4.3 Simulations . . . 41
- 4.4 Real-World Implementation . . . 47
- 4.5 Fingerprint Invisibility . . . 52
- 4.6 Conclusions . . . 55

5 Bitcoin Identification **57**
- 5.1 Background on Bitcoin . . . 57
- 5.2 Characterizing Bitcoin Traffic . . . 61
- 5.3 Designing Bitcoin Classifiers . . . 66
- 5.4 Experimental Setup . . . 71
- 5.5 Results . . . 74
- 5.6 Conclusions . . . 77

6 Traffic Analysis Attack on WhatsApp **79**

6.1 Background on Secure Instant Messaging Services 79
6.2 Attack and Threat Model 80
6.3 Attack Algorithm 82
6.4 Experimental Setup 85
6.5 Experiments 85
6.6 Conclusions 88

7 Conclusions **89**

List of Figures

1.1 Internet communications are increasingly encrypted. 2
1.2 Example application scenarios of network flow fingerprints. (F_*^I and F_*^E represent ingress and egress flows) 3

2.1 A flow correlation technique. 9

3.1 General model of flow fingerprinting systems. 18
3.2 Embedding bits 0 and 1 with TagIt. To embed 1, packets should move to darker subintervals, and to embed 0, they should be moved to lighter subintervals. 19
3.3 To ensure invisibility, TagIt fingerprinter only moves R_{move} fraction of packets into fingerprint slots. R_{move} depends on the rate of the flow being fingerprinted ($R_{move} = 0.5$ is illustrated in the figure). 21
3.4 Inserting multiple bits per interval. 22
3.5 Empirical distribution of overlap between randomly generated permutation functions with length 6000. 23
3.6 Offset synchronization of fingerprint extraction 25
3.7 False positive according to the number of embedded bits in interval. Each point in the figure is the average over 100 flows. 25
3.8 Delay distribution for different rates on a link between two distant computers (one in the U.S. and the other in Europe). 27
3.9 Extraction result for for various links (no coding). 30
3.10 Comparing analysis and simulation results for rate= 10. 30
3.11 Impact of R_{move} on invisibility and extraction rate (slot length= $8ms$). 33
3.12 Cumulative histogram of 10 flows, non-fingerprinted and fingerprinted with different slot lengths (quantization step in each figure is half of slot length). (a) and (c) have 2000 points in the X axis because slot length is 2 vs. 20 in (b)and (d), and interval length is $2sec$. 34

4.1 The network architecture of the FINN. 36
4.2 Result of increasing α on performance of FINN fingerprint in different network conditions (flow length = 100, $\ell = 2^{10}$). 42
4.3 Result of increasing ℓ and number of training data on performance of FINN fingerprint(flow length = 100). 42
4.4 Result of increasing flow length and number of training data on performance of FINN fingerprint (fingerprint length = 2^{10}). 43

4.5 FINN watermarking system. 44
4.6 Performance of the FINN watermark for different flow length. 45
4.7 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). 45
4.8 Performance of the watermark for different η. η is the ratio of watermark amplitude to network noise (flow length=50). 46
4.9 Performance of the real-time FINN fingerprint experiment from campus to various nodes (Wireless, flow length = 100). 48
4.10 Performance of the real-time FINN fingerprint experiment from Bangalore to various nodes (Wireless, flow length = 100). 49
4.11 Performance of the real-time FINN fingerprint experiment from campus to various nodes (Cellular, flow length = 100). 49
4.12 Performance of the real-time FINN watermark experiment on different bandwidths (flow length = 50). 51
4.13 Using GAN to improve invisibility. 53
4.14 K-S Test Difference for different methods (Fingerprint: flow length = 100, fingerprint length = 2^{10}, Watermark: flow length = 50). 54
4.15 We reduce the dimensionality of the samples using principal component analysis (PCA) to represent it in two dimensions. 56

5.1 Comparing different relaying in the Bitcoin network 60
5.2 Packet size distribution of Bitcoin communication messages in compact block relaying. 62
5.3 Upstream and downstream packet size distribution of Bitcoin and several popular protocols. 63
5.4 Ratio of downstream to upstream traffic volume for 5 minutes of traffic. 64
5.5 Comparing time of each block receive with time of blocks in the block chain. 65
5.6 Histogram of compact block mode components. 66
5.7 Histogram of block propagation delay. 67
5.8 Result of `sizeHist` classifier on noisy Bitcoin traffic. 72
5.9 Result of `D2U` classifier on noisy Bitcoin traffic. 74
5.10 Block detection rate using window-based classifier. 76

6.1 Attack Scenarios. Adversary wants to find the person that is chatting with client A. 81
6.2 Volume of WhatsApp traffic over time for 10 minutes. The spikes of volume occur when a large message is exchanged. 83
6.3 Event-based attacker. 83
6.4 Shape-based attacker. 84
6.5 Performance of our detectors on normal condition. 86
6.6 Comparing the result of event-based detector in different network bandwidths. 87
6.7 Result of the event-based detector when traffic is passed through different VPN servers. 88

List of Tables

1.1 Overview of Thesis 4

3.1 Fingerprint Parameters. 22
3.2 Comparing the network delay of the three links used in our simulations. 29
3.3 Trade-offs in selecting different parameters of TagIt. 29
3.4 Running TagIt on live traffic. 31
3.5 Kolmogorov-Smirnov with confidence level=95% ($\lambda = 10, r = 10$). . . 32
3.6 Average and maximum fingerprint delay per packet for different r. . . 33

4.1 FINN Parameters. 37
4.2 FINN fingerprint model hyperparameters. 40
4.3 Hyperparameters of FINN watermarking model. 44
4.4 Performance comparison of TagIt and FINN. 50
4.5 Performance of FINN watermark experiment on wireless connection for different links (False positive is fixed at 0). 51
4.6 Performance of watermark on cellular connection for different links (False positive is fixed at 0). 51
4.7 Discriminator model hyperparameters. 53
4.8 Result of clustering the fingerprinted and non-fingerprinted samples. . 55

5.1 The list of Bitcoin communication messages. 59
5.2 Number and percentage of each message in a 31 days Bitcoin traffic in compact block relaying. 62
5.3 Traffic Class Breakdown For CAIDA Dataset. 72
5.4 Result of Neural Network classifier. 77

LIST OF ABBREVIATIONS

FP False Positive

GAN Generative Adversarial Network

IPD Inter-packet delay

SD Standard Deviation

TP True Positive

Chapter 1

INTRODUCTION

The Internet has made life easier for every one of us. We shop online, connect to our friends and families, pay our bills, study. This saves a lot of time and makes our life much more convenient, but it comes at the price of our privacy. We share too much information online, and our most sensitive information is passing through the Internet, and this poses a new threat to our privacy.

To ensure the privacy of the Internet users, network traffic gets encrypted to make it harder to infer sensitive information from the traffic. Despite the use of encryption, network traffic leaks sensitive information. The wide use of encryption and similar content-obfuscation mechanisms inspires the modern techniques to infer sensitive information from the network flows, which is to solely rely on using *traffic patterns* that are not significantly impacted by encryption and network perturbations, such as packet timings and sizes, as opposed to packet contents or headers; such an analysis is broadly referred to *traffic analysis* [42, 116, 22].

In this thesis, we investigate traffic analysis for different applications and objectives, in particular, p2p cryptocurrencies, messaging applications, and stepping stone detection. In p2p cryptocurrencies, we study Bitcoin traffic to find its distinguishing attributes. Our objective is to evaluate its resilience to blocking by powerful entities such as governments. In messaging applications, we study WhatsApp messaging service to evaluate if there is privacy leakage on one-to-one communications. In anonymous communications, we study flow correlation and design algorithms that enable us to link network flows. Linking network flows is used to compromise the anonymity of Tor and similar anonymous communications.

We use two primary approaches to design our algorithms. First, we use statistical approaches in which we observe the traffic patterns of the flow or perturb some of its patterns to infer sensitive information from it. Second, we use deep learning models to extract the distinguishing features of the flows or perturb some of its patterns like timings. In the following, we explain the applications and the approaches in more detail.

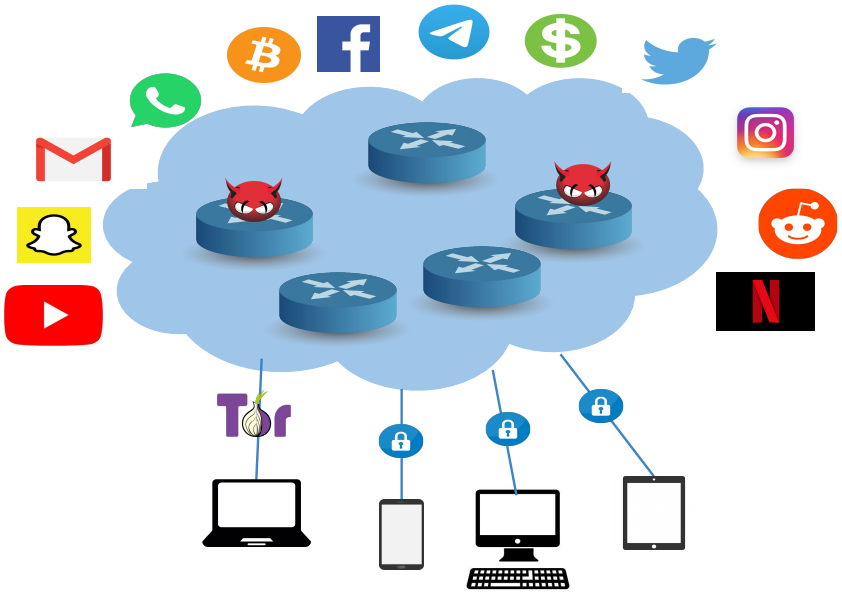


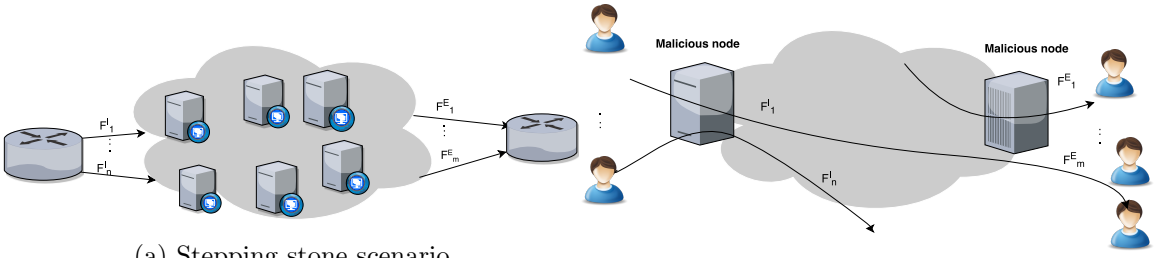
Figure 1.1: Internet communications are increasingly encrypted.

1.1 Applications of Traffic Analysis

In this thesis, we focus on several applications of traffic analysis: p2p cryptocurrencies, the security of messaging applications, and anonymous communications. We study Bitcoin traffic and evaluate its resilience to blocking by powerful entities such as ISP and government. Moreover, we analyze the WhatsApp messaging service to find out if its traffic leaks any sensitive information. The security of messaging applications is a vital issue because of the prevalence of their widespreading usage. Also, we examine flow correlation for anonymous communications. Flow correlation is to correlate the network flows entering and exiting a network in order to find the linking flows, and therefore break its anonymity. Table 1.1 shows an overview of this thesis.

Flow Correlation

One of the applications of traffic analysis that we study is to *link encrypted network flows when they pass through obfuscating proxies*. In flow correlation, we attempt to link the network flows entering and existing a network. Linking network flows is an important problem in various security and privacy applications. Particularly, it is widely known [116, 70] that linking network flows can be used by adversaries to compromise anonymity in Tor [45] and other anonymity systems [42, 82, 128, 127, 138, 72, 71, 70] by correlating the traffic patterns of ingress and egress flows. Alternatively, linking network flows has been suggested [70, 72, 71, 29, 142, 121, 130, 137, 46, 113] as a mechanism to trace back to cybercriminals who relay their traffic through previously compromised machines, known as stepping stone proxies [142]. Figure 1.2 shows these two applications.



(a) Stepping stone scenario

(b) Anonymity scenario

Figure 1.2: Example application scenarios of network flow fingerprints. (F_*^I and F_*^E represent ingress and egress flows)

Bitcoin Identification

Furthermore, Bitcoin traffic classification is another application of traffic classification that we examine. This is important for evaluating Bitcoin resilience to blocking by powerful network entities, including ISPs and governments. Bitcoin and similar blockchain-based currencies [98] have seen rapid adoption by consumers and industry because of their many applications in electronic commerce and other trust-based distributed systems. Bitcoin supports $1–$4.2B worth of transactions per day, growing steadily. Bitcoin and similar virtual currencies offer significant advantages compared to traditional electronic currencies, which include open access to a global e-commerce infrastructure, lower transaction fees, cryptographically supported contracts [20] and services [95], and transnational operations.

Given this significant importance of electronic currencies, they need to be resistant to embargoes by governments. That is, people investing in cryptocurrencies (by running businesses that rely on such currencies) should be assured that their Internet providers or governments are not able to prevent them from using their cryptocurrencies if they decide too. For the sake of argument, consider what happens if the Great Firewall of China decides to block all Bitcoin traffic overnight.

In this thesis, we investigate if and how Bitcoin's traffic can be identified through traffic analysis despite being tunneled through an encrypted channel. First, we characterize Bitcoin's traffic patterns such as rates, timings, and sizes. Comparing with other protocols, we show that Bitcoin has traffic patterns that are unique, because of the specific types of messages sent by Bitcoin peers. Leveraging such unique features of Bitcoin traffic, we design a toolset of classifiers in order to distinguish Bitcoin traffic over encrypted channels. Particularly, we use more than two month of Bitcoin and other protocols traffic over Tor [45] and three Tor pluggable transports [111], namely, FTE [48], meek [94], and obfs4 [136] to evaluate our classifiers.

Traffic Analysis Attacks on WhatsApp

Instant messaging applications have become a primary method of communication with the massive adoption of smartphones and the increase of Internet connectivity around the world. Also, they are even more prevalent in parts of the world that sending an

Table 1.1: Overview of Thesis

Applications	Approaches
Flow Fingerprinting (TagIt)	Statistical Active Analysis
Bitcoin-Hunter	Statistical and DNN-based Passive Analysis
WhatsApp Privacy Measurements	Statistical and DNN-based Passive Analysis
Flow Fingerprinting (FINN)	DNN-based Active Analysis

SMS using mobile carriers is expensive. Specifically, WhatsApp has 300 million daily active users and a total of 1.5 billion users [12]. These services enable users to send various types of messages, including video, text, voice, and files. More importantly, they provide an environment to exchange politically and socially sensitive topics, which makes them susceptible to surveillance by powerful entities like the government.

WhatsApp and similar services provide end-to-end encryption to ensure the security of their users. Despite that, it has been shown that they have security breaches. In particular, WhatsApp became a target of surveillance [9] when an attacker could use voice calling function to ring a target device and install a spying app, even if the target did not pick up the phone. Also, the call would often disappear from the device's call log. In 2019, WhatsApp reported that human rights activists and journalists were the targets of surveillance using Israeli spyware Pegasus. The spyware allowed the attackers to access passcodes and text messages in services like WhatsApp [11]. Moreover, WhatsApp was vastly blocked in China due to the Communist Party's increasing surveillance [1].

Here, we take a new direction to evaluate the privacy of users in WhatsApp messaging service. Specifically, we use traffic analysis tools to evaluate if an attacker can infer any sensitive information by monitoring the traffic of WhatsApp users.

1.2 Approaches

We apply two primary approaches to our applications. First, the statistical approach in which we manually extract the traffic features to obtains sensitive information. Second, we use DNN-based methods to find intrinsic features of the flow automatically. One of the advantages of the DNN-based technique is that it eliminates the need for an expert to select the features because it automatically extracts the features through training. Also, deep learning is useful when the pattern of old classes change, or new types of traffic emerge. They can be applied to complex traffic to extract the non-linear features [118]. The main disadvantage of DNN-based methods is that they need sufficient data and adequate computation power. Therefore, when the structure of the data is simple, or there is not enough data, it is better to use more straightforward techniques. On the other hand, using the statistical approach, we gain a better understanding of traffic since we select them manually, and we do not have the overhead of training. In the following, we explain each of these approaches in more detail.

Statistical Traffic Analysis

In the statistical approach, we attempt to infer sensitive information from traffic patterns either by observing the traffic flow (passive approach), or by slightly *perturbing* (e.g., by slightly delaying packets) in a way to embed an artificial pattern into network flows.

Previous flow correlation techniques used passive approaches [42, 142, 121, 130, 137, 46] to link network flows. Using this approach for flow correlation requires collecting long flows in order, which makes it not useful when such long flows are not available. More recently, researchers have investigated an *active* type of traffic analysis.In this approach, traffic patterns are slightly *perturbed* (e.g., by slightly delaying packets) in a way to embed an artificial pattern into network flows. This method is mainly used in flow correlation problem [128, 127, 138, 72, 71, 70]. The majority of existing works on active traffic analysis is devoted to what is called *flow watermarking* [71, 72, 113, 127, 138, 129]. Recently, Houmansadr et al. [70] proposed an alternative type of active traffic analysis called *flow fingerprinting*. While flow watermarks aim at tagging flows with a single bit of information, flow fingerprints aim at tagging flows with several bits of information. This enables one to apply different tags on different flows, therefore perform a finer-grained traffic analysis as discussed by Houmansadr et al. [70]. Intuitively, designing flow fingerprints is more challenging than watermarks as they aim at embedding multiple bits of information. We further distinguish flow watermarks and fingerprints in Section 2.1. Figure 1.2 shows the setting of a flow fingerprinting mechanism.

Previous flow fingerprinting schemes [70, 50] are *non-blind*, i.e., the fingerprinters and fingerprint extractors need to establish a control channel to continuously communicate information about the flows they intercept. This is a significant obstacle to the scalability of such systems. In this thesis, we introduce TagIt, the first *blind* flow fingerprinting mechanism. Blind mechanisms are significantly more practical and scalable than non-blind mechanisms. In a fingerprinting scenario with n ingress and m egress flows (shown in Figure 1.2), a non-blind scheme requires $O(n)$ communication between the fingerprinting parties, compared to $O(1)$ in blind mechanisms. Additionally, a non-blind mechanism imposes an $O(nm)$ computation overhead due to the pairwise-correlation of the flows observed by the fingerprinting parties. A blind scheme, however, imposes an $O(m)$ computation overhead since the extraction is performed on the individual egress flows observed by the extractor.

Deep Neural Network-based Traffic Analysis

In this approach, instead of using human-engineered features, deep learning methods are used to extract the distinguishing features of network flows through training. These methods are great in capturing the complex behavior of different traffic and do well in finding the non-linear relationship between the raw input and corresponding output and eliminate the need to have a separate feature selection and classification [118]. Note that DNN-based methods need sufficient amount of labeled data

to train well. Therefore, they are not used when there is not enough data available for training, or the traffic is simply learned by machine learning techniques that can capture the linear relationship between the input and corresponding output.

Many of previous work on traffic classification and identification use deep learning [131, 125] with a passive approach. We use a deep learning framework to distinguish Bitcoin traffic from others when there is background traffic, or traffic is tunneled through Tor or other obfuscating channels. We also use deep learning to analyze WhatsApp traffic to infer sensitive information from WhatsApp one-to-one communications. Moreover, we apply active deep learning on the flow correlation problem and introduce an active DNN-based approach for the problem of flow correlation. Our system uses a deep learning framework to embed messages within the inter-packet-delay (IPD) in a network flow and extracts the embedded message using a DNN-based decoder.

1.3 Thesis Overview

Tagging Network Flows using Blind Fingerprints — TagIt

Chapter 3 introduces the first *blind* flow fingerprinting system called TagIt. Our system works by modulating fingerprint signals into the timing patterns of network flows through slightly delaying packets into secret time intervals only known to the fingerprinting parties. Using our blind approach, we only share a secret key between the fingerprinting entities and remove the computation and communication overhead in prior work. The fingerprinting is performed to enable reliable fingerprint extraction despite natural network jitter, packet loss, reordering, and invisible to an adversary who does not possess the secret fingerprinting key. Invisibility and robustness are two sides of a coin. Thus, it is not possible to reach perfect invisibility and robustness at the same time. We show this trade-off through experiments and show that we can adjust TagIt's parameters according to the importance of each metric in our problem. TagIt makes use of randomization to resist various detection attacks such as multi-flow attacks. We evaluate the performance and the invisibility of TagIt through theoretical analysis as well as simulations and experimentation on live network flows.

Flow Fingerprinting using Neural Networks — FINN

Chapter 4 introduces the first deep learning-based fingerprinting, which uses the inter-packet-delays (IPDs) to fingerprint the network flows. Deep-learning models can automatically learn the underlying features in a flow and remove the manual feature extraction phase. We design our neural network-based model to have reliable fingerprint extraction in the presence of network jitters by learning network jitters and removing them from the fingerprinted flows.

We use Laplace distribution with different standard deviations as the jitter distribution. Network jitter was modeled as a Laplace distribution in the previous stud-

ies [102]. We evaluate the invisibility of our system and make sure that our model generates small delays making the system invisible to the adversary. Moreover, we use a Generative Adversarial Network to enforce FINN to generate fingerprinting delays that follow a Laplace distribution. Developing fingerprinting delays in a Laplace distribution improves invisibility. The small delay that we added gets lost in the network jitter and makes it challenging for an adversary to distinguish between the fingerprinting delay and the jitter. We evaluate the performance and invisibility of our model using extensive experiments and simulations on network flows. Also, we develop a real-time fingerprinting system to assess our performance on Amazon EC2 and Digitalocean nodes located worldwide.

Detecting Bitcoin Traffic Over Encrypted Channels —Bitcoin Hunter

In chapter 5, we investigate the resilience of Bitcoin to blocking by powerful network entities such as ISPs and governments. By characterizing Bitcoin's communication patterns, we use deep-learning and statistical approaches to design classifiers that can distinguish (and therefore block) Bitcoin traffic. Our classifiers can detect Bitcoin even if it is tunneled through an encrypted channel like Tor and even if there is significant background traffic, e.g., due to browsing multiple websites simultaneously. We perform extensive in-the-wild experiments to demonstrate the reliability of our classifiers in identifying Bitcoin traffic even despite using obfuscation protocols like Tor pluggable transports. We conclude that standard obfuscation mechanisms are not enough to ensure blocking-resilient access to Bitcoin (and similar cryptocurrencies). Therefore cryptocurrency operators should deploy tailored traffic obfuscation mechanisms.

Traffic Analysis Attacks on WhatsApp

In chapter 6, we analyze WhatsApp communication patterns to deduce sensitive information in one-on-one messaging. Messaging applications are a target for surveillance and censorship by governments because they provide a "secure" environment for communication between people discussing sensitive political or social subjects. The popular messaging applications utilize state-of-the-art encryption mechanisms to protect their clients. However, we argue that despite encrypting, these applications are susceptible to traffic analysis attacks. We do extensive experiments to evaluate WhatsApp messaging applications to see if one-on-one communications leak information about the member in the chat. More specifically, we want to see if an adversary who has access to one side of the conversation (either wiretapping an identified client's traffic or having access to her device) can find the other side of the chat by looking into his traffic. We use the algorithm used in [23] to identify admins or members of channels in the Telegram application.

Chapter 2

BACKGROUND AND RELATED WORK

We study traffic analysis for different applications: stepping stone detection, p2p cryptocurrencies, and messaging applications by investigating the statistical and learning-based models to infer information from their traffic. In this section, we learn about the previous techniques which solved similar problems. First, we look into flow correlation methods used for stepping stone detections. Then, we investigate traffic classification techniques to detect different applications/protocols (Bitcoin detection). Last, we study previous attacks on messaging applications.

2.1 Flow Correlation

One of the problems studied in this thesis is to *link encrypted network flows when they pass through obfuscating proxies*. In particular, this problem has been extensively studied in two contexts. First, an adversary may aim at de-anonymizing connections made through an anonymization system like Tor by *linking* the ingress and egress flows observed at various vantage points controlled by the adversary. For instance, the adversary shown in Figure 1.2b can de-anonymize a Tor connection by linking the corresponding ingress and egress flows observed on compromised guard and exit relays (or the network routers intercepting those flows). Note that such linking can not be done by comparing packet contents due to anonymization and onion routing.

A second widely studied scenario for linking network flows is to identify stepping stone attackers [29, 142, 121, 130, 137, 46, 113], which is the focus of our work. A stepping stone attacker is one who relays her attack traffic through previously compromised machines, called stepping stone proxies. Figure 1.2a shows an example scenario. Similar to the anonymity scenario, the use of encryption by stepping stone proxies prevents linking flows through packet content. Linking network flows has been studied in other scenarios as well, for instance, for detecting botmasters who control botnets trough low-latency, interactive C&C channels [115, 16].

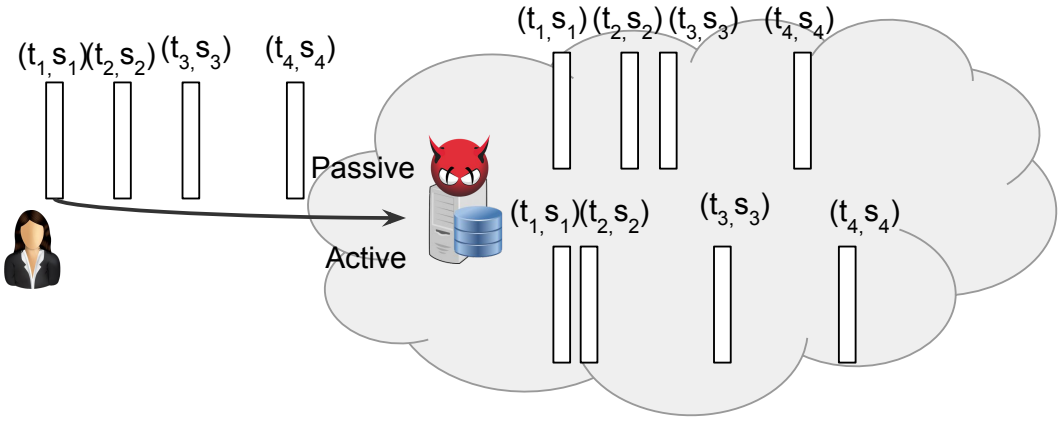


Figure 2.1: A flow correlation technique.

Here, we overview the existing work on linking network flows using traffic analysis. Figure 2.1 shows the two traffic analysis techniques.

Passive Analysis

The traditional approach for linking network flows is mainly based on observing network traffic and trying to link network flows by correlating their inherent characteristics such as packet timings, counts, and sizes [100, 68, 69, 141, 42, 142, 121, 130, 137, 46]. Chen and Heberlein [121] used *thumbprints* which is the short summary of the content of connections for correlation.

However, due to the wide use of encryption, techniques that depend on the content are no longer applicable. Zang and Paxson [142] model a network flow as a sequence of ON/ OFF intervals and used the pattern to link the flows. They used the number of consecutive OFF periods and their distance to detect stepping stones. Alternatively, Blum et al. [29] used ideas from Computational Learning Theory and analysis of a random walk to detect stepping stones. They correlated flows based on the number of packets received at any given point in time. They declared two flows to be linked if their counts of packets correlate over time. They also considered the attacker that inserts bounded chaff packets to evade detection. They are also the first to give an upper bound on the number of packets needed to detect stepping stones with a given level of confidence.

He et al. [69] studied detecting the stepping stones subject to the attacker's perturbations with constraints on the host memory or bounds on the packet delay. They compared their algorithm with previous methods, DA and DAC proposed by Blum et al. [29], and showed that their approach outperformed for fast flows. They also considered the case that the attacker adds chaff packets to evade detection. They compared their results with previous methods and showed that their approach tolerates chaff packets growing linearly by the flow's size compared to the constant chaff packets in Blum et al. [29], and Zhang et al. [141].

Such passive linking algorithms suffer from needing large numbers of packets before being able to make a reliable decision. Elices et al. [51] recently proposed a mechanism based on the Neyman-Pearson lemma that could link flows with fewer numbers of packets. Nasr et al. [102] introduced a new direction to traffic analysis called *compressive traffic analysis* to improve the scalability and apply it to the flow correlation problem. Their method used projection algorithms from signal processing to compress traffic features and perform traffic analysis on compressed features instead of the raw traffic. They reached similar performance compared to traditional approaches while providing scalability. Also, Nasr et al. [100] introduce a deep learning framework to correlate network flows, which outperforms the previous techniques and reach more than 90% true positive in linking flows that pass the Tor network using only 300 packets.

Active Analysis

The major limitation of passive analysis is not being scalable to real-world applications. For a scenario with n ingress flows and m egress flows, passive analysis requires $O(n)$ communication and $O(nm)$ computation overheads, since the collected patterns need to be cross-correlated. Active traffic analysis reduces communication and computation overheads to $O(1)$ and $O(m)$, respectively, by embedding imperceptible tags into the flows being inspected. Most active analysis techniques [71, 50, 70, 72, 113, 127, 138, 129] work by modifying packet timings, i.e., by adding artificial delays to network packets to insert invisible timing tags.

There are two types of active analysis mechanisms.

Flow Watermarking. The main body of work on active flow linking is what is known as flow watermarking. In this approach, a traffic analysis party perturbs traffic patterns in such a way to reliably encode a *single* bit of information into network flows. Wang et al. [130] were the first to propose a flow watermarking mechanism by modulating the watermark signal into the inter-packet delays (IPDs), and used four correlation functions, each with certain advantage. Wang and Reeves [129] adjusted the selected IPDs such that the quantization of the adjusted IPD had remainder of w (1 or 0) when modulus 2 was taken. To make their system robust, they embedded the watermark to m number of IPDs, and called m as *redundancy number*. They also give a bound on the maximum tolerable Perturbation to detect the watermark. RAINBOW [72] also used IPDs for watermarking but used a *non-blind* architecture. It achieves a higher detection accuracy by using a side-channel to communicate the observed packet timings among traffic analysis parties.

An alternative approach to flow watermarking is the interval-based approach in which packets are delayed into specific time intervals for watermark insertion. Most of the recent proposals for watermarking have used an interval based design [71, 113, 138, 127] due to its better resistance to packet perturbations, like packet re-ordering and drops, compared to the IPD-based approach. For instance, Yu et al. [138] used spread spectrum pseudo-noise codes to modulate the watermark signal into the rate

of packets in specific time intervals. Wang et al. [113] adjusted the packet timings to manipulate the packet counts of selective interval pairs to encode a watermark bit. If the packet count difference of a selected pair were more than a threshold, the watermark bit would be 0; otherwise, 1.Wang et al. [127] used an interval *centroid* based scheme to embed the watermarks to the flows. The centroid is the average distance of the packets from the start of the interval. They divided the flow into intervals, assign each interval to groups A and B, and manipulated the packets' timing to make the difference of aggregated centroid of group A and B positive or negative depending on the bit they intend to embed.

These interval-based watermarks are susceptible to a multi-flow attack (MFA) [80], which workd by aggregating multiple flows watermarked using the same key. Houmansadr et al. designed Swirl [71] to be resistant to MFA by making the watermarking process dependent on the network flows. Houmansadr divided the flows into two groups of base and mark intervals and used the base intervals' centroid to decide on the pattern to use for watermarking the mark intervals.

Flow Fingerprinting. Flow fingerprinting aims at embedding *multiple* bits of information on each network flow, as opposed to a single bit of information in the flow watermarking. This enables the use of flow fingerprints in scenarios with significantly larger scales than that of watermarks. For instance, while an anonymity adversary (Figure 1.2b) can use a watermark to de-anonymize a single target connection, she can use fingerprints to de-anonymize a large number of connections (e.g., by embedding distinct fingerprint tags on different ingress flows she is intercepting).

Since fingerprints embed multiple bits of information, the design of a reliable fingerprinting system is significantly more challenging than watermarking techniques. We refer the reader to Houmansadr et al. [70] for a detailed comparison of flow watermarking and fingerprinting.

Fancy [70] is the first flow fingerprinting mechanism. It extends the Rainbow [72] watermarking system through the use of various coding mechanisms to enable reliable insertion of multiple bits of fingerprints. Fancy is a—non-blind—fingerprint, i.e., the fingerprinting entities need to constantly communicate the information about the network flows they intercept through a side-channel. Non-blind fingerprinting (also non-blind watermarking) suffers from similar scalability issues of passive traffic analysis mechanisms, i.e., an $O(n)$ communication overhead and an $O(nm)$ computation overhead. This is because the non-blind fingerprinting entities need to communicate some information about the flows they intercept continuously. In Fancy, for instance, the fingerprinters send the IPDs of the flows they have fingerprinted to the fingerprint extraction entities.

Elices et al. [50] used a game-theoretic analysis to identify optimal strategies for non-blind fingerprinting and compare their performance with Fancy.

2.2 Bitcoin Identification

To the best of our knowledge, the work in this category is using passive analysis. Therefore, we only consider this approach.

Passive Analysis

There is extensive work in the literature trying to classify different applications/protocols in the network. Previously, researchers were focused on classifying the applications according to port numbers [77, 76, 88, 120] and payload [120, 36, 67, 96]. There were some problems in the previous methods, which made them change their approach. For example, many applications use uncertain port numbers or some applications may not have their port numbers registered in the Internet Assigned Numbers Authority (IANA) [112]. Some applications also use ports other than their "well-known" ports to avoid operating systems access control restrictions. Using the payload of the packets adds significant complexity and processing load. Also, it becomes impossible when the traffic is encrypted.

Moore and Papagiannaki [96] learned that using the IANA list cannot reach better than 70% byte accuracy. They also use the combination of port and payload (first KByte of each flow) and realize that they can increase their accuracy to almost 79%. They could increase the accuracy by using the rest of the flow (not just the first KByte). Sen et al. [120] studied the Kazaa P2P traffic and realized that the known port numbers could only classify 30% of the traffic. They show that using the payload can reduce the false positive and negative to 5% for most of the P2P traffic that they investigated. Another study by Madhukar and Williamson [89] states that using port numbers, they are unable to identify $30 - 70\%$ of their traffic dataset.

Modern techniques use statistical characteristics of the flows to classify the applications. These statistics include the packet timing, sizes, flow duration, packet inter-arrival time, flow idle times, etc., to distinguish between the applications. For example, [109, 44] observed distinct statistical patterns between number of applications in terms of byte, duration, arrival periodicity [109], flow duration, packet inter-arrival times and packet sizes and byte profile [44]. The authors in [38, 66, 81] also noticed different patterns such as packet inter-arrival times and packet length distribution for several applications studied. These studies started a new way of traffic classification based on traffic flow statistical properties. Note that ML techniques emerged to deal with the multi-dimensional spaces of the flow attributes. We can divide the work on IP traffic classification using ML into three categories: clustering approaches, supervised learning approaches, hybrid approaches. Note that there are some survey and comparison work in the literature [105, 78, 123, 85, 52, 67, 132, 53], authors survey the use of machine learning techniques on Internet traffic classification, or apply multiple techinques, and compare their results. We use the work of Nguyen and Armitage [105] to overview these approaches.

Clustering. Clustering algorithms were studied on [92, 140, 26, 27, 52, 56, 41] to group the traffic into number of clusters. These studies utilized clustering techniques such as Expectation Maximization algorithm [43], AutoClass, K-Means, DBSCAN, or GMM, used flow features such as byte count, idle time, packet length, duration, number of bytes, and number of packets, packet orderings, etc. For example, McGregor et al. [92] used EM algorithm to divide traffic into traffic types such as bulk transfer, small transactions, multiple transactions, etc. Zander et al. [140] separated different applications such as Half-Life online game. Bernaille et al. [26] clustered the traffic into a number of TCP applications using only the first few packets in the flow.

Supervised Learning Approaches. Roughan et al. [119] used the nearest neighbors (NN), Linear Discriminant Analysis (LDA), and Quadratic Discriminant Analysis (GDA) to classify different network applications. They use five possible categories of features: packet level, flow level, connection level, intra-flow/connection, and multi-flow connections. According to their results, flow duration (flow level) and the average packet length (packet level) were the most valuable features. They consider three classification cases: three-class classifies based on traffic type: Bulk data, Interactive, Streaming. The four-class divides into four types of applications: Bulk data, Streaming, Interactive and Transactional. Finally, the seven-class maps the traffic to HTTPS, DNS, FTP, Kazaa, RealMedia, Telnet, and WWW. Their results show that error rates increases as the number of classes increases. The three-class has the lowest error of $2.5\% - 3.4\%$ for different algorithms. The four-class error rate is from 5.1% to 7.9%, and the seven-class has the highest error of $9.4\% - 12.6\%$. Moore et al. [97], used a Bayesian technique to classify other traffic such as P2P, WWW, attack, games, bulk data, database, interactive, mail, services, and multimedia. They showed that the simple Bayesian would not provide a good accuracy (65%), and to improve their technique, they used two main refinements, Naive Bayes Kernel Estimation (NBKE) and Fast Correlation-Based Filter (FCBF), which improved the accuracy to 95%. The best trust value for an individual class of applications was 98% for WWW. The lowest was for service traffic with 44%. Moreover, in [21], authors extended the work by using the Bayesian neural network to classify Internet traffic using just header-derived statistics. They show that using Bayesian neural networks can achieve up to 99% accuracy when their train and test data are collected on the same day. Their results degrade to 95% when traffic for training and testing is captured eight months apart.

Nguyen and Armitage [104] proposed a method to classify a flow based on its last N packets, which allows the timely and continuous classification. They take out multiple sub-flows from the target flow, the one that they want to identify. Each sub-flow is chosen in places that have statistical characteristics different from the rest of the flow. Then, they train their algorithm with these sub-flows. The features are extracted from these sub-flows to represent the training data. They use the Naive Bayes algorithm and reach more than 95% recall and 98% precision using 25 packets. However, they only identify one application, UDP-based Person Shooter game - Enemy Territory [133], from other Internet applications (Web, DNS, NTP,

SMTP, SSH, Telnet, P2P, etc.).

Nguyen and Armitage [103] extended their work in [104] to address the problem of directional neutrality. They trained their model with sub-flows generated by the target application and their mirror-imaged replicas. They used Naive Bayes and Decision Tree to identify the First Person Shooter game - Enemy Territory. They showed training the algorithms with bi-directional flows would worsen the result of both classifiers. However, using their synthetic sub-flows, their recall improved up to 99%, and the precision improved to 98%. Crotti et al. [41] used three features: packet length, inter-arrival time, and packet arrival order, referred as *protocol fingerprints*, for classification, and used an algorithm based on *normalized thresholds* for classification. Their method reached 91% accuracy to classify HTTP, SMTP, and POP3 using the first few packets in the flow.

Park et al. [107] used Genetic Algorithm (GA) to select the features and tested three classifiers: Naive Bayesian with Kernel Estimation (NBKE), Decision Tree J48, and Reduced Error Pruning Tree (REPTree). Their results show that two tree-based algorithms reach better performance. Li and Moore [83] used C4.5 [114] decision tree to classify Internet traffic such as mail, gaming, database, and browsing. They reached an accuracy of 99.8, using 12 features collected at the start of the flow.

In [139] the authors used SVM to classify traffic flows into categories such as mail, buck traffic, service. They extract statistics such as packet length, byte ratio of sent and received packets, number of packets per flow, window size, and transport protocol type as their feature and showed that their method reaches more than 95% accuracy using these features. Also, in [35], the authors use SVM to classify traffic into different applications such as WWW, mail, and FTP, and showed that their approach can reach a good accuracy if they choose the classifier's parameters cautiously.

Furthermore, there is some work on using deep learning for traffic classification. For example, in [131], authors used autoencoders and neural networks to learn feature representation of their dataset containing 0.3 million records, which included 58 different protocols (except HTTP). They applied deep learning to traffic identification and anomaly detection and showed that they could identify the top 25 protocols with an accuracy of more than 90%. Also, for anomaly detection, they could classify more than 60% of the unknown traffic with a probability greater than 0.8.

Wang et al. [125] used a one-dimensional CNN to classify the ISCX VPN-nonVPN dataset's encrypted traffic. They proposed a framework that integrates the feature extraction, feature selection, and classifier, which outperformed the state-of-the-art methods in their 11 out of 12 evaluation metrics.

In [86], the authors used a deep learning approach called "Deep Packet" for the traffic characterization and application identification task, which stacks an autoencoder and CNN. They showed that their methods worked on encrypted traffic and achieved 98% and 94% accuracy on application identification and traffic characterization.

Also, some works used deep learning for malware detection [126] or to classify the

traffic of mobiles or IOT devices [90, 18, 19].

Hybrid Approaches. Erman et al. [54] presented a semi-supervised approach. They used K-Means with 64000 unlabeled flows. After clustering, they chose two random flows in each cluster to label, and with K = 400, they achieve more than 94% flow accuracy. The authors tried having five or more labeled data in each cluster and notice marginal improvements [55].

To the best of our knowledge, we are the first to attempt to distinguish Bitcoin traffic from other applications. We design several binary classifiers to detect Bitcoin traffic in the presence of small background noises. Furthermore, we utilize Neural Networks to detect Bitcoin traffic with more complex background noises such as browsing more than one website (up to 5) or running applications from CAIDA (up to 5 number of different applications).

2.3 Traffic Analysis Attacks on WhatsApp

WhatsApp and similar messaging applications are popular because they allow users to send and receive unlimited free messages. SMS was the dominant way of communication untill 2012, when smartphones become widely available. Messaging applications provided a free way of communication while SMS relied on paid telephone services [6]. According to the stats, there were 2 billion active WhatsApp and Facebook Messenger users in October 2020, which makes it the most popular messaging application [5]. WhatsApp and similar messaging applications enable the users to send and receive different messages such as audios, pdfs, images, texts, and videos in private chats, or by creating public groups. Also, they are used extensively to exchange politically and socially sensitive content, which makes them the target for surveillance by totalitarian governments and repressive regimes.

The use of these messaging applications is more important in countries such as Iran that have high surveillance on their communication. For example, during the green movement in Iran, the government cut off the text messaging services of cell phones [65]. The use of WhatsApp and similar messaging applications, which rely upon the internet rather than phone services, provides a sense of safety to the activists. Although the government can block these applications, as it occured in April 2018, when Iran blocked Telegram altogether, statistics show that majority of the users connected to Telegram through various kinds of VPNs. These applications provide security to their users through end-to-end or end-to-middle encryptions.

However, they leak sensitive data through their traffic metadata such as packet timing and sizes to an adversary that is watching the encrypted traffic of clients. Using traffic analysis low-cost techniques we can identify administrators, members of public channels[23], and individuals involved in one-on-one chats. the main reason that these traffic analysis techniques work is that these applications do not use obfuscation techniques to hide user metadata. This is because using such techniques impacts the

performance of the applications and their usability.

Using traffic analysis to infer sensitive information from messaging applications started from [40], in which the authors showed that using the metadata in Apple iMessage they could infer information about user actions (start, stop, text, attachment, and read), the length of the messages, the language of the conversation with high accuracy even in the presence of end-to-end encryption. In [108], authors used a supervised machine learning technique to show that they could effectively identify user's online activities on the KakaoTalk service. In [61], the authors developed a system named CUMMA to classify service usage of messaging applications despite traffic encryption. In [135], authors classified the in-App usage in WeChat. Their goal was to identify the actions of red packet transactions and fund transfers from encrypted network traffic. They characterized the red packet and fund transfers to distinguish them from text and other regular messages. They segmented the traffic in different bursts, each one representing a certain action such as sending/receiving a text or picture, or sending/receiving a red packet or fund transfer, and designed a classifier to distinguish the red packet and fund transfers. In [39], authors useed encrypted WhatsApp Groups messages to find out the level of engagement of the users. More specifically, they could determine in a certain time interval, which users were the most engaged ones. This is very important in groups that discuss sensitive political topics in countries with totalitarian regimes, puts those users in danger. In [23], authors used traffic analysis to find the admins and members of a Telegram channel despite the end-to-end encryption. They introduced a public countermeasure, IMProxy, that Instant messaging application clients can use to have more privacy.

Another avenue of studying messaging applications looks into the security vulnerabilities. In [122], the authors examined the current secure messaging solutions to evaluate their usability, security, and ease-of-adoption. They identified three main challenges in having a secure messaging solution: trust establishment, conversation security, and transport privacy, and showed their trade-offs: having strong trust establishments results in less usability and adoption while a more usable approach does not guarantee strong security. Johansen et al. [31, 74] comprehensively investigated the security and privacy properties (usability and security) of current approaches for end-to-end encrypted messaging. They showed that to what extent the current messaging applications achieve these properties and provide recommendations to improve them.

Chapter 3

TagIt FLOW FINGERPRINTING

Flow fingerprinting is a mechanism for linking obfuscated network flows at large scale. In this thesis, we introduce the first blind flow fingerprinting system called TagIt. Our system works by modulating fingerprint signals into the timing patterns of network flows through slightly delaying packets into secret time intervals only known to the fingerprinting parties. We design TagIt to to enable reliable fingerprint extraction by legitimate fingerprinting parties despite natural network noise, but invisible to an adversary who does not possess the secret fingerprinting key. TagIt makes use of randomization to resist various detection attacks such as multi-flow attacks. We evaluate the performance and invisibility of TagIt through theoretical analysis as well as simulations and experimentation on live network flows.

3.1 Design of TagIt Fingerprinting System

We first overview the main principles in the design of the TagIt fingerprinting system.

Timing-based fingerprinting: Similar to the large fraction of previous works on active traffic analysis, TagIt is a timing-based mechanism. That is, a TagIt fingerprinter embeds fingerprints into an intercepted flow by modifying the timings of its packets, i.e., by delaying some of the packets. This makes the fingerprints usable in scenarios where the content of the flows being linked is modified in transition, e.g., Tor connections [45] and stepping stone connections [142].

Blind design: TagIt is designed to be a blind fingerprinting system; that is, the *only* information shared between TagIt's fingerprinter(s) and extractor(s) is a fingerprinting key. This is unlike existing "non-blind" flow correlation designs [72, 70] where they need to also share some information about the intercepted flows. Particularly, in the Fancy non-blind fingerprint [70] the fingerprinter will need to continuously send the inter-packet timings of all the flows it intercepts to the Fancy extractors for the extraction mechanism to work. Figure 3.1 shows the general model of blind and non-blind fingerprinting systems.

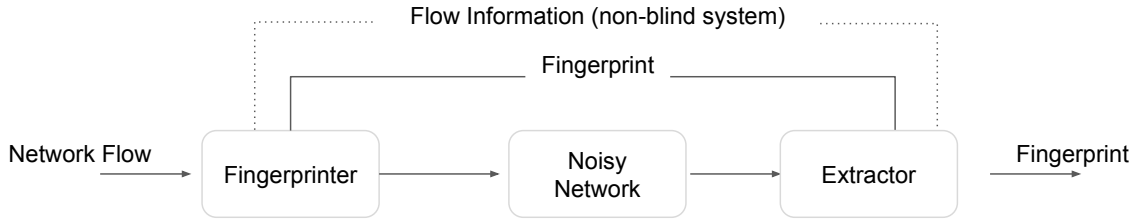


Figure 3.1: General model of flow fingerprinting systems. A blind system does not share any information about the flows between the fingerprinting parties. A non-blind system shares some information about the flows.

Blind designs are significantly more practical in real-world applications. As discussed above in Section 2.1, a blind active traffic analysis offers an $O(1)$ communication and $O(m)$ computation in a scenario with n ingress and m egress flows intercepted by the fingerprinting parties. This is while the orders of communication and computation are $O(n)$ and $O(nm)$, respectively, for a non-blind system like Fancy. Note that when comparing the orders of computation between two fingerprinting systems, one should also factor in the computation overhead of every single correlation operation, as this may differ for different systems (e.g., due to their use of different coding algorithms). This is not included in our computation order analysis, since the same correlation algorithms used in a blind system could be also used by a non-blind system.

Interval-based: There are two types of timing-based active traffic analysis systems: interval-based and IPD-based systems. An IP-based system encodes the fingerprint signal into the inter-packet delays of packets, i.e., by modifying IPDs individually. On the other hand, an interval-based approach modulates the fingerprint signal in to the counts of packets that arrive within specific time intervals. We use an interval-based structure for TagIt. This is because interval-based systems offer significantly stronger resistance to natural packet modifications, such as packet drops, repacketization, packet reordering, etc., compared to IPD-based systems that are known [72, 73, 70] to be fragile to such modifications. Therefore, a TagIt fingerprinter delays packets into specific time intervals, which we call *fingerprint* intervals, in order to fingerprint a flow. A TagIt extractor will count the ratio of the packets arriving in such intervals in order to perform fingerprint extraction.

Randomized insertion: TagIt's interval-based approach makes it robust to packet-level perturbations, as discussed above. However, Kiyavash et al. [80] demonstrate that interval-based schemes may be susceptible to an attack called the multi-flow attack (MFA). In this attack, the adversary collects multiple flows fingerprinted using an interval-based mechanism, and superimposes the flows to increase the chances of identifying the fingerprint intervals. The attack is built on the statistical distribution of packets in various intervals of an interval-based scheme.

We design TagIt in a way to be resistant the MFA attack despite its interval-based scheme. Specifically, TagIt uses a random mechanism for selecting the fingerprinting

intervals in a way that even inserting *the same fingerprint* into the same flow twice will result in different fingerprinted flows. As we will show in our analysis of Section 4.5, this makes TagIt resist the MFA by smoothing the statistical distribution of the packets on the aggregation of multiple TagIt flows.

Coding to resist noise: As described earlier, designing a reliable fingerprinting system is significantly more challenging than a watermarking system since a fingerprinting system aims to reliably transmit multiple bits of information across a noisy communication channel, unlike watermarking's single bit of information. We use two types of coding to enable reliable fingerprint extraction despite the network noise. First, we use a repetition code to resist noise due to normal network jitter. Second, we make use of standard error correction codes (like Convolutional or Reed-Solomon codes) to resist sparse, bursty channel errors that are not recoverable by normal repetition codes. We will further discuss the specific choices of our coding parameters. As will be shown in our analysis of Section 3.4, such codings result in a promising reliability for TagIt's fingerprint extraction.

Fingerprinting Scheme

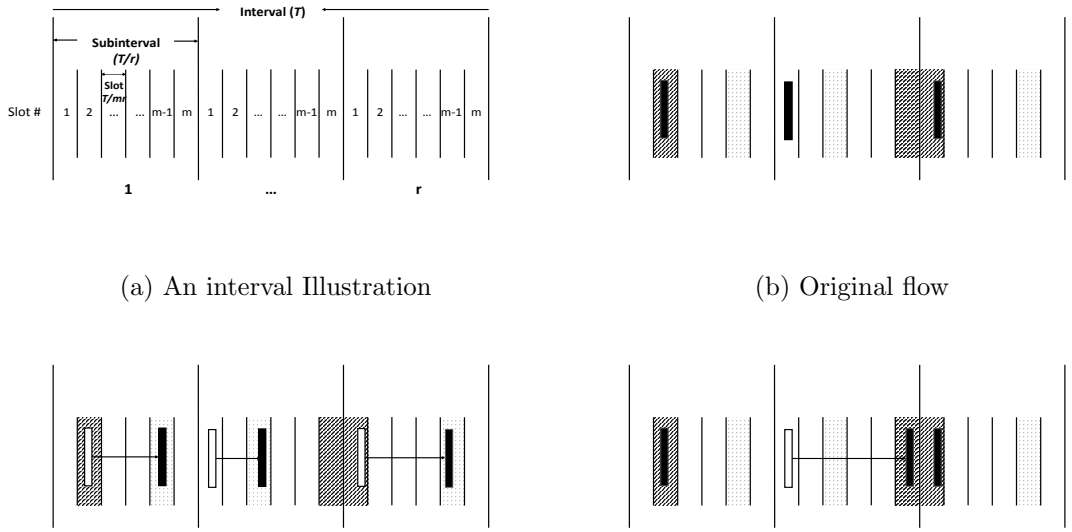
In this section, we discuss the algorithm used by a TagIt fingerprinter to fingerprint network flows. As discussed earlier, TagIt is a timing-based, interval-based scheme, therefore, it works by delaying some of the packets of a flow to be fingerprinted into specific timing intervals. In the following, we describe the details of TagIt fingerprinter; Figure 3.2 illustrates TagIt fingerprinting.

Dividing a flow into time intervals. The fingerprinter divides the time axis into a series of consecutive time intervals of lengths of T with the first interval starting at the time *offset* $0 \leq o < T$. That is, the ith interval includes the packets arriving during the time period $[o + (i-1)T, o + iT]$. The fingerprinter uses the arrival time of the first packet in the candidate flow as time zero.

Selecting fingerprint intervals. The TagIt fingerprinter embeds a flow fingerprint by delaying packets within several time intervals of that flow, which we call them *fingerprint intervals*. Suppose that the fingerprinter aims at inserting an ℓ-bits long fingerprint ID into a candidate flow. The fingerprinter converts the ℓ-bits fingerprint into $\ell^c = f/r_c$ bits of *encoded fingerprint* bits using a Convolutional or Reed-Solomon encoder, where r_c is the coding rate of our encoder and its choice will be discussed later in Section 3.4. The fingerprinter uses the first n time intervals of a flow as its fingerprint intervals, and assigns the ℓ^c encoded fingerprint bits to these intervals in order (as discussed later, we may insert multiple bits in one interval).

Dividing fingerprint intervals into slots. The fingerprinter divides each fingerprint interval (of length T) into r sub-intervals of equal length T/r. It further divides every sub-interval into m slots of length $T/(rm)$. We refer to the jth slot of the ith sub-interval as $s_{i,j}$ $(i = 1, .., r;\ j = 1, .., m)$. Figure 3.2a shows a fingerprint interval.

Secret permutation functions. A fingerprinter generates two vectors of permuta-



(a) An interval Illustration

(b) Original flow

(c) Embedding bit 0

(d) Embedding bit 1

Figure 3.2: Embedding bits 0 and 1 with TagIt. To embed 1, packets should move to darker subintervals, and to embed 0, they should be moved to lighter subintervals.

tion functions, $\Pi^0 = (\pi_0^0, \pi_1^0, ..., \pi_{r-1}^0)$ and $\Pi^1 = (\pi_0^1, \pi_1^1, ..., \pi_{r-1}^1)$, before starting the fingerprinting process, and shares them secretly with the extractor(s) as part of the secret fingerprinting key. Each π_j^i is a permutation function on $\mathbb{Z}_m = [0, .., m-1]$; for instance, $\pi_j^i : [0, 1, 2, 3, 4, 5] \rightarrow [3, 5, 1, 0, 2, 4]$ is an example permutation function for $m = 6$.

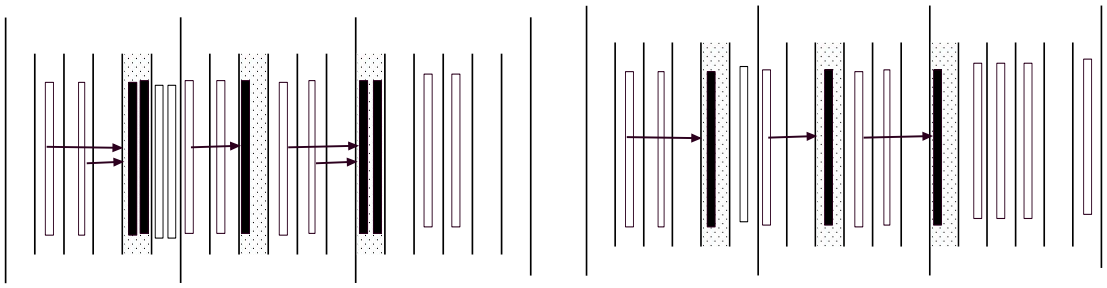
Note that the Π^0 and Π^1 functions are generated once and used for fingerprinting many flows. In Section 3.1 we will discuss the implications of the random selection of Π^0 and Π^1.

Selecting fingerprint slots. For each fingerprint interval, the fingerprinter uses the permutation functions Π^0 and Π^1 along with a *seed* to select its *fingerprint slots*. That is, for the kth fingerprint interval, the fingerprinter selects the following set of slots as the fingerprinting slots:

$$Z_k = (\pi_0^b(d_k), \pi_1^b(d_k), ..., \pi_{r-1}^b(d_k)) \tag{3.1}$$

where $b \in \{0, 1\}$ is the encoded fingerprint bit to be embedded in the kth interval, and d_k is a random seed (described later). Note that to improve resistance to exhaustive key search attacks, we use different Π^0 and Π^1 functions for different fingerprint intervals (this is analyzed in Section 3.5).

Embedding a fingerprint bit. Finally, the fingerprint bits are embedded by the fingerprinter delaying packets into the fingerprint slots. That is, the kth encoded fingerprint bit is embedded into the kth interval by delaying that interval's packets



(a) We **can** extract the embedded bit. (b) We **cannot** extract the embedded bit.

Figure 3.3: To ensure invisibility, TagIt fingerprinter only moves R_{move} fraction of packets into fingerprint slots. R_{move} depends on the rate of the flow being fingerprinted ($R_{move} = 0.5$ is illustrated in the figure).

into their nearest fingerprint slot in Z_k (the packets are delayed forward, so packets appearing after an interval's last fingerprint slot are not delayed). This illustrated in Figure 3.2.

As we will analyze in Section 4.5, for high-rate flows delaying all packets into fingerprint slots will weaken the invisibility property. We therefore only delay a fraction of the packets into fingerprint slots, R_{move}. Suppose that Δ is the length between two consecutive fingerprint slots; the fingerprinter only delays the packets arriving in the last $\Delta \times R_{move}$ section of the inter-slot interval into the second fingerprint slot. This is shown in Figure 3.3(a). Note that we need to select this parameter carefully to be able to extract the fingerprint correctly. For example, in Figure 3.3(b), it is not possible to extract the embedded bit. We discuss this parameter in Section 3.4.

Empty intervals For the intervals that we have no packet to fingerprint, we simply ignore that interval, and therefore lose the bit corresponding to that interval. Note that our choice of parameters are such that such empty intervals are rare. Also, our use of coding compensates for some of such lost bits. Alternatively, one could use the next non-empty interval to embed the corresponding bit; however, this increases the chances of total de-synchronization between the extractor and fingerprinter, e.g., if a single packets moves into an empty interval all the following bits will be lost at the extractor.

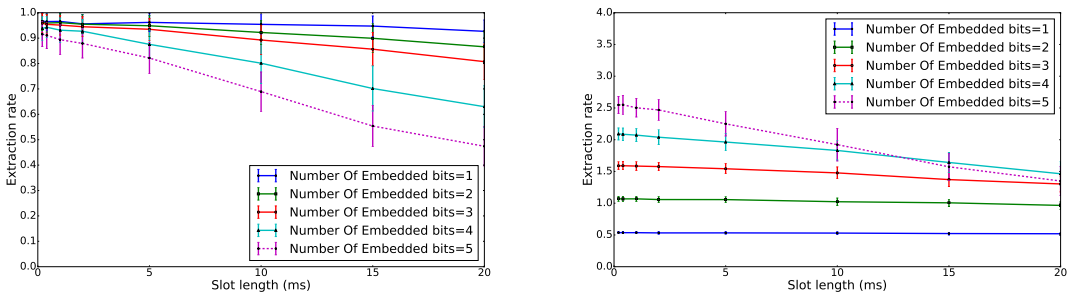
Secret key of fingerprinting. Table 3.1 summarizes the parameters of our system. Also, Table 3.1 shows the parameters that are part of the fingerprinting key. The key parameters are secretly shared between the fingerprinter and extractor, while the other parameters may be publicly disclosed.

Extension: inserting multiple bits per interval

As discussed above, TagIt uses Π_0 and Π_1 to embed bits 0 and 1, respectively. We extend the set of permutation functions to $\Pi_0, \Pi_1, ...\Pi_{S-1}$ in order to be able to embed S different message symbols, as opposed to only 2. In other words, TagIt can embed

Table 3.1: Fingerprint Parameters.

System parameters	
T	Interval length
r	Number of subintervals
m	Number of slots per subinterval
n	Number of intervals
τ	Packet extraction threshold
ρ	Fingerprint extraction threshold
ℓ	Fingerprint length
m_ℓ	Slot length
n_{bit}	Num. of fingerprint bits embedded in each interval
R_{move}	Fraction of fingerprinted packets
Secret parameters	
Π^0	Permutation for embedding bit 0
Π^1	Permutation for embedding bit 1



(a) Extraction rate for various number of bits per interval

(b) Average number of bits reliably extracted per second

Figure 3.4: Inserting multiple bits per interval.

$n_{bit} = \log_2 S$ bits of information per interval by using S numbers of permutation functions.

As expected, using more permutation functions increases the extraction complexity as the extractor will need to check more slot mappings. It also increases the chances of extraction errors: as we increase the number of permutation functions, the probability of their overlap also increases, which results in extraction errors. This can be seen in Figure 3.4a, where increasing the number of bits inserted per interval also increases the probability of error. Figure 3.4b shows the average number of bits reliably embedded per second for various numbers of permutation functions. (Note that the performance decreases with slot length as we keep the interval length fixed.)

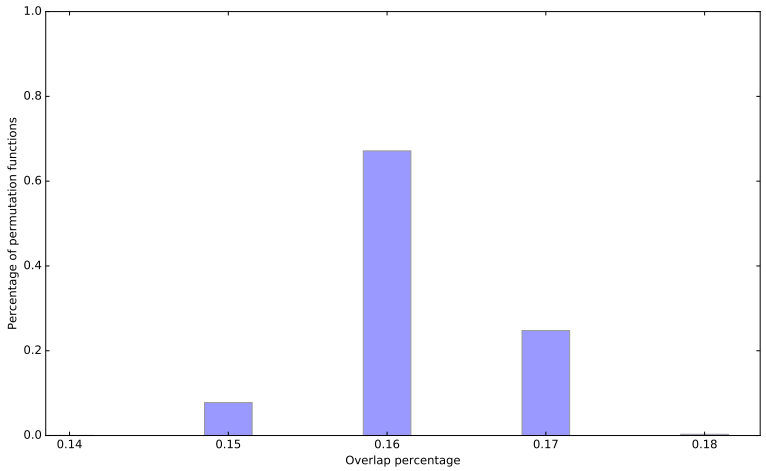


Figure 3.5: Empirical distribution of overlap between randomly generated permutation functions with length 6000.

Analysis of permutation functions

Each permutation function gives the fingerprinter m *slot mapping*s for fingerprint insertion ($\Pi^0(d)$ gives m slot mappings for bit 0 for a seed value d). However, since the permutation functions are generated randomly, it is likely that two slot mappings where one represents a 0 bit and the other represents a 1 bit have overlapping slots. A high fraction of overlapped slots between two mappings that represent different bits will increase the rate of false extractions, i.e., a 0 bit encoded into an interval may be decoded as 1. Therefore, it is important to evaluate such overlaps for random selections of $\Pi^0(d)$.

Let us compute the expected number of overlaps between two permutation functions. The probability of having $i \in [0, 1, \cdots, m-1]$ overlaps between two randomly generated permutation functions is given by

$$P_i = \binom{N}{i} \sum_{j=0}^{N-i} (-1)^j (N-i-j)!/N! \tag{3.2}$$

where $N = m!$ is the possible number of permutation functions. Therefore, we derive the expected fraction of overlaps between two permutation functions as

$$R_{overlap} = \frac{1}{m} \sum_{i=0}^{m} iP_i \tag{3.3}$$

Therefore, for $m = 6$ (used in our experiments) we have $R_{overlap} = 0.166 \simeq 1/6$. We confirm this analysis by randomly generating 40 vectors of permutation functions each of length 6000 and measuring their overlaps. As can be seen in Figure 3.5 the probability of overlap is close to 0.166.

Fingerprint Extraction Scheme

In this section, we describe how a TagIt extractor can extract fingerprints from the flows it intercepts. For a network flow intercepted by a TagIt extractor, it will either declare the flow to be "not fingerprinted", or will extract a fingerprint ID from that flow.

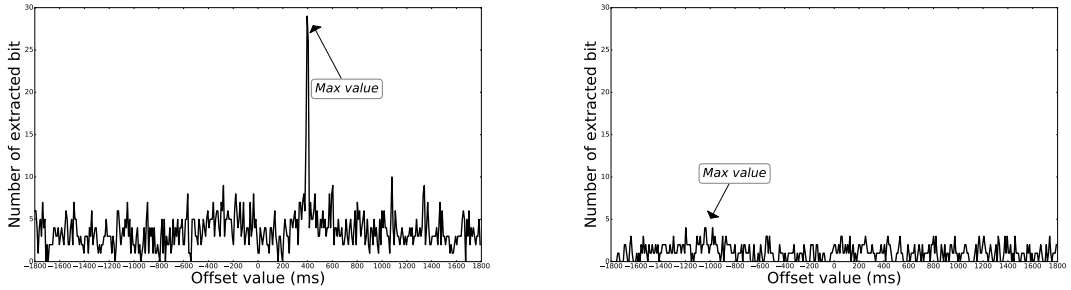
The extractor knows the fingerprinting key used by fingerprinters. As described above, the fingerprinter uses a random seed $d_k \in \mathbb{Z}_m$ to select a slot mapping for the kth interval. Therefore, the extractor will need to use $M = 2^{n_{bit}} \cdot m$ possible mappings, e.g., m mappings for the bit 0 and m for the bit 1 when $n_{bit} = 1$. For each of the possible M slot mappings, the extractor evaluates the ratio of the packets appearing in those mappings. For instance, $R^I(k)$ is the fraction of packets in interval I that appear in the slots according to the kth mapping ($0 \leq k \leq M-1$). Finally, the extractor finds the mapping k_{max} that has the maximum fraction, i.e., $k_{max} = \text{argmax}_{R^I(k)}\, k$.

If $R^I(k_{max}) > \tau$, the extractor declares the extracted bit to be the bit represented by the mapping k_{max}. Otherwise, the extractor extracts no bit from the interval I, i.e., returns a *null* bit. Note that τ is a parameter of the extractor and trade offs between the false positive and negative rates. For the extraction to be successful, we need to have $\tau \leq R_{move}$, where R_{move} is the fraction of packets moved into fingerprint slots, as defined earlier. The final stage of extraction is to use coding to correct the potentially corrupted bits. The goal of the fingerprinter is to be able to extract all fingerprint bits from a fingerprinted flow, or to correctly declare a non-fingerprinted flow as non-fingerprinted.

Extraction complexity. For each fingerprint interval, the legitimate extractor who knows the fingerprinting key, i.e., the fingerprint mappings, will need to count only M possible mappings (e.g., $2m$ for $n_{bits} = 1$). An adversary will need to guess the fingerprinting key to be able to detect the fingerprint; as we show in Section 3.5 TagIt's key has an extremely high entropy. For each key, the adversary needs to try the possible M mappings to see if any of them decode a fingerprint bit.

Another factor in the complexity of the extraction process is the complexity of the coding scheme used by the fingerprinters. However, note that since an adversary should also execute the same decoding algorithm, the coding complexity applies to the adversary as well. The coding complexity of TagIt will depend on each specific coding scheme used. For instance, a convolutional code can be decoded with a low-complexity Viterbi decoder [84, 62, 91, 58], which performs only 2^{kL} checks for each decoding operation.

Extraction Synchronization. To extract the correct fingerprint bit, we have to synchronize the received flow with the sent one. To do so, we try multiple offset values in the range $[0, T/r]$ using steps of length t, i.e., $T/(rt)$ computations. We experimentally find that setting $t = m_\ell/4$ makes the right balance between computation and synchronization accuracy. Figure 3.6 shows an extractor trying various offset values.



(a) Fingerprinted flow (b) Non-fingerprinted flow

Figure 3.6: Offset synchronization of fingerprint extraction

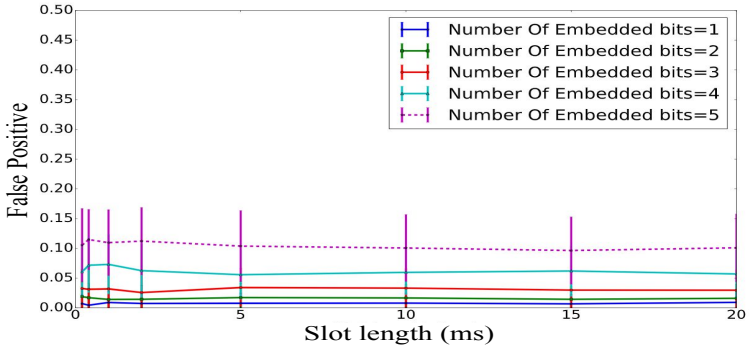


Figure 3.7: False positive according to the number of embedded bits in interval. Each point in the figure is the average over 100 flows.

3.2 System Analysis

False Extraction of Fingerprints from Non-fingerprinted Flows

We evaluate the probability of extracting a fingerprint message from a non-fingerprinted flow. In order to extract a fingerprint bit from the fingerprint interval of a non-fingerprinted flow, a τ or higher fraction of the packets within that interval should arrive within the fingerprint slots of some fingerprint mapping.

For a particular fingerprint interval, consider a specific fingerprint mapping (out of M possible mappings). For that mapping, the interval is declared to be fingerprinted with the corresponding bit if at least τ fraction of the packets in that interval arrive within the fingerprint slots of that mapping. We assume packets described by a Poisson process, i.e., the inter-arrival times are i.i.d. Therefore, for each packet, it will arrive within a fingerprint slot with a $\frac{1}{m}$ probability. Therefore, assuming P total packets in a fingerprint interval, the chances of having $\tau \times P$ or more packets within the fingerprint slots is given by:

$$F_P = 1 - CDF_{P,1/m}^{Binomial}(\lceil \tau.P \rceil) \tag{3.4}$$

where $CDF_{n,p}^{Binomial}(x)$ is the CDF function of a Binomial distribution with n number

of trials and p probability of success. We have that

$$CDF_{n,p}^{Binomial}(x) = \sum_{i=0}^{x} \binom{n}{i} p^i (1-p)^{n-i}$$

We model the arrival times using a Poisson process with rate λ (λ is the flow rate). Consequently, the number of packets in the interval follows a Poisson distribution with mean $\lambda.T$ (T is the interval length). Therefore, we average F_P for different values of P as follows:

$$F_I = M \sum_{P=1}^{\infty} \frac{e^{-\lambda.T}(\lambda.T)^P}{P!} F_P \tag{3.5}$$

Note that we also multiplied F_P with M, the number of all possible fingerprint mappings.

F_I is the probability of extracting a fingerprint bit from an arbitrary, i.e., non-fingerprinted, interval. Therefore, the chances of extracting ρ fraction of all fingerprint bits (out of total n fingerprint intervals) follows a Binomial distribution, which gives us the false positive extraction rate:

$$FP = 1 - CDF_{n,F_I}^{Binomial}(\lceil \rho.F_I \rceil) = \sum_{k=n\rho}^{n} (n,k)(F_I)^k (1-F_I)^{n-k} \tag{3.6}$$

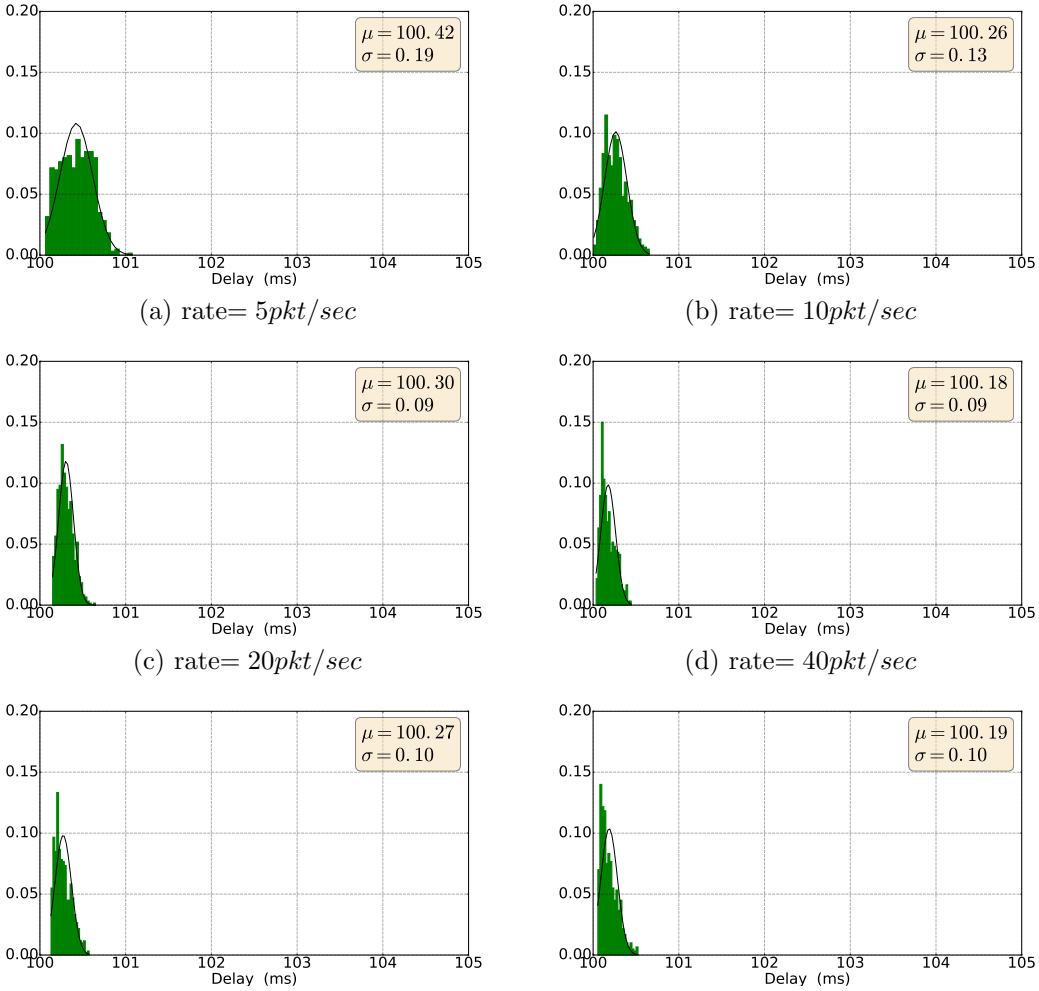
Fingerprint Error Rates

Remember that for a given fingerprint interval, the fingerprinter moves R_{move} fraction of its packets to the fingerprint slots corresponding to a particular mapping. For the extractor to be able to extract this fingerprint bit, at least τ fraction of packets should be still in the corresponding fingerprint slots. Therefore, an error happens when more than $R_{move} - \tau$ fraction of packets within the interval move out of the fingerprint slots.

An individual packet moving out of a fingerprint slot. Consider a fingerprinted packet p_i, i.e., one that has been moved to a fingerprint slot by the fingerprinter. Suppose that p_i has a distance x from the center of its fingerprint slot ($-\frac{T}{2mr} \leq x \leq \frac{T}{2mr}$). We model network noise on packets with a Gaussian distribution as suggested in previous work [110, 72] and also confirmed in our measurements shown in Figure 3.8. Therefore, the probability of p_i moving out its slot due to noise is:

$$\begin{aligned} P(p_i \text{ shifted}|x) = 1 &- \Phi_{0,1}(\frac{T/2mr - x}{\sigma}) \\ &+ \Phi_{0,1}(-\frac{T/2mr + x}{\sigma}) \end{aligned} \tag{3.7}$$

where $\Phi_{0,1}(\cdot)$ is the CDF of a Gaussian distribution with mean 0 and standard deviation 1.



(a) rate= $5pkt/sec$ (b) rate= $10pkt/sec$

(c) rate= $20pkt/sec$ (d) rate= $40pkt/sec$

(e) rate= $100pkt/sec$ (f) rate= $200pkt/sec$

Figure 3.8: Delay distribution for different rates on a link between two distant computers (one in the U.S. and the other in Europe).

Given p_i's uniform distribution in the fingerprint slot, we have:

$$P(p_i \text{ shifted}) = \frac{rm}{T} \int_{x=-T/2mr}^{x=T/2mr} P(p_i \text{ shifted}|x)dx \tag{3.8}$$

Losing a fingerprint bit. The fingerprinter loses an interval's fingerprint bit if more than $R_{move} - \tau$ fraction of its packets move out of the designated fingerprint slots, i.e.,

$$Pr_{loss} = Pr\{(R_{move} - \tau) \text{ or more frac. of packets shifted}\}$$

Assuming there are P packets in the interval, we have

$$Pr_{loss}^{P} = 1 - CDF_{P,c}^{Binomial}(\lceil (R_{move} - \tau)P \rceil) \tag{3.9}$$

where $c = P(p_i \text{ shifted})$ in (3.8). By averaging across all possible P's we have

$$Pr_{loss} = \sum_{P=1}^{\infty} \frac{e^{-\lambda.T}(\lambda.T)^P}{P!} Pr_{loss}^P \tag{3.10}$$

which is the probability of the extractor losing one particular fingerprint bit. Note that this is an upper bound for the error; for each bit there are multiple mappings (e.g., $M/2$ for $n_{bit} = 1$), and therefore the packets of the noisy interval may move into slots that correspond to another mapping of the fingerprinted bits.

Flipping a fingerprint bit. A lost fingerprint bit may be decoded as an invalid bit, e.g., a 0 bit may be extracted as a 1 by the extractor. This happens if the shifted packets move to the slots corresponding to a mapping that represents an incorrect bit. We can estimate the probability of a bit flip as:

$$Pr_{flip} = Pr_{loss} \times (M_b \sum_{P=1}^{\infty} \frac{e^{-\lambda.T}(\lambda.T)^P}{P!} F_P) \tag{3.11}$$

where F_P is given in (3.4) and M_b is the number of possible mappings for the incorrect bit.

Extraction error rate. Recall that TagIt uses an encoder, e.g., a convolutional code, to transform ℓ fingerprint bits into ℓ^c coded bits. Suppose that our encoder can correct c bits out of ℓ^c bits. Therefore, our extractor should be able to correctly extract $\ell^c - c$ or more in order to be able to decode the ℓ fingerprint bits.

Therefore the probability of the extraction error is:

$$Error_{Extraction} = 1 - CDF_{\ell^c, Pr_{loss}}^{Binomial}(c) \tag{3.12}$$

As discussed in Section 3.4, we choose our parameters such that $Error_{Extraction}$ is close to zero.

3.3 Simulations

In our simulations, we generate synthetic network flows with various rates based on Poisson processes, as commonly used in the literature. We measure network jitter between a node on our campus (which we will call "Campus") and several Planetlab nodes [25]. We pick three Planetlab nodes that represent various network conditions. Table 3.2 compares the standard deviation of delay for the three links that we used in our simulations. The delays are measured over 200 flows each containing 600 packets sent over the links.

We additionally simulate more noisy conditions (higher delay standard deviations) by adding artificial noise to the traffic using the Linux "tc" command. For each link, we measure delays for three different flow rates of 10, 100, and 200 pkt/sec.

Table 3.2: Comparing the network delay of the three links used in our simulations.

	Nodes involved	Standard deviation of delay
Link 1	Campus-Ireland	$0.083 - 0.478$ (ms)
Link 2	Campus-Switzerland	$9.773 - 12.347$ (ms)
Link 3	Campus-China	$24.917 - 30.762$ (ms)

Table 3.3: Trade-offs in selecting different parameters of TagIt.

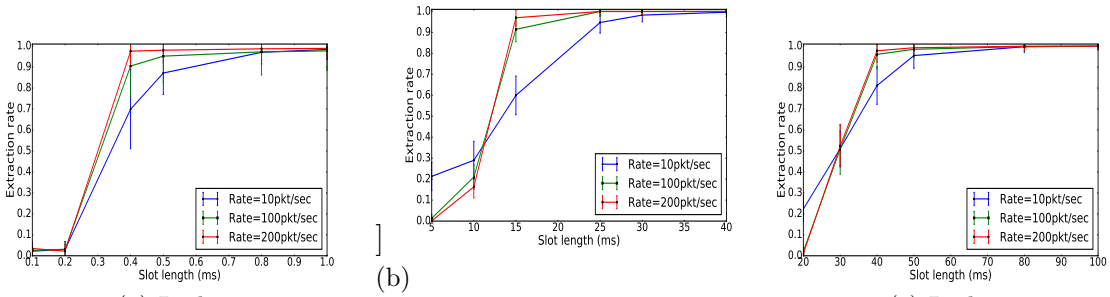
Parameter	**Trade-offs**	
	Increasing improves	Decreasing improves
T	Extraction rate	Extraction time
r	Delay, invisibility	Extraction rate
m_ℓ	Extraction rate	Invisibility
m	Invisibility, delay	Extraction rate
n_{bit}	Invisibility	False pos., Extract. time
τ	False positive	False negative
ρ	False positive	False negative
ℓ	Extract. performance	Extraction time, delay
R_{move}	Extraction rate	Invisibility

Note that we have excluded experiments on Tor. This is because our measurement of delays on Tor suggests that they significantly deviate from that of typical Internet traffic, therefore our earlier analysis does not hold. We leave deriving parameters tailored to Tor for future work.

Parameter Choices

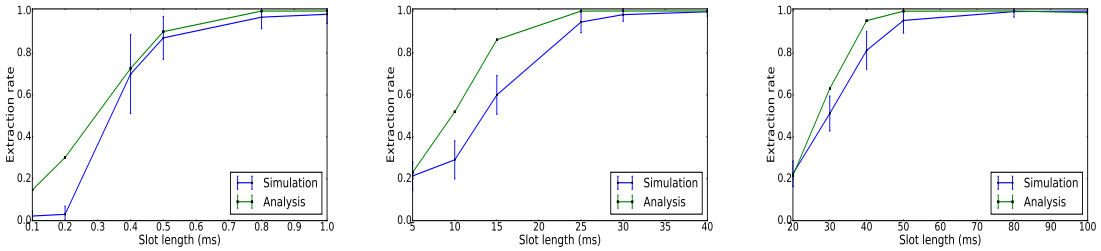
Table 3.3 shows how each of TagIt's parameters impact its performance. For instance, r makes a trade-off between false-negative and the added delay, since the maximum inserted delay is bounded by $\frac{T}{r}(2 - \frac{2}{m})$.

We pick $m = 6$ throughout the thesis. As mentioned, m trade offs between extraction performance and invisibility. The parameter T should be chosen based on the rate of the flow, since false errors are proportional to $T\lambda$. m_ℓ represents a trade-off between extraction time and delay since having small length for m_ℓ will make the synchronization step more time-consuming as discussed in Section 3.1. τ should be used to control the rates of false positive and false negative. Increasing τ improves the false positive, and decreasing it improves the false negative. Also, ρ represents a trade-off between false positive and false negative rates. Figure 3.7 suggests a maximum false extraction rate of 0.15, i.e., an expected number of $30 * 0.15 = 4.5$ false bits when $n = 30$. We therefore pick a conservative value of $\rho = 10$.



(a) Link 1 (b) Link 2 (c) Link 3

Figure 3.9: Extraction result for for various links (no coding).



(a) Link 1 (b) Link 2 (c) Link 3

Figure 3.10: Comparing analysis and simulation results for rate= 10.

Discussion of the Results

To fingerprint a flow, we randomly choose a flow delay from our delay database, and a flow from the Poisson distribution database. We embed the fingerprint on the flow, and then add the delay to the flow. We finally use the extractor (who knows the fingerprinting key) on the noisy flows. We embed various number of bits per interval on various links as they have different noise standard deviations; specifically, n_{bit} is 5, 2, and 1 for links 1, 2 and 3, respectively.

Figure 3.9 shows the extraction results for the 3 selected links. As expected, increasing slot length increases the extraction rate. Also, higher packet rates result in better extraction with the same parameters, as they offer more packets to be fingerprinted.

Comparing with analysis. Figure 3.10 compares our simulation results with that of our analysis from Section 3.2. As can be seen, our analysis presents an upperbound on the experimental extraction rate. This is due to the imperfection of our modeling of network noise (as discussed earlier), and also artifacts not included in our model such as bursty errors due to temporary network conditions. Note that our analysis does *not* aim to tightly predict the performance; instead, it intends to 1) demonstrate how various parameters trade off TagIt 's features (like FP, invisibility, etc.), and 2) enable picking the values of TagIt's parameters to be used in the experiments for various traffic rates and network conditions.

Table 3.4: Running TagIt on live traffic.

	λ	T (ms)	r	m_ℓ (ms)	R_{move}	r_c	Ave. ext.	Ave. ext. after coding
Link 1	10	1800	600	0.5	1	2/3	0.946	1
	100	270	90				0.951	1
	λ	T (ms)	r	m_ℓ (ms)	R_{move}	r_c	Ave. ext.	Ave. ext. after coding
Link 4	10	1800	15	20	1	2/3	0.82	0.996
	10	1920	16		0.6	1/3	0.943	0.98
	10	2160	18		1		0.96	1
	100	2160	18		0.6		0.93	1

3.4 Real-World Implementation

We implement TagIt on Linux to fingerprint live network flows. Our implementation is done in C++ using NFQUEUE and libnetfilter libraries [4]. We did our experiments on two different links: Link 1 from simulations, and Link 4: Campus-California with standard deviation of delay in the range $9.773 - 12.347$ ms. We use the communication system toolbox from Matlab to implement our coding algorithms. We use two coding algorithms in our experiments: Reed-Solomon codes (RS) and Convolutional codes [84].

On each of the links, we try various coding algorithms and rates to achieve an extraction rate close to 1. Note that our choice of coding algorithms and parameters are not optimal, but to demonstrate the possibility of compensating for remaining network errors through coding. We leave the investigation of optimal codes to future work.

In link 1, we embed $n_{bits} = 5$ bits in each fingerprint interval. Therefore, we use an Reed-Solomon code to correct the errors on this link. This is because RS codes are best for correcting bursty errors. We aim at fixing 5 bits of errors; we therefore use a rate 2/3 RS code with parameters $(n = 31, k = 21)$.

On the other hand, in link 4 we embed $n_{bits} = 1$ bit per interval. We therefore use the Convolutional code on this link, which works better on sparse errors (however, it works best on longer streams). Similarly aiming to correct 5 bits of errors, we try two rates of 1/3 and 2/3 with constraint lengths of 3 and [5,4], respectively. We use a Truncated termination mode for our decoder.

Table 3.4 summarizes the results on live traffic for various parameters and flow rates on the two links. The results are averaged for 100 fingerprinted flows. We pick various parameters for different tests to demonstrate the impact of parameters on performance.

Table 3.5: Kolmogorov-Smirnov with confidence level=95% ($\lambda = 10, r = 10$).

Link	T (ms)	m_ℓ(ms)	R_{move}	Pass rate			Ext. rate
				$n = 30$	$n = 20$	$n = 10$	
Link 2	360	6	0.6	1			0.52
	480	8		0.99	1	1	0.688
	600	10		0.92	0.98	1	0.745
	360	6	0.8	1			0.56
	480	8		0.94	0.99	1	0.71
	600	10		0.60	0.85	0.98	0.819
	360	6	1	0.87	0.99	1	0.68
	480	8		0.25	0.53	0.91	0.85
	600	10		0.0	0.15	0.64	0.89

3.5 Fingerprint Invisibility

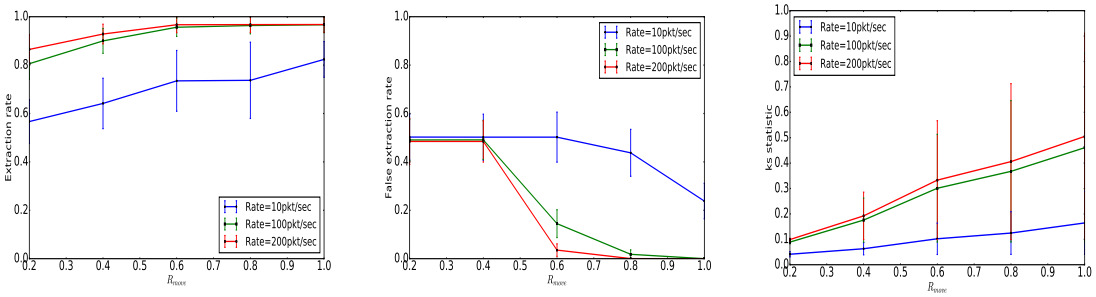
Kolmogorov-Smirnov Similarity Test

The Kolmogorov-Smirnov test compares the empirical distribution of two samples and based on their maximum distance decides if they are from the same distribution. We use the KS test to distinguish between TagIt fingerprinted flows and their non-fingerprinted versions. Table 3.5 summarizes our experiments for various parameters and flow rates. For each set of parameters (each row of the table) we generate 100 fingerprinted flows and compare each fingerprinted flow with its non-fingerprinted version using the KS test. The table shows the Table 3.5 shows the result of KS test on different links with their confidence level. The "Pass rate" column shows the fraction of fingerprinted flows (out of 100) that are declared to be from the same distribution as their non-fingerprinted version by the KS test (a higher pass rate means better invisibility). The table also shows various tradeoffs between invisibility and extraction performance, e.g., for various interval lengths, number of fingerprint bits, and the ratio of the moved packets.

Also, as described earlier, R_{move} (the fraction of packets being delayed) trades off fingerprinting performance with invisibility. Figure 3.11 shows the impact of R_{move}. As can be seen, increasing R_{move} improves fingerprinting performance (increases extraction rate and reduces false positive) at the price of degrading invisibility (reducing the KS statistic)

Single Flow Invisibility

We also use the delay imposed during fingerprinting as another metric to evaluate invisibility. Such an attack can be performed by an adversary who feeds his flows into the fingerprinting system in order to measure the difference. The worst-case delay per packet inserted by TagIt fingerprinter is given by $max_{delay} = \frac{T}{r}(2 - \frac{2}{m})$. Table 3.6 shows the average of max_{delay} for different r.



(a) Extraction rate (b) False Positive (c) Invisibility

Figure 3.11: Impact of R_{move} on invisibility and extraction rate (slot length= $8ms$).

Table 3.6: Average and maximum fingerprint delay per packet for different r.

r	T/r (ms)	Average delay (ms)	Maximum delay (ms)
10	180	84.11	300
20	90	40.55	150
30	60	28.15	100
60	30	14.44	50

Multi-flow Invisibility

We also evaluate TagIt's invisibility to the MFA attack of Kiyavash et al. [80], discussed earlier. As mentioned before, TagIt uses randomization to defeat MFA. Because of TagIt use of randomization, fingerprinted flows will have different patterns even when the fingerprint key is the same. Figure 3.12 shows the cumulative histogram for 10 flows in an interval before and after fingerprinting, for different slot lengths. As can be seen, even the use of 10 flows can not leak the fingerprint since TagIt's randomization spreads packets evenly across various slots.

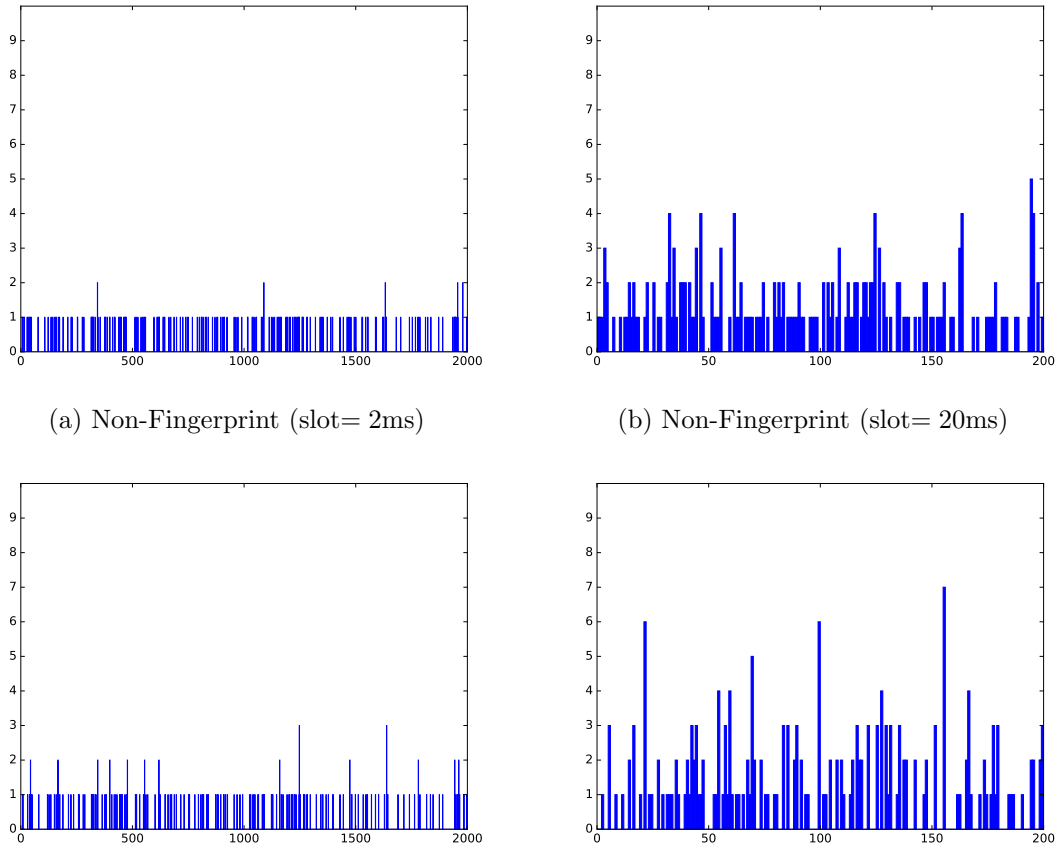
Fingerprinting Key Entropy

A trivial way to attack a fingerprinting system is to guess its fingerprinting key. Therefore, the key should have a high-entropy making it practically infeasible to be guessed. When TagIt embeds 1 bit per interval, the space of fingerprinting keys is the set of all possible permutation functions Π^0 and Π^1. Thus, the number of possible permutations is given by:

$$k_{space} = (((m)!)^{rn})^2 \tag{3.13}$$

$$\log_2 k_{space} m \tag{3.14}$$

Evaluating this for the parameters of Table 3.4 results in a key with over 7552 bits of uncertainty, which is significantly costly to be guessed.



(a) Non-Fingerprint (slot= 2ms)

(b) Non-Fingerprint (slot= 20ms)

(c) Fingerprinted (slot= 2ms)

(d) Fingerprinted (slot= 20ms)

Figure 3.12: Cumulative histogram of 10 flows, non-fingerprinted and fingerprinted with different slot lengths (quantization step in each figure is half of slot length). (a) and (c) have 2000 points in the X axis because slot length is 2 vs. 20 in (b)and (d), and interval length is $2sec$.

3.6 Conclusion

We designed the first blind flow fingerprinting system called TagIt. We extensively evaluated the performance and invisibility of TagIt through theoretical analysis. We also evaluated TagIt through extensive simulations on network traces as well as through experimenting over live network traffic.

Chapter 4

FINN Flow Fingerprinting

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 [71, 117, 72] for fingerprinting, which requires careful selection of features to manipulate for embedding the fingerprints.

4.1 Design of FINN Fingerprinting System

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 [102].
- **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

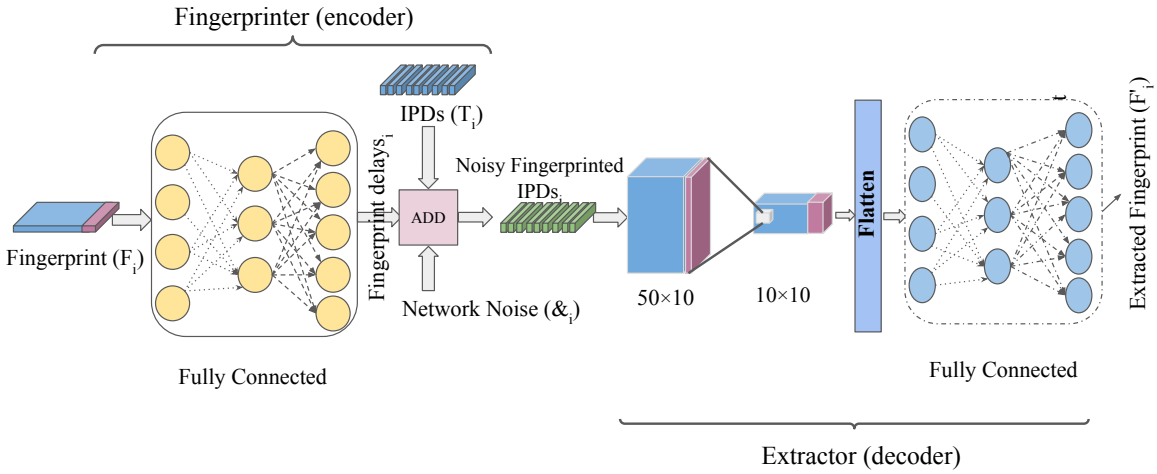


Figure 4.1: The network architecture of the FINN.

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 4.1 shows the architecture of these two components.

We show the input of the model as following:

$$Input = [F_i, \&_i] \tag{4.1}$$

Where T_i is the vector of inter-packet-delays (IPD) of the flow i, F_i is the fingerpint 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 layes 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 4.2. The fingerprinting delays are added to the T_i and $\&_i$ to create the noisy fingerprinted IPD, which would be the input for the decoder.

Table 4.1: 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

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.

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.

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 [79] to minimize the loss.

$$\mathcal{L} = \frac{I_w}{n} |\sum_{i=1}^{n} \hat{I}_i| + \frac{K_w}{n} \sum_{i=1}^{n} \sum_{c=1}^{K} -y_{o,c}^{i} \log p_{o,c} \tag{4.2}$$

4.2 Experimental Setup

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

Dataset

As we explained in Section 4.1, FINN consists of two main entities: encoder (fingerprinter) and decoder (extractor). The encoder takes 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 [93] 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:

$$\text{fingerprinting delay}_i = \{\alpha_{i0}, \alpha_{i1}, ..., \alpha_{i(n-1)}\} \tag{4.3}$$

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:

$$\sum_{n=0}^{n=(j-1)} \alpha_{in} \tag{4.4}$$

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.

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.

Table 4.2: 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

Choosing FINN's Hyperparameters

Table 4.1 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 4.1, 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 4.2 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.

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.

4.3 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.

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 4.3 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 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.

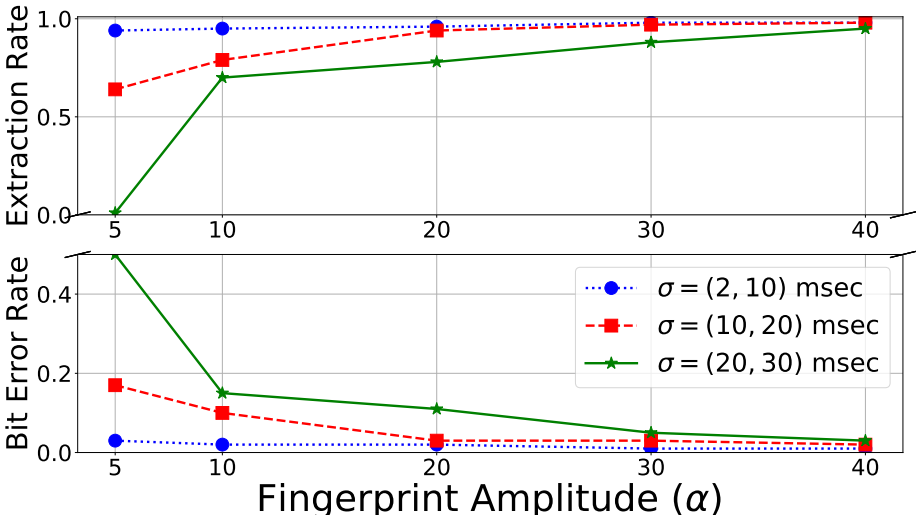


Figure 4.2: Result of increasing α on performance of FINN fingerprint in different network conditions (flow length = 100, $\ell = 2^{10}$).

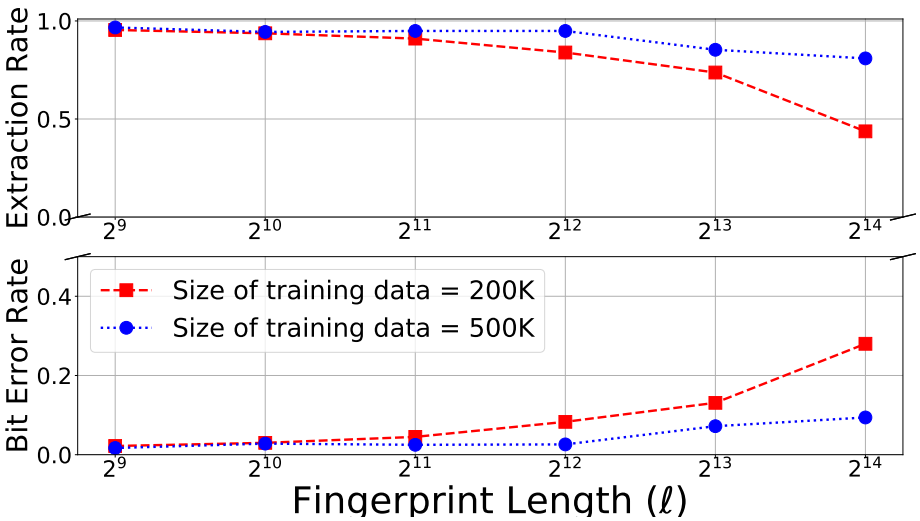


Figure 4.3: Result of increasing ℓ and number of training data on performance of FINN fingerprint(flow length = 100).

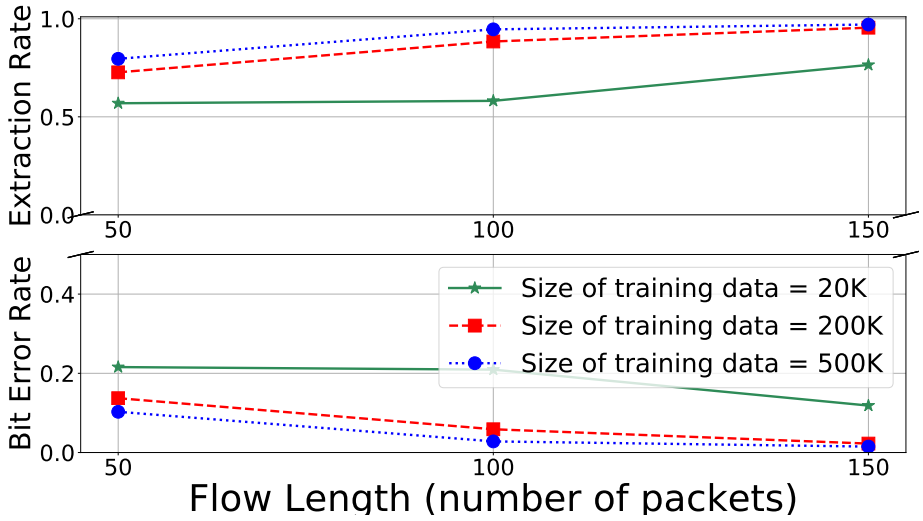


Figure 4.4: Result of increasing flow length and number of training data on performance of FINN fingerprint (fingerprint length = 2^{10}).

$$\begin{aligned} \log_2(2^{12}) \times 0.949 = 11.38 \\ \log_2(2^{9}) \times 0.966 = 8.69 \end{aligned} \tag{4.5}$$

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.

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 4.2 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.

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 4.4 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

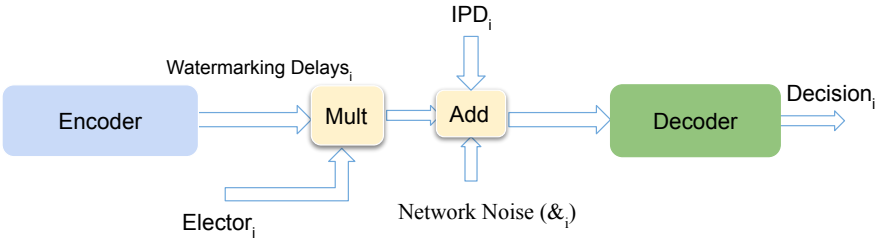


Figure 4.5: FINN watermarking system.

Table 4.3: 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 4.2	

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.

FINN as a Watermark

In this experiment, we want to evaluate our algorithm's performance when used as a watermark. 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 4.5 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

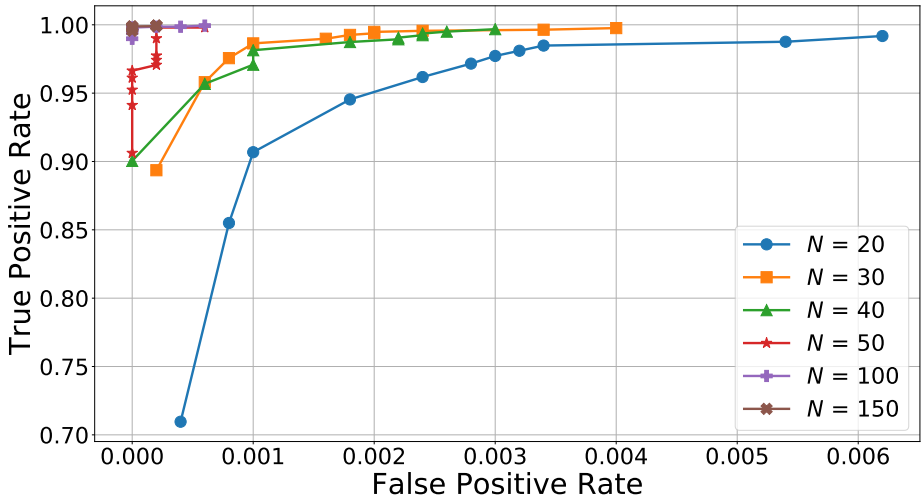


Figure 4.6: Performance of the FINN watermark for different flow length.

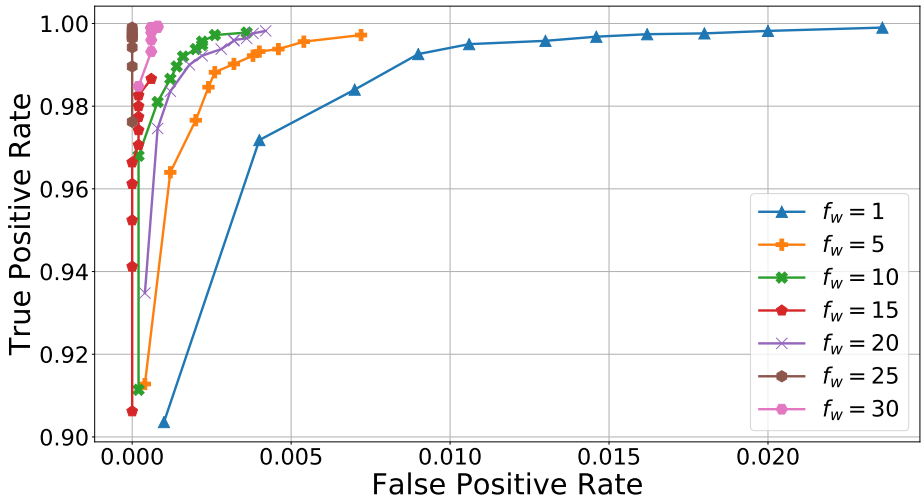


Figure 4.7: Selecting weight of false class by fixing the weight of class of true ($t_w = 1$ and flow length = 50, and f_w is increasing).

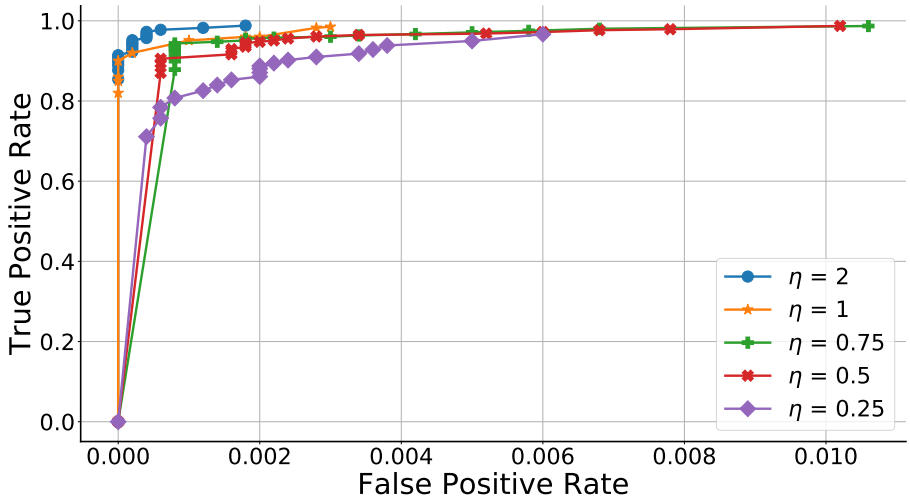


Figure 4.8: Performance of the watermark for different η. η is the ratio of watermark amplitude to network noise (flow length=50).

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.

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.

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 4.8 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 [72]; 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 4.7 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.

4.4 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 `libnetfilter_queue` library (version 1.0.4) [4]. 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 [3] 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.

Impact of Fingerprint Length on Performance

As we explained in Section 4.2, 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 4.3 showed that this training size was enough to have a generalized model. Figure 4.10 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:

$$0.962 \times \log 2^{12} = 11.54 \tag{4.6}$$

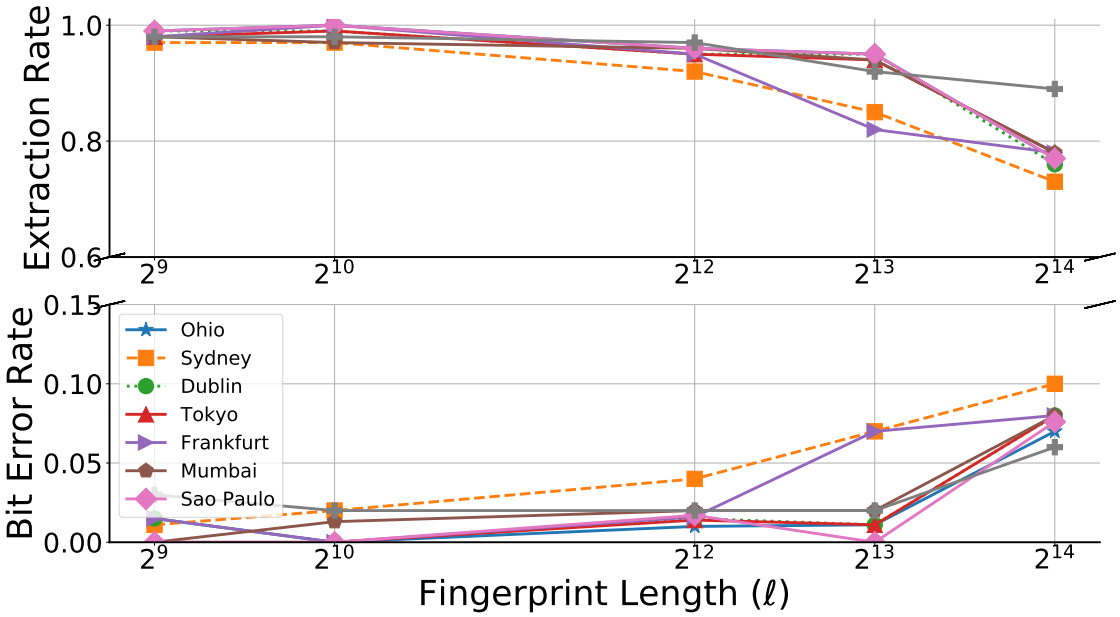


Figure 4.9: Performance of the real-time FINN fingerprint experiment from campus to various nodes (Wireless, flow length = 100).

Experiment 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 4.11 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 4.10. 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 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 4.3 shows, would improve the results.

Comparing FINN with Previous Methods

Here, we want to compare the performance of FINN with previous fingerprinting systems. TagIt [117] 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. To compare our system's performance to TagIt, we implement it in

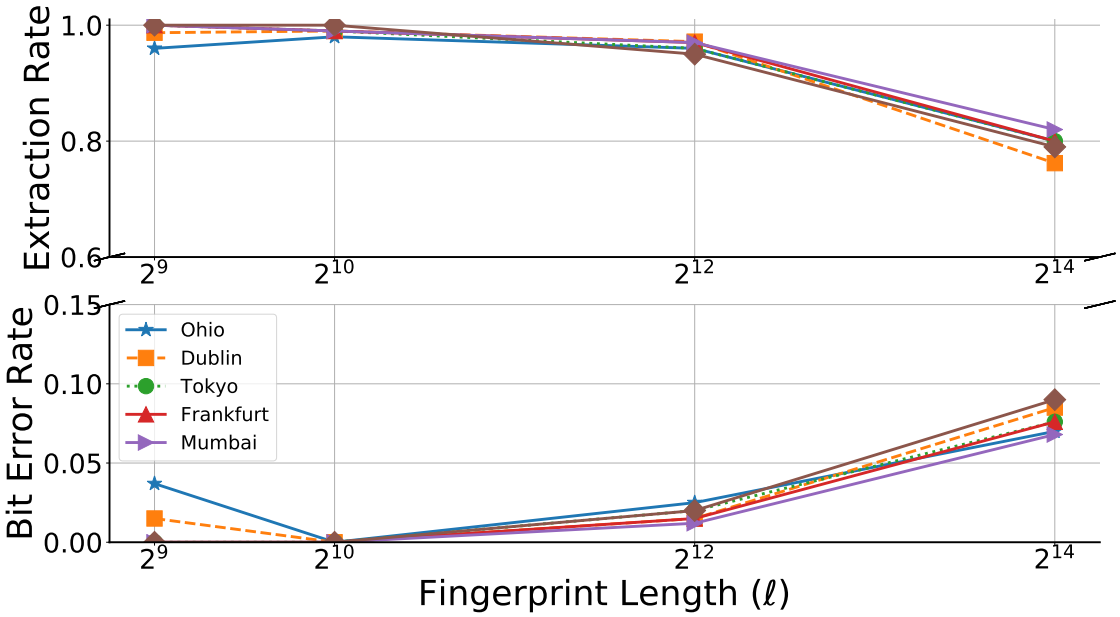


Figure 4.10: Performance of the real-time FINN fingerprint experiment from Bangalore to various nodes (Wireless, flow length = 100).

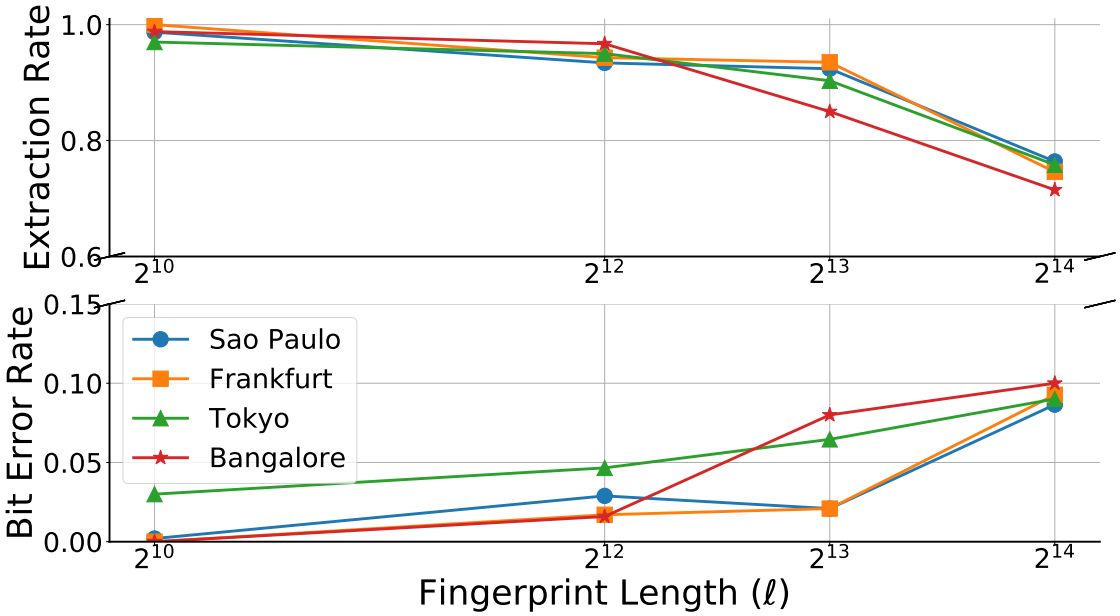


Figure 4.11: Performance of the real-time FINN fingerprint experiment from campus to various nodes (Cellular, flow length = 100).

Table 4.4: Performance comparison of TagIt and FINN.

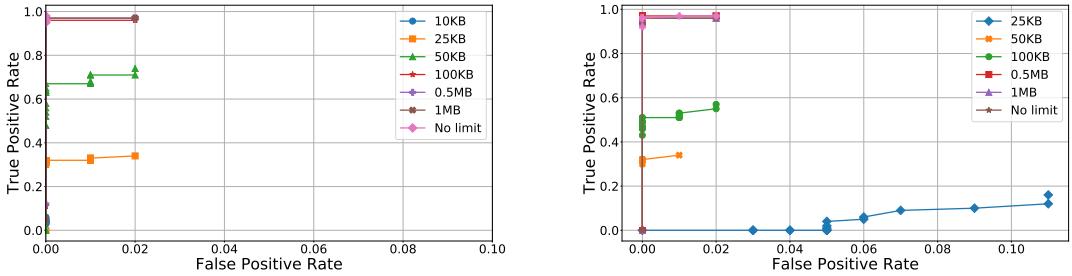
Method	ER	BER	Average # of bits	# of flows
FINN	96.2	2.1%	11.54	
TagIt (T=1620)	90.1	2.2%	5.32	190
TagIt (T=2160)	96.8	1.4%	4.13	

real-time on our Banglore 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 4.4 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).

FINN as a Watermark

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 4.5 and 4.6 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. Figure 4.12 shows 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 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.



(a) Frankfurt-Bangalore link.

(b) São Paulo-Bangalore link.

Figure 4.12: Performance of the real-time FINN watermark experiment on different bandwidths (flow length = 50).

Table 4.5: 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, Frankfurt)	0.96	
Campus - (Mumbai)	0.94	
Bangalore - (Dublin)	0.98	0
Bangalore - (Tokyo)	0.97	
Bangalore - (São Paulo, Frankfurt)	0.96	
Bangalore - (Ohio)	0.94	

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 [100] 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. Furthermore, Fancy [70] is a non-blind fingerprinting approach. Therefore, it is not fair to compare our work with it. Since non-blind techniques have access to more information, so they

Table 4.6: 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	

intuitively offer better performance.

FINN's Scalability

We compute the resources required for a stepping stone detection scenario using FINN for the CICS building at UMass. 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 subnetwork 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.

4.5 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 [117, 72] to detect watermarks added to IPDs in a flow.

Generative Adversarial Network

Generative Adversarial Network is a class of machine learning introduced by Ian Goodfellow et al. [64]. 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 [101, 102]. This method has been used in [101] to generate Laplace distribution. The generator and extractor

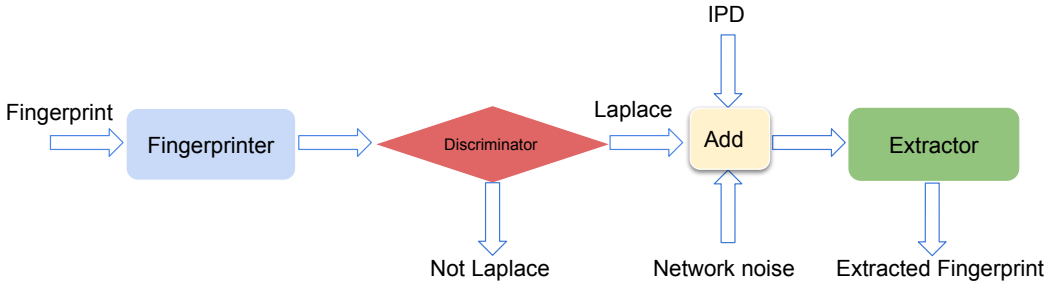


Figure 4.13: Using GAN to improve invisibility.

Table 4.7: Discriminator model hyperparameters.

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

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 4.13 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.

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:

$$D_{n,m} = \sup_x |F_{1,n}(x) - F_{1,m}(x)| \tag{4.7}$$

$F_{1,n}$ 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

$$D_{n,m} > c(\alpha)\sqrt{\frac{n+m}{nm}} \tag{4.8}$$

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

$$c(\alpha) = \sqrt{-0.5 \ln \alpha} \tag{4.9}$$

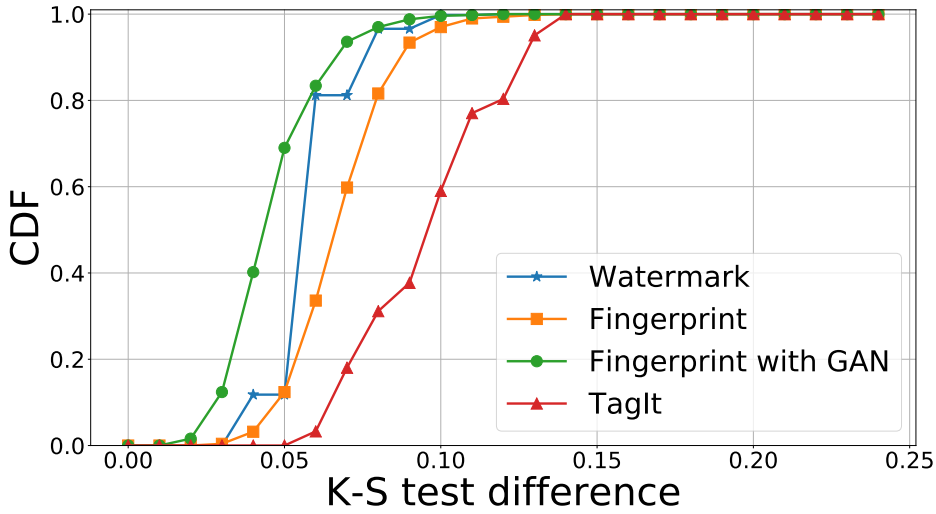


Figure 4.14: K-S Test Difference for different methods (Fingerprint: flow length = 100, fingerprint length = 2^{10}, Watermark: flow length = 50).

experiment, 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 4.14 shows the result of the Kolmogorov-Smirnov test on our data. Using (4.8), we compute the KS threshold for 0.95 confidence interval, which is 0.19205. As the Figure 4.14 shows, only 1 of our flows(0.2%) fails to pass the K-S test.

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.

Table 4.8: 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

DBSCAN is a density-based Spatial clustering that defines the clusters as connected dense regions [2]. 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 4.15 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.

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 4.8 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.

4.6 Conclusions

In this thesis, we introduced the first blind IPD-based 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 evaluate the performance of our system thorough simulations and experiments over a live network. Our experiments evaluate the impact of different parameters, and find the optimum for each one. Also, we compute the capacity of FINN, which 0.96 bit in every ten packets and is near twice is 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. In particular, this is important since it prevents us from re-training our model for every connection with different network jitter. Finally,

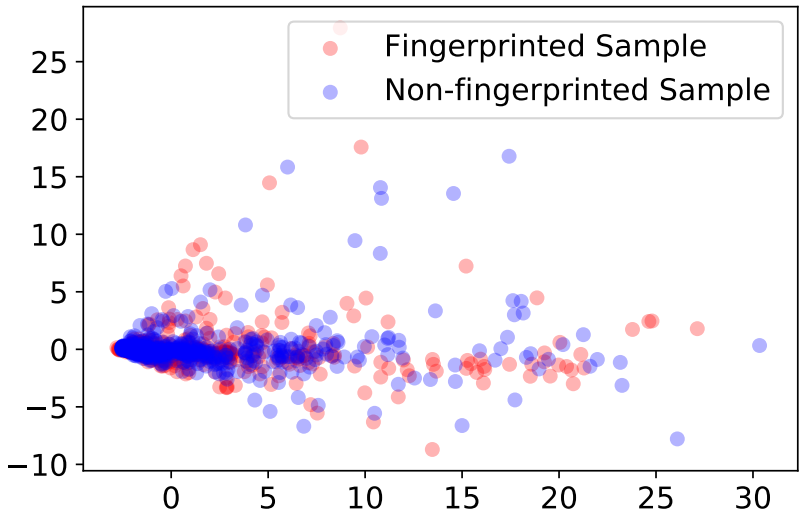


Figure 4.15: We reduce the dimensionality of the samples using principal component analysis (PCA) to represent it in two dimensions.

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 fingerprint. Here, we considered the situation that we know the incoming flows are fingerprinted and attempted to extract the embedded messages. For the future work, we want to work on scenarios that this is not the case.

Chapter 5

Bitcoin Identification

Bitcoin and similar blockchain-based currencies are significant to consumers and industry because of their applications in electronic commerce and other trust-based distributed systems. Therefore, it is of paramount importance to the consumers and industry to maintain reliable access to their Bitcoin assets. In this thesis, we investigate the resilience of Bitcoin to blocking by the powerful network entities such as ISPs and governments. By characterizing Bitcoin's communication patterns, we design classifiers that can distinguish (and therefore block) Bitcoin traffic even if it is tunneled through an encrypted channel like Tor and even if Bitcoin traffic is being mixed with background traffic, e.g., due to browsing websites. We perform extensive experiments to demonstrate the reliability of our classifiers in identifying Bitcoin traffic even despite using obfuscation protocols like Tor Pluggable Ttransports. We conclude that standard obfuscation mechanisms are not enough to ensure blocking-resilient access to Bitcoin (and similar cryptocurrencies), therefore cryptocurrency operators should deploy tailored traffic obfuscation mechanisms.

5.1 Background on Bitcoin

Bitcoin is a decentralized peer-to-peer cryptocurrency based on a white paper published in 2008 and pseudonymously signed Satoshi Nakamoto [99]. Unlike regular digital money like credit cards and money wiring, Bitcoin does not depend on a central authority for the validation of transactions. Instead, Bitcoin operates in a peer-to-peer fashion. Bitcoin transactions transfer money between Bitcoin clients. Each user is identified by an address. This address is a public key for which the user has the corresponding private key. This allows only the owner of the key to sign transactions that transfer funds out of her balance.

The transactions are broadcasted on Bitcoin's p2p network. The right order of transactions in a transactions series is crucial in order to thwart double spending and similar attacks. One classical way to keep the order of transactions is using a centralized authority. Bitcoin does so in a p2p fashion on a public ledger called the *block*

chain, which includes all Bitcoin transactions in batches called *blocks*. Each block is a set of transactions and a *proof of work*. A proof of work is a piece of data which is time-consuming and costly to generate. However, verifying proof of work is easy and is done to check certain requirements. Each block is valid if and only if its transactions and power of work are valid. The block then is distributed in the network and receiving nodes update their ledgers accordingly. Currently, the maximum size of each block is 1 MB.

In Bitcoin it is crucial that each peer's ledger be up-to-date. The peer-to-peer network has the responsibility of updating the public ledger. Only valid transaction are inserted into the ledger. To keep all peers in sync, all new transactions and blocks are continuously broadcast to all users.

Bitcoin's P2P Network

Bitcoin nodes form a random network and they connect to each other over unencrypted TCP connections. Since connections are not authenticated, peers just keep a list of their connection IP addresses. Blocks and transactions are propagated by gossip. To avoid DoS attacks, Bitcoin nodes only forward valid blocks and transactions; invalid blocks are discarded.

Bitcoin peers can be split into two types: routable and non-routable. The former are capable of accepting incoming connections, and the latter are not able to do so, for example they are behind NATs or firewalls. However, it is worth mentioning that the official **Bitcoind** software does not precisely split its functionality between routable and non-routable. Each node has up to 125 connection, up to 8 of which are outgoing. A node stays connected to a neighbor until it restarts or drops, in which case the node tries to replace it [28].

Bitcoin Protocol Messages

In this section, we introduce the key protocol messages in Bitcoin that are essential in understanding the behavior of Bitcoin traffic. We have listed the comprehensive roster of Bitcoin protocol messages in Table 5.1. We divide Bitcoin protocol messages in two groups: *synchronization messages* and *block-related messages*. Synchronization messages are referring to messages used for propagating user's addresses and transaction in the Bitcoin network. On the other hand, block-related messages focus on how blocks are advertised and how they are sent into the network.

Synchronization Messages

These messages are aimed at keeping Bitcoin peers synchronized with the rest of the Bitcoin network.

`addr`:Each peer advertises the information and IP addresses of other peers via `addr` message in the network. `addr` message contains a count and list of other peers IP addresses. Each IP address is accompanied by a timestamp showing its freshness. When

Table 5.1: The list of Bitcoin communication messages.

Message	Description
`version`	Advertise the node's version. No further communication is possible until both peers have exchanged their version.
`verack`	Reply to the `version` message.
`addr`	Send information about the known nodes of the network.
`inv`	Sent to advertise the knowledge of the peer about the known objects. It can be received unsolicited, or in reply to `getblocks`.
`getdata`	Sent in response to the `inv` message to retrieve information about the content of an object.
`notfound`	If the receiver of `getdata` cannot return the requested information, it respond with `notfound`
`getblocks`	It return an `inv` message with the list of block after the specified block in `getblocks` request
`getheaders`	It return a `headers` message with the list of block after the specified block in `getblocks` request
`tx`	Sent to describes a Bitcoin transaction in response to a `getdata` message.
`block`	`block` message is sent in response to a `getdata` message
`headers`	Return a list of block headers, in respond to `getheaders`
`getaddr`	A node sends `getaddr` to ask about the known peer from other peers
`mempool`	It asks about the transaction in mempool of other peers
`ping`	Show the TCP/IP connection is still valid.
`pong`	Response to `ping` message.
`reject`	It show a message has been rejected
`sendheaders`	Let other peers to send headers without `inv` message
`sendcmpct`	Let other peers to send compact blocks
`cmpctblock`	It used in stead of `block`, to send cmpctblock
`getblocktxn`	It indicate missing block in compact block transaction
`blocktxn`	To send missing block in compact block transaction

a peer receives a list of addresses from other peers, it has the choice of forwarding any number of them. The peer chooses the sending addresses based on the following criteria: 1) The number of IP addresses in the received message should not exceed 10, and 2) the timestamps should not be older than 10 minutes. This mechanism is applied for helping in peer discovery.

`inventory(inv):` Peers send `inv` to advertise their knowledge about the known objects, like transactions and blocks. Each `inv` message consists of a number of inventory entries and the inventory vectors itself. It can be received unsolicited, or in reply to `getblocks`. Inventory vectors are used to notify other nodes about objects they have or data which is being requested. Inventory vectors consist of the type of objects and the hash of the object.

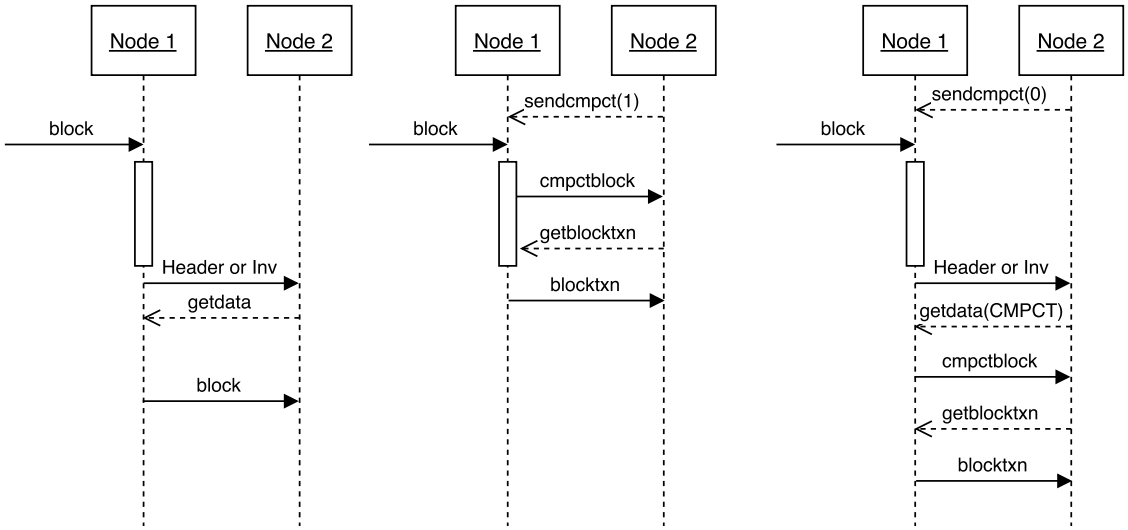


Figure 5.1: Comparing different relaying in the Bitcoin network

`getdata:` A peer sends a `getdata` message in response to the `inv` to retrieve information about the content of an object. This object could be a block or a transaction.

`tx:` This message describes a bitcoin transaction in response to a `getdata` message. Each transaction is stored in a memory pool. If a received transaction is already in the pool, or it is included in one of the blocks in the main block-chain, it get discarded.

Block-related Messages

Such messages are used to exchange Bitcoin blocks among the peers. The current Bitcoin network supports two ways of blocks propagation, full block and compact block propagation. Figure 5.1 demonstrates the flow of message communication in these two ways.

Full block propagation. The sender node first validates the block completely and then advertises its possession of the block by an `inv` message. A receiving peer that doesn't have the block, asks for it by sending `getdata` message. Finally, the sender node sends the block via a `block` message. Sending full block in the network is wasting network bandwidth since we are re-sending all of the transaction and nodes have some of transaction in their memory pool. The messages transmitted in this mode are:

`block:` It consist of block version information, previous block hash, root of a Merkle tree collection which is a hash of all transactions related to this block. Sending the new block forwarded through all the network.

Compact block propagation. From the middle of 2016 in `0.13.0` version, Bitcoin protocol began to forward blocks as compact blocks which means instead of forwarding full blocks in network, only sketches of blocks are sent. A sketch include a 80-byte block's header, the short transaction IDs used for matching already-available

transactions, and the missing transactions. After receiving the compact block, the receiving peer tries to reconstruct the block at its end, from the previously received transactions and the one just received in compact block. In this way the waste of sending each transaction twice is reduced. The advantage of compact block relaying is reducing the spikes in the bandwidth and also reduce propagation delay. The messages transmitted in this mode are:

`sendcmpct:` This message is is used to inform the receiving peer about the communication mode the sending peer has chosen. If the first byte of the message is set to 1 the sender is indicating that it wants to receive blocks as soon as possible and is working in a high-bandwidth mode. If the first byte of the message is set 0 the sender is indicating that it wants to minimize bandwidth usage as much as possible and is working in a low-bandwidth mode.

`cmpctblock:` This message is presenting a sketch of block.

`getblocktxn:` This message is used to request missing transactions by sending a list of their indexes.

`blocktxn:` This message is used to provide some of the transactions in a block, as requested.

Compact block relaying works in high and low bandwidth settings. When bandwidth is high, the receiving peer doesn't oblige the sending peers to ask for permission first. So, multiple peers can send the compact block to receiving node. Then at last the sender node sends the missing transactions by `blocktxn` message. It is worth mentioning Bitcoin works in high bandwidth mode with up to 3 peers. However, in low bandwidth mode, since bandwidth is its bottleneck, the receiving node oblige other nodes to ask for permission first. So, the sender first advertises block possession by an `inv` message. Then, the receiving node asks for the compact block by `getdata` and the sender will send the compact block by `cmpctblock` message. At last, if there is any missing transaction, the receiving node will ask for it by `getblocktxn` and the sender will send those transactions via `blocktxn`.

5.2 Characterizing Bitcoin Traffic

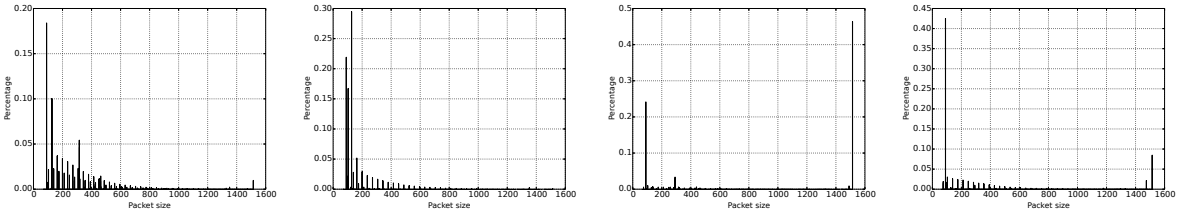
In this section, we demonstrate the unique features of Bitcoin traffic. We will show that such unique traffic patterns of Bitcoin makes it possible to reliably distinguish it from those other protocols even despite encryption and mixture with background traffic. We will use our characterization to design classifiers for Bitcoin in the following sections. All of our experiments in this section follow the experimental setup and datasets described in Section 5.4.

Proportion and Distribution of Messages

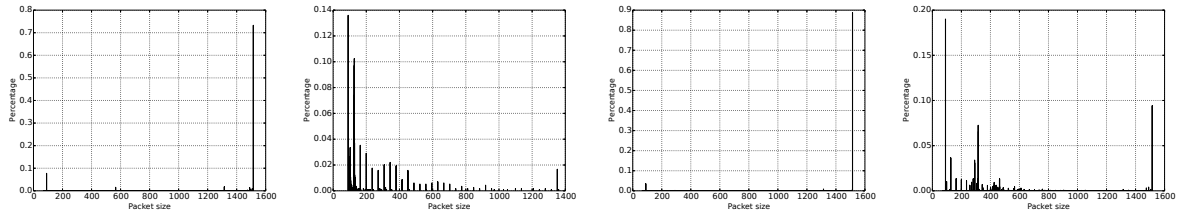
Bitcoin peers generate various kinds of messages. We show that the distribution and sizes of such messages are unique to the Bitcoin protocol, distinguishing Bitcoin traffic

Table 5.2: Number and percentage of each message in a 31 days Bitcoin traffic in compact block relaying.

Message	Packets per minute	Proportion	Min. size	Max. size	Ave. size
`inv`	173.17	27.240%	67	1514	791.87
`getdata`	13.16	2.070%	68	1514	700.33
`block`	0.94	0.330 %	74	1514	772.74
`sendcmpct`	1.60	0.253%	63	1514	782.88
`cmpctblock`	2.02	0.319%	67	1514	789.21
`getblocktxn`	0.02	0.003 %	90	1360	367.67
`blocktxn`	0.77	0.121%	67	1514	777.16
`tx`	277.72	43.688%	66	1514	790.00



(a) `inv` (b) `getdata` (c) `block` (d) `sendcmpct`



(e) `cmpctblock` (f) `getblocktxn` (g) `blocktxn` (h) `tx`

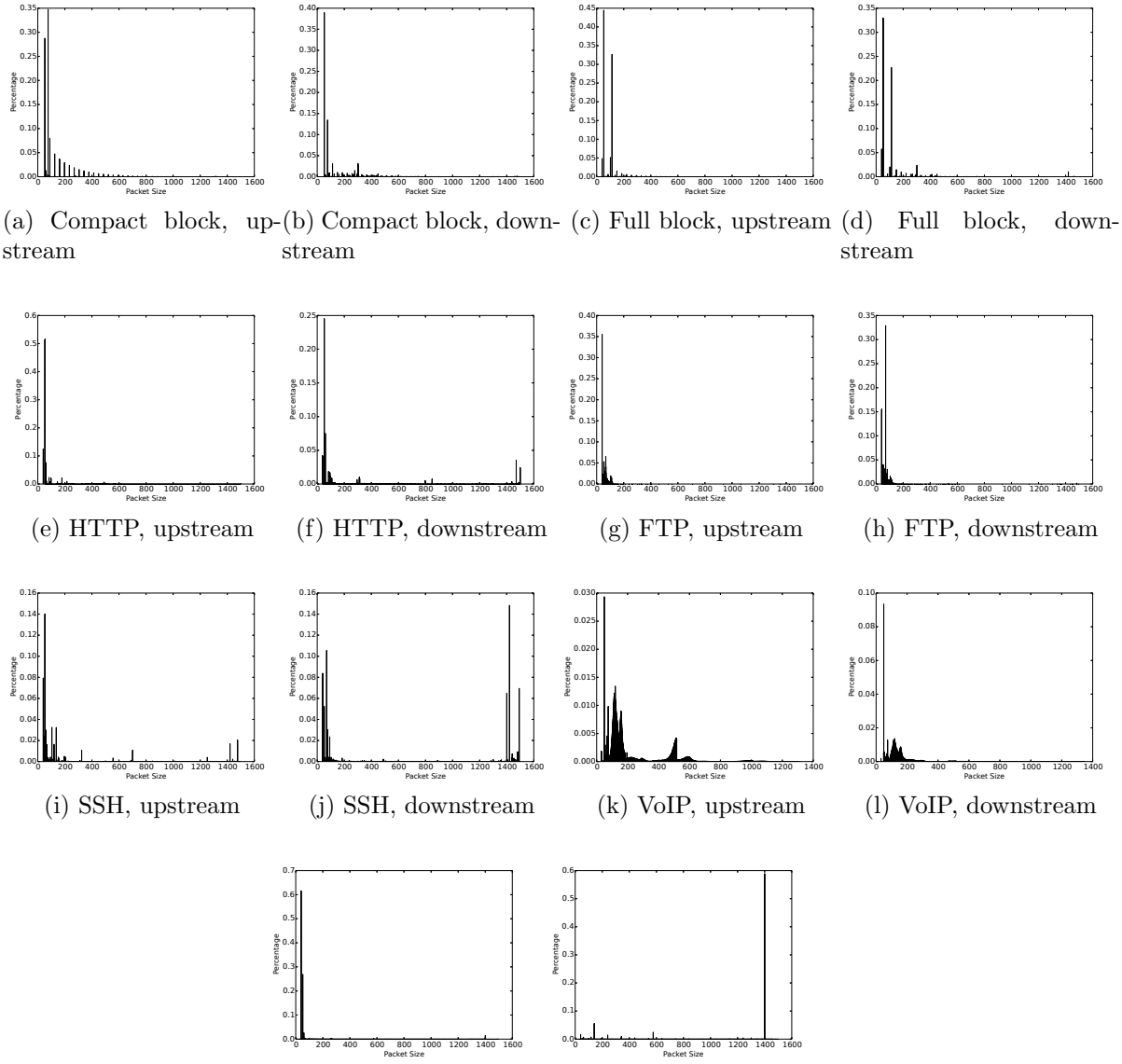
Figure 5.2: Packet size distribution of Bitcoin communication messages in compact block relaying.

reliably from other protocols.

Table 5.2 demonstrates the proportion of different messages in the Bitcoin traffic we collected for 31 days. As can be seen, `tx` and `inv` dominate with 43.6% and 27.2% of all packets, respectively. Therefore, the characteristics of these messages will shape the pattern of a Bitcoin peer's traffic.

Distribution of packet sizes. Figures 5.2(a) to 5.2(h) illustrate the packet size histogram of different types of Bitcoin messages in our collected Bitcoin traffic. As can be seen, each type of message has a distinctive traffic pattern.

Histogram of packet sizes in aggregate traffic. Figures 5.3(a) and 5.3(b) show the histogram of packet sizes in the upstream and downstream directions, re-



(a) Compact block, upstream (b) Compact block, downstream (c) Full block, upstream (d) Full block, downstream

(e) HTTP, upstream (f) HTTP, downstream (g) FTP, upstream (h) FTP, downstream

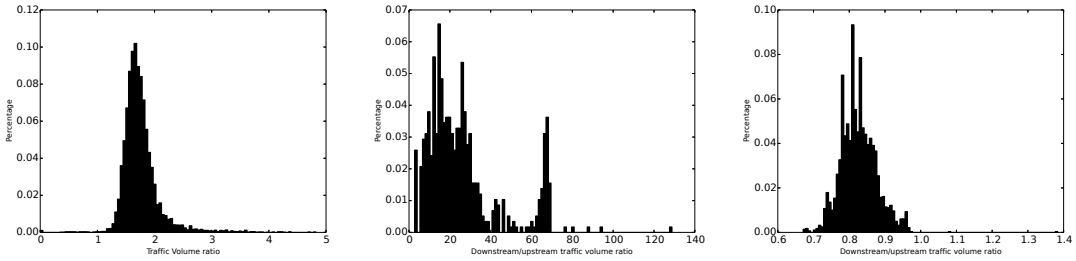
(i) SSH, upstream (j) SSH, downstream (k) VoIP, upstream (l) VoIP, downstream

(m) BitTorrent, upstream (n) BitTorrent, downstream

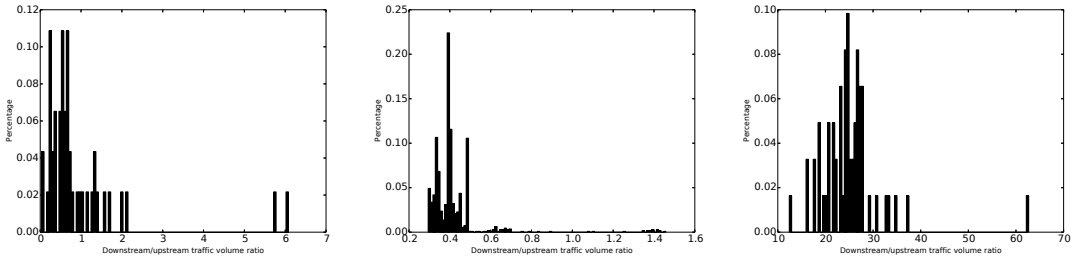
Figure 5.3: Upstream and downstream packet size distribution of Bitcoin and several popular protocols.

spectively, in compact block relaying. As mentioned before, `tx` and `inv` dominate the messages sent by a typical Bitcoin peer, therefore, their sizes (shown in Figures 5.2(a) and 5.2(h)) strongly shape the histogram of Bitcoin traffic, making them uniquely distinguishable from other protocols.

We also show histograms of Bitcoin traffic in the full block relaying mode in Figures 5.3(c) and 5.3(d). These histograms have a larger spike close to the MTU, unlike the case of compact block relaying. These are because of sending the whole block



(a) Bitcoin (b) HTTP (c) FTP



(d) SSH (e) VoIP (f) BitTorrent

Figure 5.4: Ratio of downstream to upstream traffic volume for 5 minutes of traffic.

instead of a sketch of it in the full block relaying.

Comparison to other protocols. Figures 5.3(e) to 5.3(n) show histograms of other popular protocols, collected as described in Section 5.4. Note that we look at the traffic after going through an encryption tunnel, e.g., a VPN or SSH tunnel, so the histogram includes the (small) TCP ACK packets.

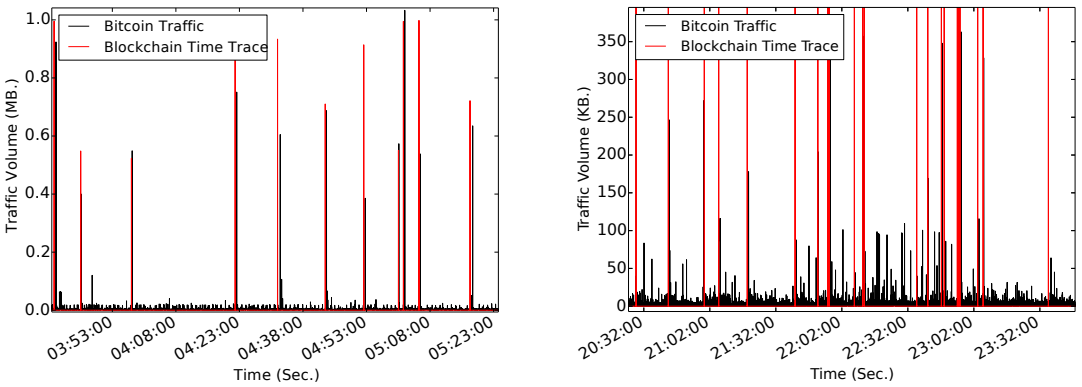
As we can see, the packet size distribution of Bitcoin is uniquely different from these other protocols, since a Bitcoin connection is composed of unique messages with specific size distributions shown before. For instance, the large number of `inv` messages shapes the overall distribution of sizes in Bitcoin traffic.

Ratio of downstream to upstream

We also measured the ratio of downstream to upstream traffic volume which is shown in Figure 5.4(a). Unlike other protocols like HTTP (shown in Figures 5.4(b) to 5.4(f)), Bitcoin traffic has a symmetric traffic volume in upstream and downstream. This is due to the fact that Bitcoin peers broadcast most of the bulky protocol messages they receive such as block and transaction announcements.

Shape of Traffic

Above we showed that the counts and sizes of packets in Bitcoin demonstrate a unique behavior. In addition to that, the shape of traffic in Bitcoin and the volume of traffic received over the time is distinguishable from other protocols.



(a) Full block

(b) Compact block

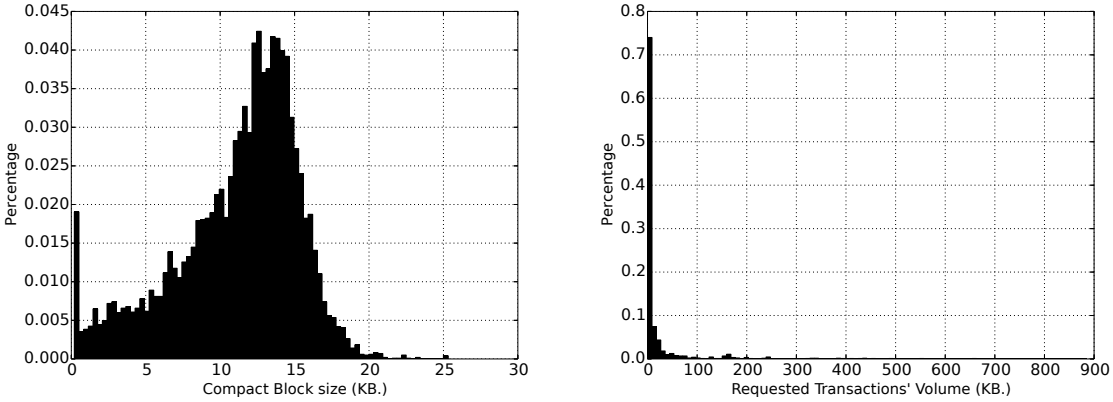
Figure 5.5: Comparing time of each block receive with time of blocks in the block chain.

Full block relaying mode. Figure 5.5(a) shows the traffic of a Bitcoin client operating in the full block relaying mode. As can be seen, the small protocol packets, mostly corresponding to `inv` and `tx` messages appear uniformly over the time. On the other hand, the Bitcoin full blocks appear as large spikes at specific points in time, i.e., once a new block is generated in the network. As can be seen, in the full block relaying mode, the blocks result in big spikes since the client will download the whole block at once, which roughly 1MB.

Compact block relaying mode. In the compact block relaying mode, it would be harder to notice the block spikes, since only a sketch of the blocks is transmitted. In this mode, transmitting a compact block in the network results in smaller spikes of 100KB. Spikes of such small sizes may also occur when unverified transactions are transmitted, which will increase the detection's false positive. Also, a Bitcoin client may operate in the high bandwidth mode, in which the receiver node asks its peers to send new blocks without asking for permissions first. This will lead to more than one peer sending the same block at the same time. This and the large volume of missed transactions result in spikes of more than 100 KB in the traffic.

Figure 5.5(b) illustrates when and how compact blocks appear on a peer's traffic. As can be seen, compact blocks appear at smaller amplitudes than the actual block size, but the behavior is also nondeterministic, since it depends on whether the client has previously received some of the transactions in that block. This intuitively makes detection of compact blocks less reliable than full blocks, as shown later our experiments. We also measure the size of compact blocks by measuring the *length* field of `cmpctblock` messages, which is shown in Figure 5.6(a). As can be seen, most of the compact blocks are as small as 15 KB (in contrast to 1MB full blocks).

We also measure the volume of transactions missing from an announced compact block (we do so based on the payload length of `blocktxn` messages). As described



(a) Histogram of the size of compact blocks (b) Histogram of the size of missing transactions

Figure 5.6: Histogram of compact block mode components.

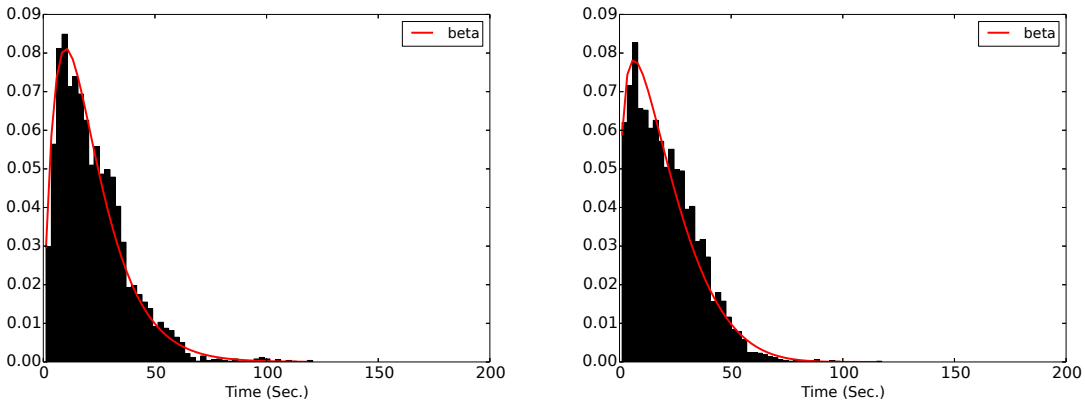
earlier, a Bitcoin client operating in the compact block mode will download such missing transactions. This is shown in Figure 5.6(b). As we can see most of the transactions have volumes less than 100 KB.

Block propagation latencies. The propagation delay in the Bitcoin network is defined as the sum of the transmission delay and the block verification in the receiving node at each hop. The transmission delay is defined as the time to exchange of *inv* and *get data* message and sending the block via *block* message.

We measure block propagation delay by subtracting the receiving time of block message and the time stamp in the header of block message. Figure 5.7(a) and 5.7(b) shows the histogram of 6000 blocks' propagation delay in compact block and full block relaying, respectively. We can model this empirical data using Beta distribution [47]. We choose a maximum for propagation delay such that 99% of beta distribution is below that maximum. According to our measurements this value equals to 65 seconds.

5.3 Designing Bitcoin Classifiers

We use the features described above to build robust classifiers for Bitcoin traffic. We aim for our classifiers to work even in the presence of encryption and background traffic, e.g., when the machine running Bitcoin is used for web browsing and runs other applications, or when the Bitcoin traffic is tunneled over Tor or VPN. For a target connection, our classifiers extract specific features from it, using which decide if the target traffic contains Bitcoin or not. Each of our classifiers extract certain features as introduced below. We evaluate the performance of our classifiers using false positive, true positive and accuracy.



(a) Compact block

(b) Full block

Figure 5.7: Histogram of block propagation delay.

Size-based Classifiers

As we said in section 5.2, the histogram of Bitcoin packet sizes has a specific pattern. According to this fact, we designed classifiers to distinguish Bitcoin traffic from other protocols traffic.

Size Histogram Classifier (`SizeHist`)

This classifier tries to correlate packet sizes of non-Bitcoin traffic with packet sizes of known Bitcoin traffic, the baseline. The classifier first divides the upstream and downstream directions in the given traffic. Then it calculates the histogram of packet sizes in each direction. More precisely, by the histogram of packet sizes, we mean the number of packets for sizes that range from 1 to MTU size. The baseline is also divided into upstream and downstream directions.

In the next step, the classifier calculates the cosine similarity as the correlation metric between the input trace and a series of Bitcoin traces in both upstream and downstream direction. Finally, to detect if the input trace is a Bitcoin traffic, classifier compares the average of correlation values with a threshold. If both of the upstream and downstream averaged correlation values are larger than their thresholds, the input trace is classified as Bitcoin traffic. Thresholds are derived as shown in the experiments section.

Tor-specific Classifier (`SizeTor`)

Some tunneling systems modify packet sizes to prevent information leakage. In particular, Tor reformats traffic into constant-sized segments called *cells*. However, as the size of a cell is smaller than the MTU of IP packets, depending on traffic volume, multiple cells can merge into single packets. This makes the number of single-cell packets and multiple-cell packets specific to different protocols tunneled over Tor.

Algorithm 1 Size Histogram Classifier

1: **procedure** SIZE HISTOGRAM CLASSIFIER
2: $B_{dwn}, B_{up} \leftarrow$ Size histogram of a Bitcoin downstream and upstream traffic
3: $V_{dwn}, Vup \leftarrow$ Size histogram of input downstream and upstream traffic
4: $cor_{dwn} = \frac{B_{dwn}.V}{|B_{dwn}|^2 \times |V_{dwn}|^2}$
5: $cor_{up} = \frac{B_{up}.V}{|B_{up}|^2 \times |V_{up}|^2}$
6: **return** $cor_{dwn}, core_{up}$
7: **end procedure**

Our `SizeTor` classifier aims at detecting Bitcoin traffic tunneled over Tor based on the distribution of single-cell packets.

As shown in Section 5.2, Bitcoin traffic consists of a large number of small-size packets (e.g., due to frequent `inv`) messages. Tor will add padding to these small packets to form a cell. Therefore, the ratio of single-size packets in Bitcoin-over-Tor is larger that regular Tor traffic, e.g., HTTP-over-Tor. Based on this, our classifier compares the ratio of single-cell packets to all packets; if the ratio is smaller than a threshold, the connection is flagged as a Bitcoin connection.

Note that while our `SizeTor` classifier is tailored for Tor, it can be adjusted for other protocols that similarly change the size of packets.

Tailoring for Tor pluggable transports. Through our experiments on Bitcoin traffic over Tor pluggable Transports, we noticed that fraction of packets that have a specific size other than cell-size (for example, size 721 in FTE) are much larger than this value in normal traffic. Furthermore, this value is much larger than the fraction of packets that are in cell size. Therefore, when implementing sizeTor on pluggable transports, we compute the fraction of packets in these specific sizes to differentiate between Bitcoin traffic and others.

Downstream to Upstream Traffic Volume Ratio (`D2U`)

As we discussed in Section 5.2, the ratio of downstream to upstream traffic volume is unique in Bitcoin. This classifier looks at t windows of Bitcoin traffic. The t parameter should be at least 10 minutes to include at least one block in it. Then the ratio of downstream to upstream traffic volume is calculated for the target traffic. If the calculated ratio is larger than a threshold, which is determined empirically, the traffic will be classified as Bitcoin traffic. The threshold is defined in a way to distinguish Bitcoin traffic from other traffic while minimizing false positive. This detection scheme will be effective even in the situation that the underlying system changes the packet sizes and packet timings, since the transmitted traffic volume in both directions would stay the same.

Shape-based Classifier

We demonstrated in Section 5.2 that the shape of Bitcoin traffic is distinct from other protocols. We therefore design a classifier that try to identify (encrypted) Bitcoin traffic based on its shape. The main intuition of our shape-based classifier is looking for changes in the traffic volume of a target user around the times of block announcements. Therefore, we assume that the classifier obtains the times and sizes of Bitcoin blocks, e.g., from the public `blockchain.info` repository, or even by running a local Bitcoin client.

Algorithm 2 illustrates our window shape correlation algorithm. For each confirmed Bitcoin block, the algorithm analyzes the volume of the target traffic during the broadcast time of that block. To do so, the algorithm defines two time windows of size ω_i around the block time, one before $(t_{block_i} - \omega_i, t_{block_i})$ and one after the block time $(t_{block_i}, t_{block_i} + \omega_i)$. The size of the window depends on the size of the block, the target client's bandwidth, etc., as evaluated later. Then, the algorithm evaluates the change in traffic during the two time intervals. For actual Bitcoin traffic with

Algorithm 2 Shape-based Classifier

procedure SHAPE BASED CLASSIFIER
  $B \leftarrow$ Block-chain time trace extracted from blockchain.info
  $V \leftarrow$ Captured traffic volume in 1 second epochs
  $N \leftarrow$ Total number of Blocks
  for each block $b_i \in B$ **do**
    $t_{b_i} \leftarrow$ Time of generation of block b_i
    $||b_i|| \leftarrow$ Block's size
    $\omega_i \leftarrow$ Window's size
    $\Delta V_i = V(t_{b_i}, t_{b_i} + \omega_i) - V(t_{b_i} - \omega_i, t_{b_i})$
    if $||b_i|| - J \leq \Delta V_i \leq ||b_i|| + J$ **then**
      $detected\ block += 1$
    end if
  end for
  return $\frac{detected\ block}{N}$
end procedure

no noise, the difference should be close to the size of the block. Therefore, the algorithm evaluates the difference of traffic volumes in the two consecutive windows and compares it to a threshold determined by the natural network jitter of the target (as discussed later). If the difference falls within a bound, the algorithm considers that block to be *detected* in the traffic of the suspected user. The algorithm attempts to detect the blocks in the traffic and evaluates the ratio of such "detected" blocks. If the ratio is above a threshold η, the target user is declared to be a Bitcoin client.

Choosing the threshold η. The threshold should be chosen based on the target user's specific network conditions such as background traffic, network noise, and

bandwidth. Therefore, the algorithm selects the η parameter according to the user's network condition.

To do so, the classifier generates N (e.g., $N = 100$) synthetic block series, which we call *ground false*s. Each ground false is generated based on the known pattern of Bitcoin blocks, e.g., by simulating blocks around 1MB roughly every 10 minutes for the full block relaying mode. More specifically, we generate the timing between blocks based on an exponential distribution with mean 10 minutes. The classifier then correlates the target traffic with each of the N ground false instances using the correlation function of Algorithm 2. Finally, the threshold η is chosen as

$$\eta > \max(CF) \tag{5.1}$$

where CF is the set of correlation values against the N ground false instances. In another words, we choose η to be larger than the largest correlation value.

Choosing other parameters. The window shape classifier also needs to choose values of the parameters ω and J. Parameter ω needs to be large enough to contain most of the traffic of a block during block propagation. Moreover, J is chosen to take into account that some of the block propagation traffic might not be downloaded in that time window. This parameters needs to be selected based on user's bandwidth and volume of background traffic , and therefore it needs to be chosen for each client separately.

Neural Network-based Classifier (NN-based)

We also use neural network-based classifiers to detect Bitcoin in the presence of a more complex background noise, e.g., browsing more than one website simultaneously. In following, we explain the feature selection phase, and then describe the design of our neural network.

Feature Selection

Again, we leverage the unique shape of Bitcoin traffic as discussed before to classify Bitcoin traffic using our NN-based classifier. To create each sample data, we divide time into time intervals of length l and, use the volume of traffic in each interval as our features:

$$V = \{v_1, v_2, ..., v_n\} \tag{5.2}$$

Note that v_i is the volume of traffic in interval i. We choose 10 minutes as the *sample size*, which is the smallest length to have at least one peak of traffic. Furthermore, to choose the interval length, we try different values of 1, 5, 10 and 20 seconds. From our experiments, we find that the interval length of 10 seconds results in the best performance. Therefore, we choose 10 seconds as the interval length (l). Since the length of each sample is 10 minutes (600 seconds), using equation $n = \text{sample size}/l$, we get an array of length 60 as our feature vector.

Designing the Model

For our neural network model, we use a multi-layer perceptron that consists of an input layer, an output layer, and two fully-connected hidden layers. The input layer has $n = 60$ neurons, which is the size of each sample data, the hidden layers have $n_1 = 32$ and $n_2 = 16$ neurons, respectively, and output layer has one neuron, which represents if the sample data contains Bitcoin traffic or not. We use Relu as the activation function of the hidden layers and sigmoid [63] for the output layer. Also, we use binary cross-entropy as our loss function, and Adam optimization [79] to minimize the loss.

5.4 Experimental Setup

We use Bitcoin Core software[1] to run full node Bitcoin clients on multiple virtual machines on a campus network. Each virtual machine is connected to the Internet through a high bandwidth. Before starting the experiments, we leave our Bitcoin clients for a few days to make sure they have downloaded up-to-date blockchain ledgers. We capture Bitcoin traffic under three different scenarios on a Linux 16.0.4 virtual machine:

Datasets

Collecting Bitcoin traffic. We use Bitcoin version 0.12.0 to capture Bitcoin traffic in the full block relaying mode and Bitcoin version 0.14.0 to capture traffic in the compact block relaying mode. We capture Bitcoin traffic for each version for a period of around a month. Specifically, we captured the Bitcoin traffic in the full block relaying mode from August 28th to October 9th, 2016, and Bitcoin traffic in the compact block mode from March 14th to April 18th, 2017.

Bitcoin tunneled through Tor. We captured Bitcoin traffic behind Tor [45] for both compact and full block modes. We also captured Bitcoin traffic in the compact block mode when traffic is tunneled through Tor and popular Tor pluggable transports of obfs4 [136], FTE [48], and Meek-amazon [94].

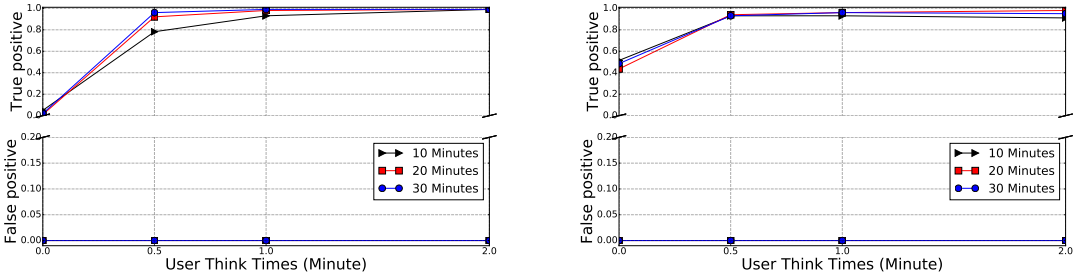
Bitcoin with background traffic. We captured Bitcoin traffic in the presence of HTTP background traffic by browsing the top 500 Alexa websites using the Selenium[2] tool while running Bitcoin software. We also collected Bitcoin traffic with HTTP background traffic for the same set of websites behind Tor and its three pluggable transports using Selenium.

CAIDA background traffic. We use CAIDA's 2018 anonymized traces[3] as a dataset for additional background traffic. We extracted the flows in this database

[1] https://bitcoin.org/en/bitcoin-core/

[2] `http://www.seleniumhq.org`

[3] `https://www.caida.org/data/monitors/passive-equinix-nyc.xml`



(a) Compact mode (b) Full mode

Figure 5.8: Result of `sizeHist` classifier on noisy Bitcoin traffic.

based on the protocol type, IP addresses and port numbers of the end-hosts. For each IP address, we consider all traffic to and from that IP as the typical traffic of that user. Table 5.3 shows the class breakdown of CAIDA dataset used in our experiments.

HTTP traffic. We collected top 500 Alexa websites using Selenium Tool. Also, we captureed these websites over Tor, and three pluggable transports (FTE, Meek, Obfs4). Moreover, we use the dataset by [100], which has collected the top $50,000$ Alexa websites over Tor.

Table 5.3: Traffic Class Breakdown For CAIDA Dataset.

Traffic Class	**Port Numbers**	**♯ of Connections**	**%of Total**
http, https	$80, 8080, 443$	745262	0.318
dns	53	1073758	0.457
smtp	25	2646	0.001
telnet	23	6958	0.003
ssh/scp	22	4928	0.002
other	–	511700	0.219
all		2345252	1.0

Metrics

We use following metrics to measure the performance of our classifiers:

- **True Positive:** True positive is the proportion of the data that our model correctly identifies Bitcoin out of all the traffic that contains Bitcoin.
- **False Positive:** False positive is the proportion of the data which did not contain Bitcoin traffic, and our model incorrectly classified as Bitcoin out of all the traffic that did not contain Bitcoin.
- **Accuracy:** Accuracy shows the proportion of correctly classified data.

Modeling Normal Users

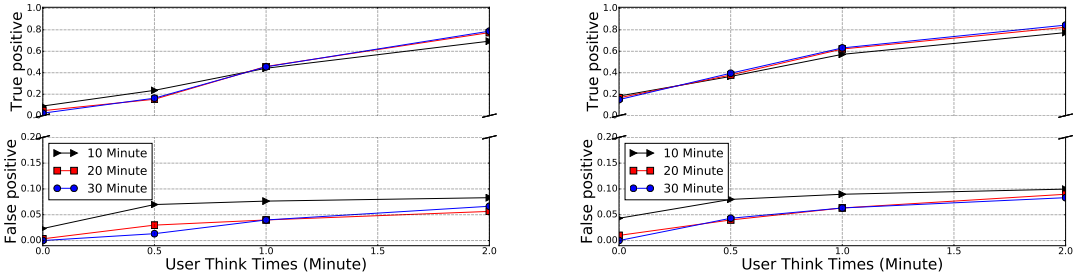
In this section, we describe four different types of users' profile that we use to evaluate the performance of our Bitcoin classifiers against. Users may have some background traffic while using Bitcoin or they can generate noisy traffic to evade detection. Therefore, we attempt to model typical behavior of users in various scenarios to evaluate the performance of our system.

- **Simple User:** A simple user is a Bitcoin client with no background noise. This type of user does not generate any network traffic other than the Bitcoin client application traffic. Therefore, all traffic of the simple user is Bitcoin traffic.
- **Simple Noisy User:** A simple noisy user is a Bitcoin client that browses **one** webpage during the time that we attempt to collect the Bitcoin traffic of the client.To control the background traffic, we introduce a parameter named think time, T, representing the amount of time that the user spends on a particular website. Note that, increasing T would decrease the background noise since there is not much traffic after a website is loaded. Thus, if T is large and the Bitcoin application is running, after the webpage is completely loaded, the simple noisy user's traffic would look like a simple user's traffic.
- **Complex Web (Complicated/Sophisticated) User:** A complex user is a Bitcoin client who simultaneously browses **multiple** websites. The complex web user is a sophisticated version of the simple noisy user. To create sample data for this user profile, we choose a Bitcoin traffic with length of sample size and accumulate noise traffic using following algorithm:

  1. Choose a random length of noise flow (k) in the range [0, *sample size*] seconds.
  2. Accumulate the noise flow to the p second of the traffic. p is selected randomly, similar to the previous step.
  3. Repeat 1.

  We repeat this process for I number of times, which represents the number of open tabs. The reason that we do not add noise from start to the end of the flow is that we want to make the background noise nonuniform, thus prevent the classifier from learning the noise and denoising the traffic.
- **Complex CAIDA User:** A complex CAIDA user is a Bitcoin client who is running one to five number of CAIDA applications, which are introduced in Table 2 simultaneously in the background. We use the same algorithm as above to create this user profile.

We define four different user profiles from simple behavior to more realistic and complicated ones. We apply each classifier on one or more number of these profiles to evaluate their performance.



(a) Compact mode (b) Full mode

Figure 5.9: Result of D2U classifier on noisy Bitcoin traffic.

5.5 Results

In this section, we implement our classifiers to evaluate their performance on user profiles described in Section 5.4 writing more than a thousands lines of code in Python. First, for each classifier, we declare the user profile(s) that we use for evaluation of its performance and the false data that we use for computing its false positive. Second, we describe the result of each classifier and give a summary and comparison of them at the end of this section.

User Profiles and False Data

For each classifier, we use a specific user profile, and depending on that, we choose the false data. As explained above, false data is the base traffic that we use to compute false positive. In other words, it is the traffic that we compare our Bitcoin traffic with. In the following, we describe these pairs for each classifier(s).

- For the window-based classifier, we use the simple user profile. Furthermore, we use HTTP which is the typical user behavior as the false data. This experiment evaluates if Bitcoin traffic can be differentiated from browsing an HTTP website.
- For the rest of the binary classifiers in this section, we use simple noisy user profile and attempt to detect the presence of Bitcoin. Note that, similar to the window-based classifier, we use HTTP for the false data. This experiment attempts to evaluate if browsing an HTTP website is enough to hide the Bitcoin traffic.
- For the neural network-based classifier, we use the complex web and complex CAIDA user for training and testing.

Note that, for the neural network classifier, we evaluate our model using $10,000$ number of test data. For rest of the classifiers, we use 500 number of test data for evaluation.

Size-based Classifiers

`SizeHist` Classifier

For this classifier, we compute the histogram of packet sizes and using a correlation algorithm we decide if the traffic contains Bitcoin or not. Because of the Bitcoin specific packet sizes, we expect to have a good performance even in the presence of background noise. We implement the `sizeHist` classifier on Bitcoin traffic in compact and full block modes using the noisy user model described in Section 5.4. Figure 5.8 shows the impact of traffic length and background noise on true positive and false positive for this classifier. As explained before, we control the noise using T. As expected, increasing T and therefore, decreasing noise enhances the classifier's performance. Figure 5.8 shows that we can reach more than 90% true positive and 0% false positive for both modes when we have 10 minutes of traffic and set T to 2 minutes. It is worth stating that we could reach similar results when we set T to 0.5 minutes and have 20 minutes of traffic.

`D2U` Classifier

We showed in Section 5.2 that downstream to upstream ratio of Bitcoin traffic can be a distinguishing factor to distinguish Bitcoin from other traffic. The D2U classifier attempts to use the symmetry between upstream and downstream of Bitcoin traffic to distinguish it from other protocols. Figure 5.9 shows the result of this classifier on noisy user profile for full and compact block modes. It indicates that increasing T and thus decreasing the background noise on Bitcoin traffic would improve the detection rate. More specifically, our true positive enhances from 0 to 80% when we increase T from 0 to 2 minutes.

Shape-based Classifier

As we explained in Section 5.3, window-based classifiers attempt to detect Bitcoin blocks using the volume of traffic downloaded at a time window around the block announcements times, and using the block detection rate it computes the true and false positive. To implement the window-based classifier, we set J and ω introduced in Section 5.3 to 100 kilobytes and 20 seconds, respectively. To set η, we compute block detection rate for Bitcoin using ground false shown in Figure 5.10.

As we explained in Section 5.3, η is the threshold that we use to differentiate Bitcoin traffic, and it should be larger than all the Bitcoin detection rates for ground false: if the block detection rate is higher than η, we classify the traffic as Bitcoin. Note that each point for Bitcoin using ground false in the figure is the average for 25 different ground false. Using this figure, we set η to 0.4. Moreover, Figure 5.10 shows the block detection rate for HTTP using ground truth and block detection rate for Bitcoin using ground truth too. Using η, we can reach detect all Bitcoin traffic through (August 28 - Oct 5) as Bitcoin. Also, we did not classify any of the HTTP traffic as Bitcoin, which results in 0% false positive. Furthermore, the performance of window-based

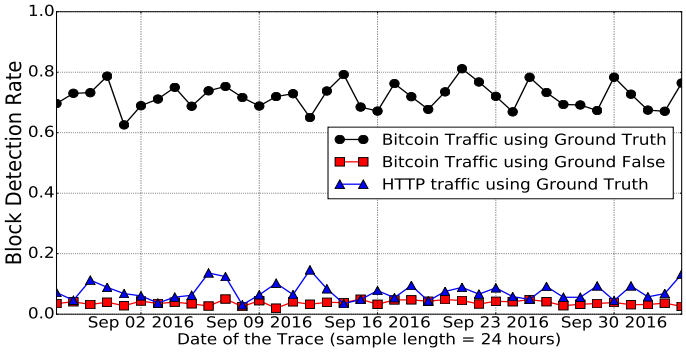


Figure 5.10: Block detection rate using window-based classifier.

classifier quickly diminishes in the presence of a small HTTP background noise since HTTP noise dominates the Bitcoin traffic and destroys the results of the classifier. Furthermore, we implement the window-based classifier on Bitcoin compact mode and learn that this classifier fails to detect Bitcoin traffic in this mode because of small block sizes, which makes it impossible to distinguish them.

Neural Network-based Classifier

In this section, we implement the NN-based classifier using Keras [37] with a Tensorflow [17] backend. We use complex web and complex CAIDA users to evaluate the NN-based classifier. To run this classifier, we need to set two parameters: learning rate and epochs. Learning rate is a hyper-parameter that controls how much we adjust the weights of the neural network model in each iteration. Moreover, epochs depict the number of times that the algorithm is run on the training data. We use the default value of 0.01 for the learning rate of Adam optimizer (lr).

For the epochs, we run our model for values ranging from 50 to 2000 and realize that using 1000 epochs allows us to obtain good performance from our classifier. Having a small number of epochs keeps the model from learning the dataset. On the other hand, having a very large number of epochs may cause over-fitting. Therefore, we need to pick this value carefully. Furthermore, increasing the number of epochs increases training time. For example, the training time increases from 20 seconds to around 4 minutes when we increase the number of epochs from 50 to 1000 when the size of the training data is 5000. Therefore, we need to take this into account when the size of training data increases.

Table 5.4 shows the result of NN-based classifier for different sizes of training data. As the table indicates, increasing the size of training data improves the results of this classifier. The accuracy of the classifier improves from 62% to 97% when we increase the size of training data from 1000 to $20,000$. Note that, true and false positive improves from 44% to 96% and 20% to 2% respectively when we increase the size of training data.

Table 5.4: Result of Neural Network classifier.

Training Size	False Positive (%)	True Positive (%)	Accuracy (%)
1000	20	44	62
5000	11	80	85
10,000	6	84	89
20,000	2	96	97

Summary and Comparison of the Results

In the following, we summarize of results for size-based, shape-based and NN-based classifiers and then, compare their performance.

- **Shape-based Classifier:** In this category, we have the window-based classifier, which results in 100% true positive and 0% false positive for full block mode when there is no background noise, but it fails to detect Bitcoin traffic on compact block mode because of the small block sizes. The performance of this classifier quickly diminishes in the presence of small noise such as simple noisy user model described in section 5.4.
- **Size-based Classifiers:** In this category, we have SizeHist and D2U classifiers.
  - **SizeHist Classifier:** Using 10 minutes of traffic with a think time of 1 minute, we can reach 0% FP and more than 90% TP in both compact and full block modes. Note that it gets a similar result for both cases when we have more than 20 minutes of traffic with a think time of 30 seconds.
  - **D2U Classifier:** This classifier reaches around 80% TP and up to 5% FP for both compact and full modes when there are at least 10 minutes of traffic and the think time is 2 minutes.

  Previous classifiers including D2U, SizeHist and window-based perform well when the background noise is negligible (think time of 2 minutes). Therefore, they are not useful when Bitcoin traffic has a large amount of background noise. To distinguish Bitcoin traffic in the presence of larger noises, we employ NN-based classifier.
- **NN-based Classifier:** To evaluate NN-based classifier, we use Complex web and CAIDA users in both modes. Our NN-based classifier is able to reach near perfect accuracy (97%) with less than 4% false positive when we increase the size of training data to 20,000. This classifier outperforms all previous ones since it is able to detect Bitcoin traffic using Complex web and complex CAIDA user explained in 5.4.

5.6 Conclusions

The reliable access to Bitcoin and similar cryptocurrencies is of crucial importance due to their consumers and the related industry. In this thesis, we investigated the resilience of Bitcoin to blocking by a powerful network entity such as an ISP or a gov-

ernment. By characterizing Bitcoin's communication patterns, we designed various classifiers that could distinguish (and therefore block) Bitcoin traffic even if it is tunneled over an encrypted channel like Tor, and even when it is mixed with background traffic. Through extensive experiments on network traffic, we demonstrated that our classifiers could reliably identify Bitcoin traffic despite using obfuscation protocols like Tor pluggable transports that modify traffic patterns. In order to disguise such patterns, an obfuscating protocol needs to apply significant cover traffic or employ large perturbations, which is undesirable for typical clients.

Chapter 6

Traffic Analysis Attack on WhatsApp

WhatsApp and similar messaging services are popular due to allowing users to send unlimited free multimedia messages. These applications encrypt the users' traffic, either end-to-end or middle-to-end, to provide security. However, these applications leak sensitive information through the traffic metadata, such as the timing and sizes of the packets. In this thesis, we investigate the traffic analysis attacks on WhatsApp. We utilize two previously introduced event-based and shape-based attacks and show that we can infer sensitive information from user traffic in WhatsApp using these simple techniques. Then, we show that using a naive countermeasure like VPN reduces the impact of these attacks. However, it does not obliterate it.

6.1 Background on Secure Instant Messaging Services

A secure Instant Messaging service (IM) is defined with two main features: 1) it deploys strong encryption, either end-to-end or end-to-middle. 2) it is not controlled by an adversary, for example, Soroush in Iran or WeChat in China. If an IM service operator cooperates with the surveillance government, there would be no need to use complicated TA attacks to infer sensitive data. For example, Iran and Russia attempted to get Telegram operators' cooperation to surveillance their people, which was unsuccessful today [8]. Note that an IM service that does not deploy strong encryption can be trivially compromised. Our TA-based attacks aim to eavesdrop the secure IMs since the non-secure ones are trivially surveilled with other trivial attacks. We specifically study WhatsApp, which is the most popular secure IM.

WhatsApp IM Architecure

Architecture. WhatsApp uses a *centroid* architecture. In a centralized architecture, clients send their messages to the IM server, and the server forwards them to the target client. All major messaging applications such as Telegram, Snapchat, Skype, Viber, Signal, etc., use this type of architecture. There are various messaging protocols for user communication, including Signal [57], Matrix [24], Off-the-Record [30], and MTProto [7]. WhatsApp uses the Signal [57] protocol to provide end-to-end encryption to voice calls, video calls, and user conversations. The signal protocol provides integrity, confidentiality, authentication, forward secrecy, participant consistency, destination validation, etc. Note that it does not offer anonymity preservation.

Security features. We discuss four main security features: integrity, confidentiality, Authentication, and Forward Secrecy.

WhatsApp provides *forward* and *future secrecy* by using Double-Ratchet protocol. Ensuring these two features provides message confidentiality even when a key is compromised [34]. Forward secrecy implies that if an adversary compromises a key, he/she cannot recover the previous keys. Similarly, future secrecy implies that compromising a key does not provide extra information about future keys. These properties ensure the confidentiality of the messages. Moreover, WhatsApp provides *end-to-end encryption* preventing the operators of the IM service from seeing users' messages. To ensure integrity, WhatsApp provides a method to authenticate the sessions between users by scanning a QR code [10]. Moreover, it uses standard authentication techniques such as authorization keys and public key certificates.

6.2 Attack and Threat Model

Similar to [23], we consider an adversary who does not block all user access to the message application or does not cooperate with the operators of the application. The adversary only monitors the encrypted traffic of the target user to deduce sensitive information from it. The information that we are interested in is the identity of the people involved in one-on-one communications. In this attack, the adversary intercepts a conversation that is believed to be politically or socially significant to find the IP address of the target in the chat. Our attack only performs traffic analysis and can be applied to all major messaging applications. Note that this is a fundamental attack that is not due to a buggy software implementation to be fixed through software updates.

Adversary

The **Adversary** is a surveillance party run by a repressive government. The adversary's goal is to find the IP address of a target individual involved in a specific one-on-one communication to punish them. The target individual could be a journalist or protestor discussing sensitive political issues in one-on-one communication.

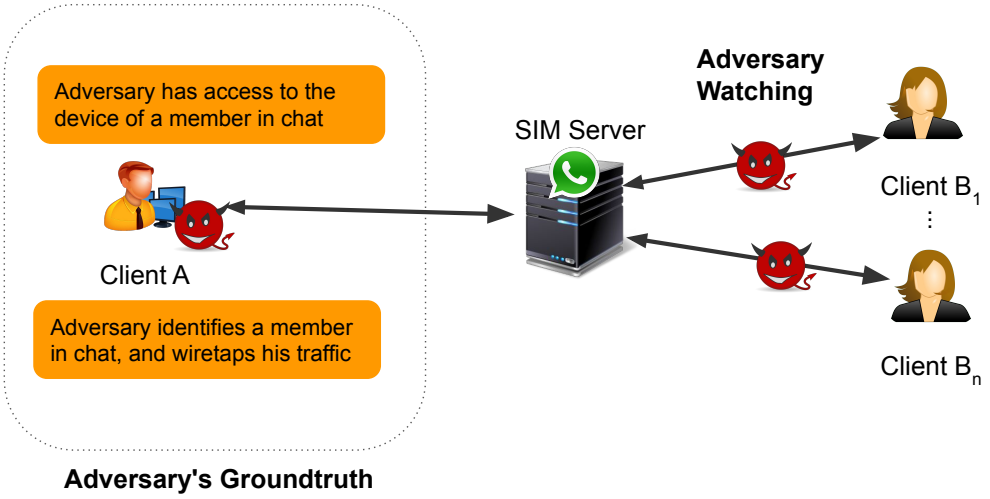


Figure 6.1: Attack Scenarios. Adversary wants to find the person that is chatting with client A.

The adversary needs to monitor the traffic of the target user by wiretapping the ISP of the targeted users to perform the attack.

Threat Model

The attack is performed on a messaging application that encrypts any communication between the clients and the servers with strong encryption and does not leak information due to security flaws in the application. For example, the application does not let hackers access the users' phone [59], or leak their personal information [60] .Also, the operators of the messaging application do not cooperate with the adversary to identify the target user.

How the Attack Is Performed

Figure 6.1 is showing the setup of the attack. The adversary intercepts client A's traffic and attempts to find the other party involved in the conversation.

Adversary ground truth. Figure 6.1 shows the setup of the attack. Assume that the adversary wants to find who the client A is speaking to. To do so, the adversary needs to have certain *ground truth* about the chat. This can be done in:

- The adversary is the client A in the one-on-one communication, therefore, it can record the messages and reply.
- The adversary is not the client A. She can intercept his encrypted traffic and records its traffic pattern. The wiretapping can be performed by the adversary who is controlling an ISP or IXP.

The adversary uses the ground truth to correlate the traffic of client A to the traffic of the target individual with the algorithm mentioned later in 6.3.

Adversary makes decisions. The adversary has the ground truth that is the client A's traffic and attempts to match it to the traffic of the wiretapped users (client B_1 to client B_n) using a detection algorithm.

6.3 Attack Algorithm

Event-based Detector

Our detector is based on the event-based detector in [23], shown in Figure 6.3, with small modifications to make it work on WhatsApp. The algorithm works by correlating the events in the ground truth with the traffic of the target user.

An event is a burst of consecutive packets that have inter-packet-delays lower than t_e seconds. It can be a single message or the number of messages sent with inter-message-delay (IMD) less than t_e. Messages include file, pdf, image, video, text, and audio. An event is defined as $e = (t, s)$ in which t is the time and s is the size of the event. The time is the timing of the last message received in the traffic burst, and the size is the volume of the traffic burst.

Figure 6.2 shows the traffic volume of WhatsApp per second in a one-on-one communication for 10 minutes. As the figure shows, there is a small traffic volume due to the text messages, handshakes, updates, etc. A peak in the traffic occurs when a comparatively large file (video, pdf, image, audio) is exchanged between the parties involved in the chat.

Remember that the adversary does not see the traffic content due to the use of encryption and only has access to the sizes and timings of the packets. The first step in correlating the traffic is to extract the events from the bursts in the encrypted traffic.

Event extraction. An event is a burst of packets in the traffic that have inter-packet-delay lower than t_e. A messaging application generates small packets due to updates, text messages, handshakes, and traffic bursts when a comparatively large message is sent. Therefore, the bursts are distinguishable by an adversary who monitors the traffic, even though she cannot see the packet contents due to encryption. We look for packets with a distance lower than t_e and consider it a burst. For each burst, an event is extracted. The sum of packet sizes in each burst gives the size of the event, and the arrival time of the last packet in the burst provides the time of the event. It is worth mentioning that if two messages are sent with IMD lower than t_e, they would be considered a single event. At the same time, some messages get divided into shorter bursts due to latencies in the network.

Detection algorithm. The adversary counts the number of the matched events between the target user flow ($f^{(U)}$) and the ground truth flow ($f^{(C)}$). The event $e_j^{(U)}$

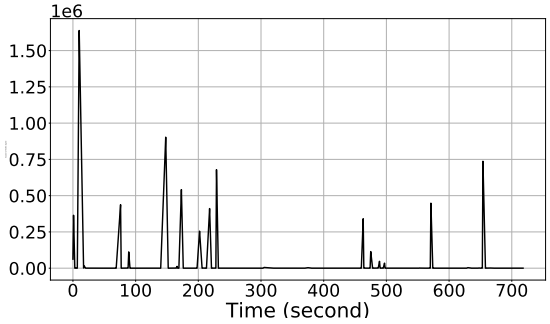


Figure 6.2: Volume of WhatsApp traffic over time for 10 minutes. The spikes of volume occur when a large message is exchanged.

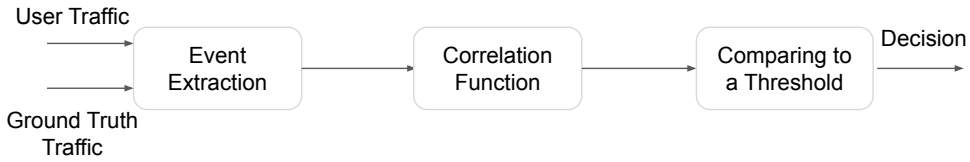


Figure 6.3: Event-based attacker.

is matched to the $e_i^{(C)}$ in the ground truth if:

- $|t_i^{(C)} - t_j^{(U)}| < \Delta$
- $|s_i^{(C)} - s_j^{(U)}| < s_j^{(U)} \times \gamma$

We make a slight change in the event-based detector introduced in [23] in matching the size of the events. They allowed the size of the burst to be a fixed offset smaller or higher than the size of the corresponding event. Here we match a burst to an event if its size is within a percentage of the event.

Assume that we want to find the event (t_i, s_i) in our traffic. We iterate over all the traffic bursts in our traffic in the time interval of $t_i - \Delta, t_i + \Delta$, and check to see if any of them matches our event. For a burst to match our event, it has to have at least $1 - \gamma$ of our event (γ is between 0 and 1). For example, if γ is set to 0.1 and the event size is $1MB$, and the Δ is 5 seconds, we need to find a burst that has a size of at least $900KB$. Note that this burst matches our event if its timing is only 5 seconds apart from the event's timing. Note that the size threshold is proportional to each event's size, which improves the result compared to choosing a fixed size threshold. Finally, the adversary computes the ratio of matched events as $r = k/N$, in which k is the number of matched events, and n is the total number of events in the ground truth. If r is above a certain threshold (η), it correlates the traffic with the ground truth.

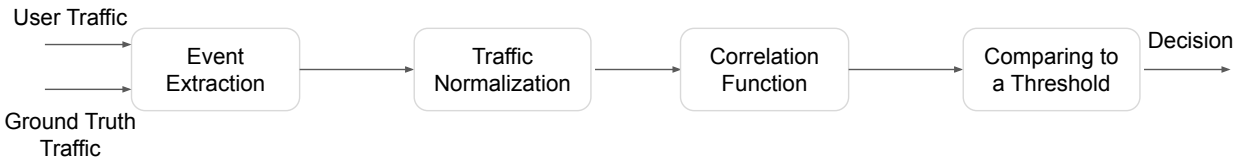


Figure 6.4: Shape-based attacker.

Shape-based Detector

The shape-based detector is designed based on the event-based method. Our shape-based detector is a combined version of the event-based and shape-based detector introduced in prior work [23], shown in Figure 6.4. First, the events are extracted from the traffic using the event-based detector's extraction phase (They only extracted the bursts and did not match them to their corresponding events). Second, the events are normalized, which we explain next, to reduce the user's bandwidth impact.

Normalizing the traffic. The shape-based detector converts the events to traffic bars. This is because the shape of the traffic is related to the user's bandwidth, and converting each event to a traffic bard removes the impact of the user's bandwidth.

To perform the normalization, we replace each event with a traffic bar with a width of t_e, and we select the height of the bar such that the area under the bar sums to the size of the event.

Then, every bar is divided into several smaller bins with a width of t_s and the height of the corresponding bar. Note that we put bins with a height of 0 and width of t_s between the bins related to different events. The new shape of traffic is a vector of the height of bins over time.

Correlating the normalized traffic. The shape-based detector correlates the ground truth's normalized traffic with the suspected users to know if they associate with each other. The equation 6.1 is used for correlation, and the $b^C = \{b_1^C, ..., b_{n_c}^C\}$ is the respective height of bins associated with the target user, and the $b^U = \{b_1^U, ..., b_{n_u}^U\}$ is the one associated with the user being tested. We compute the correlation as:

$$2 \times \frac{\sum_{i=1}^{n} b_i^C b_i^U}{\sum_{i=1}^{n} (b_i^C)^2 + \sum_{i=1}^{n} (b_i^U)^2} \tag{6.1}$$

in which, $n = min(n_c, n_u)$. This correlation returns a value of *corr*, which is between 0 and 1. We decide that two vectors are associated if the value of corr is above the threshold of η.

6.4 Experimental Setup

We collect our one-on-one conversation traffic on WhatsApp that is the most popular messaging application with more than 2 billion users globally. WhatsApp handles more than 10 billion messages daily, including 700 million photos and 100 million videos [14]. Since it does not provide an API, we use Selenium Tool[1] to send and receive messages through the WhatsApp web portal [13]. We believe this method of collecting data has an advantage over using an API to make the dataset more similar to the interaction between real users since every message is being sent in the same way a real user uses the application. Note that to simplify data collection, we send the messages in only one direction. We argue that it does not impact the result of the experiments since we will use only one side of the traffic (incoming or outgoing) to correlate the target user traffic with the ground truth.

The types of messages that we send between users are audio, video, text, and photos. The inter-message-delays between messages, which is the time between two consecutive messages in a conversation, are extracted from the experiments in [23]. Whatsapp does not allow one to create a profile using a fake number. Therefore, we used one existing personal phone number as one side of the conversation. We also purchased a phone number from Mint phone service to use as the other side of the chat. We use our detectors to learn if WhatsApp one-on-one conversation traffic leaks information above the conversation parties. Moreover, we collect data over a VPN for different VPN server locations to study the impact of tunneling WhatsApp traffic through VPN on the performance of the event-based detector. Finally, we capture data for users with different bandwidths to see how it affects the performance of our event-based detector.

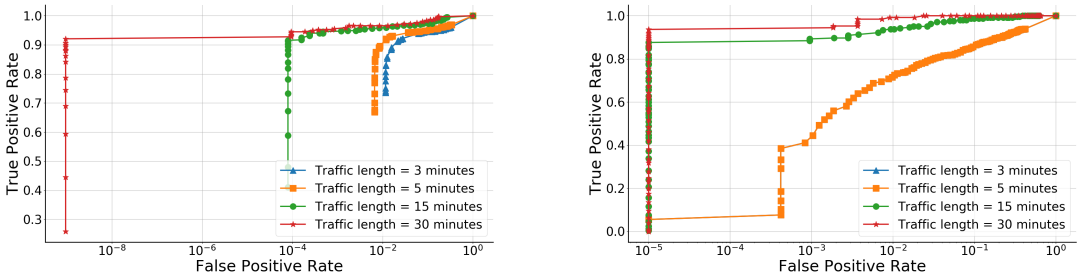
6.5 Experiments

We present our experimental setup for the attack in Figure 6.1 for the first type of adversary introduced in Section 6.2. In our attack scenario, two WhatsApp clients exchange data. We have the ground truth from one side and capture the traffic on the other side. The goal is to see if we can correlate the ground truth with the traffic. The adversary cannot see the traffic content and only has access to metadata such as packet timing and sizes. To simulate the conversation, we used Selenium on both sides to send and receive the traffic since WhatsApp does not provide an API.

Parameter Selection

We have to decide on four parameters here: t_e, t_s, γ, Δ. Here t_e is set to 1 second similar to the previous work in [23], and t_s of the shape-based detector to 0.01 seconds. γ and Δ should be chosen based on traffic characteristics. For example, when the traffic passes through a VPN, it is received with extra delay, which requires choosing

[1] `http://www.seleniumhq.org`



(a) Event-based detector

(b) Shape-based detector

Figure 6.5: Performance of our detectors on normal condition.

a larger value of Δ. Note that as we increase Δ and γ, our false positive increases along with the true positive. Therefore, it is important to choose a value that satisfies the requirement of the problem for the false positive and true positive. To choose the proper value for γ, we try values in $[0.1, 0.15, 0.2, 0.25, 0.3]$, choose the value that gives acceptable results for true and false positive. For the Δ, we try the values in $[30, 60, 100, 150]$ to find the proper value.

Normal Network Conditions

Figure 6.5 shows result for the event-based and shape-based detectors under normal conditions. As the figure shows, increasing the traffic length used for correlation improves the detection results. For the event-based detector, we fix the $\gamma = 0.15$ and the $\Delta = 30$ seconds. Fixing the true positive at 0.9, we see that we have false positive of $10^{-8.2}, 10^{-4.2}, 10^{-2}, 10^{-1.8}$ for traffic length of 30, 15, 5, and 3 minutes, respectively. We can reach 0.94 true positive and less than 10^{-3} false positive when we have 15 minutes of traffic. The shape-based classifier shows a similar result. In particular, we obtain 0.94 true positive and 10^{-3} false positive when we have 30 minutes of traffic. According to our results, we need twice as much data in event-based detectors to get the shape-based detector's similar results. For the rest of the experiments, we use the event-based detector since it outperforms the shape-based detector. Similar to the prior work in [23], the event-based detector is around *two orders of magnitude faster than* the shape-based detector. It is because the event-based classifier uses a discrete time-series of event metadata, while the shape-based uses the continuous measure of histogram over time.

Different bandwidths

To evaluate the performance of event-based detector on different user bandwidths, we use Wonder Shaper [15] (version 1.4.1) to change the upstream and downstream bandwidths of the receiver node. We observe that when bandwidth is low and the sender sends too many big files, the receiver does not receive them as bursts of events. Instead, it gets them steadily with no peeks, which we observed when the bandwidth was high. Because of that, the result of the detector immensely decreases as we

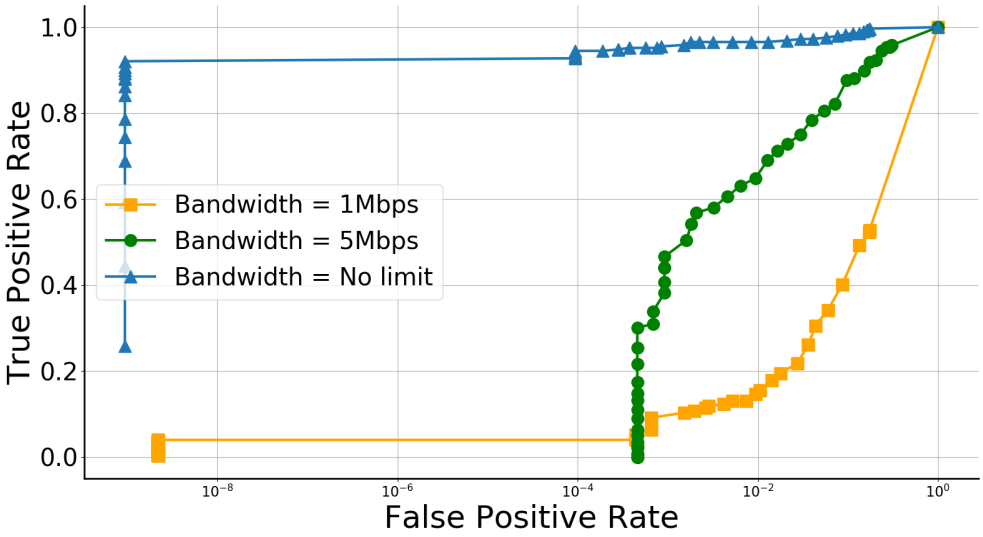


Figure 6.6: Comparing the result of event-based detector in different network bandwidths.

decrease the user bandwidth.

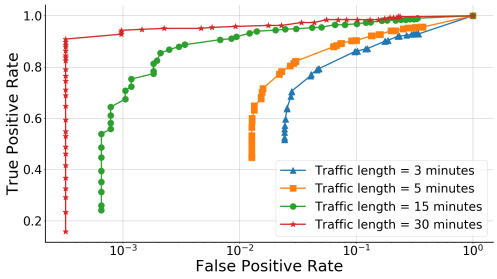
VPN as a Countermeasure

They are different options to choose a countermeasure, such as:

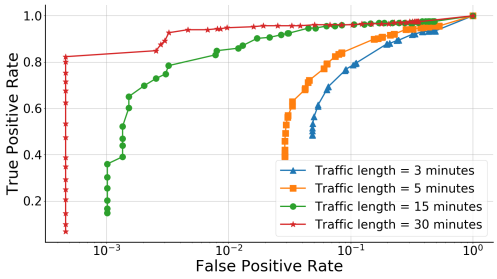
1) using an active approach to delay the events [32, 33, 49, 124, 125]. 2) using an active approach to add padding to the packets [49, 32, 33, 75, 124]. 3) using background traffic [49, 87, 106, 125, 134]. 4) using obfuscation mechanisms such as VPN, Tor.

We suspect that the reason for the effectiveness of our attacks is that WhatsApp does not use any mechanism to obfuscate its traffic. Therefore, we use multiple VPN servers to tunnel our traffic through and record their results. Figure 6.7(a)-6.7(c) shows results of our experiments over different VPN servers. As expected, our results degrade when traffic passes through a VPN due to the extra latency on the packets, but still, our results are acceptable. For example, when the traffic length is 30 minutes for the Turkey server, we obtain the true positive of 0.94, and the false positive is 10^{-3}. The false-positive becomes 10^{-2} if we aim at the same true positive, use 15 minutes of traffic. For the South Africa server, with a similar length of traffic, we obtain the true positive of 0.82 when fixing the false positive at 10^{-3}. Also, for the Japan server, having 30 minutes of traffic is enough to get the true positive of 0.83 when fixing the false positive at 10^{-3}.

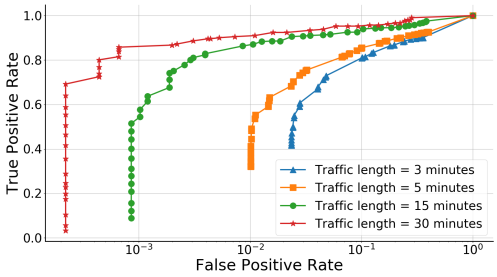
Figure 6.7(d) compares the result of the experiment on different VPN servers and without VPN. As the figure shows, passing through VPN degrades the result, and depending on the server's location due to imposing different latencies, the results



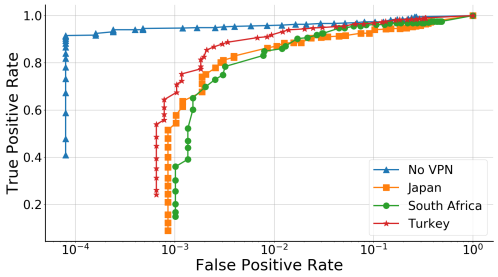
(a) Turkey



(b) South Africa



(c) Japan



(d) Comparing the result for different VPN servers

Figure 6.7: Result of the event-based detector when traffic is passed through different VPN servers.

differ. For example, the Turkey node shows better performance compared to the other two. Note that we use $\Delta = 150$ seconds and $\gamma = 0.15$ in the experiments over VPN.

6.6 Conclusions

In this chapter, we study the traffic analysis attacks on WhatsApp. Our work was an extension of Bahramali et al. [23] that investigates the attacks on Telegram. We changed their algorithms slightly to make it work on WhatsApp. We showed that using our detectors, We can infer sensitive information from the parties involved in a one-on-one WhatsApp conversation. We also showed that using VPN is not enough to protect users' privacy, and one needs to design meticulously designed obfuscations mechanisms to avoid detection. Future work can implement IMProxy introduced in Alireza et al. [23] on WhatsApp and evaluate its results.

Chapter 7

Conclusions and Future Directions

Traffic analysis is the practice of using communication patterns such as packet timings and sizes, to infer sensitive information. In this thesis, we investigated some applications of traffic analysis and designed and implemented algorithms for them.

Firstly, we studied flow correlation, which has applications in stepping stone detection and compromising of Tor and similar anonymous networks. We designed our algorithms using two approaches. First, we used the statistical approach in which we analyzed the traffic of the flow to infer sensitive information. Second, we applied deep learning to derive this information through training. We took the statistical approach to design an algorithm for flow correlation named TagIt. TagIt uses an interval-based approch to fingerprint the flows. Throughout simulations and experiments on Planetlab, we show that our approch outperforms previous fingerprinting system [70]. Also, we discuss the invisibility of our system using the K-S test, and Multi-flow attack, and show that our approach provides enough invisibility.

Also, we designed a flow correlation technique, FINN, that uses deep learning to fingerprint the network flows. Our system modifies inter-packet-delays to embed a message in the flow. We attempt to model the network jitter using our DNN-based framework to reliably extract our embedded message when it passes the jittery network. We show the performance of our system through extensive experiments over Amazon EC2 and digital ocean nodes.

We also studied the resilience of Bitcoin traffic to blocking by entities such as an ISP or a government. We characterized Bitcoin traffic patterns to find its distinguishing attributes to design a classifier that is tailored to identify Bitcoin. Through extensive experiments, we show that our classifiers can distinguish Bitcoin in presence of background traffic and despite being tunneled through obfuscation channels.

Finally, we measure the privacy of WhatsApp one-on-one communication, and its potential leakage of sensitive user data. We do measurements on the WhatsApp messaging service and analyze its potential data leakage through traffic analysis. We collect data traces of one-on-one communication on WhatsApp and run statistical

algorithms to evaluate its privacy. We also evaluate the performance of our classifier when the WhatsApp traffic is passed through VPN and show that stronger countermeasures are needed to avoid our traffic analysis attack.

Bibliography

[1] China blocks whatsapp, broadening online censorship. https://www.nytimes.com/2017/09/25/business/china-whatsapp-blocked.html.

[2] Cluster analysis. https://en.wikipedia.org/wiki/Cluster_analysis.

[3] Digitalocean. https://www.digitalocean.com/.

[4] Libnetfilter queue. http://www.netfilter.org/projects/libnetfilter_queue.

[5] Messaging Application Stat. https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/.

[6] Messaging applications. https://en.wikipedia.org/wiki/Messaging_apps.

[7] Mtproto mobile protocol. https://core.telegram.org/mtproto.

[8] Russia asks telegram to cooperate. https://www.seattletimes.com/nation-world/russian-court-telegram-app-must-cooperate-with-spy-agency/.

[9] Whatsapp discovers 'targeted' surveillance attack. https://www.bbc.com/news/technology-48262681.

[10] Whatsapp encryption overview: Technical white paper. https://scontent.whatsapp.net/v/t39.8562-34/122249142_469857720642275_2152527586907531259_n.pdf/WA_Security_WhitePaper.pdf?ccb=1-3&_nc_sid=2fbf2a&_nc_ohc=HurtY7LryYEAX_sl4zy&_nc_ht=scontent.whatsapp.net&oh=cbbd2e9a5c46fc5cd26f55996624f520&oe=608DBF19.

[11] Whatsapp says indian journalists were spied on. https://thewire.in/tech/israeli-spyware-was-used-to-spy-on-indian-activists-journalists-says-whatsapp.

[12] Whatsapp statistics. https://99firms.com/blog/whatsapp-statistics/.

[13] Whatsapp web. https://web.whatsapp.com/.

[14] Whatsapp wikipedia. https://en.wikipedia.org/wiki/WhatsApp.

[15] Wonder Shaper. https://github.com/magnific0/wondershaper.

[16] BotMosaic: Collaborative network watermark for the detection of IRC-based botnets. *Journal of Systems and Software*, 2013.

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, and e. a. Michael Isard. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[18] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapè. Mobile encrypted traffic classification using deep learning. In *Network Traffic Measurement and Analysis Conference, TMA 2018, Vienna, Austria, June 26-29, 2018*.

[19] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapè. MIMETIC: mobile encrypted traffic classification using multimodal deep learning. *Computer Networks*, 2019.

[20] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, Washington, DC, USA. IEEE Computer Society.

[21] T. Auld, A. W. Moore, and S. F. Gull. Bayesian neural networks for internet traffic classification. *IEEE Trans. Neural Networks*, 2007.

[22] A. Back, U. Möller, and A. Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In *Information Hiding, 4th International Workshop, IHW 2001, Pittsburgh, PA, USA, April 25-27, 2001, Proceedings.*

[23] A. Bahramali, R. Soltani, A. Houmansadr, D. Goeckel, and D. Towsley. Practical traffic analysis attacks on secure messaging applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2020.*

[24] A. Balducci and J. Meredith. Olm cryptogrpahic review. Technical report, NCC Group PLC, 2016.

[25] A. C. Bavier, M. Bowman, B. N. Chun, D. E. Culler, S. Karlin, S. Muir, L. L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating systems support for planetary-scale network services. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings.*

[26] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian. Traffic classification on the fly. *Comput. Commun. Rev.*, 2006.

[27] L. Bernaille, R. Teixeira, and K. Salamatian. Early application identification. In *Proceedings of the 2006 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2006, Lisboa, Portugal, December 4-7, 2006.*

[28] A. Biryukov, D. Khovratovich, and I. Pustogarov. Deanonymisation of clients in bitcoin p2p network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* ACM.

[29] A. Blum, D. Song, and S. Venkataraman. Detection of Interactive Stepping Stones: Algorithms and Confidence Bounds. In *RAID*, 2004.

[30] N. Borisov, I. Goldberg, and E. A. Brewer. Off-the-record communication, or, why not to use PGP. In V. Atluri, P. F. Syverson, and S. D. C. di Vimercati, editors, *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004.*

[31] H. A. C. Johansen, A. Mujaj and J. Noll. Comparing implementations of secure messaging protocols (long version). 2017.

[32] X. Cai, R. Nithyanand, and R. Johnson. Cs-buflo: A congestion sensitive website fingerprinting defense. In G. Ahn and A. Datta, editors, *Proceedings of the 13th Workshop on Privacy in the Electronic Society, WPES, 2014.*

[33] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014.*

[34] S. H. Calvin Li, Daniel Sanchez. Whatsapp security paper analysis. `https://courses.csail.mit.edu/6.857/2016/files/36.pdf`.

[35] J. Cao, Z. Fang, G. Qu, H. Sun, and D. Zhang. An accurate traffic classification model based on support vector machines. *Int. Journal of Network Management*, 2017.

[36] T. Choi, C. Kim, J. Yoon, J. Park, B. Lee, H. Kim, H. Chung, and T. Jeong. Content-aware internet application traffic measurement and analysis. In *Managing Next Generation Convergence Networks and Services, IEEE/IFIPNetwork Operations and Management Symposium, NOMS 2004, Seoul, Korea, 19-23 April 2004, Proceedings.*

[37] F. Chollet. keras. `https://github.com/fchollet/keras`, 2015.

[38] k. claffy. *Internet traffic characterization*. PhD thesis, UC San Diego, Jun 1994.

[39] M. Cotacallapa, L. Berton, L. N. Ferreira, M. G. Quiles, L. Zhao, E. E. N. Macau, and D. A. Vega-Oliveros. Measuring the engagement level in encrypted group conversations by using temporal networks. In *2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, United Kingdom, July 19-24, 2020*. IEEE, 2020.

[40] S. E. Coull and K. P. Dyer. Traffic analysis of encrypted messaging services: Apple imessage and beyond. *Computer Communication Review*, 44(5):5–11, 2014.

[41] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli. Traffic classification through simple statistical fingerprinting. *Computer Communication Review*, 2007.

[42] G. Danezis. The traffic analysis of continuous-time mixes. In *Privacy Enhancing Technologies, 4th International Workshop, PET 2004, Toronto, Canada, May 26-28, 2004, Revised Selected Papers*.

[43] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B*, 1977.

[44] C. Dewes, A. Wichmann, and A. Feldmann. An analysis of internet chat systems. In *Proceedings of the 3rd ACM SIGCOMM Internet Measurement Conference, IMC 2003, Miami Beach, FL, USA, October 27-29, 2003*. ACM.

[45] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*.

[46] D. L. Donoho, A. G. Flesia, U. Shankar, V. Paxson, J. Coit, and S. Staniford. Multiscale stepping-stone detection: Detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. In *RAID*, 2002.

[47] R. Durrett. *Probability: theory and examples*. Cambridge university press, 2010.

[48] K. Dyer, S. Coull, T. Ristenpart, and T. Shrimpton. Protocol Misidentification Made Easy with Format-transforming Encryption. In *CCS*, 2013.

[49] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE Symposium on Security and Privacy, SP 2012*.

[50] J. A. Elices and F. Pérez-González. The flow fingerprinting game. In *2013 IEEE International Workshop on Information Forensics and Security, WIFS 2013, Guangzhou, China, November 18-21, 2013*.

[51] J. A. Elices and F. Pérez-González. A highly optimized flow-correlation attack. *CoRR*, 2013.

[52] J. Erman, M. F. Arlitt, and A. Mahanti. Traffic classification using clustering algorithms. In *Proceedings of the 2nd Annual ACM Workshop on Mining Network Data, MineNet 2006, Pisa, Italy, September 15, 2006*.

[53] J. Erman, A. Mahanti, and M. F. Arlitt. Internet traffic identification using machine learning. In *Proceedings of the Global Telecommunications Conference, 2006. GLOBECOM '06, San Francisco, CA, USA, 27 November - 1 December 2006*. IEEE.

[54] J. Erman, A. Mahanti, M. F. Arlitt, I. Cohen, and C. L. Williamson. Semi-supervised network traffic classification. In L. Golubchik, M. H. Ammar, and M. Harchol-Balter, editors, *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2007, San Diego, California, USA, June 12-16, 2007*.

[55] J. Erman, A. Mahanti, M. F. Arlitt, I. Cohen, and C. L. Williamson. Offline/realtime traffic classification using semi-supervised learning. *Perform. Eval.*, 2007.

[56] J. Erman, A. Mahanti, M. F. Arlitt, and C. L. Williamson. Identifying and discriminating between web and peer-to-peer traffic in the network core. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*.

[57] K. Ermoshina, F. Musiani, and H. Halpin. End-to-end encrypted messaging protocols: An overview. In *Internet Science - Third International Conference, INSCI 2016, Florence, Italy, September 12-14, 2016, Proceedings*.

[58] B. F. U. Filho, R. D. Souza, C. Pimentel, and M. Jar. Convolutional codes under a minimal trellis complexity measure. *IEEE Trans. Communications*, 2009.

[59] WhatsApp Security Flaw. `https://www.cnn.com/2019/05/14/tech/whatsapp-attack/index.html`.

[60] WhatsApp Security Flaw. `https://www.pymnts.com/news/security-and-risk/2020/telegram-breach-exposed-millions-user-data/`.

[61] Y. Fu, H. Xiong, X. Lu, J. Yang, and C. Chen. Service usage classification with encrypted internet traffic in mobile messaging apps. *IEEE Trans. Mob. Comput.*, 2016.

[62] G. Garrammone. On decoding complexity of reed-solomon codes on the packet erasure channel. *IEEE Communications Letters*, 17(4):773–776, 2013.

[63] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[64] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks, 2014.

[65] Moghanizadeh, S. (2013). "The role of social media in Iran's Green Movement". https://gupea.ub.gu.se/bitstream/2077/33898/1/gupea_2077_33898_1.pdf.

[66] T. L. Grenville, T. Lang, G. Armitage, P. Branch, and H. yi Choo. A synthetic traffic model for half-life. In *in Australian Telecommunications, Networks and Applications Conference (ATNAC*, 2003.

[67] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: automated construction of application signatures. In *Proceedings of the 1st Annual ACM Workshop on Mining Network Data, MineNet 2005, Philadelphia, Pennsylvania, USA, August 26, 2005*.

[68] T. He and L. Tong. Detecting encrypted stepping-stone connections. *IEEE Transactions on Signal Processing*, 2007.

[69] T. He and L. Tong. Detecting encrypted stepping-stone connections. *IEEE Trans. Signal Processing*, 2007.

[70] A. Houmansadr and N. Borisov. 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*.

[71] A. Houmansadr and N. Borisov. SWIRL: A scalable watermark to detect correlated network flows. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*.

[72] A. Houmansadr, N. Kiyavash, and N. Borisov. 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*.

[73] A. Houmansadr, N. Kiyavash, and N. Borisov. Non-Blind Watermarking of Network Flows. *IEEE/ACM Transactions on Networking*, Aug 2014.

[74] C. Johansen, A. Mujaj, H. Arshad, and J. Noll. The snowden phone: A comparative survey of secure instant messaging mobile applications (authors' version). *CoRR*, 2018.

[75] M. Juárez, M. Imani, M. Perry, C. Díaz, and M. Wright. Toward an efficient website fingerprinting defense. In *Computer Security - ESORICS, 2016*.

[76] T. Karagiannis, A. Broido, N. Brownlee, kc claffy, and M. Faloutsos. Is P2P dying or just hiding? [P2P traffic measurement]. In *Proceedings of the Global Telecommunications Conference, 2004. GLOBECOM'04, Dallas, Texas, USA, 29 November - 3 December 2004*.

[77] T. Karagiannis, A. Broido, M. Faloutsos, and K. C. Claffy. Transport layer identification of P2P traffic. In *Proceedings of the 4th ACM SIGCOMM Internet Measurement Conference, IMC 2004, Taormina, Sicily, Italy, October 25-27, 2004*.

[78] H. Kim, K. C. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet traffic classification demystified: myths, caveats, and the best practices. In *Proceedings of the 2008 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2008, Madrid, Spain, December 9-12, 2008*.

[79] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, 2014.

[80] N. Kiyavash, A. Houmansadr, and N. Borisov. Multi-flow attacks against network flow watermarking schemes. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, 2008.

[81] T. Lang, P. Branch, and G. Armitage. A Synthetic Traffic Model for Quake 3. June 2004.

[82] B. N. Levine, M. K. Reiter, C. Wang, and M. K. Wright. 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*.

[83] W. Li and A. W. Moore. A machine learning approach for efficient traffic classification. In *15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2007), October 24-26, 2007, Istanbul, Turkey*.

[84] J. H. V. Lint. *Introduction to Coding Theory*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3rd edition, 1998.

[85] R. Liu and X. Yu. A survey on encrypted traffic identification. In Z. Tian, L. Yin, and Z. Gu, editors, *CIAT 2020: International Conference on Cyberspace Innovation of Advanced Technologies, Virtual Event / Guangzhou, China, December 5, 2020*. ACM.

[86] M. Lotfollahi, R. S. H. Zade, M. J. Siavoshani, and M. Saberian. Deep packet: A novel approach for encrypted traffic classification using deep learning. *CoRR*, 2017.

[87] X. Luo, P. Zhou, E. W. W. Chan, W. Lee, R. K. C. Chang, and R. Perdisci. HTTPOS: sealing information leaks with browser-side obfuscation of encrypted flows. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*.

[88] A. Madhukar and C. L. Williamson. A longitudinal study of P2P traffic classification. In *14th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2006), 11-14 September 2006, Monterey, California, USA*.

[89] A. Madhukar and C. L. Williamson. A longitudinal study of P2P traffic classification. In *14th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2006), 11-14 September 2006, Monterey, California, USA*. IEEE Computer Society.

[90] M. L. Martín, B. Carro, A. Sánchez-Esguevillas, and J. Lloret. Network traffic classifier with convolutional and recurrent neural networks for internet of things. *IEEE Access*, 2017.

[91] R. J. McEliece and W. Lin. The trellis complexity of convolutional codes. *IEEE Trans. Information Theory*, 1996.

[92] A. McGregor, M. A. Hall, P. Lorier, and J. Brunskill. Flow clustering using machine learning techniques. In C. Barakat and I. Pratt, editors, *Passive and Active Network Measurement, 5th International Workshop, PAM 2004, Antibes Juan-les-Pins, France, April 19-20, 2004, Proceedings*, Lecture Notes in Computer Science.

[93] Caida Dataset. `https://www.caida.org/data/monitors/passive-equinix-nyc.xml`.

[94] Meek Pluggable Transport. `https://trac.torproject.org/projects/tor/wiki/doc/meek`.

[95] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, Washington, DC, USA. IEEE Computer Society.

[96] A. W. Moore and K. Papagiannaki. Toward the accurate identification of network applications. In *Passive and Active Network Measurement, 6th International Workshop, PAM 2005, Boston, MA, USA, March 31 - April 1, 2005, Proceedings*.

[97] A. W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2005, June 6-10, 2005, Banff, Alberta, Canada*.

[98] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2008.

[99] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[100] M. Nasr, A. Bahramali, and A. Houmansadr. 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*.

[101] M. Nasr, A. Bahramali, and A. Houmansadr. Defeating dnn-based traffic analysis systems in real-time with blind adversarial perturbations. In *USENIX Security*, 2021.

[102] M. Nasr, A. Houmansadr, and A. Mazumdar. 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*.

[103] T. Nguyen and G. Armitage. Synthetic Sub-flow Pairs for Timely and Stable IP Traffic Identification. In *Australian Telecommunication Networks and Application Conference 2006*.

[104] T. T. T. Nguyen and G. J. Armitage. Training on multiple sub-flows to optimise the use of machine learning classifiers in real-world IP networks. In *LCN 2006, The 31st Annual IEEE Conference on Local Computer Networks, Tampa, Florida, USA, 14-16 November 2006*. IEEE Computer Society.

[105] T. T. T. Nguyen and G. J. Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials*, 2008.

[106] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In Y. Chen and J. Vaidya, editors, *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society, WPES, 2011*.

[107] J. Park, H. Tyan, and C. J. Kuo. Ga-based internet traffic classification technique for qos provisioning. In *Second International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP 2006), Pasadena, California, USA, December 18-20, 2006, Proceedings*. IEEE Computer Society.

[108] K. Park and H. Kim. Encryption is not enough: Inferring user activities on kakaotalk with traffic analysis. In H. Kim and D. Choi, editors, *Information Security Applications - 16th International Workshop, WISA2015, Jeju Island, Korea, August 20-22, 2015, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2015.

[109] V. Paxson. Empirically derived analytic models of wide-area TCP connections. *IEEE/ACM Trans. Netw.*, 1994.

[110] P. Peng, P. Ning, and D. S. Reeves. On the secrecy of timing-based active watermarking trace-back techniques. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*.

[111] Tor: Pluggable Transports. `https://www.torproject.org/docs/pluggable-transports.html.en`.

[112] Internet Assigned Numbers Authority. `https://www.iana.org/assignments/service-names-port-numbers`.

[113] Y. J. Pyun, Y. H. Park, X. Wang, D. S. Reeves, and P. Ning. Tracing traffic through intermediate hosts that repacketize flows. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA*.

[114] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[115] D. Ramsbrock, X. Wang, and X. Jiang. A first step towards live botmaster traceback. In *RAID*, 2008.

[116] J.-F. Raymond. Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems. In *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*. Springer-Verlag, LNCS 2009, 2000.

[117] F. Rezaei and A. Houmansadr. Tagit: Tagging network flows using blind fingerprints. *PoPETs*, 2017.

[118] S. Rezaei and X. Liu. Deep learning for encrypted traffic classification: An overview. *IEEE Communications Magazine*, 57(5):76–81, 2019.

[119] M. Roughan, S. Sen, O. Spatscheck, and N. G. Duffield. Class-of-service mapping for qos: a statistical signature-based approach to IP traffic classification. In A. Lombardo and J. F. Kurose, editors, *Proceedings of the 4th ACM SIGCOMM Internet Measurement Conference, IMC 2004, Taormina, Sicily, Italy, October 25-27, 2004*.

[120] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*.

[121] S. Staniford-Chen and T. L. Heberlein. Holding intruders accountable on the internet. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995*, 1995.

[122] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. Sok: Secure messaging. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015.

[123] P. Velan, M. Cermák, P. Celeda, and M. Drasar. A survey of methods for encrypted traffic classification and analysis. *Int. J. Netw. Manag.*, 2015.

[124] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *Proceedings of the 23rd USENIX Security Symposium, 2014*, 2014.

[125] T. Wang and I. Goldberg. Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In E. Kirda and T. Ristenpart, editors, *26th USENIX Security Symposium, 2017*.

[126] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng. Malware traffic classification using convolutional neural network for representation learning. In *2017 International Conference on Information Networking, ICOIN 2017, Da Nang, Vietnam, January 11-13, 2017*, 2017.

[127] X. Wang, S. Chen, and S. Jajodia. 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*.

[128] X. Wang, S. Chen, and S. Jajodia. 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*.

[129] X. Wang and D. S. Reeves. 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*.

[130] X. Wang, D. S. Reeves, and S. F. Wu. 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*.

[131] Z. Wang. The applications of deep learning. 2015.

[132] N. Williams, S. Zander, and G. J. Armitage. A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *Comput. Commun. Rev.*, 2006.

[133] Wolfenstein: Enemy Territory. `https://wolfenstein.fandom.com/wiki/Wolfenstein:_Enemy_Territory`, 2007.

[134] C. V. Wright, S. E. Coull, and F. Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2009.

[135] F. Yan, M. Xu, T. Qiao, T. Wu, X. Yang, N. Zheng, and K. R. Choo. Identifying wechat red packets and fund transfers via analyzing encrypted network traffic. In *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2018, New York, NY, USA, August 1-3, 2018*. IEEE, 2018.

[136] Yawning. Obfsproxy4. `https://github.com/Yawning/obfs4/blob/master/doc/obfs4-spec.txt`, 2015.

[137] K. Yoda and H. Etoh. 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*.

[138] W. Yu, X. Fu, S. Graham, D. Xuan, and W. Zhao. 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*.

[139] R. Yuan, Z. Li, X. Guan, and L. Xu. An svm-based machine learning method for accurate internet traffic classification. *Information Systems Frontiers*, 2010.

[140] S. Zander, T. T. T. Nguyen, and G. J. Armitage. Automated traffic classification and application identification using machine learning. In *30th Annual IEEE Conference on Local Computer Networks (LCN 2005), 15-17 November 2005, Sydney, Australia, Proceedings*.

[141] L. Zhang, A. G. Persaud, A. Johnson, and Y. Guan. Detection of stepping stone attack under delay and chaff perturbations. In *Proceedings of the 25th IEEE International Performance Computing and Communications Conference, IPCCC 2006, April 10-12, 2006, Phoenix, Arizona, USA*. IEEE, 2006.

[142] Y. Zhang and V. Paxson. Detecting stepping stones. In *9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000*.