TÉCNICO LISBOA

Distributed system for cooperative deanonymization of Tor circuits

Pedro Manuel Torres Pires de Medeiros

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Nuno Miguel Carvalho dos Santos

January 2021

Acknowledgments

Firstly, I thank my advisor, Professor Nuno Santos, for welcoming me as his student and for his enthusiastic guidance throughout this work. I thank Diogo Barradas for all the support and also guidance who was readily available at all times. I thank Bernardo Ferreira and Bernardo Portela for the fruitful discussions along with the support in hardware. I thank Professor Luís Rodrigues for the discussions and ideas at beginning of this project. I thank my family for the endless support throughout my studies. Lastly, I thank all my friends who have turned the last few years into a delightful journey.

Resumo

A tecnologia dos dias de hoje permite que alguém interessado (por exemplo um ISP) monitorize as comunicações de qualquer utilizador, sendo que estas podem levar à descoberta de informação sensível como os ideais políticos, preferências do consumidor, condição de saúde entre outras informações privadas. Redes de anonimato como o Tor foram desenhadas de forma a circundar técnicas de análise de tráfego ou vigilância de redes, mantendo o anonimato dos utilizadores quando navegam na internet. Além destas propriedades o Tor permite construir serviços especiais que se mantêm anônimos na rede e permitem assim manter ambos o cliente e o servidor anônimos. Infelizmente estes serviços podem também ser usados para a prática de atividades ilegais o que fomenta a necessidade de criação de ferramentas que permitam quebrar o anonimato de fluxos ilegais na rede Tor. Nesta tese estudamos técnicas que permitam quebrar o anonimato de fluxos ilegais em que ambos o cliente e o servidor são anônimos. O nosso trabalho faz uso de resultados recentes que mostram que é possível aplicar ataques de correlação de tráfego utilizando redes neuronais profundas. Com base nestes resultados apresentamos um sistema distribuído, cooperativo de de-anonimização de tráfego, onde vários ISPs podem contribuir partilhando informações acerca dos fluxos que interceptam permitindo assim a realização de correlação de tráfego. Fazemos uma avaliação detalhada deste sistema onde utilizamos fluxos gerados sinteticamente com base em serviços onion Tor. Apresentamos também uma extensão ao sistema onde os dados cedidos pelos ISPs são mantidos privados e apenas quando uma correlação é encontrada o fluxo em questão é de-anonimizado assegurando o anonimato e privacidade dos restantes fluxos envolvidos no pedido.

Palavras-chave: Tor, correlação de tráfego, redes neuronais, aprendizagem profunda, serviços onion

Abstract

Current technology allows an interested party (for instance, an ISP) to closely monitor the communications of any user which, in turn, may allow to uncover data such as the user political views, consumer preferences, health condition, and other private information. Anonymity networks, such as Tor, have been designed to defeat network surveillance or traffic analysis activities, and preserve the anonymity of users when they use the Internet. Not only that Tor allows the construction of special services that also remain anonymous on the network making both the client and server anonymous. Unfortunately, these services can also be used to support illegal activities, which raises the need for tools that can help in de-anonymizing illegal Tor flows. In this work we study techniques that allow to de-anonymize illegal flows where both the client and the server are anonymous. Our work leverages on recent results that show that it is possible to perform traffic correlation based on deep neural networks. This paves the way for designing distributed, cooperative, de-anonymization tools, where multiple ISPs can contribute by sharing information regarding flow sets and then perform traffic-correlation. We perform an in depth evaluation of such a system where synthetic flows were generated based on real fully anonymous Tor onion services. We also present an extension to the system where the data provided by the ISPs is kept private and only when a correlation is found, the target flow can be de-anonymized while ensuring that the anonymity of the other flows involved in the query is preserved.

Keywords: Tor, traffic correlation, neural networks, deep learning, onion services

Contents

Acknowledgments . . . iii

Resumo . . . v

Abstract . . . vii

List of Tables . . . xi

List of Figures . . . xiii

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Goals . . . 2

1.3 Contributions . . . 3

1.4 Document Roadmap . . . 3

2 Background and Related Work **5**

2.1 Background on Tor and Onion Services . . . 5

2.2 Overview of Tor Deanonymizing Techniques . . . 9

2.3 Traffic Correlation Attacks . . . 11

2.4 Traffic Correlation using Neural Networks . . . 13

2.5 Neural Networks with Encrypted Data . . . 16

3 Torpedo **21**

3.1 System Overview . . . 21

3.2 Ranking Stage . . . 25

3.3 Correlation Stage . . . 27

3.4 Privacy Preserving Extensions . . . 31

3.5 Implementation . . . 34

4 Harvesting an Onion Dataset **39**

4.1 Onion Service Crawling and Processing . . . 39

4.2 Synthetic Generation of Onion Service Traffic . . . 41

4.3 Dissecting Onion Service Traffic . . . 44

4.4 Summary . . . 49

5 Evaluation **51**

5.1 Methodology and Metrics 51

5.2 Ranking Stage Evaluation 53

5.3 Correlation Stage Evaluation 55

5.4 Full Pipeline Evaluation 61

5.5 Privacy-Preserving Extensions Evaluation 65

5.6 Summary 68

6 Conclusions **69**

6.1 Achievements 69

6.2 Future Work 70

Bibliography **71**

List of Tables

2.1 Comparison between encrypted neural network systems. 19

3.1 Ranking stage's network architecture. 26
3.2 Correlation stage network architecture. 27
3.3 Combination methods results for multiple examples. 30
3.4 Common Keras models parameters. 35

4.1 Selected onion service pages distributions. 41
4.2 Final dataset characteristics. 44

5.1 Original DeepCorr dataset size. 58
5.2 Original DeepCorr results by epoch. 58
5.3 TPR for single network with 1/2 the original DeepCorr size for our generated dataset with a maximum FPR of 10^{-2}. 59
5.4 Results for ensemble model 1/2 the original DeepCorr size for our generated dataset. . . 59
5.5 TPR of single networks varying kernel and strides. 60
5.6 TPR of single networks varying kernel and strides. 61
5.7 Evolution of ACC with flow size. 64
5.8 Evolution of ACC with multiple aggregation methods. 64
5.9 Evolution of both training and prediction times with multiple model modifications. 65
5.10 TF-Encrypted enabled network vs base line. 66
5.11 TF-Encrypted enabled optimized networks. 66
5.12 TF-Encrypted same machine multiple processes. 67
5.13 TF-Encrypted different machines (2vCPU) within same location. 67
5.14 TF-Encrypted different machines (4vCPU) within same location. 68
5.15 TF-Encrypted different machines different locations. 68

List of Figures

2.1 Example Tor circuit enabling Alice to connect anonymously with Bob. 6
2.2 Onion service with connection established. 7
2.3 Illustration of a realistic Tor circuit spanning multiple ASes. 8
2.4 Traffic correlation attack model. 11
2.5 Simple neural network. 14
2.6 Convolution illustration. 15
2.7 Network architecture of DeepCorr's CNN. 16
2.8 TF Encrypted architecture. 18

3.1 Torpedo system components: client, probe, and correlator. 22
3.2 Probe component rotating log and query response. 23
3.3 Correlator stages processing input flow pairs. 25
3.4 Reduce input vector size techniques. 29
3.5 Input padding modifications. 33
3.6 Input reshaping modification. 34
3.7 Both modifications combined. 34

4.1 Collected onion service pages distributions. 41
4.2 Sample capture model. 42
4.3 Full trace hello world onion service. 45
4.4 Requests zoom hello world onion service. 46
4.5 First and second requests to hello world page zoom. 47
4.6 Full trace 400KB onion service. 48
4.7 Full trace 900KB onion service. 48
4.8 2nd and 3rd requests' close up for 400KB onion service. 49

5.1 CDF concurrent pairs. 53
5.2 CDF up to 16 seconds time difference pairs, uncorrelated only match the same onion. . . 54
5.3 CDF up to 16 seconds time difference pairs. 55
5.4 First results on realistic dataset. 56
5.5 Results on realistic dataset for holdout 60% to 40%. 57
5.6 Results on final dataset. 57

5.7 TPR for single networks with 1/2 the original DeepCorr size with our generated dataset. . . 60
5.8 Single network results varying kernel and strides. 61
5.9 Results for both stages combined. 62
5.10 Results for both stages using combined training. 62

Chapter 1

Introduction

This thesis addresses the problem of traffic correlation on anonymity networks such as Tor. It studies the use of deep neural networks to perform traffic correlation. Although traffic correlation attacks are well known, they mostly target client-only anonymity scenarios. We will focus on the scenario where both the client and server preserve their anonymity during communication. We present a thorough evaluation of traffic correlation based on models combining both deep and regular neural networks for client and server anonymous communication.

1.1 Motivation

Current technology allows an interested party (for instance, an ISP) to closely monitor the communications of any user over the Internet. Even if the communication is encrypted, the destination of packets and multiple statistical properties of network flows may allow an eavesdropper to uncover data such as the user political views, consumer preferences, health condition, and other private information [1–4]. Anonymity networks such as Tor [5] have been designed to defeat network surveillance or traffic analysis activities, and preserve the anonymity of users when they use the Internet. Tor works by using an overlay network of relay nodes that forward encrypted packets in a virtual circuit. Packets are encrypted using multiple, nested, layers of encryption, such that each relay can only extract information regarding the next hop in the virtual circuit but cannot extract information regarding the origin or the final destination of the packet. In this way, the first node of the Tor circuit knows the user IP but does not know the final destination, and the last node knows the final destination but does not know the user IP. Besides providing sender anonymity, Tor also provides the ability for a given network destination to remain anonymous. These fully anonymous Tor circuits, that ensure both sender and receiver anonymity, perform twice the number of hops on the Tor network.

The Tor network is popular amongst a large number of legitimate users, such as individuals that seek to ensure their right to privacy, or journalists that need to report facts under adversarial conditions. Unfortunately, Tor can also be used to support illegal activities, such as drug dealing, illegal gun selling, or child pornography [6–10] just to name a few. This raises the need for tools that can selectively deanonymize Tor flows and help in the prosecution of miscreants involved in illegal activities. Previous work has shown that encrypted traffic analysis techniques enable the correlation of flows entering the Tor network with flows exiting the Tor network to de-anonymize a flow [11–15]. Some of the earlier approaches in the literature focused on the analysis of packet timing and length distributions in order to correlate traffic flows [11, 16]. More recent work leverages machine learning techniques with manually selected features in order to train a flow correlator [14]. Currently, state-of-the-art approaches take advantage of convolutional neural networks to automatically learn how flows are related to each other, ultimately performing flow correlation with significant success [15].

However promising, these techniques only target traffic tunneled through sender anonymous circuits and that are destined to typical websites which do not ripe the benefits of destination anonymity. In this work we will apply such traffic correlation techniques to fully anonymous communication through Tor, which is where most illegal activities take place.

1.2 Goals

This project aims at exploring approaches that allow for the de-anonymization of Tor circuits that provide both sender and receiver anonymity. Although previous work has achieved high accuracy when correlating regular Tor flows, no work has successfully tackled the problem of the de-anonymization of fully anonymous Tor circuits. Existing systems also do not provide any security or privacy guarantees on the data being used to perform correlation. This may pose itself as a burden when envisioning the wide deployment of a flow correlation tool among privileged entities in the network which are typically unwilling to share network data, e.g., ISPs and Law Enforcement Agencies (LEAs). Our goal is to perform traffic correlation attacks when both the sender and the receiver are anonymous by means of Tor Onion Services. We also present an extension to our system that improves upon privacy guarantees on the flow data being used.

> *Goals:* This project focuses on the design and implementation of a protocol and system that will allow one to perform Tor circuit de-anonymization when both the sender and receiver are anonymous. The system shall be able to correlate fully anonymous Tor circuits with reasonable results. We expect to achieve these goals while being able to process large amounts of data in a fair amount of time and without sacrificing the accuracy of traffic correlation. The system should also employ a protocol to allow for multiple ASes and authorities to work together. An extension to the system shall also be able to provide privacy guarantees on flow data provided by cooperating ASes.

In order to reach the proposed goal we started by collecting a baseline for state-of-the-art traffic correlation techniques based in convolutional neural networks. The second major step to achieve our goal comprised on the collection of a large-scale dataset of fully anonymous flow pairs which, to the best of our knowledge, has not yet been performed in the literature. The third step to achieve our goal focused on the design of a system that can process fully anonymous circuits with high performance and precision. The final step to achieve our goal introduced preliminary work on several extensions whose objective is to increase the privacy-preserving properties of our system.

1.3 Contributions

This thesis describes, implements and evaluates Torpedo, a system that allows for de-anonymization of fully anonymous Tor circuits where both the sender and receiver are anonymous. As a result, the thesis makes the following contributions:

- Describes Torpedo, the first system that applies traffic correlation attacks to Tor onion traffic.
- Innovative correlator design that separates the correlation process into two distinct phases allowing for high throughput of correlations.
- Describes and implements preliminary privacy-preserving extensions to the system.
- Delivers a thorough evaluation of the system performance and precision showing that we can achieve high throughput while maintaining reasonable precision.
- Delivers a dataset of correlated flow pairs from fully anonymous circuits generated by accessing Tor onion services, both static and dynamic which can be used in future scientific work.

1.4 Document Roadmap

The rest of this document is organized as follows: Chapter 2 presents all the background and related work. Chapter 3 describes the proposed Torpedo architecture and how we implemented a prototype of such architecture. In Chapter 4 we describe how the dataset was collected and characterized. Chapter 5 presents the evaluation and results of the implemented system. Finally, Chapter 6 concludes the thesis and presents possible future work.

Chapter 2

Background and Related Work

This chapter goes through some essential background material that is necessary to undestand the contributions of this thesis. We start by describing the Tor network and how the Tor onion services work. Then we proceed by presenting the related work on previous Tor deanonymization attacks, traffic correlation specific attacks, and finally the use of neural networks to perform such correlation attacks.

2.1 Background on Tor and Onion Services

This section provides some necessary background on the Tor anonymity network, focusing in particular on the mechanisms responsible for enabling anonymous communications at the IP level. We start by introducing the basic techniques employed by the Tor system for providing sender anonymity, i.e., Tor circuits. Then we present how these mechanisms have been further leveraged for providing receiver anonymity with the introduction of Tor onion services. Lastly, we provide a general picture of how Tor circuits are laid out and how their traffic can be observed by Autonomous Systems (ASs) on the Internet.

2.1.1 Providing Sender Anonymity

Tor is one of the most popular anonymity networks whose primary goal is to provide sender anonymity. Essentially, it allows a client to establish a TCP/IP connection with a receiver while preserving the client's IP address anonymous from the receiver. To achieve this, Tor implements a variant of the *onion routing* protocol [17]. Onion routing relies on multiple intermediary nodes – called *relays* – to help forward a client's messages to the receiver while hiding the client's IP address and requiring minimal trust in each individual relay. To this end, onion routing uses multiple encryption layers wrapping the message such that the intermediary relays cannot observe the contents of the message or learn the full route of the message within the relay network between sender and receiver, as this would break anonymity in case one such relay had been compromised.

Tor provides sender-anonymous TPC/IP tunnels named *circuits*. Figure 2.1 shows how a circuit is established by a client (Alice) involving multiple relay nodes of the Tor network. Tor circuits are typically three nodes long, having an *entry*, *middle* and *exit* relay nodes, and are constructed by the Tor client

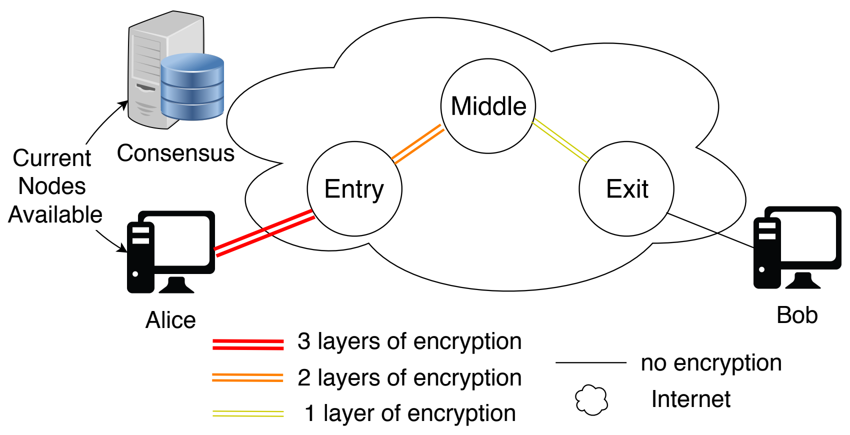


Figure 2.1: Example Tor circuit enabling Alice to connect anonymously with Bob.

who arranges three different symmetric keys with each respective node upon the circuit establishment protocol. Once this protocol has been successfully completed, the client can use the newly established circuit to tunnel IP packets down to some arbitrary receiver (e.g., Bob). To hide the packet size distribution, IP packets are encapsulated and encrypted into constant 512-byte cells. The cells are then delivered to the entry node, and will eventually reach the exit node (through the middle node). At this point, the exit node opens a TCP/IP connection with the intended receiver so as to exchange all original packets between the client and the receiver effectively serving the role of proxy to the client.

More specifically, the circuit establishment protocol works as follows. The client selects the three relay nodes by consulting a public directory named *consensus*. Anyone can contribute to the Tor network by deploying relay nodes across the world and publishing some necessary identifying information about each relay on the consensus, namely its IP address and a public key. Each relay must keep the respective private key secret. After selecting the relay nodes, the Tor client leverages the relay nodes' public keys to negotiate a pairwise symmetric key with each relay using onion routing encryption. The public keys are used to authenticate each relay. Once the symmetric keys have been set up, the circuit is now established and data transmission can be initiated. For a given cell, the client encrypts it first with the exit relay key, then with the middle relay key, and finally with the entry relay key. Then it forwards the resulting message to the entry node. Each node allocated to the circuit will then progressively "unpeal" the message by decrypting it and sending the decrypted payload to the next hop in the circuit. The exit relay removes the last layer of encryption and can now know the final destination of the message. Responses are propagated in the circuit in the inverse sequence and encrypted in the reverse order. Given that the exit node does not know the IP address of the client, and the receiver only observes the IP address of the exit node, sender anonymity is guaranteed.

2.1.2 Providing Receiver Anonymity

A second desirable property wanted by the Tor designers was the ability to provide receiver anonymity. In fact, although in the scenario presented above the client remains anonymous, the receiver's IP address must necessarily be exposed so that the client can contact it in the first place. In fact, once a Tor circuit

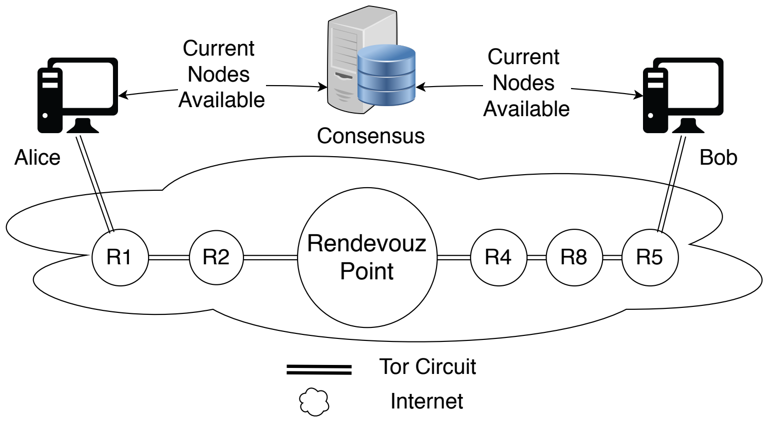


Figure 2.2: Onion service with connection established.

has been established, the exit node needs to learn the IP address of the receiver so that it can proxy the client's connections accordingly. Hence, the basic Tor circuits do not provide receiver anonymity.

To overcome this limitation, Tor introduced a new mechanism called *onion services* [5]. It enables the deployment of public servers that can be reachable through TCP/IP to provide standard services (e.g., over HTTPS or FTPS) while preserving the IP address of the server anonymous. To achieve this end, the key idea is to leverage Tor circuits as the building block so that both the sender and the receiver create two independent circuits which connect to an intermediate relay node called *rendezvous point* (RP). By executing a set of specific protocols, a client willing to connect to a receiver uses not the IP address of the receiver, but a onion service address based on a public key owned by the receiver. Using this address, the client can choose any relay in the Tor network to be elected as the RP and establish a circuit with the RP. At this point the RP does not know the IP address of the client. The receiver has then the ability to locate the RP, and establish its own circuit to the RP. Because a Tor circuit provides sender anonymity, the RP does not know the real IP address of the receiver. Once the RP is connected to both the sender and receiver through independent circuits, all it does is to forward the traffic between both endpoints, therefore providing mutual anonymity. This protocol is briefly illustrated in Figure 2.2. Due to its complexity, we omit many details, e.g., about the name resolution of onion services and location of the RP by the receiver. For more details, we refer the interested reader to Dingledine et al. [5].

In virtue of its strong anonymity properties, Tor has captured the attention of individuals engaged in illicit activities and constitutes today one of the foundations of the "dark web" [18]. In particular, Tor has been used for running anonymous online marketplaces of illicit goods behind onion services such as Silk Road [6], and allowing end-users to access them and purchase such goods anonymously. Past studies [7–10] have reported numerous other illicit activities fostered by web-based onion services, e.g., involving violence, illegal weapons and drugs, child pornography, hacking, and illicit finances. Unsurprisingly, this has prompted the governmental authorities to investigate illegal affairs involving Tor communications, and to look for ways to deanonymize suspicious Tor circuits. To better understand the challenges of such an endeavor, we present in more detail what are the main assumptions that make Tor difficult to deanonymize.

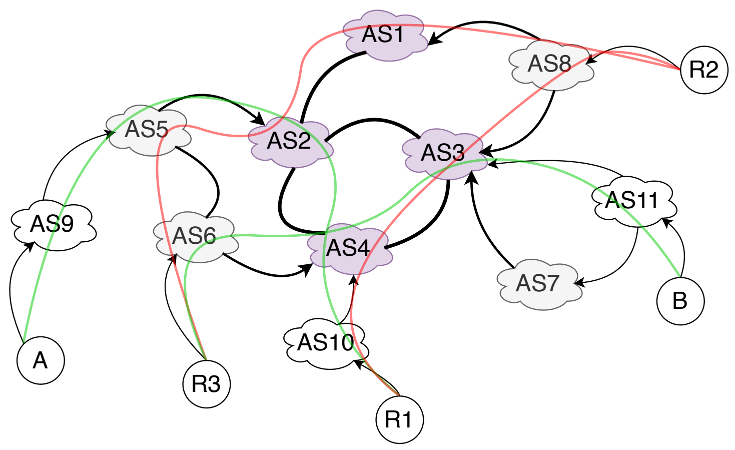


Figure 2.3: Illustration of a realistic Tor circuit spanning multiple ASes.

2.1.3 The Security Foundations of Tor

The security foundation of Tor lies on the assumption that no single organization will be able to control the entry node and the exit node of a given circuit. Otherwise, if such an adversary exists, it can correlate the volume and the inter-packet arrival time of packets exchanged through that circuit with high probability, and obtain the associated IP addresses of the communicating endpoints. This is possible because (a) the amount of traffic tunneled through the circuit is the same, and (b) the relay nodes tend to forward the packets as fast as possible in order to improve the performance.

In theory, it is not even necessary for an adversary to compromise the entry and exit nodes of a given circuit in order to deanonymize it, but simply the ability to intercept / observe the traffic exchanged by these nodes. For instance, if by any chance the entry and exit nodes of a circuit are connected to the same Internet Service Provider (ISP), the ISP finds itself in a unique position to be able to observe the traffic exchanged between these relays. As a result, that ISP is able to deanonymize, not only that circuit, but all Tor circuits that leverage those two particular relays as entry and exit nodes.

In practice, however, hardly a single ISP will ever be able to intercept both the entry and exit nodes of a given circuit. This is because Tor tends to select relay nodes connected to different ISPs. Figure 2.3 represents how a realist Tor circuit is allocated on the Internet. The Internet is composed of multiple interconnected networks maintained by network providers named Autonomous Systems (ASes). Normally, ASes are categorized into three classes: Tier-1, Tier-2 and Tier-3 ASes. Tier-1 ASes are the largest type. They cover very large geographical regions and are responsible for the Internet backbone. Tier-1 ASes allow smaller Tier-2 and Tier-3 ASes to use their networks, for which they have to pay. Tier-2 ASes often represent large organizations or Internet Service Providers (ISPs). Tier-3 ASes refer to those that do not provide internet access to any other AS, typically local ISPs or small organizations. Figure 2.3 shows a circuit created by Alice (A) to contact Bob (B), and it uses three relay nodes: R1 connected to AS10, R2 connected to AS8, and R3 connected to AS6. In these conditions, deanonimyzing this circuit is challenging for three main reasons:

Privacy reasons: Deanonymization can be possible if AS10 and AS6 agree to exchange information about the traffic of their own networks. They can try to match traffic patterns between R1 and R3 allowing them to correlate the respective circuit's endpoints. However, ASes tend to be very reluctant in exchanging any information whatsoever about their networks, in particular about network topology and traffic. On the one hand, ASes want to protect the privacy of their own clients. On the other, they want to secure their networks and preserve their competitive advantage with respect to their competitors.

Political reasons: Data sharing between ASes may also be conditioned by legal restrictions imposed in their base countries. For instance, if AS10 and AS6 are located in the USA and in Germany, respectively, it is necessary to consider the laws of both countries and even take into account international agreements established between them. Furthermore, exchanging this kind of information normally requires the intervention and cooperation of law enforcement authorities from both countries, which may be challenging to be achieved in practice.

Operational reasons: Since Tier-2 and Tier-1 ASes aggregate more traffic, another possibility would be to inspect the traffic at these networks. For instance, in Figure 2.3, because AS4 is responsible for forwarding traffic from AS10 and AS4 can also observe the packets by R1 and R3. However, due to the high volume of traffic handled at the core by Tier-2 and Tier-1 ASes, it is very difficult for them to collect and analyze flows. It requires a considerable (and costly) amount of processing power at the network elements (e.g., switches) which makes it hard to perform these operations at such a large scale. For all these reasons, Tor circuits remain very difficult to deanonymize by the law enforcement authorities.

2.2 Overview of Tor Deanonymizing Techniques

Given that we aim at enabling law enforcement authorities to deanonymize Tor circuits using traffic correlation techniques based on neural networks, this section provides an overview of some representative known attacks to Tor. Then we progressively funnel our presentation of the related work to describe the state of the art in traffic correlation using neural networks.

Over the past 15 years since its inception, the Tor network has been subjected to many attacks designed mostly with the purpose of deanonymizing end-users' circuits or onion services. In tandem, Tor has evolved and incorporated new defenses against some of these attacks. In this section, we provide an overview of the most important categories of deanoymizing techniques and attacks. To this end, we follow the taxonomy proposed by Evers et at.[1] which categorizes existing Tor deanonymizing techniques into the following seven categories taking into account their method and goal. Each category can also be sorted into groups based on two different perspectives: (1) attacks can be *passive* or *active* depending on whether the adversary simply observes the traffic (the former) or actively manipulates it (the latter); (2) attacks can be *single-end* or *end-to-end* depending on whether the adversary can monitor or control one of the circuit's entry or exit nodes (the former) or both edges of the circuits (the latter):

Correlation attacks: This category of attacks is conducted by an adversary in a passive fashion, and comprises the focus of our work. In the next section, we will provide an in-depth description of correlation attacks on Tor. In a nutshell, correlation attacks allow an adversary eavesdropping the network between

[1] https://github.com/Attacks-on-Tor/Attacks-on-Tor

the client and the entry relay, and between the exit node and the server, to correlate observed traffic flows and confirm that a given client and server are communicating in the same circuit. To date, multiple end-to-end correlation attacks have been proposed in the literature, including those based in statistical methods [19], application-layer exploits [20], and traffic analysis [21].

Congestion attacks: Congestion attacks are end-to-end active attacks which enable an adversary to determine which relays make part of a specific circuit initiated by a client. The core insight of this technique is tied to the fact that an adversary can measure latency differences in the traffic flow of a target client after congesting selected Tor relays. Examples are the attacks introduced by Murdoch and Danezis [12] and by Evans et al. [22].

Timing attacks: On par with congestion attacks, end-to-end timing attacks are conducted in an active fashion and enable an adversary to determine which server a client is communicating with. Such attacks are based on the manipulation and/or measurement of traffic patterns flowing from the entry to the exit node. For instance, the attack of Chakravarty et al. [23] first holds on the ability to create fluctuations in the traffic destined to the victim by using a colluding end-point. Then, the adversary identifies the client by detecting the perturbations it has introduced in the network across the existing pool of Tor relays.

Fingerprinting attacks: Tor does not significantly modify the shape of traffic patterns and closely preserves the packet timing and volume characteristics which are exclusive to a given web page. This makes it possible for an adversary to launch single-end and passive fingerprinting attacks in order to identify which webpage [24, 25] or onion service [26, 27] is being accessed by a client. Similarly, fingerprinting techniques can be used to distinguish onion service activity from regular Internet browsing over Tor, leading to the possible deanonymization of the onion service clients and servers [28].

Denial of service attacks: An adversary can perform single-end active attacks aimed at exhausting important resources in the Tor network. Typically, such attacks aim to jeopardize the performance of users' connections, or to force a degradation in the security of the established circuits. For instance, Jansen et al. [29] show that it is possible to abuse Tor's congestion and flow control mechanisms to put targeted relays out of action and to divert users to malicious relays in control of the adversary. In contrast, the CellFlood attack [30] impairs the ability of selected Tor relays to serve circuit creation requests by sending a few computationally expensive circuit creation requests towards those relays.

Supportive attacks: The goal of these attacks is not to deanonymize or impair the connection of Tor users, but to help setting up a later attack that does. In the past, Winter et al. [31] have studied the nefarious effects of Sybil identities on the Tor network, i.e., relays that are controlled by a single operator, showing that they can be used for facilitating end-to-end traffic correlation attacks or to tamper with clients' traffic. In a different approach, Li et al. [32] describe an attack aimed at shortening the rotation time of clients' guard nodes, i.e, Tor entry relays which are specially selected as the point of entry in the anonymous network. The attack increases the chance for a malicious relay to be selected as a client's entry relay and to aid in profiling the client's traffic.

Revealing onion service attacks: Due to the possibility of serving sensitive contents on onion services, these have been subject to attacks aimed at revealing their location. As an example, the attack by

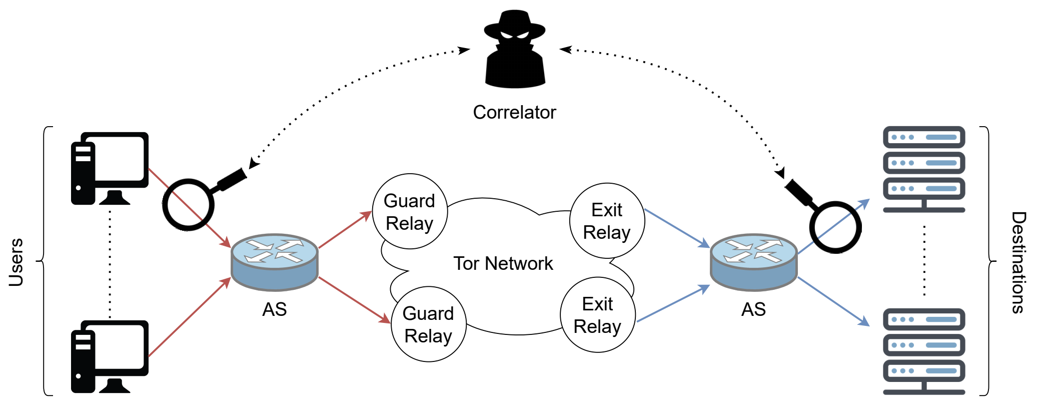


Figure 2.4: Traffic correlation attack model.

Biryukov et al. [33] increases the likelihood of a malicious relay to be chosen as the first node to be connected to a onion service's server and to trivially identify the server's location. In contrast, Zander and Murdoch [34] propose an approach for locating a onion service's servers by correlating clock skew changes caused by a flood of requests to the same onion service.

This section has briefly covered several categories of attacks against Tor, eventually leading to the deanonymization of users. The next section delivers an in-depth exposition over existing traffic correlation attacks which touch the crux of our approach for deanonymizing Tor circuits.

2.3 Traffic Correlation Attacks

This section provides an overview of multiple correlation attack models. Traffic correlation consists in assessing whether two samples of network flows gathered in separate areas of the network correspond to the same connection. Besides multiple applications of flow correlation, such as enabling the detection of stepping-stone attackers [35], correlation attacks have a prominent impact on the security of the Tor network, and enable the effective deanonymization of Tor circuits. A successful traffic correlation attack has the ability to deanonymize not only Tor clients but also servers running behind Tor onion services.

Figure 2.4 illustrates how such an attack can be performed when a user uses the Tor network. An attacker observes both end connections of the Tor circuit but does not need to observe the full path of the circuit traversing the Tor network. By applying these correlation attacks to the observed flows at each end, the attacker can to determine if the flows captured belong to the same connection. If that is indeed the case, since the attacker knows the IPs at each end of both flows, it has essentially deanonymized the user and it now knows it's IP and the IP it is accessing to.

Typically, traffic correlation attacks can be perpetrated by finding similarities between the packet sizes and inter-packet timing characteristics of packets pertaining to a given flow. For the specific case of Tor, the use of packet size features is largely useless for the purpose of traffic correlation as fixed sized cells are used to propagate data between endpoints. However, in order to maintain low-latency, Tor does not significantly obscure inter-packet timing, allowing an adversary with a privileged location in the network to succeed in performing traffic correlation attacks. We now describe a number of influential flow correlation techniques which may be used to launch an attack against the Tor network.

Based on generic statistical traffic features: The first techniques that were used for performing traffic correlation attacks employed simple statistical features about the flows (e.g., packet time differences). In 2004, Levine et al. [11] demonstrated through simulations that low-latency mix systems – including systems based on onion routing like Tor – are prone to timing attacks. Essentially, they study a correlation between the timings of packets seen at different relays of a mix network (e.g., at an entry and exit relay in a Tor circuit) based on the difference between the arrival time of a packet and the arrival time of its successor. If the two relays are in the same path, there should be a correlation between the time differences observed at each relay. Murdoch and Danezis [12] focused their work specifically on the Tor network and leveraged network delays. Since Tor relays propagate cells in a round-robbin fashion, circuits with no cells to be currently propagated are kept inactive. Hence, the more active circuits exist on a relay, the more latency will be introduced in each individual connection transiting through that relay. This latency difference can then be used to perform correlation and determine if a certain Tor relay is routing a given flow. This attack, however, cannot locate the Tor client in question since it is only devised to probe Tor relays. Yet, it can greatly impair a user's anonymity by identifying the relays of circuits.

Based on traffic models: Shmatikov et al. [13] build realistic models based on HTTP traces in order to analyze the resilience of low-latency anonymity networks against inter-packet timing correlation attacks. Their methodology is as follows. To perform the attack, 60 seconds of traffic are split into a limited amount of fixed-sized time windows. Then, the number of packets observed during each time windows is counted. Finally, the obtained sequences of packet counts generated for each flow are compared through cross-correlation. Providing that the correlation coefficient surpasses a given threshold, the attacker can determine that both samples correspond to the same flow.

Based on traffic watermarking: Houmansadr et al. [36] built a watermarking scheme that enables the correlation of traffic flows. They introduced small delays in a flow, unnoticeable to the user but which can be correlated in the future. The proposed system records the original flow, the inter-packet timing when it arrives at the watermarker and then introduces a negligible delay in each packet. Once the flow arrives at the correlator, the recorded inter-packet timings from before are compared with the observed flow to perform the correlation. Such a watermarking scheme can be employed to launch a correlation attack within the Tor network. To deploy such an attack, an adversary can deploy a malicious server that watermarks traffic so that it can be correlated further in the Tor circuit, possibly identifying the client.

Based on asymmetric traffic analysis: Sun et al. [14] describe a set of attacks which enable a state-level adversary in control of an AS to obtain visibility over the traffic flowing multiple ASes, and to correlate Tor traffic in order to deanonymize specific Tor circuits. This system uses two main insights. The first insight is that a malicious AS can interfere with the BGP protocol in order to divert and intercept traffic which would not typically cross the AS. This interference is performed by means of BGP interception, where a malicious AS first advertises a subset of IP addresses that it does not own making closer traffic pass through it instead of following the path to the destination. Then, for preventing a client from experiencing an interruption of the connection, the malicious AS forwards the diverted traffic to the correct

AS. The second insight consists in leveraging asymmetric traffic analysis, a variation of typical traffic analysis attacks. In this variation, the attacker only needs to observe incoming or outgoing traffic at both ends of communication to perform the correlation. The attack is performed by gathering the sequence of incoming and outgoing cells, as well as the duration of activity in each circuit. In their work, authors perform correlation with the help of machine learning algorithms, namely decision trees and the k-NN algorithm. The above combinations of attacks allow an adversary in control of a single malicious AS to deanonymize a Tor circuit. However, the described attack requires the observation of a large amount of data before being able to achieve acceptable results. In fact, to achieve an accuracy higher than 80%, it is necessary to inspect more than 30 MB worth of data. In many cases, it may be infeasible to observe such an amount of data due to the nature of traffic used in many activities performed over Tor.

Based on compression techniques: Recently, Nasr et al. [21] have introduced *compressive flow correlation*, an approach that enables a passive adversary to correlate traffic flows while resorting to a small bandwidth, storage, and computation footprint. To achieve this goal, the authors applied linear projection algorithms to sparse traffic features, such as inter-packet timing and packet sizes, effectively producing a compressed representation of such features. These compressed flow features are then used to perform the correlation which means that the communication and storage consumed decreases, thereby improving scalability. However, performing traffic correlation on compressed data is slower when comparing against the optimal solution.

Based on neural networks: The latest traffic correlation technique uses deep learning to learn features about the dynamics of the Tor network and achieve higher accuracy in flow correlation attacks. DeepCorr [15] is a state-of-the-art system that allows an attacker to train a convolution neural network (CNN) on network flows. Importantly, DeepCorr offers a higher accuracy in flow correlation than previous approaches, while simply requiring the observation of the first 300 packets of data pertaining to a given flow. This represents a major improvement over the technique previously proposed by Sun et al. [14]. Next, we describe this technique in more detail since we follow a similar approach in our work.

2.4 Traffic Correlation using Neural Networks

The compelling results achieved by DeepCorr [15] are attributed to the fact that the system trains a CNN on real world data which allows the model to learn intrinsically how the Tor network operates. For a 900 packets flow observation, DeepCorr achieves an accuracy of 96%. In contrast, previous systems like the one proposed by Sun et al. [14] (named RAPTOR) rely on generic statistical flow features to perform the correlation, missing information that is not captured by such analysis. When observing the same amount of data, RAPTOR achieves only a 4% accuracy when correlating traffic flows. Next, we describe neural networks, deep neural networks, and DeepCorr. Finally, we explore encrypted neural networks.

2.4.1 Neural Networks

Neural networks have the ability to learn classification problems by processing labeled data and applying the learned functions to previously unseen data. Hence, they can be used to classify flow pairs and perform correlation attacks. Neural networks consist of an architecture of nodes and arrays of weights and biases. The input to the network is mapped to the first layer and the output of the network is the result on the last layer nodes. The evaluation of the network occurs by multiplying the value of a given node by a given weight, summing the corresponding bias and propagating the result to the next corresponding node in the network.

Figure 2.5 illustrates a simple network with 2 inputs, 1 output, and a single hidden layer composed of 3 nodes. Arrows represent the weights and biases associated to that connection. A node can have multiple inputs and outputs. The value of the node corresponds to the sum of all its inputs which is then passed through an activation function to arrive at the final value for that node. There are multiple examples of activation functions, the most common being ReLU which is also the one used in all our models activation functions. These functions essentially apply some kind of filter/mapping to a given value. The ReLU function for input v can be described as *the maximum value between 0 and v meaning negative values will be pulled to 0*, i.e. if the sum of all inputs for the first node on the hidden layer of the example would be -0.4 then the resulting node value would be 0 where as if the sum were to be 0.6 then the resulting node value would be 0.6.

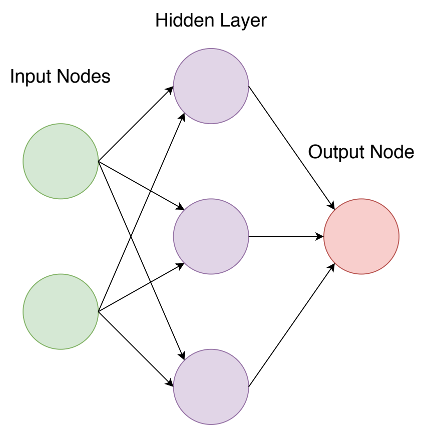


Figure 2.5: Simple neural network.

Neural networks learn by comparing the predicted results with the expected output. This learning process essentially determines the error in the network output and adjusts the weights and biases in a direction that reduces this error. Hence a model may only be as good as the data used during training. To train a model such as DeepCorr the expected result is 1 if the given example is a correlated pair and 0 if the given example is an uncorrelated pair. The outputs of the network, however, will rarely be exactly 1 or 0. Instead the result will be between these two values and the difference between the expected result is used to further tweak the network, i.e., during training we may have a correlated pair with output 0.84. Even though this result is much closer to 1 than to 0, the training process will adjust the network weights such that in a future evaluation the result for this pair is even closer to 1.

2.4.2 Convolutional Neural Networks

Convolutional neural networks expand on the previous concept by adding special layers before the fully connected network. DeepCorr makes use of such convolutional layers to extract features from the input. The convolutional operations consists of sliding a given filter through the input and extracting the resulting

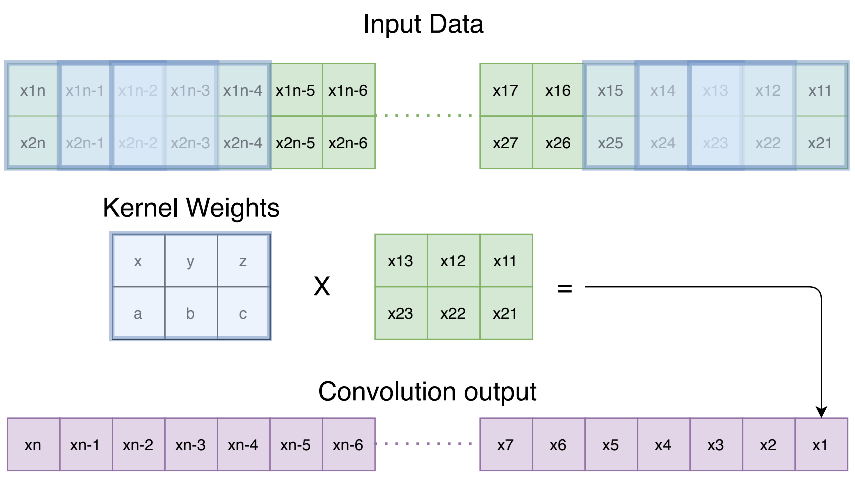


Figure 2.6: Convolution illustration.

computation to construct a new data vector. These filters are also know as *kernels* and are especially good at recognizing certain patterns in the input. These patterns can be seen as features that instead of being manually programmed by the designer of the network are learned by the network it self. It is this property that makes convolutional neural networks especially good at processing images as they learn to recognize patterns that wouldn't have been described through manually programmed features.

Figure 2.6 illustrates an example of this computation for the first convolutional layer of DeepCorr. We illustrate one kernel with dimensions (2,3) sliding through the input with a stride of (1) and generating the corresponding output. The variables of the model being trained in these computations are the kernel weights represented by x, y, z, a, b, and c in the figure. The same ReLU activation function is used here to process the result of the convolutions. Although the output may seem smaller than the input, this is the result of just one kernel for the first convolutional layer which employs in DeepCorr's case 2000 distinct kernels albeit all with the same dimension. To reduce these computed features, deep neural networks make use of another type of layer in this case a *max pooling layer*. Max pooling refers to the operation of reducing the input size by only keeping the maximum value on a given window. The concept is similar to the action of the sliding kernels however in this case there are no weights to train and there is only one sliding window hence producing less data then its input.

This resulting vector is than fed to the following convolutional layer and the cycle repeats. Once there are no more convolutional layers we proceed to the fully connected layers similar to the previous network example. The last resulting vector is now mapped to the input layer of the fully connected part of the network. This is done by mapping every value of the resulting vector (usually a vector of vectors which is flatten to one single vector) to a single input node in the first layer of the fully connected network. We then proceed with the evaluation of the network as described before.

2.4.3 DeepCorr Architecture

We now deliver a brief exposition about the functioning of CNNs using DeepCorr [15], used for the purpose of correlating Tor traffic, as our running example. Seen as a black box, this CNN operates in

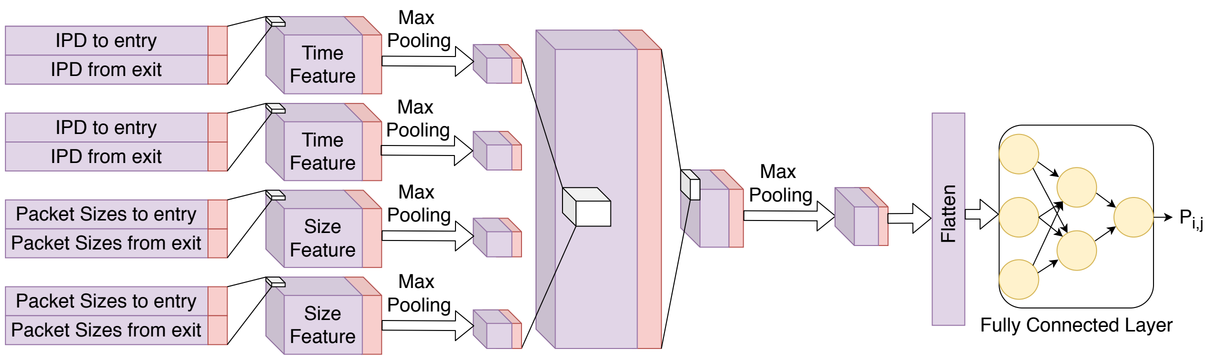


Figure 2.7: Network architecture of DeepCorr's CNN.

two phases: a *training phase* and a *testing phase*. The training phase aims at programming the CNN in order to learn whether two flows given as input are correlated or not, i.e., belong to the same Tor circuit. To this end, the CNN is fed with training information consisting of a large number of flow pairs. This includes two types of flow pairs: *correlated flow pairs* and *uncorrelated flow pairs*. The former consists of pairwise segments of the same Tor connection (e.g., ingress and egress segments). The latter consists of two arbitrary flows that are not part of the same Tor connection. The first class is labeled with value 1 and the second with value 0. During the testing phase, the CNN is then provided with an unlabeled pair of flows and produces as output the probability of two flows being associated, i.e., that these correspond to the entry and exit segments of a Tor circuit.

The internal structure of a CNN can be seen as a pipeline comprising several layers of data processing, typically *convolution, pooling, and fully connected* layers. The concrete configuration of the network tends to be highly specific to the target application domain. The architecture of DeepCorr's CNN is illustrated in Figure 2.7. The CNN takes as input four features for each flow – two vectors of inter-packet timings and two vectors of packet sizes – and is composed of two layers of convolution and pooling, and three layers of a fully connected neural network. The first convolution layer aims to capture the similarities between the input vectors that are expected to be correlated for associated Tor flows. The second convolution layer captures overall traffic features from the combination of timing and size information.

DeepCorr was tested by collecting a large dataset of flows using multiple virtual machines to connect to the 50,000 top Alexa websites via Tor network. This method ensured the real Tor network was used without collecting data pertaining to real users of the network. DeepCorr's authors have collected traffic for 2 weeks then waited 3 months and collected again traffic for a month. By using this time interval between collection the authors were able to infer that an attacker only needs to retrain DeepCorr system once per month to achieve good results.

2.5 Neural Networks with Encrypted Data

In the past few years, neural networks have been applied to a range of application domains yielding exceptional results in many fields [37], such as time-series predictions, anomaly detection, computer vision, speech recognition, and natural language processing. The fact that many applications deal with privacy-sensitive data (e.g., health or finance sectors) prompted the development of special neural networks that can be used with encrypted data [38–41]. Recently, several schemes have been proposed for allowing a CNN to operate with encrypted data. Such approaches ensure that both inputs and outputs are encrypted and therefore cannot be learned by the platform provider where the CNN is operated. While these schemes support multiple use cases, the most common is prediction-as-a-service (PaaS), a service where a provider trains a neural network and clients submit samples to be classified by the network while still preserving the confidentiality of their data. The key technical insight to allow CNNs to work with encrypted data lies in the observation that most operations performed by the CNN layers consist in multiplications and additions, and that some existing cryptography primitives, such as homomorphic encryption and multi-party computation, allow these computations to be performed without sharing knowledge about the raw data. Next, we provide an overview of the most interesting solutions that use neural networks with encrypted data.

CryptoNets [38]: This was the first practical system to demonstrate the feasibility of processing encrypted data through neural networks. The main motivation for this work was that of being able to outsource the processing of sensitive data using neural networks to computing platforms owned by third parties while preserving the confidentiality of both data and results. Essentially, the idea is to encrypt the data before shipping it to the third party, where it is processed using CryptoNets. The resulting values are encrypted in such a way that only the original data owner will be able to decrypt the value of predictions. CryptoNets were found to preserve data privacy while still being able to attain high accuracy results on the example network over the MNIST dataset[2]. The performance of the system was acceptable thanks to the optimizations performed, but orders of magnitude larger than normal neural networks. The size of messages exchanged by the system have also showed a drastic increase when comparing plain data with the encrypted counterpart. **MiniONN [41]:** The main goal of this system is to transform typical neural networks into oblivious neural networks supporting privacy-preserving predictions with reasonable efficiency. To attain this goal, MiniONN leverages a protocol based both on HE and two-party computation (2PC) to allow for performance gains while keeping both the input and the model private. In 2PC, two entities work together to compute a result while keeping their inputs private. Essentially, 2PC simulates a trusted third party that computes a function over both inputs, and returns a result to both participants without leaking any data between the two. In this particular scenario, this protocol is executed between a cloud-based prediction service, where the neural network model is held, and the client, who submits the encrypted inputs in order to learn the corresponding predictions.

When compared with CryptoNets, MiniONN achieves better results both in latency and message size. The authors show a 230-fold reduction in latency when considering the MNIST dataset and an 8-fold reduction in message size. It is worth noting that, since CryptoNets can process multiple requests at the same time, it can outperform MiniONN by 6-fold higher throughput. This however can only happen if the requests are from the same client and it can wait large amounts of time between requests and responses. Lastly, although the privacy guarantees offered by both systems to the client are identical, CryptoNets requires changes to how the neural networks are trained, whereas MiniONN supports all

[2]http://yann.lecun.com/exdb/mnist/

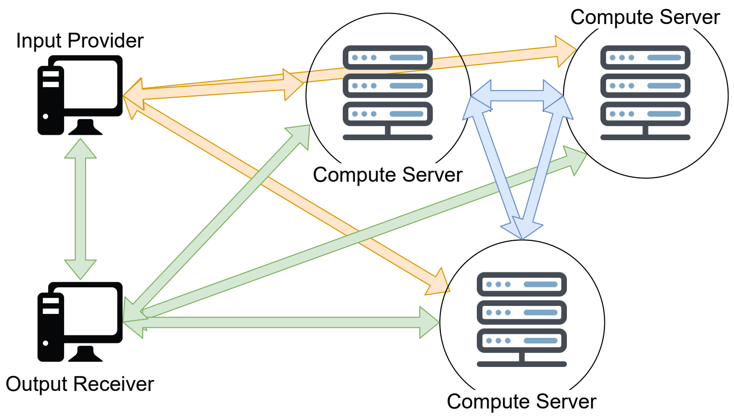


Figure 2.8: TF Encrypted architecture.

operations typically used by neural network designers.

Gazelle [39]: This recent system delivers higher performance than existing approaches by employing a new protocol and modifications to the behavior of typical neural networks. For instance, in contrast to the encryption scheme adopted by CryptoNets, which has prohibitively large computational costs, Gazelle employs a simpler (and faster) scheme named *additively homomorphic encryption* (PAHE). PAHE allows for packing multiple plaintexts into a single ciphertext, and then performing encrypted operations on this ciphertext. This scheme also benefits from the fact that no ciphertext to ciphertext multiplication is needed, just has in CryptoNets, further increasing the performance of matrix multiplications. Gazelle also greatly reduces the communication complexity of systems such as MiniONN.

In the foundation of the protocol used in Gazelle lies a combination PAHE and garbled circuits (GC). To evaluate the non linear activation layer GC is used. This circuit receives three vectors, *r*, *r*' and *x*+*r*. It is divided into three blocks: the first block computes the arithmetic sum of *r* and *x*+*r*, the second block computes, e.g., the ReLU activation function, and the third block adds *r*' to the result to obtain C's share of the output, *y* = ReLU(*x*+*r*)+*r*', given that S's share is *r*'. At this point, each party possesses an additive share of the input to the next iteration of the protocol, being S's share *r*' and C's share *y*. We are again at the beginning of the evaluation of a linear layer, and C starts by again encrypting its share, *y*, which in turn resets the noise introduce in the previous iteration by the homomorphic operations. This protocol continues until all layers have been evaluated.

TF Encrypted [3]: An open source project named TF Encrypted has leveraged some of the recent developments that allow neural networks to operate over encrypted data. It focuses on two main use cases: a simple prediction where the input is encrypted, and a private weight training for training the network using encrypted data. In particular, TF Encrypted provides a framework for enabling encrypted deep learning readily available for the community of TensorFlow[4] and Keras[5] developers. Both Keras and TensorFlow are widely used open-source libraries for supporting neural network and machine learning tasks. Through the use of TF Encrypted, one can build a neural network through a programming model similar to Keras with the added value of being able to perform secure predictions.

To achieve data privacy, TF Encrypted makes use of multiparty computation techniques such that

[3]https://tf-encrypted.io/
[4]https://www.tensorflow.org/
[5]https://keras.io/

Feature	CryptoNets	MiniONN	Gazelle	TF-Encryted
Offline computations only	yes	no	no	no
Full activation function	no	yes	yes	yes
Provides framework	no	no	no	yes
High throughput multiple data	yes	no	no	no
Low latency SP	no	yes/no	yes	yes
Supports existing CNN	no	yes	yes	yes
Small message / input size	no	yes	yes	yes
Fully hides CNN	yes	no	no	no

Table 2.1: Comparison between encrypted neural network systems.

only the respective data owner has full access to the clear data. Its architecture is based on three compute servers that never learn the clear inputs of the computations they are performing. Input providers distribute partial shares of their input among the three compute servers such that they can proceed with evaluating the network model on the input without actually learning the input. Figure 2.8 illustrates a simple setup with a single input provider, a single output receiver and three compute servers. This setup refers to the classification of a given input provided by the input provider. The model being used has been previously trained and is replicated on each of the compute servers. The classification process starts by the input provider distributing the respective input shares among the compute servers (yellow arrows). During the classification the compute servers perform the heavier computations but the input provider as an active role that allows the model evaluation to progress (yellow and blue arrows). The final result of the classification can now be unveiled by the output receiver to which both the input provider and the compute servers send their respective shares (green arrows), of the final result that allows its decryption by the output receiver.

The downside of using such a model is that the input provider has an active role during classification as the protocol depends on its interactions to progress. As a result of this active message exchange between parties, TF Encrypted performance is directly affected by the network infrastructure connecting parties. On the other hand we can have as many input providers and output receivers as required, i.e. two ASes and two LEAs as respectively input providers and output receivers.

Since it is compatible with the behavior of TensorFlow and Keras, TF Encrypted has an important advantage in the context of our work, namely that DeepCorr's authors provide an implementation of their neural network (see Figure 2.7) for TensorFlow. TF Encrypted can also compute important activation functions, such as ReLU, which is vastly used in DeepCorr's implementation. Some potential shortcomings of TF Encrypted, however, include its modest results in terms of runtime prediction and reduction in the accuracy of predictions due to the use of approximations within the neural network.

Discussion: An overview of the properties exhibited by the systems described in this Section can be found in Table 2.1. This table highlights the most relevant features for the considered systems: *offline computations only* refers to the ability of performing classifications without the need for both parties to communicate besides exchanging inputs and outputs, *full activation function* indicates the ability of computing linear activation functions, like ReLU, without performing approximations, *provides framework* indicates whether or not the system has a usable implementation, *high throughput multiple data* tells the ability of the system to process multiple requests in parallel, *low latency SP* means low latency for a single input in this case to perform a single prediction, *supports existing CNN* represents the ability of the system to support existing CNN configurations, *small message/input size* represents the input size after being encrypted and when applicable small message sizes if the system needs to exchange messages between parties while performing the classification, and *fully hides CNN* represents the system's ability to hide not only the weights of the neurons but also the size of the layers and the number of layers.

It is possible to observe in the table that, in general, systems evolved into a cooperative online computation method instead of just relying on HE due to its computational cost and input size. While this approach compromises the privacy of the neural network size, it still effectively hides the weights of the neurons and the input is still completely hidden. These choices also allow for attaining a better performance and smaller message/input sizes. TF Encrypted materializes these systems by providing a usable framework over TensorFlow.

By means of neural networks with encrypted traffic we can employ traffic correlation attacks based on such networks schemes. This paves the way for designing and implementing a privacy preserving extension for our system. TF Encrypted poses the ideal candidate for such an extension as it provides a usable framework very similar to Keras which is based on Tensorflow, used to develop DeepCorr.

Summary

This chapter presents an overview over the Tor network and Onion services which provide both sender and receiver anonymity. It introduces attacks that can be done against anonymity networks such as Tor and focuses on traffic correlation attacks. Within these attacks the state of the art uses deep neural networks to perform the correlation of regular Tor traffic flow pairs, which we will leverage in the next chapter to design Torpedo, a distributed system that allows LEAs to perform correlation of Tor onion service traffic. We also presented neural networks that can compute over encrypted data which will represent the foundation of a privacy preserving extension of the Torpedo system.

Chapter 3

Torpedo

In this chapter, we present the design and implementation of Torpedo, a system that provides a federated service for the correlation of onion service traffic. Section 3.1 presents the system from a high level perspective and describes its components. Sections 3.2 and 3.3 analyze in depth both stages of Torpedo's correlator pipeline. In Section 3.4, we present some privacy preserving extensions to our system. Finally, Section 3.5 explains how we have implemented a prototype of our system.

3.1 System Overview

This section provides an overview of the Torpedo system. Figure 3.1 illustrates a system deployment on a concrete (and simplified) usage scenario. Torpedo consists of three distinct components: *client*, *probe*, and *correlator*. It also involves three types of participants: LEAs, ASes, and a Data Processing Provider (DPP). The client consists of a piece of software that offers a flow query interface to LEAs. It enables LEAs to submit queries to the system aimed at determining correlated flow pairs that belong to accesses from Tor users to onion services. The client sends these queries to the DPP which is responsible for running the correlator logic. The correlator implements a two-stage pipeline of CNNs which is fed with flow pair information from participating ASes based on the criteria indicated in the query. This flow pair information is collected by Torpedo's probes that the participating ASes have deployed in their local networks. The correlator's CNN pipeline outputs the results of the flow pair correlation and sends these results back to the originating client. The client displays a list of flow pair ids and respective IP addresses along with the probability that each flow pair belongs to a Tor user's access to a given onion service.

To illustrate how Torpedo works, consider the particular example depicted in Figure 3.1. It represents Tor circuits traversing several ASes where a client A is exchanging traffic with B an onion service. The circuit enters the Tor network in AS1 and leaves it in AS2. Since it is an onion circuit both R1 and R'1 are Tor guard nodes respectively for the client and onion service. As A and B are respectively connected to the Internet through AS1 and AS2, both these ASes know the IP of the client and onion service respectively. In this example, AS1 is based in Portugal and AS2 in Germany. Each AS runs its own probe and is supervised by a LEA of its respective country. To use Torpedo, the security officers

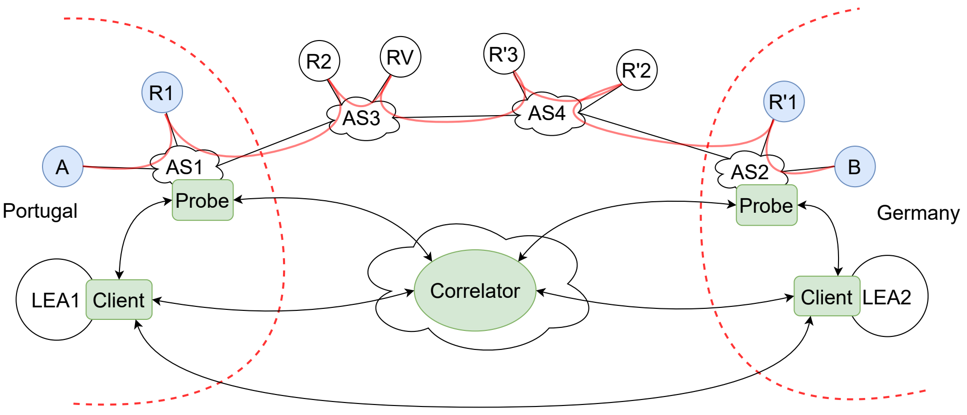


Figure 3.1: Torpedo system components: client, probe, and correlator.

of Portuguese and German LEAs – LEA1 and LEA2 respectively – run their own client instances. The correlator is hosted by a DPP which in this case is a commercial Predictions-as-a-Service cloud platform.

Consider, for instance, that LEA1 is interested in identifying correlated flows that belong to Tor onion service traffic entering in AS1 and exiting in AS2 on 28/12/2020 around 00h22. Since AS1 is based in Portuguese territory, traffic information pertaining to AS1 is readily available to be queried. However, given that AS2 is deployed in German land, it needs to obtain adequate permissions from LEA2. To execute such a query, the LEA1 officer submits the query to the local client. Both LEA1 and LEA2 clients execute a protocol that prompts the LEA2 officer asking for permission to observe AS2. If granted, the clients connect to the probes of each respective AS (AS1 and AS2), which look up their local logs, select the flows that match the query (i.e., satisfy the date and time constrains states in the query), and select the features of the matching flows for further correlation. The selected features are then forwarded to the correlator. The correlator runs them through it's correlation model, and returns the results back to LEA1 and LEA2 clients, which display the outcome of the query to the officers of both LEA institutions. Next, we clarify the design goals of Torpedo and explain in more detail how each component works.

3.1.1 Design Goals

In designing Torpedo, we were driven by the following main design goals:

- **Enable cooperative / federated Tor onion traffic correlation:** We aim to allow multiple ASes and LEAs across the globe to collectively and voluntarily participate in the correlation of Tor onion traffic correlation for the main purpose of cybercrime investigation.
- **Favor the correlation accuracy in detriment to correlation coverage:** Given that this system's goal is to assist the judicial authorities in cybercrime investigation, we design our system to deliver reliable and high accurate results (i.e., low false positives) trading off, if necessary, accuracy for coverage (i.e., tolerating higher rates of false negatives).
- **Deliver good performance for query execution:** Despite the potentially large number of flow pair combinations that may need to be correlated at a given time frame, our system should return query results in usable time periods, i.e., in the order of minutes.

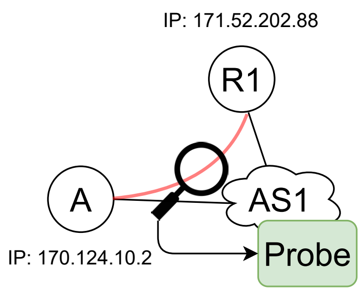
- **Susceptible to operate in a privacy-preserving manner:** In addition to designing our system based on a trusted party which whom all ASes can share flow data for correlation purposes – i.e., the DPP – we aim to design the correlation logic such that it can be implemented in a privacy preserving way where the flow correlation can be performed without the need for ASes to reveal the raw trace data to any trusted party.

Next, we provide more details on how each component is designed. We assume that all participants in the system, LEAs, ASes and the DPP trust each other.

3.1.2 Probe and Trace Collection

The probe is a component that must be deployed by every AS enrolled in the Torpedo system. It is responsible for logging relevant features of the local Tor traffic , namely: time of first packet, total number of packets, packet sizes, inter-packet timing distributions of flows, and the source / destination IPs. It also participates in the process of flow correlation upon request by authoritative LEA clients.

Probes must continuously record these features into a rotating log. Not all flows need to be recorded, though. Given that we are targeting the Tor network, specifically onion service connections, only Tor traffic information needs to be collected and stored in real time. Further more traffic traversing middle nodes is not relevant since they do not reveal the real IPs of the participating parties in the observed flow. As onion circuits imply a fully anonymous communication both the client and the server are exchanging traffic with a guard node thus we can focus feature collection only on traffic from or to guard nodes of the network which the list is publicly available. Although maintaining a real time log may seem as an obstacle, recent literature shows such logging activities can be performed very efficiently in AS-level networks [42]. Upon request by a client, the probe module queues and sends relevant flows from the rotating log to the correlator by means of a secure channel.



Query: Source IP: 171.124.10.2 Date Time: 1609114926000+1000

Source	Destination	Date Time	Features
170.14.120.56	171.52.202.88	1609114912456	Features
169.145.65.3	80.95.114.250	1609114912457	Features
		⋮	
170.124.10.2	122.232.32.153	1609114926109	Features
		⋮	
170.124.10.2	171.52.202.88	1609114926810	Features
170.124.10.2	171.52.202.88	1609114926919	Features
149.147.73.8	80.95.114.250	1609114926923	Features

Figure 3.2: Probe component rotating log and query response.

As a way of example, Figure 3.2 illustrates a Torpedo probe running on AS1 and collecting log entries. We can see that the probe module can record the traffic being exchanged from A to the Tor relay R1 but also that the probe collected entries for other IPs. Notice that all destination IPs are Tor guard relays as we are only interested in traffic traversing these nodes. Even though AS1 does not control all possible guard nodes, it can still record any traffic that it's clients exchange with guard nodes out of it's domain. Notice also that being a rotating log the oldest entry, marked yellow, will be deleted after a new entry is ready to be logged. The query in question refers to IP 170.124.10.2 hence the relevant log entries (marked green) will be queued for the correlator component.

3.1.3 Client and Query Usage Scenarios

Torpedo operates by responding to queries made to the system by LEAs. It takes as input a given query which calls more inputs from ASes involved in the resolution of the query. The queries refer to traffic traversing the Tor network where a client is accessing onion services. The output should be the IPs that match the query specifications.

There are the following use cases for queries to be issued to the system: (i) LEAs have suspicion on a given Tor user and want to deanonymize its traffic to onion services, (ii) LEAs have suspicion on a given onion service and want to deanonymize the Tor users that have accessed it, (iii) LEAs controls a honey pot onion service and want to deanonymize clients accessing it, and (iv) LEAs issue client requests to a targeted onion service and want to deanonymize its IP. All these use cases can be considered variants of the following two: (a) LEAs have a given flow sample and want to determine the IPs of the corresponding correlated flow (Tor client side or onion service side), or (b) LEAs have a given IP and want to determine the IPs exchanging traffic with it.

Thus, in a query, a LEA should specify the suspicious IP or flow along with relevant metadata such as ASes targeted, date and time of analysis. With this the system collects relevant data from the ASes and performs a correlation attack between the input flows to produce an output of corresponding IPs. Specifically, the client establishes a secure connection to engaging parties in response to a given query. Once a query is given as input to the system the client interprets the given query and contacts relevant ASes to initiate the fetching of flows. ASes queue flows that match the query specification i.e. from/to a given IP at a given time. For example, in Figure 3.2, the client submitted a query based on source IP (170.124.10.2) and a date / time. The client can now request the correlator to correlate all candidate flows obtained from the ASes' probes contacted by the client. Once the correlation process ends the results are sent from the correlator to the client and the query request ends.

3.1.4 Correlator and Flow Pair Correlation

The final component, the correlator, runs on a trusted third party – the DPP – and it is responsible for performing the flow correlation operations. It takes as input a set of flows sent by the ASes involved in a given query and outputs a probability of correlation between pairs of these flows.

Internally, the correlator is composed of a two-stage pipeline where each stage is implemented by its own neural network with a particular function. We designate these stages by *ranking stage* and *correlation stage*. The former is the first stage of the pipeline and it is tailored to pre-process a high volume of flow pairs very efficiently and return an ordered list of the pairs which are more likely to be correlated. The second correlation stage is dedicated to deep traffic analysis as it performs the

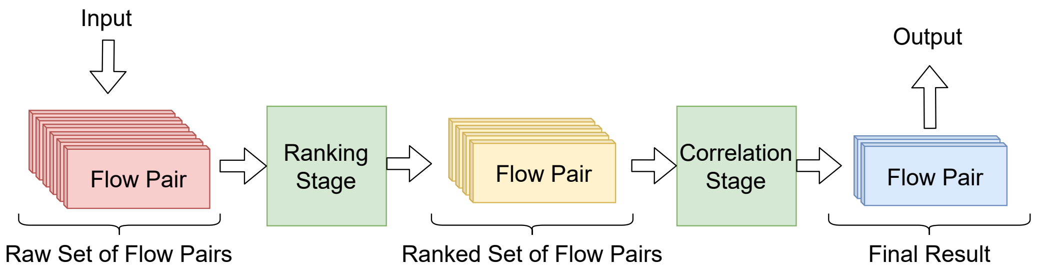


Figure 3.3: Correlator stages processing input flow pairs.

actual correlation of the flow pairs. The rational behind separating the correlator in two distinct phases rather than one only is its improved performance and the customizability. First, our two stage approach allows us to process large volumes of data in the ranking stage and use it as a filter for selecting the most probable correlated pairs to input to the correlation stage, which due to its nature takes orders of magnitude longer to process. Another benefit of this approach is allowing each stage to focus on a subset of the flow features leading to better over all performance of the model.

Figure 3.3 illustrates how both stages interact with one another to produce a final result of the possibly correlated flow pairs. The input to the ranking stage is the set of all pairs provided to the client by ASes for a given query. This stage processes all these pairs and sends its output of ranked pairs to the correlation stage. The ranking stage also filters out certain pairs that do not meet the minimum threshold to progress into the next and final stage. Finally, the correlation stage analyzes these pairs and outputs the final set of possibly correlated flows with their associated probability of correlation. The following two sections explain in detail how each of these stages work.

3.2 Ranking Stage

The ranking stage is composed of a simple neural network as described in Table 3.1. This neural network takes three inputs (the green circles in Figure 2.5) which correspond to three features: *packets in forward direction*, *packets in reverse direction* and *time of capture*. These features are derived from the flows of each pair to be evaluated by the correlator:

- Packets in forward direction refers to the difference between the number of packets sent from the first flow and the number of packets received at the second flow;
- Packets in reverse direction refers to the same as in the forward direction but in the reverse order being packets sent from the second flow and received by the first flow;
- Time of capture refers to the difference of time between the first packet of each flow.

The neural network then uses three fully connected hidden layers only. Each hidden layer contains a specific number of nodes (the purple circles in Figure 2.5). This number is indicated in Table 3.1 as “Size”. All the layers of this neural network use the ReLU activation function. The reduced number of layers of the neural network along with our feature selection allow for a highly efficient evaluation of flow

Layer	Details
Input Layer	Size: 3
Fully Connected 1	Size: 300, Activation: ReLU
Fully Connected 2	Size: 80, Activation: ReLU
Fully Connected 3	Size: 10, Activation: ReLU
Output Layer	Size: 1, Activation: Sigmoid

Table 3.1: Ranking stage's neural network architecture.

pairs. This efficiency is desirable as the system may have to correlate large amounts of flow pairs due to the combinatorial nature of pairs. For instance if two ASes capture each 100 candidate flows for a given query, then we are looking to 100 x 100 pairs to evaluate. We can see how fast this total number of pairs grows in relation to the number of captured flows per AS.

The output of this stage consists of a number ranging from 0 to 1 which implies the probability of correlation for a given pair. This result can then be used to select the pairs that are most probably correlated and should be further evaluated by the correlation stage. We can perform this candidate selection by means of a configurable threshold or by amount of flows to evaluate in the correlation stage. For instance we can have a threshold T where pairs with a result above T are selected as candidates. This is ideal when there are few constrains on the number of pairs that the correlation stage can process, but we can also order pairs by their result and select candidates by the most probable N when there is an imposed limit of a maximum of N pairs that can be processed by the correlation stage.

Previous attempts: Before arriving at this architecture we have iterated over multiple models.

1. Our first attempt was based on manually programmed heuristics, the intuition being that if the flows were correlated then an equal amount of packets should be observed at both ends and there should be a minimum and maximum interval for the difference between times of capture. However, experimental validation has shown that the number of packets observed at the client was invariably different from the number of packets observed at the onion service's end mostly due to packet aggregation at a relay level. This operation may depend on the network load, hence a static approach is not desirable, leading us to the use of machine learning techniques that would allow the model to learn the relations between the inputs that make a correlated pair.

2. Then we used a neural network similar to the one described above but with 6 inputs instead of 3. These inputs refer to the time of first packet, packets sent, and packets received of each flow (3 inputs per flow). However, there was a great disproportion between the values of the inputs which impaired the network's learning process. We applied a correction factor to reduce the differences between inputs and cap the most significant bits of the flows' absolute times but with little success. We overcome this problem by changing to 3 inputs and using the differences between these values. By passing the differences instead of the absolute values we are directing the network toward learning the differences between flows which is our intended purpose as these differences are the key to distinguishing correlated from uncorrelated pairs.

Layer	Details
Input Layer	Size: Flow Length*8
Convolution Layer 1	Kernel num: 1000 Kernel size: (2,30) Stride: (2,1) Activation: Relu
Max Pool 1	Window Size: (1,5) Stride: (1,1)
Convolution Layer 2	Kernel num: 500 Kernel size: (4,10) Stride: (4,1) Activation: Relu
Max Pool 2	Window Size: (1,5) Stride: (1,1)
Fully Connected 1	Size: 1500, Activation: ReLU
Fully Connected 2	Size: 400, Activation: ReLU
Fully Connected 3	Size: 50, Activation: ReLU
Output Layer	Size: 1, Activation: Sigmoid

Table 3.2: Correlation stage network architecture.

3. We have also experimented with another learning algorithm – Random Forests – using the same inputs as the neural network described above. This version of the classifier did converge albeit with overall worst performance. For this reason, we abandoned it in favor of the neural network architecture presented in Table 3.1.

3.3 Correlation Stage

The correlation stage makes use of a highly more complex model than the ranking stage. This is because it focuses on traffic analysis by processing the inter-packet timings and packet sizes of the two flows in both directions for a given pair. It is implemented by a CNN which consists of a modified version of the DeepCorr's CNN (see Section 2.4.3). This change was required to make the CNN work on our hardware.

Table 3.2 describes the parameters of each layer of the CNN. The input is composed of 8 vectors of length N where N is the number of packets the model is trained on. From these 8 vectors 4 relate to the first flow and the other 4 to the second. These vectors contain the inter-packet timings for the incoming traffic, inter-packet timings for the outgoing traffic, packet sizes for the incoming traffic and packets sizes for the outgoing traffic. The network presents two convolutional layers and three fully connected layers. The convolutional layers are aimed at training filters that will recognise patterns in the traffic data inputted to the network. The output of these filters is then fed to the fully connected network to process and learn the relations between them.

- The intuition behind the first convolutional layer is that it allows the network to learn features that correlate the same characteristics of both flows. To achieve this, we use a kernel size of (2,30) where we are processing 2 input vectors at a time, observing 30 packets from each at a given step i.e. the first 30 inter-packet timings of the outgoing traffic in the first flow and the first 30 inter-packet timings of the incoming traffic in the second flow. By using a stride of (2,1) we ensure this filters only match intended features and do not mix inter-packet timings with packet sizes nor incoming traffic with incoming traffic and vice versa, this also states that the filters will slide through the packets one by one.

- In the second convolutional layer this separation between inter-packet timings and size features is no longer present as the kernels are now processing data that came from both size and timing inputs. This layer serves for the network to learn more complex functions about the input flows that process both types of input (time and size related) together. We then proceed to the fully connected network which is composed of 3 layers before the final output which is again a number of 0 to 1 relating to the probability of that given pair being correlated.

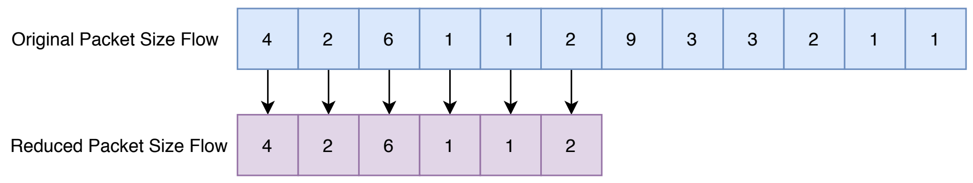
The main differences between our network model and the original DeepCorr's is the size of each layer. We reduced the number of kernels and neurons in both the convolutional and fully connected layers, respectively, so that the network could be used with our available hardware. Further details in our evaluation (see Section 5) show that this modification did not negatively affect the network's precision. Next, we present some alternative designs that we have explored.

3.3.1 Alternative Variants of the Baseline CNN

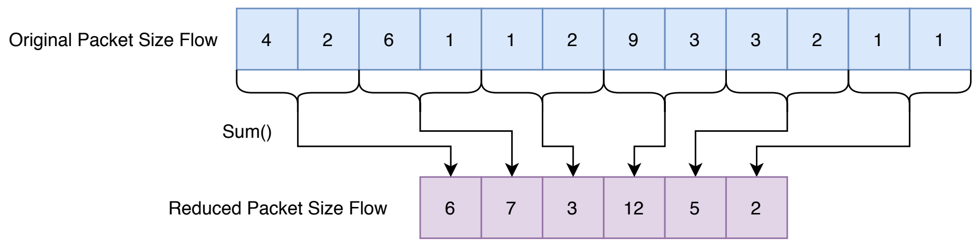
We now elaborate on alternative correlation stage designs that sacrifice precision for performance. Although we expect the variants presented next to perform worse in terms of precision than the presented correlation stage model, they may have a role in pre-processing large volumes of data similar to the ranking stage. One can use these networks to more quickly process the output of the ranking stage when dealing with extremely large amounts of possible correlated pairs. It is up to the user to decide what compromises it wants to make in terms of performance vs precision and possibly even exchange the correlation stage model by one that correlates faster but is less precise.

Flow vector manipulations: In the original DeepCorr paper, the authors claimed a 94% accuracy by extending the flow length provided to the network from 300 packets (80% claimed accuracy) to 450 packets. This is expected as the network has essentially 50% more data to perform the correlation and to further distinguish between otherwise seemingly correlated flows. This nevertheless interesting result may be not so much of an advantage in our setting due to the nature of the traffic we are correlating. Onion services tend to be much smaller than traditional websites consequently leading to smaller flows. Hence, we studied several flow length manipulations that are better suited for our workload. Figure 3.4 illustrates all flow manipulations that we considered and explain next:

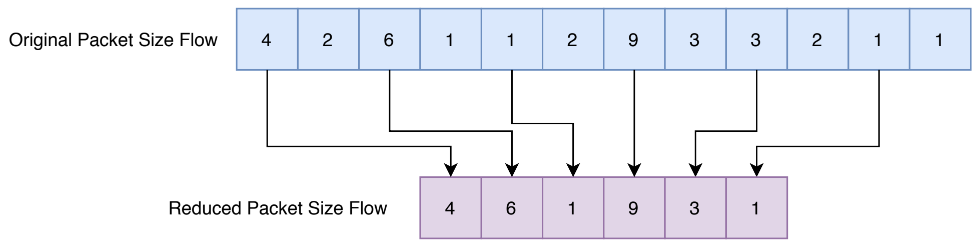
- *Packet capping*: A first strategy to increase the throughput of the CNN is by reducing the input size of the original packet size vector to a smaller size. This greatly affects the performance as the number of operations and convolutions is intrinsically related to the flow length of the input.



(a) Packet Capping



(b) Packet Aggregation by sum



(c) Packet Slotting

Figure 3.4: Reduce input vector size techniques.

However, one can also expect a drop in precision as there is less information on the input for the network to accurately distinguish similar pairs.

- *Packet aggregation by sum*: One solution to this lack of information is to reduce the flow length but, instead of capping the flows, aggregate multiple packets data into a single position of the input vector. This allows for a smaller input vector to contain the data of a larger observed flow. This aggregation is primarily done at the size features keeping the inter-pack timings untouched. At the size feature we can sum multiple packet sizes to only use one space in the vector essentially simulating a large packet corresponding to the sum of the packet sizes it aggregates.

- *Packet slotting*: Another solution that we explored is to, instead of capping the input flows by only using the first N packets information, remove every other packet from the input essentially reducing the total size of the flow while allowing the network to see packets further in time. However, this does not increase the amount of information we are providing to the network as the removed packets information is never embedded in the input as opposed to the sum aggregation technique.

Reduction of stride in convolutional layers: Another variation of the model that improves its performance is to reduce the stride used in the convolutional layers. Given that the convolutions are the most computing expensive part of evaluating this model, any modification that reduces these computations will significantly increase its performance. By increasing the stride we are essentially reducing the number of convolutions performed at each kernel of that layer. The original model uses a stride of (2, 1) for the first convolutional layer. By increasing the second coordinate of the stride to 2 instead of 1, we

a	b	Method A $\frac{a+b}{2}$	Method B $\frac{\sqrt{a^2+b^2}}{\sqrt{2}}$	Method C $\sqrt{a*b}$
0.10	0.10	0.10	0.10	0.10
0.50	0.10	0.30	0.36	0.22
0.80	0.10	0.45	0.57	0.28
0.90	0.10	0.50	0.64	0.30
0.90	0.50	0.70	0.73	0.67
0.90	0.80	0.85	0.85	0.85
0.90	0.90	0.90	0.90	0.90

Table 3.3: Combination methods results for multiple results of network a and b of ensemble model.

essentially halve the number of convolutions performed at this layer. It also means that we are know stepping 2 packets at a time instead of one which can negatively affect the precision of the model.

Reduction of the number of kernels: A final variation that we explored also leverages the number of convolutions performed by the model but instead of increasing the stride it reduces the number of kernels used. If we use 500 kernels on the first convolutional layer instead of 1000 we are also halving the number of convolutions performed at the first layer. However, now we are only generating half the features on the first layer meaning that the model is more restricted on the patterns it can recognise. As a result, this effect may hinder the precision.

3.3.2 Exploring CNN Ensembles

Besides all the above models where a single network is used for the correlation stage we have also designed a model based on two distinct but similar networks where the final result would be the ensemble of both networks output. Our expectation was to achieve higher results by having the two networks learn different features from the flow pairs that would reach a given true positive rate with a lower false positive rate as opposed to the single network approach. The rational being that correlated pairs would yield similar results on both networks but uncorrelated pairs would only confuse a given network while the other could still correctly classify it.

To this end, we devised multiple combinations primarily by changing the stride between the networks. The original strides as stated before are (2,1) and (4,1) for the first and second convolutional layers respectively. We combined multiple networks of the original stride with new networks of strides (2,2) and (4,1) and networks of stride (2,1) and (4,2). These configurations may lead the different networks to learn different kernels that recognise the same patterns which will intern produce agreeable results when a pair is indeed correlated but produce dissimilar results when the pair is uncorrelated.

For combining the results of both networks in the ensemble models we devised 3 different methods each encouraging different strategies (see Table 3.3). Assuming a being the result from the first network in the ensemble model and b being the result from the second network in the ensemble model the following combining strategies were considered: $\frac{a+b}{2}$, $\frac{\sqrt{a^2+b^2}}{\sqrt{2}}$, $\sqrt{a*b}$. Taking the second entry on Table 3.3, we have a given flow pair produce a result of 0.5 on network a and 0.1 on network b. These correspond to the probability of that given pair being correlated. We can see that for the different combination methods (A,B or C) the final result of the model changes. This final result corresponds to the model output which is the probability of that given pair being correlated. The effects of each method are as follows:

- **Method A:** As Table 3.3 shows, method A does not take any influence by the disparity of the results of both networks as expected being it the simple arithmetic mean. For instance the forth entry on the table presents highly dispar results between both networks (0.90 and 0.10) yet the final result for method A is 0.50 corresponding to the mid point of the outputs from networks a and b despite their disparity. We can take this result as somewhat a baseline for the next two methods.
- **Method B:** Method B influences the result by adjusting it towards the higher result from both networks. Taking the same fourth entry as example we can now see that this method pulls the final result, 0.64, towards 0.9. Yet when the results from both networks are closer together method B approximates its final result to the one of method A as can be seen in both the first and last two entries of Table 3.3. Although it is counter-intuitive, it might be interesting to evaluate how the ensemble models will perform under this combination method as it is expected to yield higher false positive rates due to its optimistic nature.
- **Method C**: Finally method C is the opposite of method B in as it skews the result towards the lower network result as we intended. This method is expected to produce lower false positive rates as it prioritizes less dis-par results. Taking for instance the result of all combination methods for results 0.90 and 0.10 of networks a and b of the ensemble model we can see that method A and B attribute a final result of over 0.5 even though one network only gave a result of 0.1 for that pair. In contrast, method C, notices this disparity and yields only 0.3 as final result for that pair.

Although the ensemble models seemed promising the results presented later in chapter 5 will prove this method performs worse than a single CNN in our setting of onion service traffic correlation.

3.4 Privacy Preserving Extensions

In the design that we presented so far for the Torpedo's correlator component, there are no guarantees of data privacy on both the input and the result of the correlations. This means that the ASes are required to share the locally collected flow information with a trusted third party: the DPP. However, as explained in Section 2.1.3, this requirement may deter many ASes from participating in the Torpedo system as it forces them to share sensitive data with third parties. To overcome this limitation, we designed the correlator's CNN pipeline so that it can also operate in a privacy-preserving setting. It particular, Torpedo supports a set of extensions that allows the system to perform correlations on encrypted traffic data as well as not learning the result of the correlation process. This essentially ensures data privacy to the original flows provided by ASes; neither the correlator nor the LEAs will learn any features of the traffic. It also hides the correlation results from both the ASes and the correlator, only making it available to the LEAs. Next, we present our basic approach to attain these properties and then present our preliminary design of these extensions which involve the substitution of the pipeline's CNNs presented above.

3.4.1 Technical Approach and Tradeoffs

When operating in privacy-preserving mode, both stages of the correlator's pipeline will make use of a cryptographic primitive that allows computations over encrypted data, namely *multi-party computation*. By using multi-party computation (MPC) protocols we can ensure that each party is the only one that has access to its data in clear and that the computation result is only decrypted by designated parties. To implement the CNNs' logic on MPC, we adopted the TF Encrypted framework (see Section 2.5).

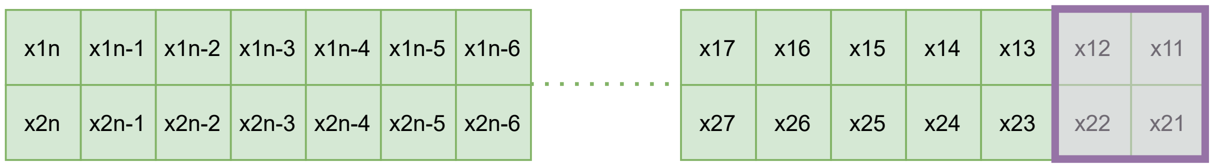
MPC, however, comes at the cost of performance. While simple additions are relatively fast, multiplications require much more computational power. Unfortunately, multiplications represent the majority of neural network computations. Another factor that will slow down the system is the communication delays between parties. Whereas in our baseline shared correlation setting all data can be present on the DPP's machine, in a privacy-preserving correlation mode the participating parties – notably the ASes – have an active role during the correlation process. Hence, the round trip times will create execution overheads as the data needs to be sent around during the correlation process.

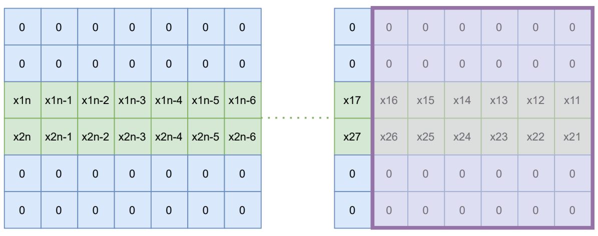
To mitigate the overhead imposed by the MPC operations we can employ the optimized model variations described in the section above which improve the performance at the expense of precision. Given the usage scenario intended for our system, this may end up being an acceptable middle ground. The idea being that, in the course of an investigation, a LEA can overcome the failure to detect a given correlated pair by performing multiple measurements over time. We assume LEAs will not perform any rash judgement solely based on a single observation. Instead, Torpedo will allow them to perform multiple observations of suspected traffic which will increase the confidence on the final result even though individual correlations may have reduced precision.

The geographic location of the involved parties is also an important factor of the system performance as it is directly correlated to round trip time of messages exchanged between them. The only modification that can be performed at the correlation level is to increase the amount of data processed each time mitigating the overhead imposed by the delay of message exchanges, i.e. by processing 100 flow pairs instead of 1 on each round we are effectively minimizing the impact of the message delays as messages contain data refereeing to 100 pairs instead of a single pair. This limitation may be taken into account when parties are setting up for correlation where the best performance comes when all the involved machines are closer to each other therefore LEAs and ASes may use this result when choosing suitable machines to run their correlation computations. Next, we present our preliminary design for the Torpedo's privacy preserving extensions for each correlator's stage.

3.4.2 Privacy-Preserving Ranking Stage

The first attempt at implementing a privacy preserving model for the ranking stage used SCALE-MAMBA[1], a framework that allows one to construct simple python like scripts that are then computed in a multi-party computation setting preserving information about the inputs of each party. This initial framework was chosen by the simplicity of translation from python code and by the fact that initial ranking stage approaches relied on simple computed binary heuristics instead of a more complex neural network model.


(a) Original


(b) Input padding 0s

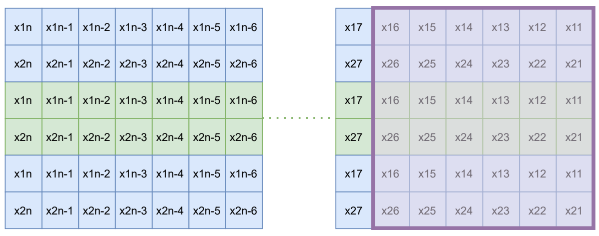

(c) Input padding repeat

Figure 3.5: Input padding modifications.

However, when we adopted a neural network for the ranking stage, we changed to the TF-Encrypted framework. This is a natural choice as otherwise we would have to implement all the infrastructure for evaluating neural networks on top of SCALE-MAMBA. Converting the previous Keras model to a suitable implementation in TF-Encrypted is straightforward in this ranking stage model which does not make use of convolutional layers. The interface provided by TF-Encrypted allows for a direct translation of the previous Keras implementation to the new TK-Encrypted and works as expected.

3.4.3 Privacy-Preserving Correlation Stage

Porting our CNN of the correlation stage to TF-Encrypted was considerably more troublesome. Although TF-Encrypted is very resourceful, it does not provide support for rectangular kernel shapes, only square ones. This in turn restricts both convolutional layers to square kernels as opposed to the original (2,30) and (4,10) rectangular ones.

To deal with this restriction, we have two natural options: either increase the first dimension or reduce the second one. Increasing the first dimension of the kernels to process the same amount of data in the second dimension (30, 30) (10, 10) is not adequate because the first dimension in the first convolutional layer represents the aggregation of the closely correlated features from each flow pair. Hence a more reasonable option would be to reduce the second dimension on both layers so that it matches the first, to produce the following kernel dimensions (2,2) and (4,4). With this modification the model can be evaluated by TF-Encrypted in a privacy preserving setting. However it comes at a cost of a possible degradation of precision. Reducing the second dimension in the first convolutional layer essentially translates to the kernels only processing two packets from both flows at a time as opposed to the original 30 packets at a time. This may cause the network to deviate from important traffic characteristics such as burst of packets.

Faced with this challenge we devised two possible modifications that would allow for the network

[1] https://homes.esat.kuleuven.be/ nsmart/SCALE/

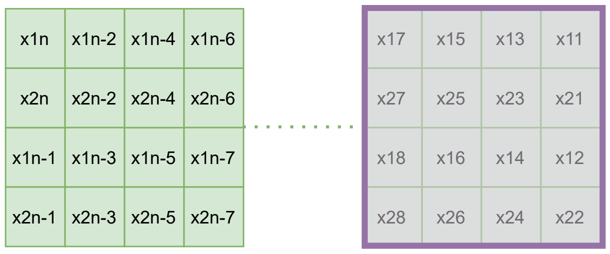


Figure 3.6: Input reshaping modification.

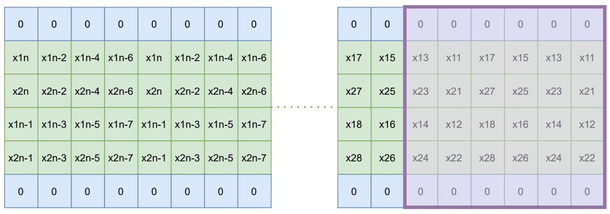


Figure 3.7: Both modifications combined.

to process the same larger amounts of packets at a time while still maintaining the intended model structure. These are input padding and input reshaping.

- *Input padding* refers to essentially adding extra vectors to the input so that a kernel of (30,30) would still only see the first two intended vectors and the extra data is essentially padding. Figure 3.5 illustrates this padding technique with the two possible padding schemes we consider, the kernel size being represented in purple. The simplest approach would be to add extra vectors of 0s and another approach which instead of filling the input with 0s repeats the input as it's own padding.

- *Input reshaping* refers to a sacrifice of stride (packets skipped per kernel slide) to allow more packets at a time to be processed by rearranging the input structure. Figure 3.6 illustrates how reshaping compromises stride in order to extend the number of packets processed at a time, with the kernel size again illustrated in purple.

Figure 3.7 illustrates both modifications being used together. We can see that we are processing double the amount of packets as in the solo input padding example but with a same size kernel (same purple kernel area), albeit with the cost of jumping 2 packets per kernel slide instead of 1 possibly affecting the model precision.

3.5 Implementation

This section describes the technology that we used to implement the models presented in the sections above. We also provide some relevant details about the training and testing of our models.

3.5.1 Development Frameworks

We focused on implementing the correlator component composed by the two stages of its pipeline. We used the same development framework to build both the neural networks as it allows us to focus on the

Parameter	**Value**
Loss Function	Binary Cross Entropy
Optimizer	Adam
Dropout Probability	0.4
Padding	VALID
Learning Rate	10^{-4}

Table 3.4: Common Keras models parameters.

design of the model without worrying about specific implementation problems. The framework that we used is Keras version 2.2.4-tf which works as an interface to TensorFlow version 1.15.0. The testbed for both training and testing consisted of a single machine running Ubuntu 18.04 and both correlator stages were implemented using python 3.6. Although both models could be constructed using only TensorFlow the use of Keras is not only simpler but will facilitate the migration to the privacy preserving setting later. The ranking stage training implementation is 191 lines long with the testing implementation being 199 lines long. The ranking stage model is however defined with only 11 lines of code. The correlation stage training implementation is 457 lines long with the testing implementation being 207 lines long. The correlation stage model is however defined with only 24 lines of code.

Keras allows a direct construction of the models by simply specifying the parameters of each layer, it then handles the generation of all the weighs matrixes by itself. Besides the previously stated parameters for each model (ranking and correlation stage), both models share the following parameters (when applicable): *loss function*, *optimizer*, *dropout probability*, *padding*, and *learning rate*. Table 3.4 lists the values used for both models as described below:

- The *loss function* is used to compute the error between the predicted output of the model and the expected result of the given example in the training dataset. In both models the loss function of choice was binary cross entropy as it is suited for data where there are only two possible classes, in our case correlated or uncorrelated.

- The *optimizer* is responsible for updating the models weights with regards to the learning rate based on the loss of a given result. Both models use the well known optimizer, Adam [43], which uses a stochastic gradient descent method that allows for scalability when using large amounts of data as in the case of neural networks.

- *Dropout probability* refers to the probability of a given neuron being excluded from a given prediction during training. It's aim is to prevent over fitting of the network to the training dataset. Overfitting essentially being the network memorizing all training examples but not generalizing the problem to unseen data. In both models we kept the dropout probability equal to 0.4.

- *Padding* refers to padding techniques applied to convolutional layers. Padding may occur when the input is not a multiple of the kernel. For all convolutional layers padding was set to 'VALID' which essentially means no padding is performed. When the kernel is reaching the end of a given vector it simply drops the last packets that did not fit in a complete kernel convolution.

- *Learning rate* refers to how fast the network converges when adjusting the weights during training. A smaller learning rate provides a higher granularity of search for the best results but also takes longer to converge. Larger learning rates may converge the weights faster but get easily stuck at local best results. In both cases a learning rate of 10^{-4} was used. We may also note that higher learning rates were leading to the correlation stage not converging to reasonable results.

As a final step in the implementation description, the result of both neural networks is passed through a sigmoid function to produce a final result between 0 and 1.

3.5.2 Training Implementation

As for any neural network based system, Torpedo's correlator models need to be trained before they can be used for actively correlating flows (i.e., testing). Training is performed exactly the same way for both correlation settings, i.e., with and without privacy-preserving extensions. We kept the original 200 epochs found on the DeepCorr github code[2].

For the correlation stage, for every 3 epochs, we evaluate the model to access its current accuracy. The accuracy is calculated as in the original DeepCorr paper by accessing how many flow pairs were correctly flag as correlated but to flag a pair we take the highest output of the network for a set of pairs where one is the original correlated flow and N other are uncorrelated flows matching the first flow from the original pair with any other second flow. Although this metric does not accurately represent the precision of the model we use it to save the model with the best scores in it. Other saving techniques that we tested such as by highest F1 score, lowest loss, or simply the latest epoch model proved to be less accurate in identifying the best performing model, although in many cases the highest accuracy model was also the latest epoch. This mid training evaluation is performed using the training dataset although there is no learning occurring when we do so, keeping the results fair.

Besides this training customization we also mention that, for both training and testing, a prepossessing phase of the dataset occurs before the flows are passed to the correlation stage network. Due to the disparity in order of magnitude between the inter-packet timings and the packet sizes we multiply both by an adjusting factor that not only brings both features closer but also close to 0 which is relevant for the network training capabilities. Inter-packet timings are thus multiplied by 10^3 and sizes by 10^{-3}.

Summary

This chapter has explained the design of Torpedo, a system that leverages traffic correlation attacks to deanonymize communications between Tor clients and Tor onion services. Torpedo allows for LEAs and ASes to cooperate in the deanonymization Tor onion circuits. The system design employs a two stage approach that enables large amounts of flow pairs to be processed in useful time periods. We also presented a privacy preserving extension that allows the system to function over encrypted data without learning the traffic characteristics it is correlating. Finally we described an implementations of

[2]https://github.com/SPIN-UMass/DeepCorr

the correlator module of the system which will be thoroughly evaluated in the following chapters. The next chapter refers to the collection and analysis of the dataset we will use in the system evaluation.

Chapter 4

Harvesting an Onion Dataset

Our evaluation of Torpedo encompasses the ability to perform traffic correlation using real-world traffic between Tor clients and onion services. Collecting such a representative dataset is in itself a non trivial task as it is not possible to intercept the Tor traffic at the endpoint of existing onion services. In this chapter, we describe how we overcome this challenge by generating a realistic onion traffic dataset that we then use in our empirical evaluation. This process involved several steps which we progressively unfold in this chapter. We begin by explaining how we collected real samples of existing onion service pages (Section 4.1), and then how we pre-processed the collected data for generating a synthetic dataset containing traces of Tor client accesses to emulated onion services (Section 4.2). Lastly, in Section 4.3, we describe our methodology to characterize the traffic generated by onion services.

4.1 Onion Service Crawling and Processing

To generate and collect realistic correlated onion traffic samples, we need to recreate instances of real-world onion services in a controlled environment. This section describes how we collected and characterized real samples of onion service pages to have a representative subset that we can use.

4.1.1 Gathering a Real-World Onion Service Dataset

In order to generate a realist setting for capturing onion service traffic it is essential that the generated traffic by the respective onion services is representative of real web pages. Current literature [44–46] describes the characteristics of onion services as being substantially different from a regular clearnet website. Web pages are usually much smaller in size and many do not link to any outside website or service [44, 45] however 20% seem to import resources from the clearnet [46].

Our approach to gathering such a realistic dataset is by crawling the space of existing onion services. By using copies of such onion service pages our dataset will accurately represent traffic that is being generated by real accesses to onion services which differ from regular web traffic in the content they serve. As explained below, to gather these pages our crawling methodology is based on existing and proven techniques [26]. This ensures that our dataset does not suffer from assumptions that would have otherwise be made when synthetically generating the onion service web pages.

Crawling methodology: To obtain realist web page samples we crawled the dark web for onion services and collected a copy of their index HTML page along with all the assets (images, css, javascript) necessary to fully recreate the pages. Our crawling strategy is reminiscent of the crawling procedure of Overdorf et al. [26] and was done by accessing publicly available .onion URLs. These URLs were fetched from a well known dark web search engine Ahmia[1]which provided 11033 unique .onion URLs. The first crawl fetched just the HTML text and allowed us to assess how many services were actually online and if there were any duplicates (.onions pointing to the same onion service page).

Onion service pre-processing: From the original 11011 services, 1270 did not respond. After comparing the HTML files fetched there were enormous amounts of duplicate onion services, only 1447 did not have any duplicate, with 7860 onion URLs representing at least 10 duplicates of any given service and 5877 onion URLs being at least 100 duplicates, echoing the findings from Burda et al. [47]. These duplicates were often from onion services providing some kind of easy money technique such as bitcoin multipliers or prepaid credit cards most probably fishing scams [48]. After accounting for these duplicates we are left with 1936 unique onion services although still many represent scams very similar to the ones stated above, differing only slightly in HTML. We then proceeded to check the HTML and request the missing assets that would allow for an accurate reproduction of all the pages.

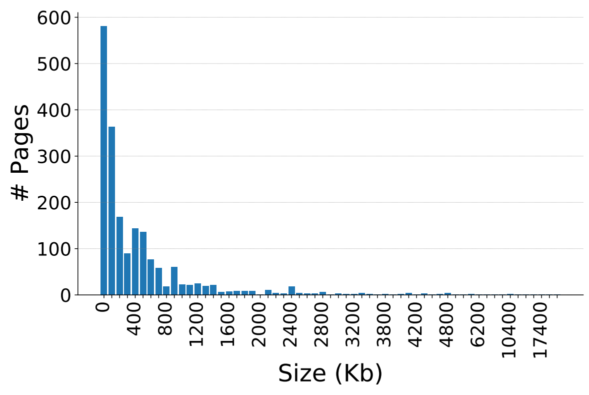
4.1.2 Onion Services Characterization

After collecting our dataset of crawled onion service pages, we performed an additional characterization step. This step allows us to categorize the existing onion services according to their most relevant characteristics. Based on these categories, we can then select a subset of representative onion services which can be used for creating onion service mock-ups and generating synthetic onion service traffic.

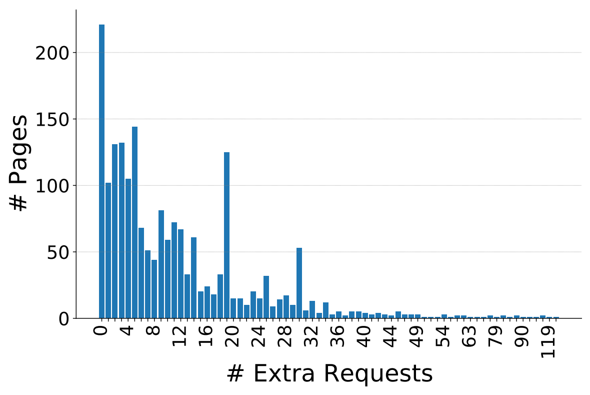
To provide a realistic setting without utilizing all the 1936 onion services collected, we categorized the pages in regards to two different aspects that would mostly impact the traffic patterns generated when accessing a particular onion service. These are the *total size of the page* and the *number of extra requests* (i.e., assets such as javascript, images, css) required to fully construct the page. We consider these two features to be the most relevant since: i) the total size of the page including all assets directly impacts the size of the generated traffic, larger pages generating more Tor cells to fully complete the request, and ii) the number of extra requests also directly impacts the generated traffic since for each asset referenced in the page, the client needs to explicitly make an extra request to get it. This in turn is going to generate more distinct packet bursts in the client-onion service interaction.

Figure 4.1 (a) shows the distribution of pages per size grouped in intervals of 100 KB. We see that the average size falls in between 500-600KB but the median size is in the interval of 100-200KB. Figure 4.1 (b) shows the distribution of pages per extra requests to fully render the page. It shows that the average number of extra requests is 11. However, the median extra request count is about 7. By combining

[1]https://ahmia.fi/



(a) Sizes



(b) Extra Links

Figure 4.1: Collected onion service pages distributions.

Size(KB)	Number of Pages	Extra Requests	Number of Pages
0-100	11	0	4
100-200	7	1-3	8
200-300	3	4-6	6
300-400	2	7-9	4
400-500	3	10-12	3
500-600	3	13-15	2
600-700	1	18-19	3
700-800	1	25	1
900-1000	1	30	1

Table 4.1: Selected onion service pages distributions.

these 2 metrics, we have selected a total of 32 pages to be representative of the whole set. The sizes of the selected pages ranged between 0-900KB and extra requests ranged from 0-30. Table 4.1 provides further details on both features selected for the set of representative onion services pages. The first 2 columns refer to the size of the pages, i.e. the first entry states we have 11 onion service pages that are within 0-100KB in size. The last 2 columns refer to the number of extra requests per page, i.e. the second entry states we have 8 onion service pages that require 1 to 3 additional assets from the server to fully render. Note that there is no restriction between the size and extra requests meaning a page can be 0-100KB and have 30 extra requests or, in the other extreme, be 900-1000KB and have no extra requests albeit in both cases no other page in the set could respectively have 30 extra request nor be 900-1000KB in size. Having this subset of web pages that represent the onion service space we then proceeded to setting up a testbed that allow for the collection of the onion service traffic generated when accessing this pages through the Tor network.

4.2 Synthetic Generation of Onion Service Traffic

This section describes our setup of a realistic environment which allowed us to collect onion service traffic samples at both the Tor client and onion service side. It enabled the construction of a dataset of correlated flow pairs. We start by describing the architecture of the clients and onion services, then

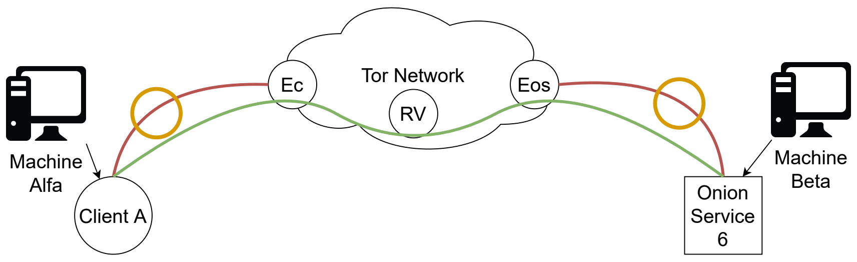


Figure 4.2: Sample capture model.

proceed to the two different experiments of static onion service pages and realistic onion service pages.

4.2.1 Testbed Overview

To collect onion service traffic we deploy a similar configuration to the one depicted in figure 4.2. We control both machines Alfa and Beta such that we can observe and collect traffic locally to their respective entry node in the Tor network. As a sample capture machine Alfa emulates a client, A, and machine Beta serves an onion service (OS), 6. A correlated flow pair is generated by accessing OS 6 through client A and capturing the corresponding traffic at machines Alfa and Beta. The flows captured at each machine form a correlated flow pair. Note that even though we control both the client and the onion service alike, they use real the Tor network to communicate with each other as in a real onion service. We simplify the onion circuit illustrated by the green line, which enters the Tor network through the entry relay of client A (Ec), traverses multiple nodes inside the Tor network including the rendezvous (RV) and finally reaches the onion service exiting the Tor network through the onion service entry relay (Eos).

Testbed setup: To collect multiple correlated flows with different characteristics we deployed multiple onion services and clients around the globe. To deploy both onion services and clients we used 8 different machines in the following locations: Amsterdam, Finland, London, Los Angeles, Sao Paulo, Singapore, Sydney and Tokyo. Each of these machines served both as a client and an onion service provider. The capturing process proceeds sequentially for each of the 8 clients in each machine. Each client accesses all onion service pages hosted on the remaining machines 25 times per page. Note that a single onion service provider can serve multiple onion service pages with different characteristics effectively emulating multiple onion services.

4.2.2 Kinds of Emulated Onion Service Pages

We describe the generation of two different kinds of onion service pages: *statically generated* and *realistic* onion service pages. They allow us to gather valuable insights over different characteristics of onion services. We aim at reporting differences between onion services which serve a set of (possibly cross-domain) assets from those which simply serve HTML content.

Statically generated onion service pages: This setup uses randomly generated text files of different sizes to emulate the onion services on our testbed. In this setting we only have a single onion service per machine making a total of 9 onion services. Each machine hosted a single text file and text file sizes ranged from 100-900 KB in steps of 100 KB. Each client accessed 25 times a given onion service while refreshing its circuit every time, effectively emulating a different client for each access (this is so because a new circuit is created). The captures are started prior to any onion service contact by the clients. Thus, the samples contain the traffic generated by establishing a connection between the client and onion service on both sides. It includes a client's interactions with the Tor Directory Table, an Introduction Point, and a Rendezvous Point, and the onion service's interactions with an Introduction point, and a Rendezvous Point. Accesses to onion services were made using curl and routing it through the Tor proxy on the client.

Realistic onion service pages: In this setup of the dataset we use proper HTML web pages ranging from simple HTML-only pages to more complex pages including multiple assets such as images, css, javascript etc. The web pages used are real copies of existing onion services which we crawled using publicly available .onion URLs. From the crawled web pages we select a representative set of 32 unique pages to populate the 9 machines serving onion services. The criterion for this 32 page set is described above in section 4.1 when we discuss our crawling methodology and onion services collected. Next, we provide mode details on the method for generating onion service pages.

4.2.3 Dataset of Realistic Onion Service Traffic

In this setting each client accesses a given page 25 times. Each onion service hosts 4 distinct pages simulating 4 different onion services. Notice that while in the previous iteration `curl` could be used to simulate a onion service access a different approach has to be taken here since we now want to also capture traffic generated by requesting assets referenced by the HTML page which are only needed when fully rendering the page. Hence, curl does not perform these requests. One could use `curl` once more to request any present references in the main HTML file to generate the desired traffic.

Instead, we used the Tor Browser itself just as a regular onion service user would do. This makes this setup highly realistic as not only are the pages themselves from real onion services, but also the client-side traffic is generated by the most commonly used and recommended browser to navigate onion services. To control the Tor Browser we used Selenium [49], a tool that allows one to take full control of the browser programmatically while simulating a regular user. This allows us to automate the requests to onion services programmatically using the Tor Browser which fully renders the page requesting both HTML and all necessary assets to do so.

Traffic collection procedure: We performed multiple iterations until we have finally stabilized our setup.

1. In the first iteration, we were relaunching the browser for every new client request we made. Our intention was to clear the browser's caches and avoid interference between different client requests. However, this also generated considerable amounts of additional traffic that made the captures very noisy. This is because, when the browser is launched, it makes a series of requests of its own, a more in depth analysis of the behaviour of the browser is also described below.

2. The second iteration solved this issue by restarting the browser only when changing the target page being requested. The browser would remain opened for the 25 requests to a given onion

Characteristic	**Value**
Onion services	8
Onion service pages	32
Pages per onion service	4
Clients	8
Requests per page	50
Pages per client	28
Emulated onion services	32
Emulated clients	224
Total number of flow pairs	11 200

Table 4.2: Final dataset characteristics.

service page and only when changing this target page it would be restarted. A delay was also added before starting the capture procedure allowing any browser related requests to complete.

3. To further improve the quality of the dataset in a third iteration we removed the circuit establishment to the onion service out of the capture as this step should be identical in all the clients connections hence not adding any relevant information when correlating clients' accesses to onion services. To accomplish this we simply make a first not captured request to the onion service which will establish the onion circuit and only then make the captured request which will reuse the previously established circuit to the onion service. This is also more realistic as a given user is expected to navigate through the onion service hence producing much more traffic beyond the onion circuit establishment protocol than just the first request that generates this extra traffic.

4. In final fourth iteration we made 50 instead of 25 accesses to each onion service page to increase the data available for generating the datasets.

Final dataset characteristics Table 4.2 enumerates the characteristics of the final dataset. A total of 8 machines were serving a single instance of an onion service with 4 distinct pages. A total of 8 client machines were used however since we were refreshing the clients Tor circuit once all requests to a given page are complete we are effectively emulating 224 distinct clients. Each emulated client performed a total of 50 requests to the same onion service page giving a total of 11 200 accesses when considering all the 224 emulated clients. This represents 22 400 distinct flows with 11 200 correlated flow pairs.

4.3 Dissecting Onion Service Traffic

The dataset that we have collected was used primarily for evaluating Torpedo's correlator pipeline. To better understand the results of our evaluation – which will be presented in the next chapter – we find it particularly useful to gain some insight on the patterns that get generated whenever a Tor client accesses an onion service. We will see that some of these patterns help to explain the effectiveness of the traffic features that we use in Torpedo's neural networks for efficient flow pair correlation. In this section, we aim to study the network traffic generated by accessing onion services and identifying traffic patterns and features that may influence traffic correlation attacks. We start by describing the experiment we performed for this purpose, and then we present our analysis of the captured traffic.

4.3.1 Experiment Overview

We carried out an experiment that aims to compare the number of packets being exchanged between a Tor client and an onion service. We use two different machines, one for hosting the client and another for hosting the onion service, and Selenium to emulate a client accessing the onion service. The packets exchanged both endpoints are transmitted through the Tor network.

The experiment consists of capturing the generated traffic at both endpoints and make an account of the evolution of the number of packets sent / received by the client / onion service. We first start by initializing the Tor process on the client and wait some time (10 seconds) until the resulting chatter with the Tor network ends. Then we initialize the Tor browser and wait for another 10 seconds before making any web browsing requests. Lastly we begin the browsing session on the Tor browser by performing a series of 5 consecutive requests to our onion service separated by a 5 seconds interval. Each request is performed after first evicting the browser's cache so that the website is fully loaded from the server.

We used three different onion services with different characteristics each. One is a simple "hello world" web page with a single html file without referencing any other files/assets from the server. The other two are copies of real onion services one with 400KB of data to load and the other with 900KB of data, both containing multiple references to other files in the server to fully render the page (CSS, JS, Images, etc). All three onion services were hosted on the same machine. The client was always excecuted on the same hardware. The machines running the client and onion services where located in different physical locations around the globe. Next, we highlight our most interesting findings.

4.3.2 Traffic Pattern 1: Tor Client Fingerprint

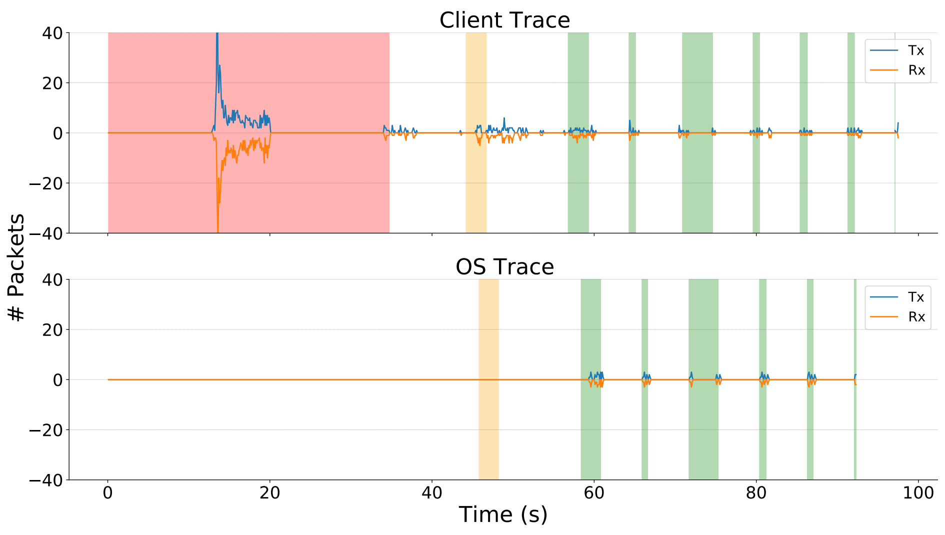


Figure 4.3: Full trace hello world onion service.

Figure 4.3 plots the transmitted and received packets over time for the entire duration of the experiment for the first onion service (Hello World HTML page). We can see a large burst of packets being transmitted and received by the client when we first start Tor (highlighted red on the chart). This is expected as the client needs to establish its circuits in this phase as well as contact Tor's directory authorities. As a result, large volumes of traffic are generated. Some additional traffic is generated in comparison to using the standard Tor browser due to the use of Stem, which is a python library that we used to control Tor. This library spawns more circuits at startup then usual. However, this effect does not impact the experiments as these circuits are not used at all during our browsing session.

The second phase of the experiment is to launch the Tor browser. This action is highlighted yellow in said chart both at the client – where the browser was executed – and at the onion service – for visualizing the corresponding time frame at the destination endpoint. We see that launching the Tor browser causes some packets to be exchanged between the browser and the Tor network even though no page has been loaded from the onion service. This traffic may be caused by the browser searching for updates and contacting services essential to its functionality. However we did not confirmed the purpose of this traffic.

Lastly, the client performs a series of 5 requests to the onion service (each highlighted green) spaced by 5 second intervals. As expected, there is a lag between the first request where the circuit to the onion service needs to be established and subsequent requests where this circuit is already established and can be reused for the request. Most importantly, we can clearly identify a distinctive "preamble" at the client side which consists of the traffic for starting up the Tor client and the Tor browser. This preamble can be used as a fingerprint to distinguish client-side flows from onion-side flows.

4.3.3 Traffic Pattern 2: Onion Circuit Establishment Fingerprint

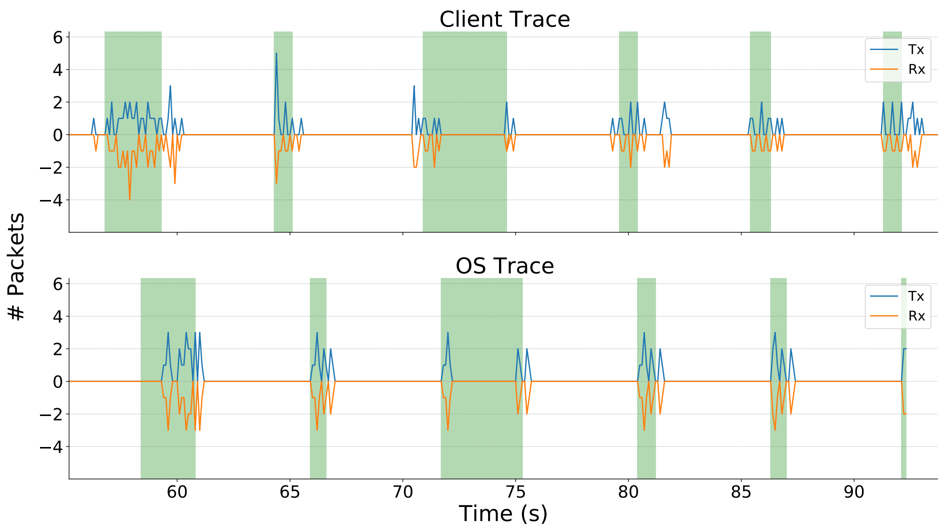
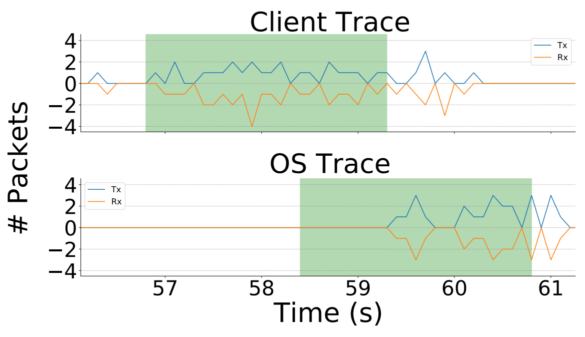


Figure 4.4: Zoom in series of requests.

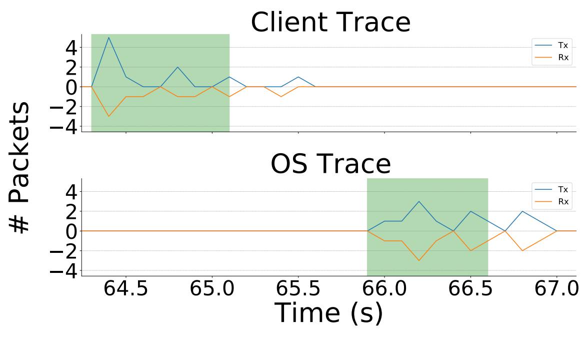
Figure 4.4 zooms in the previous plot for a more in-depth view on the requests. We can indeed see a larger number of packets being exchanged on the first request by both client and server as expected.

Notice also that there are significantly more packets exchanged at the client for the first request then at the onion service, this is a consequence of how onion services operate and how an onion circuit from a client to the onion service is created. When the onion service is first created it establishes circuits to all its introduction points and publishes these introduction points to a Tor directory. This interaction only happens at this startup phase for the onion service hence it does not show up in our plots. The onion service is now ready for accepting connections.

When our client first wants to connect to the onion service it needs to know its introduction points first, which it can by contacting a Tor directory. Then the client needs to contact the introduction point that the onion service has already set up. Only then both parties create a Tor circuit to the chosen rendezvous point and start exchanging messages. This extra effort for the client is clearly visible in the amount of extra packets exchanged for the onion circuit establishment when compared to the onion service for the same request (see the first green region in Figure 4.4 around 40 and 60 seconds for each endpoint). Subsequent requests do not generate nearly as much traffic, which is expected as the onion circuit is already established. We can also observe a trend of more packets being observed at the client side than on the onion service, which may represent possible control cells sent by the client to the network and may also be explained by possible relay aggregation more Tor cells on the onion service side.



(a) First Request.



(b) Second Request.

Figure 4.5: First and second requests to hello world page zoom.

Figure 4.5 (a) and (b) plot a close up of the first and second requests, respectively, where we can clearly see the difference between them. In the former case, it is necessary to build the onion circuit; in the latter, such a circuit has already been established. It is also clear that the client, even for subsequent requests, is exchanging more packets with the network. Results from the experiment with larger onion service pages are similar with the exception of the traffic being exchanged which is considerably more.

4.3.4 Traffic Pattern 3: Tor Requests Fingerprints

Figures 4.6 and 4.7 plot respectively the experiment for the onion services with 400KB and 900KB pages. Both pages are based on realistic onion services previously crawled as described in Section 4.2. In both cases the first request has a start up phase taking longer and subsequent requests are faster as already observed in the previous section. We can now see how the HTML page is clearly fetched first and only then images and assets are requested as the charts present an initial burst of packets but also

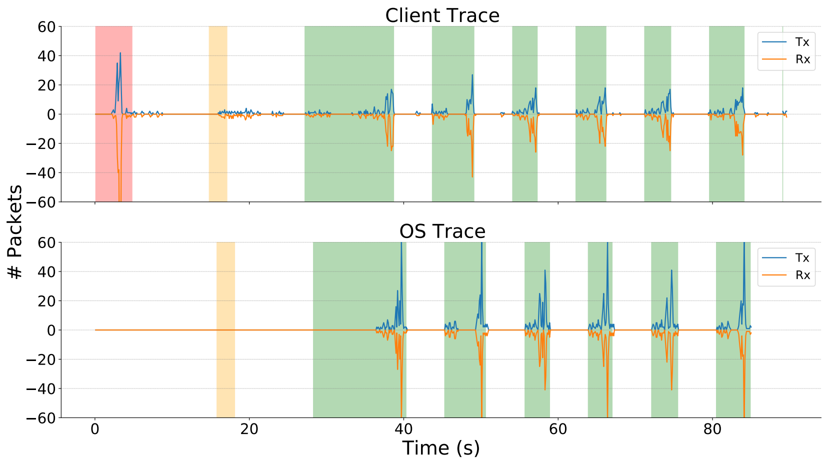


Figure 4.6: Full trace 400KB onion service.

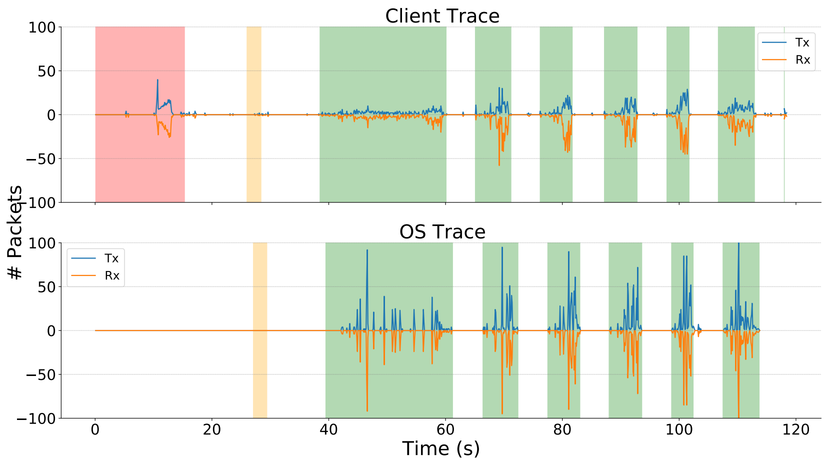


Figure 4.7: Full trace 900KB onion service.

subsequent bursts representing the assets required to fully render the page. This change is present in both sides, client and server, and generates substantially different traffic patterns depending on the page being loaded as can be seen when comparing both onion services, 400KB and 900KB.

Figure 4.8 zooms in on the second and third request to the 400KB onion service. We can see a clear initial request phase where fewer packets are being exchanged between both client and server. This is most likely referring to the HTML page as in these requests the onion circuit is already established. Once this initial phase is complete there are noticeable packet bursts which represent the assets referred in the HTML that the client is now fetching to fully render the page. This is also noticeable by the total time of the request. Whereas in the static simple onion page the request completed in less than a second (when the onion circuit was already established) in this case it takes about three seconds to complete since not only the overall data being fetched by the client is larger but also by the delay introduced in the client having to make multiple requests to fully complete the onion service page. Another key aspect that can be seen in Figure 4.8 is how requests to the same exact onion service page produce slightly different traffic patterns. We can see that the overall volume of traffic exchanged in both requests is similar as expected but even though the same circuit is being used there is still a noticeable difference in packet bursts between the second and third requests. This difference is key to enable a correlator model to distinguish between accesses to a same or very similar onion service.

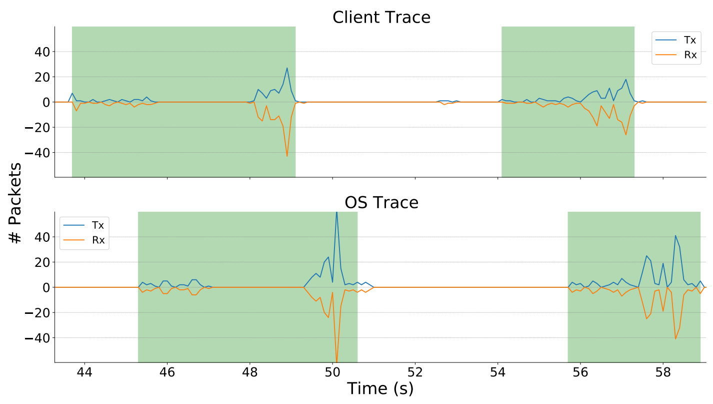


Figure 4.8: 2nd and 3rd requests' close up for 400KB onion service.

The past three sections thoroughly analysed the traffic generated at both the client and server side when accessing an onion service. We observed multiple sections of the flows generated and concluded that not every segment is relevant when performing traffic correlation attacks. The initial Tor startup phase as well as the browser startup generate lots of traffic that as no use in a correlation attack. The first request to an onion service also presents an initial phase at both the client and server where the patterns generated will not distinguish between different flows as they correspond to the creation of an onion circuit similar in every access to an onion server. The key segments to perform accurate traffic correlation thus correspond to accesses being made when the circuit is already established. This is especially evident in smaller onion service pages where the circuit establishment traffic is similar in volume to the page it self. Larger onion service pages may not suffer as much when correlating the first request as opposed to subsequent requests as the volume of the page is significantly larger than the circuit establishment traffic.

4.4 Summary

This chapter has described how we generated a dataset of onion service traffic while maintaining a realistic representation of existing onion services. We also provide an analysis of the traffic captured that helps us to understand the network interaction between the client and onion service with the Tor network. This provides valuable insights on which specific segments of Tor requests can be used for accurate correlation attacks. The next chapter presents a thorough experimental evaluation of Torpedo making use of the aforementioned dataset and insights.

Chapter 5

Evaluation

This chapter presents the evaluation of Torpedo. The end goal of our system is to provide a tool that can correlate Tor onion services traffic while maintaining a *high precision* and *reasonable performance*. By precision we mean the accuracy of the system in regards to its correlation capabilities. In other words, we want to assess how accurately our system can identify correlated and uncorrelated pairs as well as how often it confuses both classes. By performance we mean how long the system takes to process a given amount of data. We focus specifically on assessing the correlator pipeline and the specific models of each of its stages. Each stage will be evaluated in a slightly different manner as in the ranking stage we want to gauge how the ranking performs in contrast to no ranking order, and in the correlation stage we want to measure the true positive rate in contrast to a low false positive rate. Next, after we clarify our methodology and metrics, we present the evaluation results for each of these stages independently, and then the results of their combined operation when linked together in a full pipeline mode. Lastly, we present some preliminary evaluation results of Torpedo operating in the privacy-preserving mode.

5.1 Methodology and Metrics

This section describes how the collected trace dataset (see Section 4.2.3) is used to generate a suitable training and testing dataset and how the system will be evaluated.

Uncorrelated pair generation: The collected flow pairs only contain correlated pairs as expected. However, to train our classifiers uncorrelated pairs are essential as the network needs to learn how to distinguish between both classes. To generate an uncorrelated pair we can simply match the flow captured at the client with a flow from another connection. This scheme allows us to not only generate arbitrary numbers of uncorrelated pairs but also have some control over the distinguishability between pairs. This leads us to four distinct categories of uncorrelated pairs:

a) A client flow is matched with the onion service flow from another client to another onion service page; this essentially maximizes the distinguishability as the full circuit is completely different as well as the generated traffic;

b) A client flow is matched with an onion service flow from that same client but to a different onion service page, where although the generated traffic is also completely different the guard nodes of the client is the same which may reduce the distinguishability;

c) A client flow is matched with an onion service flow to that same exact page, but the onion service flow was generated by another client, and therefore the circuit is totally different;

d) A client flow is matched with an onion service flow to that same exact page; the onion service flow was generated by the same client but from a different access which makes the circuit up until the rendezvous point exactly the same.

Cases c) and d) further reduce the distinguishability when compared to a) and b). Case d) is less interesting since it represents a user requesting the same page using the same circuit.

Ranking stage evaluation datasets: For evaluating the model of the ranking stage, we devised a series of specific datasets. All generated datasets consist of 99 uncorrelated flow pairs per correlated flow pair. The choice of uncorrelated pairs was made by considering several combinations, either by restricting the accessed onion service or by manipulating the time difference between the accesses.

Test and train dataset partitioning: Throughout the evaluation process the dataset has to be divided into a train and test dataset. Our experiments use one of two holdout sets with a distinct split: 60% training, 40% testing or 15% training, 85% testing. The first holdout set corresponds to the typically approach in generic holdout sets for machine learning. The second holdout set simulates a special environment where the access to training data is severely diminished. The uncorrelated pairs are randomly generated according to the number of negative samples requested. We used multiple sizes of negative samples mostly 9 but also 99 and 199 for the train dataset. Test datasets also generate uncorrelated pairs in a similar way to the train dataset. However, we always used 199 negative samples. Besides random generation, we also devised four different categories for the test dataset in some experiments as described above where client and onion service specific restrictions apply.

Precision metrics: For evaluating the precision, we use two metrics: *true positive rate* and *false positive rate*. True positive rate (TPR) refers to the number of correctly identified correlated pairs over the total number of correlated pairs in the dataset. False positive rate (FPR) refers to the number of uncorrelated pairs that erroneously have been marked as correlated over the total number of uncorrelated pairs in the dataset. The relation between TPR and FPR provides an effective method for comparing models, and it is inline with the goal of achieving the maximum possible TPR for the minimum possible FPR.

Performance metrics: For evaluating the performance of our system, we focus on the time the model needs to correlate a certain amount of flow pairs (*testing time*) as well as the time required for training (*training time*). This will allow the comparison of different strategies in terms of performance and compromises that can be made to improve performance at the cost of precision. To also take a more hardware independent approach we will provide relative speedup results in comparison to a base line since absolute times of both training and classification will vary greatly depending on the setup, one can however expect similar speedup rates between the experiments presented.

5.2 Ranking Stage Evaluation

This section presents a thorough evaluation of the ranking stage implemented by the Torpedo correlator's neural network presented in Section 3.2 (see Table 3.1). The ranking stage assigns each pair a score corresponding to the probability of correlation. By setting a configurable threshold, only the flow pairs that have a resulting score above that threshold will be selected to the correlation stage. Our evaluation aims to study if the ranking stage yields good results by consistently giving higher scores to correlated pairs as opposed to uncorrelated ones. To access the impact of the target flows characteristics we varied the dataset in two dimensions: *volume of traffic* and *time disparity*.

The following results show that Torpedo's ranking stage can indeed improve the system performance by pre-filtering most uncorrelated flows. It also improves the TPR to FPR ratio even when considering accesses to the same onion service which the correlation stage struggles to produce high accuracy results. Next we present our findings as we vary each of said dimensions individually and then combined.

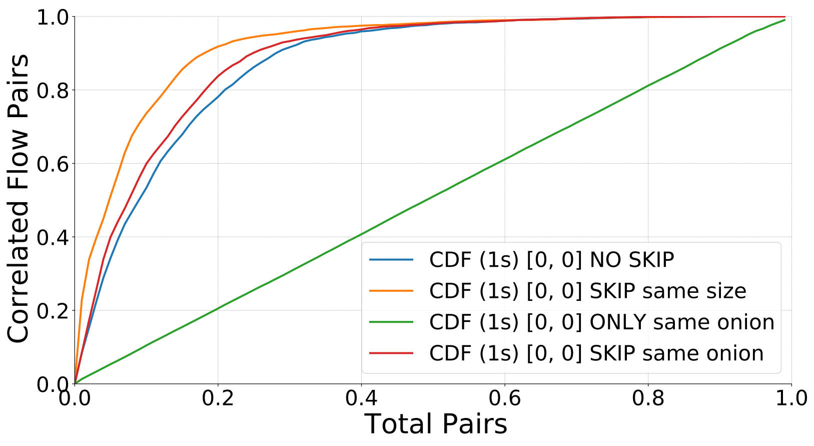


Figure 5.1: CDF concurrent pairs.

Varying the volume of traffic: The results for this experiment are shown in Figure 5.1. In this chart, as in the following ones, we plot the CDF corresponding to the percentage of correlated flow pairs relative to the total number of pairs in various categories. Each category is consistently labeled as:

- "NO SKIP" refers to no restriction on how uncorrelated pairs are generated;
- "SKIP same size" forbids flows to onion services of the same size to form an uncorrelated pair;
- "SKIP same onion" forbids flows accessing the same onion service to form an uncorrelated pair;
- "ONLY same onion" forces all uncorrelated pairs to accesses to the same onion service only;

The chart also provides a "[X,Y]" associated with each line. This range refers to the forced interval of time difference between flows in a pair. For instance, X=0 and Y=1 means that flows can deviate a maximum of 1 second (0 to 1 randomly distributed) from the original time difference of the correlated pair. In this first experiment all flow pairs have minimal time deviations meaning we are essentially simulating 99 flows that run concurrently to the original correlated pair. In total, there are 460 000 pairs with similar time difference for each instance of the dataset classes. By fixing all these flow pairs around the same point in time, the different categories will then explore what happens as a function of the volume of traffic returned by various onion services.

Unsurprisingly, Figure 5.1 shows an almost straight line for the “ONLY same onion” class. This result occurs because it is very difficult for the network to distinguish between accesses to the same onion service page (same overall size therefore very similar amounts of packets exchanged) at the same time. When no restrictions apply (i.e., class “NO SKIP”), we can expect to capture close to 80% of the correlated flow pairs, while only seeing 20% of the total number of pairs. When we forbid the same page or pages with similar size, the number of flow pairs needed to observe 80% of correlated pairs decreases as expected. This shows that even for flow pairs that are seemingly correlated, when only considering the time difference of first packet, the neural network is able to achieve fairly reasonable results. We can then see that the neural network of the ranking stage can greatly reduce the search space of flow pairs that the CNN of the correlation stage needs to process.

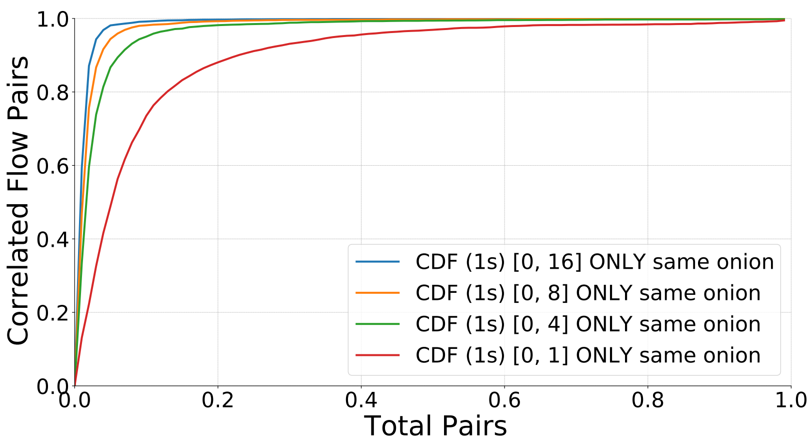


Figure 5.2: CDF up to 16 seconds time difference pairs, uncorrelated only match the same onion.

Varying the flow pairs’ time disparity: By forcing all uncorrelated pairs to be a match between flows accessing the same onion service page we can effectively simulate 99/Y possible pairs per second targeting a given onion service, where Y is the maximum time deviation in an interval [0,Y]. For example, considering Y=1 second gives 99/1 = 99 pairs per second. Figure 5.2 plots the results of this experiment which are still very promising for real world data. For a time space of 16 seconds (blue line), we can capture 85% of correlated pairs by just observing 2% of the total pairs. In absolute terms, from the 9200 pairs selected at this stage, 3910 of those are indeed correlated. If we consider a maximum time difference of 1 second (red line), we are looking at over 20% for the same 2% of flow pairs, which means 920 of the total 9200 would be indeed correlated pairs.

Varying both dimensions: Figure 5.3 plots the results for our last experiment. If we consider 460K possible flow pairs occurring in a time space of 16 seconds (red line) we can capture 95% of correlated pairs by just observing 2% of the total pairs. In absolute terms it would mean that from the 9200 pairs

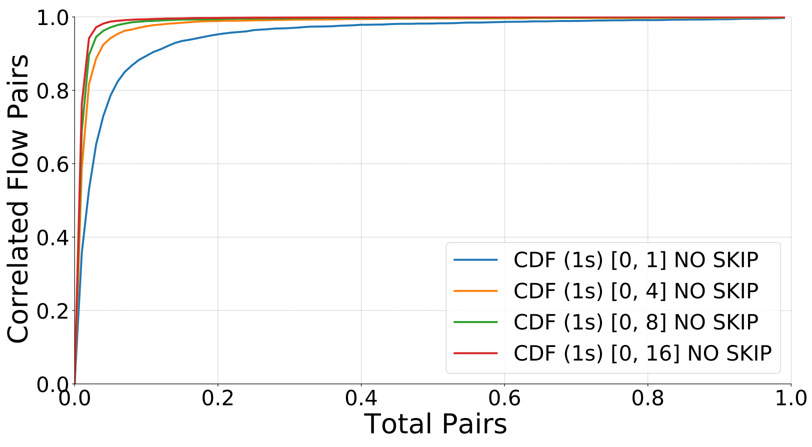


Figure 5.3: CDF up to 16 seconds time difference pairs.

selected at this stage, 4370 of those are indeed correlated. If we consider a maximum time difference of 1 second (blue line) we are looking at a little over 50% for the same 2% of flows which means 2300 of the total 9200 would be indeed correlated pairs. This however assumes no restrictions on the selection of uncorrelated pairs whereas the previous experiment reduced distinguishability by enforcing the "ONLY same onion" policy. Next, we focus on the evaluation of Torpedo's correlation stage.

5.3 Correlation Stage Evaluation

The correlation stage aims to correlate flow pairs through the inter packet timings and packet sizes of each flow. This operation is implemented in the second stage of the Torpedor correlator's pipeline. In this section, we start by presenting our evaluation of the convolutional network that implements this function in Torpedo. This is the baseline CNN introduced in Section 3.3, and it is fully characterized by the parameters listed in Table 3.2. Then, we provide additional results that refer to the evaluation of alternative CNN architectures, namely the original DeepCorr model (Section 5.3.2) and other models (Section 5.3.3) that we have explored before converging into the architecture of Torpedo's baseline CNN.

5.3.1 Evaluation of the Baseline Correlator CNN

To evaluate the baseline correlator CNN, we performed several experiments using our most realistic dataset, which uses real onion service web pages and respective assets, and where the Tor client requests were issued from the Tor browser itself. We present three different instance of the same model evaluation with different dataset holdouts.

Holdout 15% 85% experiment: Figure 5.4 shows the plot of true negative rates (TNR) for the uncorrelated categories and true positive rate (TPR) for the correlated category. This graph clearly shows where the network struggles to correctly classify pairs. We can see that for flow pairs where each flow was from a different onion service access, the network has a TNR of 99% while also achieving a TPR of 34% (red and orange lines are within 0.1% of each other). However, when uncorrelated pairs represent accesses

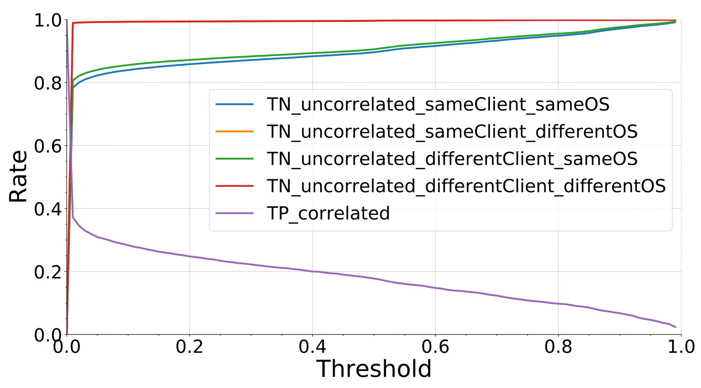


Figure 5.4: First results on realistic dataset.

to the same onion service the TNR drastically decreases, meaning we would be flagging many uncorrelated pairs as being correlated. The side-effect of this is that innocent users would probably be unjustly incriminated. Another conclusion we initially made from this particular experiment is that changing the client (therefore the circuit) did not significantly affect the classifier (lines green and blue) as they only differed by 2%. This conclusion, however, was based on a first iteration of this dataset where clients changed the circuit before every request. This change means that the same client was also using a different circuit just as a different client would, hence the proximity between these results.

Holdout 60% 40% experiment: We performed this same experiment, but now using the 60% training 40% testing dataset instance instead of the previous 15% training 85% testing. The reasoning for this change is that even though 15% training 85% testing may more accurately represent a real world case where a more limited amout of data could be captured for training purposes, the total number of pairs we collected may be too small to use such drastic training dataset reductions leaving the network very little data to learn from. This proved indeed to be somewhat the case as the following results show. Figure 5.5 plots the same rates as before, but now from a network trained using the 60% training 40% testing dataset. We can clearly see an increase in TPR from 34% to 57% for a slightly higher TNR for different onion service accesses of now 99.1%. We can also observe that the TNRs for uncorrelated same onion service accesses increase their distance from one another (green and blue lines). This result may come from the fact that during training there are intrinsically more examples of uncorrelated flow pairs where the access is originated from another client.

Final dataset experiment: Our last experiment represents the final iteration of our evaluation of Torpedo's baseline correlator CNN. To overcome the shortcomings from the previous experiments where clients always changed circuit and we noticed a significant increase in TPR for higher volumes of training data, our final dataset now does not establish a new Tor circuit before each request to the same onion service (reuses the same browser instance as well without caching) and we doubled the accesses that clients made to each onion service from 25 to 50. We also kept the use of a 60% to 40% ratio for our train and test datasets respectively. Figure 5.6 plots the new results for this experiment.

As shown in Figure 5.6, there are significant differences when compared to the plots from the two previous attempts. Starting with a significant increase in TPR, where we can now achieve 73% while

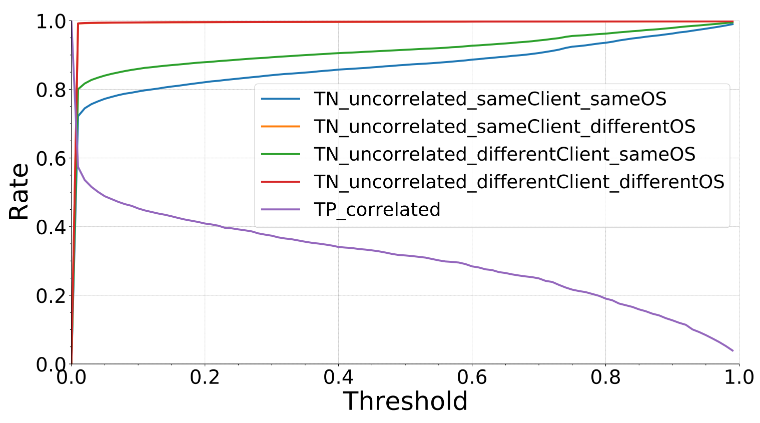


Figure 5.5: Results on realistic dataset for holdout 60% to 40%.

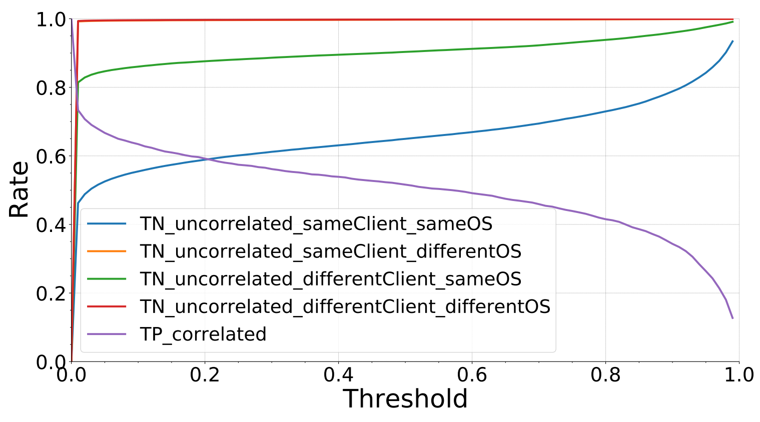


Figure 5.6: Results with the final dataset.

maintaining a TNR for uncorrelated flow pairs from different onion services above 99.2%, this increase can be attributed to the higher data available for training as well as true examples of same client uncorrelated traffic. We can now clearly see the effects of changing clients (changing circuit) and using the same client (same circuit), namely that there is a considerable gap between these two lines, respectively green and blue. As expected, the TNR for the uncorrelated pairs where we match two flows that access the same onion service from the same client is significantly lower as there is little variation between the flows in these pairs. As the circuit and volume of traffic is the same, any existing traffic differences that allow the CNN to correctly classify these pairs are caused by the variations in the delays introduced by the Tor relays allocated to the circuits. These delays will be typically influenced by the Tor network workload experienced by each relay.

When we analyze the counterpart, where uncorrelated pairs match the traffic for the same onion service but from different clients, we gain 35% more TNR. This shows that the classifier has indeed learned to recognize patterns in the traffic. Had it not, we would expect a similar result to the blue line. Interactions between client and server are intrinsically connected to the circuit that connects them since different circuits generate different delays in packets. This provides enough information for the network to make a more informed classification of the given pair.

Dataset	Correlated Pairs	Uncorrelated Pairs
Train	1324	263 476
Test	6000	1 194 000

Table 5.1: Original DeepCorr dataset size.

Epoch	TP Rate	FP Rate
Epoch 5	40.03%	6.3×10^{-4}
Epoch 11	49.96%	7.7×10^{-4}
Epoch 16	45.85%	3.7×10^{-4}
Epoch 22	53.06%	6.3×10^{-4}
Epoch 28	48.95%	4.8×10^{-4}
Epoch 33	57.11%	9.8×10^{-4}
Epoch 39	55.45%	8.6×10^{-4}
Epoch 44	50.26%	6.2×10^{-4}

Table 5.2: Original DeepCorr results by epoch.

As for final observation, in contrast to previous experiments, we can now achieve a TNR of 99.9% with a TPR of 24% for uncorrelated pairs with different onion service accesses, as well as 12.7% of TPR for a 99% TNR when considering uncorrelated pairs that match flows accessing the same onion service.

5.3.2 Evaluation of the DeepCorr CNN

DeepCorr was designed to perform traffic correlation on Tor traffic to regular websites (non onion services) and achieved high results in doing so. Although we focus on a different goal, i.e., on Tor onion traffic correlation, given that our solution is based on DeepCorr, we have tried to reproduce it and use its original CNN as starting point for our system. However, we faced several challenges in reproducing DeepCoor's original results. We briefly make an account of our experience with testing DeepCoor's CNN as it may be useful for other researchers interested in adopting it in the future.

The DeepCorr's authors claim that their system can achieve a TPR of 80% while maintaining a FPR of 10^{-3} for flow pairs with 300 packets of length. The CNN training time reported for achieving reasonable results is about a single day when training on an NVIDIA TITAN X GPU (12GB RAM max at time of paper release). However, in spite of their claims, we could not reproduce their findings. Our testbed consisted of a single machine with 110GB of RAM, 6 vCPU and an NVIDIA P40 24GB. We used the code released by the DeepCorr's authors and modified it to reflect the parameters described in the original DeepCorr paper. We also used their dataset, being careful to generate the training sets in a similar fashion as implemented in their code. As a result, the training dataset consisted of only 1324 pairs which has greatly reduced the training time. We kept the same number of uncorrelated pairs per correlated pair of 199, refer to Table 5.1 for detailed dataset information.

However, even though DeepCorr's authors as stated that their model could fit in a single 12GB RAM GPU, we were not able to run their provided code with the stated CNN parameters on a GPU with less than 24GB of ram. This was not possible even after reducing our batch size to 1. We also tried reducing

Negative Samples	Stride [2,1] [4,1]	Stride [2,1] [4,2]	Stride [2,2] [4,1]
9	22.71%	28.39%	23.24%
99	17.88%	-	-

Table 5.3: TPR for single network with 1/2 the original DeepCorr size for our generated dataset with a maximum FPR of 10^{-2}.

Combination	TP Rate	FP Rate
[2,1] [4,2] and [2,2] [4,1]	14.71%	5.5×10^{-3}
[2,1] [4,2] and [2,1] [4,1]	19.92%	8.3×10^{-3}
[2,2] [4,1] and [2,1] [4,1]	21.56%	9.4×10^{-3}

Table 5.4: Results for ensemble model 1/2 the original DeepCorr size with our generated dataset.

floating point precision from float32 to float16 to reduce the memory used by the model. However, this change also drastically reduced the network performance, which was certainly not desired. There is no mention in the original paper about the number of epochs used during training so we preserved the approach implemented in the provided code of 200 epochs. However, a single epoch in our setting was taking 1h35m. After running the model for over 69 hours and reaching epoch 44, we could not afford the expenses of continuing the training process for the full 200 epochs.

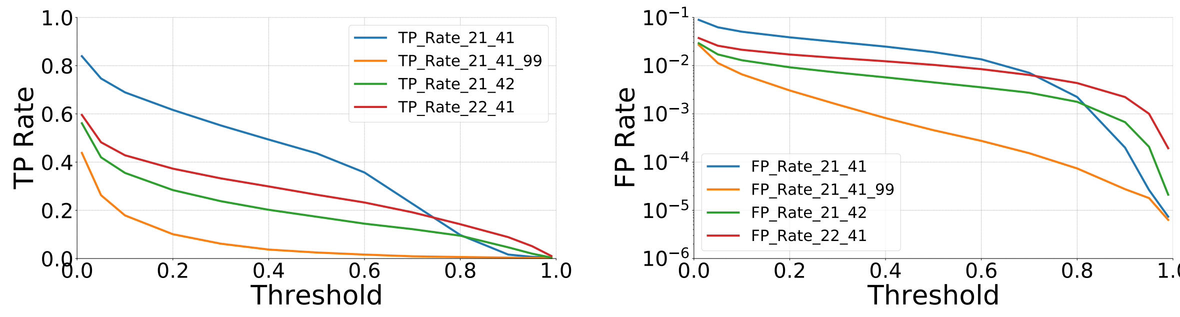
Table 5.2 shows the TPR as well as the FPR of the network at each evaluation epoch. In DeeCoor's original paper, for a 300 packet long flow the authors were able to reach an 80% true positive rate for a false positive rate of 10^{-3}. Our best result only reaches 57% true positive rate for a false positive rate of 9.8×10^{-4}, more results in Table 5.2. This represents a 29% drop in performance besides the enormous training time. This excessively long time to train the model and loss of performance explain why we had to devise a customized version of DeepCorr's original CNN for Torpedo.

5.3.3 Evaluation of Other CNN Variants

We conducted a series of tests with half the original kernel size with and without ensemble combination. Although initial testing seemed promising the ensemble model did not significantly improved the correlation results. These results refer to the first iteration of the dataset which contained only 8 different onion services and all are random text files with varying sizes as stated in the dataset section.

Tables 5.3 and 5.4 show the results for both single network and ensemble models with multiple configurations. Here we can see that training with 99 negative samples did not improve the classifier performance thus we abandoned this idea continuing only with 9 negative samples for training. The best result comes from a single network with strides [2,1] and [4,2] which may suggest that this configuration is slightly better than its counterpart [2,2] and [4,1]. This improved performance when increasing the granularity of the sliding kernel in the first convolutional layer may suggest that the network benefits from evaluating directly correlated features. This is what occurs in this first layer where the kernel slides on the packets outgoing from one pair incoming in the other as opposed to the second layer where more abstract features may be generated as we are now processing both time and size features together.

The apparent differences that seem to exist between individual single network approaches are not



(a) TP Rate

(b) FP Rate

Figure 5.7: TPR for single networks with 1/2 the original DeepCorr size with our generated dataset.

Kernel Size	Stride [2,1] [4,1]	Stride [2,1] [4,2]	Stride [2,2] [4,1]
[2,30] [4,10]	22.95%	28.60%	27.99%
[2,20] [4,8]	-	28.64%	29.65%
[2,10] [4,5]	-	24.54%	23.47%

Table 5.5: TPR of single networks varying kernel and strides.

that large when we account for the difference in FPRs. Figure 5.7 provides a visual comparison showing that with higher TPRs there is an associated increase in FPRs. This has led us to ultimately choosing the [2,1] [4,1] variation as it presented the best flexibility by varying the threshold, being able to give high TPR but also a low FPR by adjusting the chosen threshold.

We also performed a series of evaluations but taking into account the location when generating uncorrelated flows. This has effectively increased the distinguishability between the uncorrelated flows since the onion services are location-dependent. In the following tests we also tested the hypothesis of using different kernel sizes for the convolutional layers as well as the ensemble models originated by combining these different kernel networks.

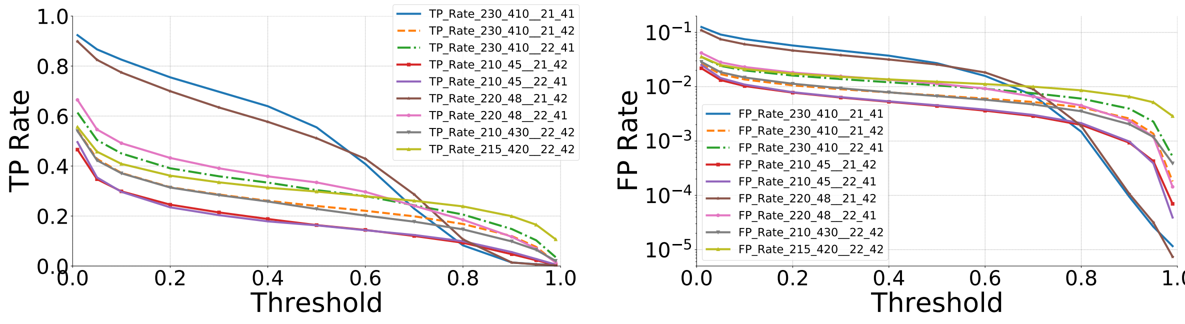
For the single network approach multiple kernels were tested as presented in Tables 5.5 and 5.6. We varied both the strides and kernel sizes. For the kernels we tried multiple variations of the original by reducing the length of packets processed at each time (kernel index 1). No significant improvements were obtained and slight variations in performance seen in the tables can be attributed to variations in the false positive rate which again was kept at a maximum of 10^{-2}. Figure 5.8 shows that all graphs follow a similar trend not improving their TPR without sacrificing FPR.

Ensemble models follow a similar trend to the previous experiments where they actually performed worse in all cases when compared to the single network approach. Even though we expected higher TPR in these tests when compared to the non-location aware previous experiment the improvement was low and can probably be attributed to slight variations in the dataset generation. This may be explained by the fact that we are only considering up to 300 packets even though our static onion services range in size from 100KB to 900KB which the difference may not be captured by our sampling.

It is worth noting that we evaluated our smaller network against the same dataset used for the previous DeepCorr baseline establishment. The results were similar if not better than the original DeepCorr

Kernel Size	Stride [2,2] [4,2]	Stride [2,4] [4,2]
[2,10] [4,30]	28.37%	-
[2,15] [4,20]	-	23.85%

Table 5.6: TPR of single networks varying kernel and strides.



(a) TP Rate (b) FP Rate

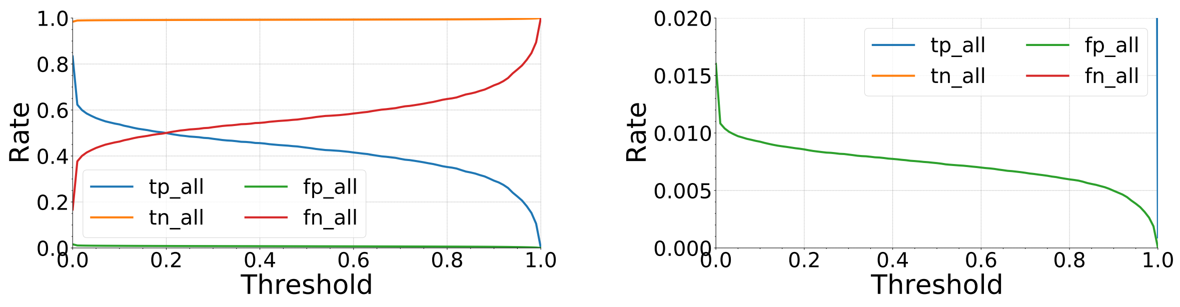
Figure 5.8: Single network results varying kernel and strides.

design. The disparity between regular Tor flows and onion service flows can be explained by the fact that this captures all include the protocol for establishing the onion service connection which cannot provide any useful information to the classifier despite being processed by it. Another big factor comes from these static onion services serving static random text files which do not accurately represent real onion service web pages. This may seem not that all relevant since the traffic is always encrypted and data is divided in equal sized cells meaning only the total size of the page should matter however that is not the case due to the way a website is requested. In any webpage access (assuming no cached assets), the client first requests the base HTML file but then also individually requests all assets referenced in the base HTML to fully render the page. This is a key difference as it implies more traffic generated from the client to server to carry the extra requests along with the responses to these requests by the server making for a much more interactive process than just a single file request. A final key difference is that in our dataset we consider multiple accesses to the same onion service whereas the DeepCorr dataset collection method suggests that a given website was only accessed once.

5.4 Full Pipeline Evaluation

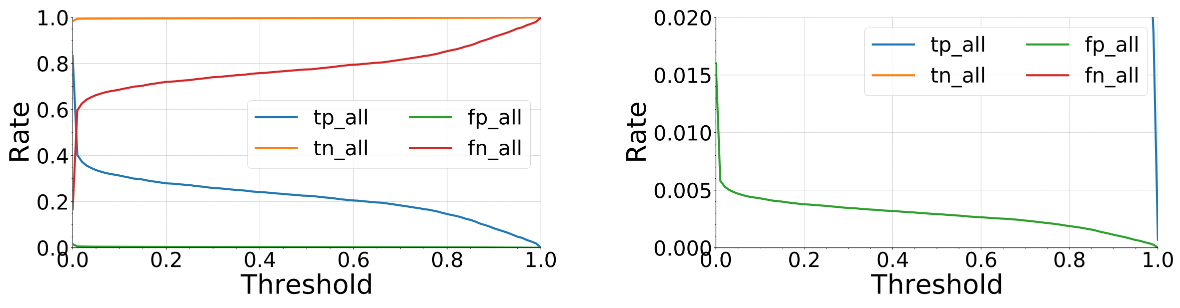
This section focuses on the analysis of the results where both stages work in conjunction to correlate traffic. The results were taken by combining both stages where all flow pairs are first ranked using the ranking stage and then compared to a fixed optimised threshold. The pairs that meet the threshold are then passed onto the second stage where traffic analysis is performed to provide a final correlation probability score. All plots below depict TP/FP/TN/FN rates against the threshold of the second stage.

Full pipeline with separate training: Figure 5.9 (a) plots true positive, true negative, false positive and false negative rates for all instances of uncorrelated pairs. We can see that as expected false positive rates are low for all the threshold range. This is to be expected has there is no absolute time adulteration



(a) Full plot. (b) Zoom on false positive.

Figure 5.9: Results for both stages combined.



(a) Full plot. (b) Zoom on false positive.

Figure 5.10: Results for both stages using combined training.

on this experiments and thus the ranking stage can accurately filter most of uncorrelated pairs. As a consequence we can also see that even for a threshold of 0 we only capture 83% of all correlated pairs as some are lost in the ranking stage as well. This setting provides a TPR of 57% for a FPR of 9.89×10^{-3}. In the other end of the spectrum we can achieve a FPR of just 1.44×10^{-5} for a TPR of 9.21×10^{-4}. A zoomed in version of the previously mentioned plot can be seen in Figure 5.9 (b) where we can see the evolution of false positives while varying the threshold of the second stage.

Full pipeline with combined training: We replicated the previous experiment but on a second stage that had been trained using data already analyzed by the ranking stage. In this setting the same uncorrelated pair generation method is used, however, only pairs that meet the desired threshold of the ranking stage are used during training of the correlation stage. The rationale behind this method is to train the correlation stage on pairs that the ranking stage cannot accurately distinguish. In the previous experiment the correlation stage had been trained on both pairs that could accurately be classified by the ranking stage and pairs that could not, by focusing the training of the second stage on pairs that the ranking stage struggles to correctly classy we can expect to maximize their combination.

Figure 5.10 (a) and (b) plot our results. We can see that indeed the FPR is lower, but so it is the TPR. This feature can be used to tune the network to the user needs as we can now achieve 7.1% of TPR with a FPR of 9.6×10^{-4}. Comparing for a similar FPR of 5.8×10^{-3} we now achieve 40.3% of TPR where before we only reached 34.5%. These results may be improved by segmenting the training dataset. Since we used the same dataset for training both stages, when training the correlation stage the ranking stage predictions were made on its training data which may disguise the network's difficulty in dealing with unseen data.

Performance analysis: To assess the total time of query response of the system we devised the following formula based on the time it takes to evaluate each of the raking and correlation stages in seconds per flow (s/f) and on the threshold of ranking stage propagation:

$$Tt = X \times Rp + X \times Th \times Cp$$

Tt is the total time in seconds for evaluating a given query. X is the number of flow pairs that have been given has input. Th is the threshold value of the ranking stage that controls the number of selected pairs which are passed over from the raking stage to the correlation stage. Rp and Cp correspond respectively to the time it takes to evaluate a given flow pair on the ranking stage and on the correlation stage, respectively. The first operand in the addition ($X \times Rp$) represents the time of execution for the ranking stage and is fixed for a given Rp and X. The second operand in the addition ($X \times Th \times Cp$) represents the time of execution for the correlation stage which for a fixed X and Cp can be customized by the Th parameter. According to our experiments, the ranking stage is able to achieve a performance of Rp=3×10^{-5} s/f whereas the correlation stage can only reach Cp=3.5×10^{-3} s/f. Both of these performances have been measured on a machine with an NVIDIA T4 16GB GPU, 2vCPU Intel, 32GB RAM. Considering both Rp and Cp presented above, for a Th value of 2% and a total of 1M pairs (X) we would have a total computation time of query of approximately 101 seconds, i.e., less than 2 minutes.

Main findings: In summary, we present two distinct constructions of the same system with slightly different characteristics that can be used by the users to better suit their needs. The system is able to achieve reasonable true positive rates while maintaining very low false positive rates. Further optimizations that compromise precision for performance will be analysed in the next section.

5.4.1 Optimizations

To accelerate the correlation process, in this section we explore other possible optimizations/modifications to our system that will allow for processing larger volumes of data without sacrificing the precision as much as one would expect. All experiments bellow use the original DeepCorr dataset.

For some of the measurements reported below, to assess the precision of our optimized model we adopt another metric: the *accuracy* (Acc) which we borrow from DeepCorr[1]. The accuracy represents the number of correctly identified correlated pairs divided by the total number of correlated pairs. In this case, however, the method for identifying a correlated pair is based on DeepCorr's, which is fundamentally different. Os opposed to identifying a correct correlation by checking if the result returned by the CNN for a given pair is greater than some threshold, in this case we pick the pair with maximum probability given by the CNN considering a set that includes the uncorrelated pairs and the correlated pair. So, even if the returned probability was not high enough to supersede the threshold, if the score for

[1] `https://github.com/SPIN-UMass/DeepCorr`

ACC	Input Length	Flow Observed
55.8%	100	100
41.1%	50	50
39.2%	25	25

Table 5.7: Evolution of ACC with flow size.

Aggregation Method	ACC	Input Length	Flow Observed
Sum 2	56.3%	100	200
Sum 3	55.5%	100	300
Sum 2	46.2%	50	100
Sum 3	48.6%	50	150
Sum 4	53.3%	50	200
Sum 2	36.2%	25	50
Sum 4	43.8%	25	100
Sum 8	43.6%	25	200

Table 5.8: Evolution of ACC with multiple aggregation methods.

this pair is the highest, then it will still be counted as a correct prediction. This also applies in the case of having multiple probabilities that would trigger the threshold we will only consider the highest.

O1: Flow size reduction: This optimization corresponds to the “packet capping” optimization introduced in Section 3.3.1 (see also Figure 3.4). Table 5.7 shows the results for flow sizes. Although reducing the flow size drastically improves the performance of the model, it also greatly impacts its accuracy. This result is expected as we are effectively reducing the information the network has to correlate two flows.

O2: Packet aggregation functions: Corresponds to a second optimization presented in Section 3.3.1. Table 5.8 shows the results when different aggregation functions are applied also varying the total input sizes. Even though a flow size of 50 had previously resulted in an accuracy of 41%, by aggregating the packet data we can now reach a maximum of 53.3% accuracy, which is close to the performance of a normal input length of 100 packets. The same can be said to an input size of 25 aggregated packets being close to the performance of a full 50 packet input flow. We can thus conclude that by aggregating packet data we can improve the performance of the network while maintaining similar accuracy values.

O3: Correlation model fine-tuning: We tested several variations of the model by changing the kernel and stride parameterization of the CNN as shown in Table 5.9. This table presents the time of training per epoch and the total time for classifying a given amount of flow pairs for each parameterization. For all the results we used a total of 33390 flow pairs for training and 23547 for testing, the hardware configuration was always the same consisting of a single NVIDIA T4 16GB GPU, 2vCPU Intel, 32GB RAM. We can see that the performance of the network is intrinsically connected to the amount of data being processed (flow length) as well as to the stride for the kernels. However changing the kernels by themselves did not significantly affected both times. The flow length is expected to have this impact as there are intrinsically less computations to be made and linearly affect the input size: as we can see, a reduction of 1/3 in the input size produces approximately a 1/3 reduction in both training and testing. Changing the kernel does not seem to severely hinder the performance. A possible explanation is that

Kernels	Strides	Flow Length	Time Evaluating(s)	Time per Epoch(s)
[2,30][4,10]	[2,1][4,1]	300	83	428
[2,30][4,10]	[2,1][4,1]	200	56	245
[2,30][4,10]	[2,1][4,1]	100	28	101
[2,30][4,10]	[2,1][4,1]	50	16	35
[2,20][4,10]	[2,1][4,1]	300	83	437
[2,10][4,10]	[2,1][4,1]	300	83	447
[2,2][4,10]	[2,1][4,1]	300	92	456
[2,30][4,10]	[2,2][4,1]	300	47	200
[2,30][4,10]	[2,4][4,1]	300	31	100
[2,30][4,10]	[2,8][4,1]	300	18	51

Table 5.9: Evolution of both training and prediction times with multiple model modifications.

the computations being performed when evaluating the kernel are all similar whether we are evaluating a small [2,2] kernel or a larger [2,30] and can be fully parallelized therefore taking approximately the same time to complete. Changing the stride also greatly affects both times as we are effectively computing less convolutions overall. When comparing a stride of [2,1] to a stride of [2,2] we are essentially halving the number of times a kernel is computed. Hence we observe a direct influence in the execution times whereby duplicating the stride in one axis results in approximately half both training and testing times.

5.5 Privacy-Preserving Extensions Evaluation

This section describes our preliminary results to implement the privacy-preserving mode for Torpedo. We used the multiparty computation framework, TF-Encrypted, to conduct our experiments. Next we present our results when testing on a single machine and when testing on a distributed environment.

5.5.1 Performance Analysis in a Standalone Setting

TF-Encrypted can be used for implementing the neural networks of both stages of the Torpedo's correlator pipeline: the ranking stage and the correlation stage. We will concentrate only on evaluating the performance of TF-Encrypted for the CNN that implements the correlation stage as this is the one is the most CPU and network intensive. In these experiments, we use the original DeepCorr dataset. The network is also slightly different as it uses the configuration present in the available code also provided by the authors were strides were [2, 2] and [2, 2] instead of [2, 1] and [4, 1] as described in the paper, the first convolutional layer kernel also differs where it is now [2, 20] instead of the described [2, 30]. Nevertheless we establish a baseline for this configuration and build upon this result.

Infusing TF-Encrypted into DeepCorr: As explained in Section 3.4.3, it was necessary to adapt the CNN to be compatible with TF-Encrypted. Table 5.10 plots the results for the baseline and direct translation of the model to a compatible TF-Encrypted version where kernels were reduced from respectively [2, 20] and [4, 10] to [2, 2] and [4, 4]. Surprisingly we can see that the network actually performed 1.5%

Kernel Size	Accuracy	Training Time	Evaluation Time
[2,20] [4,10]	72.5%	30H	1H20M
[2,2] [4,4]	73.6%	18H	40M

Table 5.10: TF-Encrypted enabled network vs. baseline.

Optimization	Kernel Size	Accuracy	Training Time	Evaluation Time
Input padding 0s	[20,20] [4,4]	77.4%	20H	35M
Input padding 0s	[30,30] [4,4]	76.8%	21H	50M
Input padding repeat	[20,20] [4,4]	75.8%	19H	45M
Input padding repeat	[30,30] [4,4]	74.2%	22H	50M
Input reshaping	[4,4] [4,4]	74.1%	17H	1H40M

Table 5.11: TF-Encrypted enabled optimized networks.

better than it's non privacy-preserving variant. Nevertheless we proceeded to evaluate the various forms of optimizations described earlier.

Table 5.11 plots the results for the various optimizations made to the original model to allow larger amounts of data to be processed at the first convolutional layer (see Section 3.4.3). We can see that even though the baseline non TF-Encrypted enabled model benefits from the original kernel sizes, the results from the optimized TF-Encrypted enabled models are better. We can also observe that changing the first kernel size from 20 to 30 as suggested by DeepCorr's authors did not produce better results as one would expect. However, there is a noticeable difference when processing just 4 packets at a time (reshape model) when compared to the higher volumes of 20 and 30 packets (padding models) suggesting that indeed the network is impaired by the number of packets processed at each convolution. Another aspect that may seem a surprise is the repeat padding vs the 0 padding models. The 0 padding models produced better results which may be due to some confusion introduced by repeating the input. By having 0s instead the network can focus the weight training on smaller amounts of data which may converge better and quicker hence producing betters results for similar epoch iterations of training. We can conclude that the optimizations do enhance the overall precision of the system while maintaining a similar performance in regards to time of training and classification.

5.5.2 Performance Analysis in a Distributed Setting

Given that in a realistic deployment of Torpedo, the privacy-preserving correlation would be performed by several participating parties connected over the Internet, we performed additional experiments that emulate that scenario. Our main goal is to assess the impact of the network latency between the intervening parties to the overall performance of the system. We consider that correlation process as supported by TF-Encrypted will involve 3 compute servers. We performed these experiments resorting to a reduced size convolutional neural network implemented using TF-Encrypted. All tests were performed classifying 300 random flow pairs with a length of 100 packets. The CNN used is similar in structure to the one described in DeepCorr github[2] code excepting that we used 1/8 of the original layers' size.

[2]`https://github.com/SPIN-UMass/DeepCorr`

Batch Size	Classification Time	Memory Usage
1	479s	5GB
6	480s	8GB
12	479s	11GB
30	483s	23GB

Table 5.12: TF-Encrypted same machine multiple processes.

Batch Size	Classification Time	Memory Usage: Master	Server 0	Server 1	Server 2
1	1177s	7GB	1GB	1GB	1GB
6	1229s	7GB	2GB	3GB	3GB
12	1089s	8GB	4GB	6GB	6GB

Table 5.13: TF-Encrypted different machines (2vCPU) within same location.

Testing on a single machine: To establish a baseline comparison point, we first tested the framework running on a single machine with 30GB of RAM and 8 vCPU cores based on Intel Skylake lineup of Xeon CPUs. We experimented with the following batch sizes: 1, 6, 12, 30. We found no apparent difference between the classification times for the 300 pairs while varying the batch size. However, the memory consumed on the host machine would somewhat linearly increase with batch size being respectively 5, 8, 11 and 23 GB. Table 5.12 presents the absolute results for this experiment.

Testing with multiple machines on a local cluster: These next tests were performed with the machine described above as master, input provider, and model provider. The remaining 3 compute servers were powered by similar machines with 10GB of RAM and 2 vCPU all in the same cluster (low network latency, around 2-3ms between hosts). Table 5.13 shows the results of this experiment. We found that the classification times tripled while in this setting compared to running in the same machine. The difference in classification time between different batch sizes seems to vary slightly with the best result (lowest classification time) being with the highest batch size of 12. Again a linear memory consumption was observed but only in the 3 compute servers.

Table 5.14 shows the results for the same experiment but this time compute servers have 4 vCPU cores instead of 2 allowing us to access the scalability of the system and by increasing from 2 to 4 the classification times essentially were twice as fast also suggesting a linear scalability factor. This experiment shows even less variability between classification times in between batch sizes.

Testing with multiple servers connected over the Internet: As a final setting we approached a more realistic scenario (albeit a very pessimistic one) where the three compute servers were located around the globe (USA, London, and Australia). This led to a latency increase of two orders of magnitude now ranging from 180ms to 280ms between hosts. The same 4 core machines were being used in this setting along with the same master. As shown in Table 5.15, the increase in classification time followed a similar trend to the latency, increasing by an order of magnitude. However, it was now heavily affected by the batch size as expected. In this setting, we only tested for a batch size of 1 and 6 but the classification time between these two alternatives almost halved, representing a 41% decrease in classification time when increasing the batch size from 1 to 6. We can expect a similar improvement when increasing again

Batch Size	Classification Time	Memory Usage: Master	Server 0	Server 1	Server 2
1	629s	7GB	1GB	1GB	1GB
6	601s	7GB	2GB	3GB	3GB
12	590s	8GB	4GB	6GB	6GB
30	606s	-	-	-	-

Table 5.14: TF-Encrypted different machines (4vCPU) within same location.

Batch Size	Classification Time	Memory Usage: Master	Server 0	Server 1	Server 2
1	13200s	7GB	1GB	1GB	1GB
6	7800s	7GB	2GB	3GB	3GB

Table 5.15: TF-Encrypted different machines different locations.

from 6 to 12 as times where still very above the base line for a the same location setting. Continuing the improvement of Torpedo's privacy preserving mode is left for future work.

5.6 Summary

This chapter presented the experimental evaluation of the correlator component of Torpedo covering both individual stages and the full pipeline. It also presents preliminary evaluations on the privacy preserving extensions to our system. Although there is a significant drop in the precision of the models when compared to regular web site traffic the system can still provide relevant results to the use case as it is intended. Having a low false positive rate is essential to prevent false incrimination, but if a user is consistently using the network for illegal activities a criminal investigation making use of multiple correlations will increase the confidence in the obtained results. The next chapter concludes this document by summarizing the main findings of this thesis and introducing some directions for future work.

Chapter 6

Conclusions

Anonymity networks such as Tor allow users to remain anonymous when navigating the Internet. In particular, Tor deploys a protocol that allows both the sender and receiver to remain anonymous while communicating. These services represents the majority of the "dark web" where illegal activities such as illicit marketplaces, child pornography, finance fraud among others take place. This raises the need for authorities to de-anonymize these Tor illegal flows.

In this thesis, we have described the design and implementation of Torpedo, a distributed, cooperative, de-anonymization system where ASes and law enforcement agencies cooperate to perform traffic correlation on Tor onion service flows. By adapting existing traffic correlation attacks based on CNNs to regular Tor traffic, Torpedo allows efficient correlation attacks to be performed on onion service traffic.

6.1 Achievements

Our achievements are as follows:

- We present Torpedo, the first system that performs traffic correlation attacks on Tor onion service traffic. Torpedo leverages recent results on traffic correlation attacks based to convolutional neural networks that target regular Tor flows and re-purposes these techniques to onion service traffic where both the sender and receiver remain anonymous.

- We further present an innovative system architecture based on a two-stage approach that allows Torpedo to process high volumes of data while maintaining reasonable precision on flow correlation. This architecture allows the system to be tweaked by the user to better suit theirs needs allowing a compromise between performance and precision.

- We performed a thorough evaluation of the Torpedo correlation component and show that the system is able to process large volumes of data in small periods of time, while maintaining a reasonable true positive rate and a low false positive rate.

- We devised an extension providing privacy guarantees to the system and we implemented and evaluated an early prototype. This extension allows Torpedo to perform traffic correlation on encrypted flow pairs while not requiring ASes to completely trust the correlator.

6.2 Future Work

We identify multiple aspects of the presented system that demand further improvements.

First, there is a need to fully implement the system with regard to the security protocols that assure a secure channel between all intervening parties.

Second, our current evaluation assumes there is a clear way of intercepting individual flows communicating with a given onion service. If multiple users simultaneously access a given onion service it is not clear how client-specific flow capture and discrimination can be performed. Thus, a deeper study is need to access the system in such conditions.

Third, the prototype implemented in this thesis focused solely on the correlator component of Torpedo and does not implement the client and probe modules. One direction for future work comprises the implementation and evaluation of these modules. In particular, there is a pressing need to assess the probe module ability to flag and log Tor guard relay traffic in real time. This involves the design and implementation of all the security protocols related to the communication between parties involved in flow de-anonymization, as well as the support for the mentioned query types by the Torpedo components.

Fourth, all modules should be implemented and fully integrated with TF-Encrypted to support the privacy-preserving correlation of flows. The thorough evaluation of this setup comprises another direction for future work.

Finally, there is the need to assess the coverage of onion service Tor traffic that the cooperating ASes leveraging Torpedo can achieve. This involves the study of which ASes concentrate a larger number of guard nodes as well as which ASes concentrate large fractions of hosted onion services.

Bibliography

[1] N. V. Verde, G. Ateniese, E. Gabrielli, L. V. Mancini, and A. Spognardi. No nat'd user left behind: Fingerprinting users behind nat from netflow records alone. In *IEEE International Conference on Distributed Computing Systems*, 2014.

[2] Scott Coull, Charles Wright, Fabian Monrose, Michael Collins, and Michael Reiter. Playing devil's advocate: Inferring sensitive information from anonymized network traces. In *Network and Distributed Systems Security Symposium*, 2007.

[3] Ting-Fang Yen, Xin Huang, Fabian Monrose, and Michael K Reiter. Browser fingerprinting from coarse traffic summaries: Techniques and implications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2009.

[4] Jing Yuan, Zhu Li, and Ruixi Yuan. Information entropy based clustering method for unsupervised internet traffic classification. In *IEEE International Conference on Communications*, 2008.

[5] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[6] Nicolas Christin. Traveling the silk road: a measurement analysis of a large anonymous online marketplace. In *International Conference on World Wide Web*, 2013.

[7] Kyle Soska and Nicolas Christin. Measuring the longitudinal evolution of the online anonymous marketplace ecosystem. In *USENIX Security Symposium*, 2015.

[8] Ryan Hurley, Swagatika Prusty, Hamed Soroush, Robert J Walls, Jeannie Albrecht, Emmanuel Cecchet, Brian Neil Levine, Marc Liberatore, Brian Lynn, and Janis Wolak. Measurement and analysis of child pornography trafficking on p2p networks. In *International Conference on World Wide Web*, 2013.

[9] Gabriel Weimann. Going dark: Terrorism on the dark web. *Studies in Conflict & Terrorism*, 2016.

[10] Matteo Casenove and Armando Miraglia. Botnet over tor: The illusion of hiding. In *International Conference On Cyber Conflict*, 2014.

[11] Brian N Levine, Michael K Reiter, Chenxi Wang, and Matthew Wright. Timing attacks in low-latency mix systems. In *International Conference on Financial Cryptography*, 2004.

[12] Steven J Murdoch and George Danezis. Low-cost traffic analysis of tor. In *IEEE Symposium on Security and Privacy*, 2005.

[13] Vitaly Shmatikov and Ming-Hsiu Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *European Symposium on Research in Computer Security*, 2006.

[14] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. RAPTOR: Routing attacks on privacy in tor. In *USENIX Security Symposium*, 2015.

[15] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. Deepcorr: Strong flow correlation attacks on tor using deep learning. In *ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[16] L. Overlier and P. Syverson. Locating hidden servers. In *2006 IEEE Symposium on Security and Privacy*, 2006.

[17] Paul Syverson, Gene Tsudik, Michael Reed, and Carl Landwehr. Towards an analysis of onion routing security. In *Designing Privacy Enhancing Technologies*, 2001.

[18] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. The onions have eyes: A comprehensive structure and privacy analysis of tor hidden services. In *International Conference on World Wide Web*, 2017.

[19] George Danezis. Statistical disclosure attacks. In *IFIP International Information Security Conference*, 2003.

[20] Stevens Le Blond, Pere Manils, Abdelberi Chaabane, Mohamed Ali Kaafar, Claude Castelluccia, Arnaud Legout, and Walid Dabbous. One bad apple spoils the bunch: Exploiting p2p applications to trace and profile tor users. In *USENIX Conference on Large-Scale Exploits and Emergent Threats*, 2011.

[21] Milad Nasr, Amir Houmansadr, and Arya Mazumdar. Compressive traffic analysis: A new paradigm for scalable traffic analysis. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[22] Nathan S Evans, Roger Dingledine, and Christian Grothoff. A practical congestion attack on tor using long paths. In *USENIX Security Symposium*, 2009.

[23] Sambuddho Chakravarty, Angelos Stavrou, and Angelos D Keromytis. Traffic analysis against low-latency anonymity networks using available bandwidth estimation. In *European symposium on research in computer security*, 2010.

[24] Tao Wang and Ian Goldberg. On realistically attacking tor with website fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2016.

[25] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *USENIX Security Symposium*, 2016.

[26] Rebekah Overdorf, Mark Juarez, Gunes Acar, Rachel Greenstadt, and Claudia Diaz. How unique is your. onion?: An analysis of the fingerprintability of tor onion services. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[27] Andriy Panchenko, Asya Mitseva, Martin Henze, Fabian Lanze, Klaus Wehrle, and Thomas Engel. Analysis of fingerprinting techniques for tor hidden services. In *Workshop on Privacy in the Electronic Society*, 2017.

[28] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit fingerprinting attacks: Passive deanonymization of tor hidden services. In *USENIX Security Symposium*, 2015.

[29] Rob Jansen, Florian Tschorsch, Aaron Johnson, and Björn Scheuermann. The sniper attack: Anonymously deanonymizing and disabling the tor network. In *Network and Distributed Security Symposium*, 2014.

[30] Marco Valerio Barbera, Vasileios P Kemerlis, Vasilis Pappas, and Angelos D Keromytis. Cellflood: Attacking tor onion routers on the cheap. In *European Symposium on Research in Computer Security*, 2013.

[31] Philipp Winter, Roya Ensafi, Karsten Loesing, and Nick Feamster. Identifying and characterizing sybils in the tor network. In *USENIX Security Symposium*, 2016.

[32] Quangang Li, Peipeng Liu, and Zhiguang Qin. A stealthy attack against tor guard selection. *International Journal of Security and Its Applications*, 2015.

[33] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. Trawling for tor hidden services: Detection, measurement, deanonymization. In *IEEE Symposium on Security and Privacy*, 2013.

[34] Sebastian Zander and Steven J Murdoch. An improved clock-skew measurement technique for revealing hidden services. In *USENIX Security Symposium*, 2008.

[35] Baris Coskun and Nasir Memon. Online sketching of network flows for real-time stepping-stone detection. In *Annual Computer Security Applications Conference*, 2009.

[36] Amir Houmansadr, Negar Kiyavash, and Nikita Borisov. Rainbow: A robust and invisible non-blind watermark for network flows. In *Network and Distributed Systems Security Symposium*, 2009.

[37] M.R.M. Manickam, M. Mohanapriya, Sandip Kale, M. Uday, P. Kulkarni, Y. Khandagale, and S.P. Patil. Research study on applications of artificial neural networks and e-learning personalization. *International Journal of Civil Engineering and Technology*, 2017.

[38] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, 2016.

[39] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security Symposium*, 2018.

[40] Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin, and Gavin Uhma. Private machine learning in tensorflow using secure computation. 2018.

[41] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[42] Luís Rodrigues Salvatore Signorello Fernando Ramos André Madeira Diogo Barradas, Nuno Santos. Flowlens: Enabling efficient flow classification for ml-based network security applications. In *(NDSS 2021) [To appear]*.

[43] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[44] Massimo Bernaschi, Alessandro Celestini, Stefano Guarino, Flavio Lombardi, and Enrico Mastrostefano. Spiders like onions: On the network of tor hidden services. In *The World Wide Web Conference*, 2019.

[45] Gareth Owenson, Sarah Cortes, and Andrew Lewman. The darknet's smaller than we thought: The life cycle of tor hidden services. *Digital Investigation*, 2018.

[46] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. The onions have eyes: A comprehensive structure and privacy analysis of tor hidden services. In *Proceedings of the 26th International Conference on World Wide Web*, 2017.

[47] Pavlo Burda, Coen Boot, and Luca Allodi. Characterizing the redundancy of darkweb .onion services. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, 2019.

[48] Changhoon Yoon, Kwanwoo Kim, Yongdae Kim, Seungwon Shin, and Sooel Son. Doppelgängers on the dark web: A large-scale assessment on phishing hidden web services. In *The World Wide Web Conference*, 2019.

[49] Sagar Shivaji Salunke. *Selenium Webdriver in Python: Learn with Examples*. 2014.