UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Privacy-preserving Deanonymization of Dark Web Tor Onion Services for Criminal Investigations

Daniel José Ferreira Ângelo

Mestrado em Engenharia Informática

Dissertação orientada por:
Professor Doutor Bernardo Luís da Silva Ferreira

2022

Agradecimentos

Esta dissertação não é, na realidade, o trabalho de um único indivíduo. Nunca poderia ter terminado este projeto se não fossem as pessoas que estiveram aqui para me apoiar ao longo destes últimos anos. Dedico-lhes os seguintes parágrafos.

Em primeiro lugar, gostaria de agradecer ao meu orientador, o Professor Bernardo Ferreira, que me ajudou e guiou ao longo de todo o projeto com a sua inesgotável paciência e sabedoria, incentivando-me a persistir e concedendo-me confiança total para concluir esta dissertação.

Agradeço também ao Bernardo Portela, ao Diogo Barradas, ao João Vinagre e ao Nuno Santos pelos conselhos valiosos que me indicaram principalmente no início deste projeto, bem como aos meus outros colegas do Torpedo: o Afonso Carvalho, o André Tse, a Daniela Lopes, a Inês Macedo, o Pedro Medeiros e o Pedro Vieira, que puderam esclarecer algumas das minhas dúvidas nas nossas produtivas reuniões virtuais.

Agradeço a todos os meus antigos professores. O conhecimento que me transmitiram aliado ao seu dom de ensinar é uma das principais razões pelas quais escrevo esta tese e algo que seguramente não posso subestimar.

Também não posso esquecer todos os amigos e colegas feitos ao longo deste percurso. Guardarei para sempre as memórias únicas dos momentos que vivi com eles.

Agradeço em particular ao meu amigo Diogo, que é uma das pessoas mais engraçadas e genuínas que conheço. O facto de partilharmos vários interesses em comum fez com que nos conectássemos com tremenda facilidade desde uma idade invulgarmente jovem e atualmente ainda valorizamos imenso essa conexão.

Estas palavras são agora dedicadas à Sofia, a minha namorada e melhor amiga, a pessoa que me conhece melhor do que ninguém, que me fez sorrir sempre que sentia que não conseguiria concluir este trabalho. A Sofia não só acreditou que eu iria superar todas as adversidades, como também me apoiou com a sua energia radiante ao longo deste árduo, mas gratificante desafio.

Por fim, nunca poderei expressar plenamente a gratidão que tenho à minha família, principalmente pelas inúmeras ocasiões em que estiveram aqui para me apoiar e pela educação que me proporcionaram. Só eles sabem desde o início o quanto isto realmente significa para mim. Este esforço seria sempre em vão se não estivessem aqui.

Este trabalho foi financiado pela Fundação para a Ciência e Tecnologia (FCT) com bolsas (LASIGE) UIDB/00408/2020, UIDP/00408/2020 e (DAnon) CMU/TIC/0044/2021.

Resumo

Atualmente é possível monitorizar algumas comunicações pela Internet que extraem informações confidenciais dos utilizadores. O tema emergente da privacidade dos dados preocupa diversas pessoas, que começam a valorizar cada vez mais as garantias por detrás das redes de anonimato, uma vez que estas conferem o direito à privacidade e conseguem muitas vezes contornar a vigilância e censura *online*.
O Tor é precisamente uma das redes de anonimato mais populares do mundo. Os utilizadores desta plataforma variam de dissidentes a cibercriminosos ou até mesmo cidadãos comuns simplesmente preocupados com sua privacidade. Esta rede baseia-se em mecanismos de segurança avançados que fornecem fortes garantias contra ataques de correlação de tráfego que podem de-anonimizar os seus utilizadores e serviços.
Uma rede Tor utiliza, geralmente, 3 nós que fazem *relay* entre um cliente e um servidor, isto é, encaminham pacotes cifrados através desse circuito Tor recém-criado, sendo que cada nó é responsável por decifrar os pacotes que chegam à sua própria camada e encaminhar o *payload* para a próxima até chegar ao servidor de destino. Assumindo este modelo, o servidor não preserva o seu anonimato, uma vez que seu endereço IP é divulgado em claro pelo último nó do circuito, ou seja, o que se encontra imediatamente antes de si.
Esta limitação pode ser resolvida através de serviços *onion* (OS), que essencialmente asseguram o anonimato do cliente e do servidor, com a composição de dois circuitos Tor independentes através de um nó intermédio chamado RP, que faz *relay* para ambas as entidades (cliente e servidor). Portanto, para de-anonimizar uma sessão de OS, um adversário deve poder controlar os nós de entrada e de saída de um determinado circuito ou então os circuitos que interligam o cliente e o serviço *onion*. Desse modo, um ataque de correlação de tráfego e consequente de-anonimização de uma sessão de OS apenas pode ocorrer se pelo menos uma destas condições se verificar.
O Torpedo é, pelo que sabemos, o primeiro ataque de correlação de tráfego no Tor que visa de-anonimizar as sessões de OS. De forma federada, os servidores pertencentes a fornecedores de serviço de Internet (ISPs) espalhados pelo mundo podem processar *queries* de de-anonimização de IPs específicos. Com a abstração de uma interface, essas *queries* são submetidas por um operador com a finalidade de de-anonimizar serviços *onion* e clientes.
O fluxo de execução de *queries* do Torpedo é constituído por uma *pipeline* de várias etapas, que pode ser dividida em duas fases principais: a fase de *filtering*, onde os ISPs pré-processam os fluxos observados em diversos *vantage points* da rede e a fase de *matching*, onde se tentam encontrar sessões que envolvam o mesmo cliente e OS após a filtragem inicial. Os resultados preliminares revelaram que o ataque era bem sucedido ao identificar os endereços IP das sessões de OS com alta confiança (sem falsos positivos).
Ainda assim, o Torpedo exigia que os ISPs partilhassem tráfego sensível dos seus clientes entre si. Considerando o cenário provável de que os nós de entrada e de saída de um determinado circuito não pertencem

ao mesmo ISP, torna-se fulcral para o sucesso do ataque que os ISPs atuem cooperativamente caso o adversário pretenda correlacionar o tráfego. No entanto, uma das principais limitações atuais do Torpedo é o facto de os ISPs serem obrigados a partilhar dados confidenciais dos seus clientes caso decidam unir-se para de-anonimizar uma sessão de um serviço *onion*. Isto acaba por ser uma desvantagem considerável que impede que o ataque atinja todo o seu potencial, visto que os ISPs não estão interessados em partilhar dados sensíveis entre si. Para além disso, também é relevante considerarmos as limitações da jurisdição de dados (RGPD) e restrições legais de tal envolvimento.

A resposta para este problema reside em computação multipartidária (MPC), que consiste essencialmente num algoritmo criptográfico que dispensa o requisito de confiança entre entidades que participam na mesma computação, mas que não se confiam mutuamente. De modo a atingir este objetivo, algumas propriedades devem ser asseguradas, tais como privacidade e correção. Ou seja, apenas será divulgado um *output* comum a todos os participantes da computação e não os valores privados dos demais (privacidade), sendo que o *output* tem de estar efetivamente correto, de acordo com o que é suposto calcular (correção).

Adicionalmente, também considerámos que poderia ser benéfico a inclusão de privacidade diferencial (DP) no treino do modelo. Um algoritmo é considerado "diferencialmente privado" quando, ao observarmos o *output*, não conseguimos afirmar se algum *sample* individual foi adicionado ou removido do *dataset* de treino original. Portanto, o principal objetivo deste método é garantir que as informações confidenciais no *dataset* não são *leaked*. Podemos alcançar isso através da introdução de um valor de *noise* aleatório predefinido durante o treino, que dificultará qualquer tentativa posterior de um adversário expor os dados privados.

Embora o ataque não exija necessariamente a aplicação desta técnica de privacidade diferencial para ser bem sucedido, a sua incorporação pode ser interpretada como um tópico de potencial futura investigação na área de preservação de privacidade em aprendizagem automática. Para além disso, este método também pode ser aplicável num contexto em que o Torpedo eventualmente atue como uma *framework* de de-anonimização que auxilie investigações criminais em vez de atacar indiscriminadamente a rede Tor.

Desse modo, neste trabalho, tentamos, então, complementar a investigação desenvolvida anteriormente com a introdução e estudo de técnicas de preservação de privacidade em aprendizagem automática. A solução visa desenvolver e avaliar um novo vetor de ataque ao Tor que consiga preservar a privacidade dos dados de cada participante envolvido nas computações, permitindo, então, que os ISPs cifrem o seu tráfego de rede antes de fazerem a correlação. Estes servidores não possuem a totalidade do modelo, mas apenas "shares" (resultantes da computação multipartidária), sendo ainda que o próprio operador também realiza inferências privadas sem ter conhecimento do mesmo.

Em mais detalhe, testamos, avaliamos e recorremos a uma *framework* de MPC orientada a aprendizagem automática que é executada por cima do Torpedo (TF Encrypted) e também desenvolvemos uma extensão preliminar para treinar o modelo com recurso a DP ao usarmos o TF Privacy. Descrevemos o funcionamento dos programas implementados e analisamos, em particular: a fase de pré-processamento dos dados, o treino do modelo com ou sem DP, a subsequente distribuição de "shares" aos ISPs e as inferências privadas que o operador submete.

Após definirmos a metodologia dos testes e delinearmos não só as especificações de *hardware*, como também as localizações geográficas de cada máquina, damos início à avaliação da nossa solução. Con-

cluímos que o desempenho e a precisão do sistema foram pouco afetados ao executarmos computação multipartidária entre os ISPs, uma vez que foi possível processar uma quantidade razoável de fluxos e obter, apesar de tudo, tempos de treino e de inferência aceitáveis, bem como uma elevada taxa de verdadeiros positivos e reduzida taxa de falsos positivos. No entanto, o mesmo não se verificou, pelo menos no que diz respeito a precisão, quando introduzimos adicionalmente uma quantidade predefinida de *noise* nos gradientes do modelo, ao realizarmos o treino com privacidade diferencial. Ainda assim, a adoção de técnicas de preservação de privacidade em aprendizagem automática no contexto de um ataque de correlação de tráfego a um sistema de grande escala como é o Tor constitui um dos maiores feitos desta tese. Reservamos, contudo, para possível trabalho futuro a expansão e aprimoramento do *setting* de privacidade diferencial.

Palavras-chave: Tor, correlação de tráfego, preservação de privacidade em aprendizagem automática, computação multipartidária, privacidade diferencial

Abstract

Tor is one of the most popular anonymity networks in the world. Users of this platform range from dissidents to cybercriminals or even ordinary citizens concerned with their privacy. It is based on advanced security mechanisms that provide strong guarantees against traffic correlation attacks that can deanonymize its users and services.

Torpedo is the first known traffic correlation attack on Tor that aims at deanonymizing onion services' (OS) sessions. In a federated way, servers belonging to ISPs around the globe can process deanonymization queries of specific IPs. With the abstraction of an interface, these queries can be submitted by an operator to deanonymize OSes and clients. Initial results showed that this attack was able to identify the IP addresses of OS sessions with high confidence (no false positives).

However, Torpedo required ISPs to share sensitive network traffic of their clients between each other. Thus, in this work, we seek to complement the previously developed research with the introduction and study of privacy-preserving machine learning techniques, aiming to develop and assess a new attack vector on Tor that can preserve the privacy of the inputs of each party involved in a computation, allowing ISPs to encrypt their network traffic before correlation.

In more detail, we leverage, test and assess a ML-oriented multi-party computation framework on top of Torpedo (TF Encrypted) and we also develop a preliminary extension for training the model with differential privacy using TF Privacy.

Our evaluation concludes that the performance and precision of the system were not significantly affected by the execution of multi-party computation between ISPs, but the same was not true when we additionally introduced a pre-defined amount of random noise to the gradients by training the model with differential privacy.

Keywords: Tor, traffic correlation, privacy-preserving machine learning, multi-party computation, differential privacy

Contents

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Context and Motivation . . . 1
1.2 Objective . . . 2
1.3 Document Structure . . . 2

2 Background and Related Work 5

2.1 Tor . . . 5
2.1.1 Motivation . . . 5
2.1.2 Network Overview . . . 5
2.1.3 Onion Services . . . 6
2.1.4 Security . . . 6
2.1.5 Attacks . . . 7
2.1.6 Traffic correlation attacks . . . 8
2.2 Neural Networks . . . 9
2.2.1 Introduction . . . 9
2.2.2 Convolutional Neural Networks . . . 11
2.2.3 DeepCorr's CNN . . . 11
2.3 Torpedo [23] . . . 12
2.3.1 Challenges . . . 13
2.3.2 System Overview . . . 13
2.3.3 Use Cases . . . 14
2.3.4 Workflow . . . 14
2.3.4.1 Filtering phase . . . 14
2.3.4.2 Matching phase . . . 15
2.3.5 Countermeasures and Limitations . . . 16
2.4 Multi-party Computation . . . 16
2.4.1 Introduction . . . 16
2.4.2 General-purpose frameworks . . . 18
2.4.2.1 SPDZ [22] / SPDZ-2 [36] . . . 18

2.4.2.2 MASCOT [35] . . . 19
2.4.2.3 SCALE-MAMBA [9] . . . 20
2.4.2.4 MP-SPDZ [34] . . . 20
2.4.2.5 Comparison . . . 20
2.4.3 Machine learning based frameworks . . . 21
2.4.3.1 ABY^3 [44] . . . 21
2.4.3.2 SecureNN [64] . . . 21
2.4.3.3 FLASH [15] . . . 22
2.4.3.4 TF Encrypted [21] . . . 22
2.4.3.5 CrypTen [38] . . . 23
2.4.3.6 Comparison and Discussion . . . 23
2.5 Differential Privacy . . . 24
2.5.1 Definition . . . 24
2.5.2 Machine Learning Practical Application . . . 25

3 Design of the Solution 27
3.1 General Approach . . . 27
3.1.1 Design Goals . . . 27
3.1.2 Plain-text Operations . . . 27
3.1.3 Private Operations . . . 28
3.1.4 Architecture Overview . . . 29
3.2 Security and Trade-offs . . . 29
3.2.1 Multi-Party Computation Impact . . . 30
3.2.2 Differential Privacy Impact . . . 30

4 Implementation 33
4.1 Project Structure . . . 33
4.2 Data Pre-Processing . . . 33
4.3 Training Parameters . . . 34
4.3.1 DP-Exclusive Training Parameters . . . 35
4.4 Neural Network of the Privacy-Preserving Ranking Stage . . . 36
4.5 Neural Network of the Privacy-Preserving Correlation Stage . . . 37
4.6 Training without DP *vs* Training with DP . . . 38
4.7 Secure Model Serving and Private Inferences . . . 39
4.8 Workflow Summary . . . 40
4.9 Frameworks . . . 40

5 Results and Evaluation 43
5.1 Methodology . . . 43
5.2 Benchmarking . . . 44
5.2.1 Hardware Specifications . . . 44
5.2.2 Geographical locations . . . 45

5.3 Evaluation of the Privacy-Preserving Ranking Stage . . . 45
5.3.1 Accuracy . . . 45
5.3.2 Loss . . . 48
5.3.3 Evaluating the performance for the standard settings . . . 48
5.3.4 Evaluating the performance for different hardware specifications . . . 48
5.3.5 Evaluating the performance for different geographical locations . . . 49
5.3.6 Precision . . . 50
5.4 Evaluation of the Privacy-Preserving Correlation Stage . . . 51
5.4.1 Accuracy . . . 51
5.4.2 Loss . . . 51
5.4.3 Evaluating the performance for the standard settings . . . 53
5.4.4 Evaluating the performance for different hardware specifications . . . 54
5.4.5 Evaluating the performance for different geographical locations . . . 55
5.4.6 Precision . . . 56
5.5 Preliminary Performance Evaluation for the DP extension . . . 56
5.6 General security key-points . . . 58

6 Conclusion **59**
6.1 Contributions . . . 59
6.2 Future Work . . . 59

Bibliography **65**

List of Figures

2.1 Architecture of a simple neural network 10
2.2 Architecture of DeepCorr's CNN [47] 12
2.3 Query execution workflow in Torpedo [23] 14

3.1 General architecture of the privacy-preserving Torpedo solution 29

5.1 Model accuracy plots for the privacy-preserving ranking stage 46
5.2 Model loss plots for the privacy-preserving ranking stage 47
5.3 Model accuracy plots for the privacy-preserving correlation stage 52
5.4 Model loss plots for the privacy-preserving correlation stage 53

List of Tables

2.1 Simple comparison between general-purpose MPC frameworks 21
2.2 Simple comparison between machine learning based MPC frameworks 24

4.1 Neural network architecture of the privacy-preserving ranking stage 36
4.2 Neural network architecture of the privacy-preserving correlation stage 38

5.1 Settings for the hardware specifications of the machines 44
5.2 Settings for the geographical locations of the machines 45
5.3 General performance metrics evaluated for different numbers of epochs and batch sizes in the privacy-preserving ranking stage default setting 48
5.4 Evaluation of the training and testing times for different hardware specifications in the privacy-preserving ranking stage 49
5.5 Evaluation of the testing time for different geographical locations in the privacy-preserving ranking stage 50
5.6 Privacy-preserving ranking stage precision metrics evaluated for 20 private inferences varying the correlation threshold 51
5.7 General performance metrics evaluated for different numbers of epochs and batch sizes in the privacy-preserving correlation stage default setting 54
5.8 Evaluation of the training and testing times for different hardware specifications in the privacy-preserving correlation stage 55
5.9 Evaluation of the testing time for different geographical locations in the privacy-preserving correlation stage 55
5.10 Privacy-preserving correlation stage precision metrics evaluated for 10 private inferences varying the correlation threshold 56
5.11 General performance metrics evaluated for different noise multipliers in the privacy-preserving ranking stage for the MPC+DP setting 57
5.12 General performance metrics evaluated for different noise multipliers in the privacy-preserving correlation stage for the MPC+DP setting 57

Chapter 1

Introduction

This chapter will be dedicated to explaining the context, motivation and objectives of this thesis, presenting the inherent problem of the investigation, as well as a brief glimpse of some methods that will be considered to tackle it. Finally, the document structure will also be detailed.

1.1 Context and Motivation

Presently, it is conceivable to monitor communications over the Internet that extract sensitive user information. With the growing concern regarding data privacy, several Internet users appreciate the guarantees behind anonymity networks, which provide right to privacy and circumvent online surveillance and censorship [26]. The Onion Router (Tor) [24] is among the largest of those networks [27]. As it will be thoroughly studied in Section 2.1, it employs an overlay of, generally, 3 relay nodes between a client and a server, that forward encrypted packets through the newly established Tor circuit, being each node responsible for decrypting packets from its own layer and forwarding the payload to the next one until those packets are fully sent to the destination server. Assuming this model, the server does not preserve its anonymity, since its IP address is disclosed in plain-text by the last node of the circuit. This limitation can be tackled with onion services (OS) (Section 2.1.3), which essentially provide both client and server anonymity by composing two independent Tor circuits with an intermediate relay node called rendezvous-point (RP) for both entities (client and server). Therefore, in order to deanonymize an OS session, an adversary has to either own both the entry and exit nodes of a given circuit or the circuits that interconnect the client and the OS. Hence, a traffic correlation attack and consequent OS deanonymization can merely occur if at least one of these conditions is met.

Torpedo [23] is an attack on Tor that follows this approach, allowing the deanonymization of OS sessions. It is deployed in a federated fashion, by a group of ISPs that collect and analyse Tor traffic through ML-based classifiers which subsequently correlate these sessions, as it will be described in more detail in Section 2.3. The previous research reveals vastly positive initial results regarding correlation precision, i.e., whenever Torpedo correlates a flow, it is able to identify the IP addresses of both endpoints of that OS session. The Internet Service Providers (ISPs) are accountable for managing the servers that monitor the network and processing deanonymization queries submitted by an operator. Considering the likely scenario that the entry and exit nodes of a given circuit don't belong to the same ISP, then it becomes vital for the success of the attack that the ISPs act cooperatively if the attacker wants to perform traffic

correlation. However, one of Torpedo's major current limitations is the fact that the Internet Service Providers are required to share confidential clients' data with each other if they decide to work jointly in order to deanonymize an OS session. This ends up being a considerable drawback that prevents the attack from reaching its full potential, since ISPs are not interested in sharing sensitive data between each other and are not willing to lose competitor advantage against their opponents. Besides, it is still relevant to consider the data jurisdiction limitations and legal restrictions of such an involvement. Therefore, this work focuses in building a privacy-preserving version of Torpedo, adopting a technique called Multi-Party Computation (MPC), that, as detailed in Section 2.4, primarily consists in a cryptographic protocol that dismisses the trust requirement between mutually untrusted parties of a computation. In order to accomplish this goal, some properties have to be ensured, such as privacy and correctness, i.e., it will only be disclosed an output value common to all the participants of the computation and not the others' private inputs (privacy) and the output value has to be indeed correct, according to what is supposed to compute (correctness).

1.2 Objective

Currently, it is not yet trivial to guarantee the aforementioned properties without some kind of compromise, that generally resides between security and performance, where the existing most secure MPC frameworks tend to perform slower. The goal of this work is to study the state-of-the-art regarding both general-purpose and machine learning oriented MPC frameworks, characterize these solutions and create a module for Torpedo that leverages and evaluates the chosen framework, aiming for a privacy-preserving solution that meets its requirements. Furthermore, we also introduce a preliminary extension for this system with the addition of differential privacy techniques, that constitute an alternative privacy-preserving method.

Firstly, we started by researching and studying the state-of-the-art and background of diverse topics, such as Tor, neural networks, Torpedo, multi-party computation and also differential privacy. This lead to the starting point of the design of the solution, where we tried to delineate our work plan and how we would implement it, which was precisely the next step and perhaps the lengthiest of them all. Finally, we had to define the methodology of our tests and evaluate the implemented system.

1.3 Document Structure

- **Chapter 2** will contextualize the domain of the research and its purpose is to analyse the state-of-the-art literature and background of the Tor network (Section 2.1), neural networks (Section 2.2), Torpedo [23] (Section 2.3), multi-party computation (Section 2.4) and differential privacy (Section 2.5).

- **Chapter 3**, on the other hand, will focus in describing a theoretical analysis behind several design choices that preceded the implementation of the solution, discussing the general approach that was followed (Section 3.1) and some security and trade-offs that had to be taken into consideration (Section 3.2).

- **Chapter 4** presents a full walk-through of the entire implementation phase, providing details regarding the project structure (Section 4.1), the data pre-processing phase (Section 4.2), the training parameters of the neural networks (Section 4.3), the architecture of these neural networks for both the privacy-preserving ranking and correlation stages (Sections 4.4 and 4.5), a comparison between public training with or without DP (Section 4.6), a summary of the complete workflow (Section 4.8) and, finally, a list of the different development frameworks that were used (Section 4.9).
- **Chapter 5** exposes the results and evaluates the solution. In particular, we start by defining a methodology for the experiments that we decided to test the system (Section 5.1). We also establish some settings for the hardware specifications and geographical locations of the machines (Section 5.2). After that, we present the obtained results and analyse them for both the privacy-preserving ranking and correlation stages (Sections 5.3 and 5.4).
- **Chapter 6** concludes this thesis, highlighting its most notable contributions (Section 6.1) and suggesting possible lines of future work and improvement (Section 6.2).

Chapter 2

Background and Related Work

2.1 Tor

In this section, we'll first discuss how one of the most popular anonymity networks in the world was created, what are its main motivations and uses, a brief overview of the protocol itself, as well as the mechanism that allows not only the sender but also the receiver to remain anonymous in a communication (onion services) and some security assumptions that have to be taken into account in this network. Lastly, we'll also categorize some types of attacks that can occur in Tor and explore their state-of-the-art in the literature.

2.1.1 Motivation

Tor is an anonymity network that was originally deployed as a second-generation onion routing project [24]. In its inception, the primary purpose was to protect government communications. However, at the present, this network is used by military, journalists, activists, law enforcement officers and cybercriminals [55]. The motivation behind most of Tor uses is related to the rising concern about online privacy protection. For example, Tor as a tool may be widely adopted by citizens who are trying to freely and securely browse the web without being censored. However, as much as there are good intentions behind some of its uses, there are also people who can exploit the features of this network in order to deploy or consume illicit activity [19, 65]. This can be accomplished by the use of Tor Onion Services (OSes), which are basically being adopted as an infrastructure with regards to make the so called "Dark Web" a reality. However, before we overview OSes' functioning, we should firstly explain how Tor works.

2.1.2 Network Overview

Tor's architecture tries to provide a great degree of anonymity to the sender, ensuring that by, typically, constructing a three-hop path through the network, also known as circuit, employing a layered encryption model comparable to onion routing [42]. In short, Tor allows a client to establish a TCP/IP connection to a receiver while keeping the IP address of the sender anonymous from the receiver. These TCP/IP connections of a particular client are tunnelled between the 3 nodes of the circuit (also known as relays), where the first one corresponds to the entry node, the second to the middle node and the third to the exit node. This onion routing strategy is a great way to forward messages in an anonymous way, since it

requires each node of the circuit to unwrap the encryption layer of the received payload before forwarding it to the next node. Thus, it makes it impossible for a single node to deduce both where the request came from and where it goes. While the entry node can infer the identity of the initiating client, the exit node can infer the destination server when it decrypts the received payload. However, in order to break the anonymity of the sender, a particular user has to be the owner of both nodes in a given circuit.
The circuit is formed as follows. Firstly, the client consults the relay nodes from a public document named *consensus* [53] and picks 3 of these nodes at random. Every node has some identifying parameters, such as the IP address and the public key, while keeping the private key for themselves. These public keys are used to authenticate each relay node and to negotiate an exchange of the symmetric keys that will be required to encrypt and decrypt each encryption layer on the relays. In order to hamper traffic analysis attacks, IP packets are encapsulated into fixed 512-byte cells and for each cell, the client encrypts it first with the symmetric key of the exit node, then with the key of the middle node and finally with the key of the entry node. This way, the messages will be delivered in the correct order for each relay node, where the upper layer will be decrypted by the entry node, the middle one by the second node and the third by the exit node before delivering it to the destination server. What guarantees anonymity of the client is the fact that the exit node does not know the IP address of the sender and the receiver only knows the IP address that came before (the IP address of the exit node).

2.1.3 Onion Services

While Tor's architecture makes it feasible for the sender to keep its anonymity, the same does not apply for the receivers, where we have to consider additional complexity. This is due to the fact that once a Tor circuit is established and right after the exit node unwraps the last encryption layer, that same node is able to uncover the IP address of the receiver in plain-text before forwarding the cell to that IP, which does not provide anonymity of the receiver.
This problem is addressed with a mechanism called *onion services*. In order to achieve this, we have to take some inspiration from the fact that a Tor circuit preserves the anonymity of the sender. Therefore, if we build two independent circuits and connect them to an intermediate relay node called *rendezvous point* (RP), we can also maintain the receiver anonymous. So, the client instead of connecting itself to the receiver, attempts to establish a connection to an onion service, that is basically built upon the public key used by the receiver node. From then on, it just picks a relay from the network (the RP) and establishes a circuit with it, making it unfeasible for the RP to know the IP address of the client. Meanwhile, the receiver of the communication, after locating the RP, does the same process described above, which is also materialized by the creation of an independent circuit to the RP. Since the RP is connected to both the sender's and the receiver's IP addresses independently, it does not know which one is the receiver and which one is the sender, so it just forwards traffic for both endpoints, keeping them anonymous in that process.

2.1.4 Security

The security and anonymity of Tor relies on a simple assumption: no single organization will own both the entry and exit nodes of a given circuit. If that happens to be the case, then that organization can

correlate the circuit traffic with a very high probability (e.g., measuring the inter-packet arrival time and packet volume) to discover the endpoints of the communication i.e., the IP addresses of the sender and the receiver. This, however, is not as easy as it seems, because generally, Internet Service Providers (ISPs) control some nodes of the network, but Tor tends to select relays connected to different ISPs to make it almost unfeasible to deanonymize the traffic. In practice, this results in a cluster of networks: the so-called Autonomous Systems (ASes), where ISPs are interconnected to each other. These ASes make deanonymizing Tor circuits extremely challenging for several reasons. Let's suppose there are 2 ASes involved in a specific Tor circuit, where the entry node is in AS1's network and the exit node is in AS2's. In practice, different ASes have to agree to exchange information about the traffic on their network. Therefore, AS1 and AS2 have to cooperate so they want to correlate traffic, but that also means that they have: to jeopardize the privacy of their clients; take into account the global or local political legislation on data privacy and invest a lot of financial resources to increment the processing power of their networks. Thus, these reasons make it actually demanding and discouraging for ASes to deanonymize Tor circuits.

2.1.5 Attacks

This section will be focused on some attacks that can occur in Tor. We will first categorize these attacks and then provide some academic background and reference the state-of-the-art of each one of their types. In particular, we will be focused on traffic correlation techniques. Since Torpedo [23] is contained within the scope of this kind of attacks, we are skipping them in this section and thoroughly discussing them in the following one.

Fingerprinting attacks: In a fingerprinting attack, an attacker leverages the fact that Tor traffic has some unique characteristics, e.g., the packet timing or the packet volume for a given web page or onion service. This allows an attacker to identify which OS [49] or website [51, 61] a certain client is requesting.

Denial of Service attacks: An adversary can exhaust the Tor network, jeopardizing the performance of the connections and degrading the quality of service as well as the security of the circuits itself. For example, the CellFlood [11] attack shows that, by sending some computationally demanding circuit setup requests to specific Tor relays, it is possible to weaken the performance of those same relays to serve other requests that they receive. Another type of attack was studied by Jansen et al. [31], that explored the capability of a generic attacker to exploit Tor's congestion and flow control mechanisms to take down import relays and reroute users to malicious relays controlled by the adversary.

Timing attacks: During a timing attack, an attacker can manipulate both the entry node and the exit node of a specific client and by correlating flow patterns from both nodes, it is possible to know which server the client is communicating with. An example of this attack is the one done by Chakravarty et al. [17], where a server colludes with the adversary. By analyzing the bandwidth probing nodes placed near the boundaries of some ASes, the attacker can then find the bandwidth pattern used by the colluding server and subsequently trace that connection back to the client.

Congestion attacks: In these types of attacks, an adversary tries to determine the relays that constitute the circuit to be used by a particular client. This can be achieved by measuring the latency differences in

traffic flow in the client by congesting relays while a website controlled by the attacker serves that target. When Tor was not still a huge network, Murdoch et al. [46] achieved something like this by letting an adversary connect to a corrupt server and serve a particular target. Then, this server sends traffic to the client using a specific pattern. If the latency pattern of a relay matches the latency pattern of the corrupt server traffic to the client, then we can assume that that relay is part of the circuit of the client. This attack is obviously unfeasible today, as there are around 7000 relay nodes on Tor [54].

Supportive attacks: These attacks do not aim at deanonymizing Tor users or harming a Tor network connection, but rather assisting a subsequent attack that manages to accomplish that. A Sybil attack is when an adversary is able to own several relays, simplifying traffic correlation or website fingerprinting, e.g. The effectiveness depends on the consensus weight, i.e. the amount of traffic an adversary is able to observe from those relays. In 2016, Winter et al. [66] showed that this kind of attacks are not trivial to defend against without a central authority, so they developed a tool called *sybilhunter*, that analyses archived Tor data in order to categorize the Sybil groups that it finds.

2.1.6 Traffic correlation attacks

In this type of attack, it is assumed that an adversary is able to eavesdrop the network between the client and the entry node and then between the exit node and the server, i.e. in both endpoints and also in the first and last relays of a given circuit. It is not required to observe the whole circuit to be able to correlate the traffic. This process allows the attacker to correlate the detected traffic and deanonymize the client and the server, since the entry node knows the IP address of the source and the exit node knows the IP address of the destination, so it becomes possible to deduce that they are communicating through the same Tor circuit [16]. These attacks can usually be accomplished by the comparison of the inter-packet delays and packet sizes of the flows. If these values are similar enough for a given pair of flows, then they are probably correlated. We will now contextualize some flow correlation attacks on Tor.
The Bad Apple attack [13] is a traffic correlation attack at the application level, that consists in two phases: the exploitation of an insecure application that retrieves the IP address of a Tor client and the exploitation of Tor itself to return usage information of that IP address on a given application. What makes this attack a prominent threat is the fact that it just requires an insecure application to possibly compromise the anonymity of a particular user. If an attacker owns the exit nodes of a circuit that is being used by a normal Tor user and already retrieved the IP address of that client using the malicious application, then if he later observes traffic on that same circuit, he can correlate that user with the destination server observed at the exit node. In just 23 days, Le Blond et al. [13] were able to deanonymize 10000 IP addresses of Tor clients, using BitTorrent as the nefarious application. After this first step, they can profile the subsequent requests made by the users.
In 2017, Nasr et al. [48] proposed a traffic correlation approach, which they called *compressive traffic analysis*, that allowed an adversary to correlate flows only by computing the compressed traffic features instead of the raw features, since the latter usually demand high storage and a overhead in both computations and communications that we can't afford if we want our system to be scalable. So, the authors leveraged the idea of random and deterministic linear projection algorithms to compress traffic features, such as packet sizes and inter-packet delays. This way, these features get compressed and the correlation

process is not as resource-intensive, therefore enhancing scalability. The performance, however, may get deteriorated, since the correlation is being performed on compressed data.
RAPTOR [63] created a new paradigm for traffic correlation techniques, since this attack can be launched by ASes in order to jeopardize user anonymity, so we have to assume a resourceful adversary in the threat model. There are some dynamics of the Internet itself, such as routing asymmetry and churn that are going to be abused in order to carry on this attack. For example, Sun et al. [63] took advantage of the former to get better chances of observing traffic from (at least) one of the directions of the endpoints, either ingress or egress traffic. They knew it would be possible to obtain the TCP SEQ# and ACK# of the packets exchanged during the communication, since the TCP headers are not encrypted in both endpoints, so it becomes feasible for a malicious AS to intercept and deanonymize the client and the server, without having the need to follow both directions of the traffic. Then, RAPTOR also exploits the intrinsic dynamics of the churn, e.g., rearrangements of BGP paths, that allow ASes to observe extra traffic. Every one of these changes may grant a malicious AS new opportunities to perform asymmetrical traffic analysis, thus increasing the deanonymization chances of additional Tor clients. Finally, RAPTOR can also hijack or intercept the BGP itself. In a BGP hijack, the malicious AS just announces to the network a prefix that does not belong to it, so, as a consequence, many users that want to communicate to the legitimate AS that owns that given prefix, communicate instead with the AS owned by the adversary. A BGP interception is even more dangerous, because it not only does this, but also inspects and forwards the traffic to the specific destination, keeping the client connection alive. By performing a BGP hijack, the attacker is able to detect which users are using specific guard relays, while a BGP interception combined with the last attack allows the launch of more asymmetrical traffic analysis attacks, providing exact deanonymization of clients. RAPTOR is able to achieve great precision, but it requires, however, the observation of a huge amount of network flows, which may be impractical in a real scenario.
DeepCorr [47], as we'll explore in more detail in Section 2.2.3, is still the state-of-the-art regarding traffic correlation attacks on Tor utilizing neural networks. It utilizes a CNN (Convolutional Neural Network) that is trained on flows, leveraging some characteristics of them, such as the packet sizes and inter-packet delays, in order to correlate these flows in the subsequent inference phase. It achieves great precision results and does not need to observe as much data as RAPTOR [63].

2.2 Neural Networks

In this chapter, we'll explore the topic of neural networks and their capability to be applied in classification problems. We'll firstly discuss how these networks function, then we'll expand their definition to incorporate convolutional neural networks (CNN) too and, finally, we'll also specifically review DeepCorr's [47] CNN, since Torpedo's [23] approach is closely inspired by it.

2.2.1 Introduction

According to Gurney [30, Chapter 1], a neural network (NN) is "an interconnected assembly of simple processing elements, units or nodes, whose functionality is loosely based on the animal neuron". These networks take in data, train the nodes to recognize the patterns that exist in that data and then predict the output for similar datasets. We can take advantage of this workflow to classify flow pairs from Tor.

Generally, we can divide neural networks into three types of components: *the input layer*, *the hidden layer* and *the output layer*. Figure 2.1 exemplifies a simple neural network, where we have 2 input nodes, a single hidden layer with 5 nodes and 1 output node. As we can see, both input nodes have channels connecting them to the hidden layer and these channels have a value designated *weight*, that will be relevant to the next node they are linked to in the propagation. We multiply these weights by the values of the input nodes and add a specific value of the node they are connected to in the hidden layer called *bias*. The result will be then passed as input to an activation function to determine if the corresponding node in the hidden layer will be indeed activated or not. If the former is true, it can transmit data to the next nodes in a process that is known as *forward propagation*.
Let's imagine an elementary example, where we have a neural network that is slightly more complex than the one in Figure 2.1. In this particular neural network we are trying to predict if the input matches a certain geometric figure (e.g., a square or a pentagon). The output nodes will correspond to all the possible geometric figures that this neural network is able to identify. Let's suppose the output node with the larger value after an iteration of forward propagation was the one corresponding to a pentagon, but the real input was supposed to be a square. In this example, the ground truth would be represented by a 1 for the pentagon and a 0 for all the other geometric figures (just the square if there were no more). The error is calculated by the ground truth of each figure minus the predicted output they got after the forward propagation. This way, there can be made some adjustments to the network in order to correct the prediction through *backpropagation*, where the weights and biases are adjusted, thus reducing the errors and approaching the expected result. It is important to note that during the dataset training phase, many of the output values may not be so close 1 or 0, so this rectifications must be iterated in such a way that in the dataset testing phase, the values are the closest possible to the ground truth.

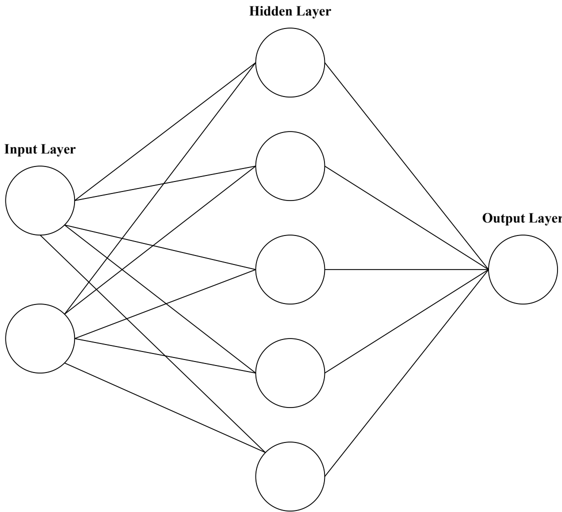


Figure 2.1: Architecture of a simple neural network

2.2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of NNs in which the network employs a linear operation on two functions of a revalued argument that is called convolution [29, Chapter 9]. In traditional neural networks, all output nodes interact with all input nodes. In Convolutional Neural Networks, however, this is not the case, since these networks have sparse interactions, which means that they only run a subset of nodes smaller than the whole input in filters designed *kernels*. For instance, let's consider we want to build a regular neural network for image detection. We can assume that the input layer will require millions of nodes (e.g., a 1920x1080 image has more than 2 million pixels). If we assume *m* inputs and *n* outputs, this means that, in practice, we will deal with a $O\ (m \times n)$ runtime due to the $m \times n$ computations, which can easily become quite demanding, bearing in mind the number of input nodes in this particular scenario. With the application of CNNs in this case, we can still detect meaningful features from an image with kernels that are constituted by considerably less pixels than in the original hypothesis. Therefore, if we consider *k* as the number of connections to each output node, where $k \ll m$, then the computational effort will be shortened. Besides, we don't have to store so many parameters, hence reducing the memory requirements for the network itself [29, Chapter 9]. As a result, the performance will be substantially better, which validates CNNs as feasible options for image detection and recognition [18].

In a CNN, the hidden layers are the ones that are performing convolution, which, in this case, involves a scalar product of the kernel with the layer's input matrix, returning a value that will be added to a new data vector (the convolution output). The activation function of this product is commonly ReLU (Rectified Linear Unit) [7] and it essentially determines the maximum value between 0 and a given input *x* (basically $f(x) = max(0, x)$) [41]. Let's presume that a negative value was passed to the convolution output. Using ReLU as the activation function, this value would become 0. If the value was already positive, then it would stay the same. The kernels will then slide through strides of the input, applying the same scalar product and activation function to execute the convolutions, adding those values to the new data vector. In order to reduce the number of features to be computed, we have to introduce another concept: a *pooling layer*. Usually the data is pooled by a maximum or average value, so this max or average pooling layer is then responsible for decreasing the initial input size by only merging the maximum or average value on a particular window into a single node in the next layer, respectively. Accordingly, this will result in less data compared to the initial input. The data vector will then receive the next convolution layer and this process will be iterated until there are no more convolution layers. Finally, we can proceed to the next step, where the layers become fully connected, this being achieved by flattening the multiple vectors into a single one. Only after this, the network is ready to be evaluated.

2.2.3 DeepCorr's CNN

This subsection will be focused on DeepCorr's CNN [47], that, like Torpedo's [23], is also being used with the purpose of correlating Tor traffic flows. Nasr et al. [47] chose this type of network, since they are known to achieve a great performance on time series [58, Chapter 4] and Tor flows can be easily modelled as so.

This CNN, similarly to many other neural networks, also has two phases: the training and testing phases.

The first one has to process a large quantity of flow pairs, some correlated, i.e., belonging to the same Tor connection, and others uncorrelated. In the testing phase, pairs of flows are given to the network and the CNN evaluates the probability of them being correlated or not, wherein the output is expressed between the values 1 (correlated) and 0 (uncorrelated), respectively.
Figure 2.2 illustrates the architecture of DeepCorr's CNN. It first receives four features for each flow (two vectors for the inter-packet delays and two for the packet sizes of the flows). Then, these features go through a first layer of pooling and convolution, where there is extracted some information that might indicate a possible correlation between two adjacent Tor flows (that are expected to be correlated). After this step, it comes the second convolution layer, that tries to capture overall traffic patterns through a combinatorial way of all size and timing features from the flows. This output is then flattened and fed to a fully connected network, composed by three layers.

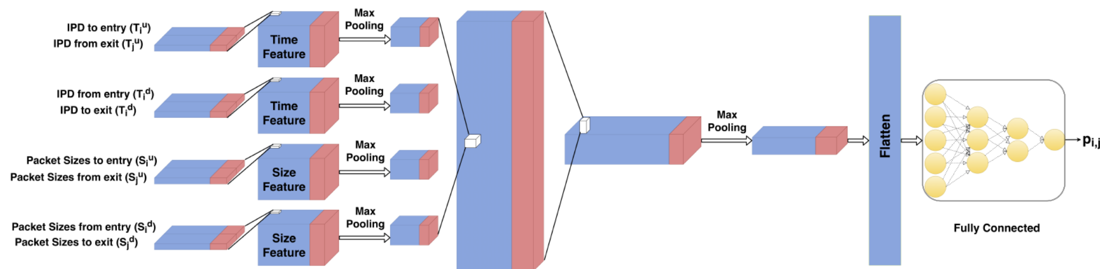


Figure 2.2: Architecture of DeepCorr's CNN [47]

When compared to other solutions for Tor flow correlation attacks like RAPTOR [63], DeepCorr outperforms the former by a great margin of precision of correlated flow pairs in the same settings (96% *vs* 4%). This is due to the fact that Nasr et al. [47] considered the dynamic and intricate essence of noise in Tor, so instead of developing a generic statistical correlation algorithm, they leveraged the properties of CNNs for time series [58, Chapter 4] and, like described above, built a convolutional neural network that could learn a correlation function to match flows on arbitrary circuits and destinations, regardless of whether or not those same circuits and destinations are part of the training set. Besides, DeepCorr's performance gets even better when considering longer flow observations with larger training sets, being also quicker than previous work to achieve reasonable true positive rates and precision values [47]. Finally, unlike other deep learning algorithms such as Fully Connected Neural Networks (FCNNs), Recurrent Neural Networks (RNNs) or Support Vector Machines (SVMs), CNNs provided the best flow correlation performance, since they are more fit to capture complex features from raw input data as well as larger amounts of data than the other algorithms [29, Chapter 9].

2.3 Torpedo [23]

Torpedo [23] is a distributed system that can be deployed as a traffic correlation attack on Tor, which provides global deanonymization of OS (Onion Services) sessions. In the next sections we'll briefly point out some challenges and overview its design and all of its phases. Lastly, we are also going to

discuss some countermeasures for the attack as well as a limitation of its current version in order to motivate the introduction of MPC techniques to mitigate some shortcomings.

2.3.1 Challenges

Torpedo demonstrates the feasibility of deploying a traffic correlation attack on Tor and the repercussions it may generate for its users. It turned out to be a real challenge, because unlike other existing traffic correlation attacks [46, 47, 63], that focused their efforts to analyse *individual* Tor circuits, Torpedo instead correlates *OS traffic*, which makes it considerably harder, since an OS session is the product of two interconnected Tor circuits: one established with the client and the RP and the other established with the OS and the RP. Furthermore, there were even more challenges that included:

- The differentiation of Tor OS traffic from non-OS traffic;
- The client concurrency at OS endpoints;
- The scalability requirements of the system despite the possible large flow samples of both clients and OSes.

In the next section, we'll discuss Torpedo's design in detail and explain how it overcame some of these challenges.

2.3.2 System Overview

Firstly, Torpedo is composed by three different components: query interface, filtering nodes and matching nodes. Moreover, it also involves three agents: the operator (attacker), ISPs and optionally CSPs (Cloud Service Providers).
The query interface is just a simple software abstraction that allows the operator to submit deanonymization queries to the system, resulting in an output where the matched OS sessions that meet the conditions of the query are displayed.
The filtering and matching nodes, however, are utterly linked to the main structure of Torpedo's architecture. On one hand, we have the filtering nodes, which are deployed by ISPs on specific points. Their goal is to observe Tor flows and filter samples from OS traffic. They constitute a system that can always be subject to change, since ISPs have the option to add or remove many of these nodes incrementally from particular vantage points. A key point that has to be highlighted is the fact that these ISPs only retain authority and management accountability over the nodes deployed in their network (regardless of their geographic location), so they are basically seen as *untrusted entities* to each other.
The matching nodes' job is to process the deanonymization queries submitted by the operator and can be hosted by ISPs or by external trustworthy CSPs. They leverage the computations already retrieved from the filtering nodes and perform the correlation process to deanonymize OS sessions.
Both of these last two components will have have their specific sections (2.3.4.1 and 2.3.4.2), where the Filtering and Matching Phases are going to be explored in a slightly more detailed way.

2.3.3 Use Cases

Generally, there are three use cases for Torpedo:

- **Sniper operations:** Probe an identified OS, in order to monitor future traffic, track down the owners or simply shut it down;
- **Sting operations:** Identify OSes accesses by specific clients, where the queries use the source as the IP of the client they are looking for and the destination as the IP of the targeted OS;
- **Decoy operations:** Setup of a honeypot OS, to gather IP addresses who connect to it to potentially track them down later.

The following query is an example of a sting operation, where the attacker checks if a specific user with IP address x.x.x.x is accessing a specific OS with IP address y.y.y.y between 8PM and 9PM of June 06, 2022.

```
MATCH FROM x.x.x.x TO y.y.y.y WHERE time.window
between 06-06-2022 8PM and +1h RET status
```

2.3.4 Workflow

Regardless of the way operators interact with the query interface, Torpedo follows a query execution workflow that relies on a multi-stage pipeline architecture with two separate stages: filtering phase and matching phase, as we can see in Figure 2.3.

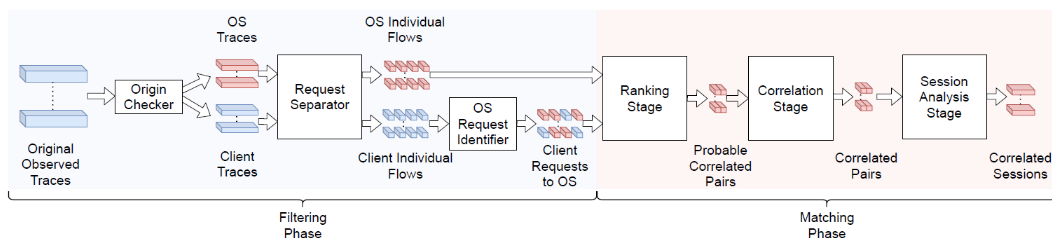


Figure 2.3: Query execution workflow in Torpedo [23]

2.3.4.1 Filtering phase

The filtering phase is responsible for pre-processing Tor flows that are gathered in specific points. We can even deepen and divide this phase into three sub-phases: Origin Checker, Request Separator and OS Request Identifier.

The Origin Checker tries to differentiate flows generated by normal Tor users from flows generated by OSes. That way, we can subsequently focus only on flow pairs that involve a client and an OS. As it was already studied in the literature, it is possible to distinguish some communication patterns of Tor clients and OSes [50], since the latter have to publish their addresses to Tor directories after establishing the connection to their guard node, generating substantially more traffic in that process.

The Request Separator, as the name suggests, breaks down the flow bursts collected in the Origin Checker and isolates the client requests from the OS responses in each of those bursts, aiming to generate a simple sample to be used in the following stages of the workflow.
The OS Request Identifier has the goal to filter all the clients' requests to OS sites, thus excluding all the requests towards regular sites. It is relatively easy to make this distinction as Tor clients create extra circuits when they connect to different OSes. Besides, they also generate a higher disparity of bin byte counts than non-OS sites, making it more approachable for the classifiers to discern between both classes. After all of these 3 steps, the filtering phase is over and the final output is a set of Tor flows that start in specific clients, go towards particular OSes and the other way round.

2.3.4.2 Matching phase

The goal in this phase is to find sessions involving the same client and OS, i.e., identify which clients are connected to which OSes. Like the previous phase, this one also has three stages: Ranking stage, Correlation stage and Session Analysis stage, where each stage has to cope with a more granular level of flow information.
In the Ranking Stage, for each flow pair will be given an output score ranging from 0 to 1, which represents the confidence that a given pair is indeed correlated. Given two flows (A and B, where A $\neq B$), the confidence scores are calculated through a simple neural network with three fully connected layers, where there are: packets in a forward direction (e.g., the difference of packets from A to B), packets in a reverse direction (e.g., the difference of packets from B to A) and the time of capture, that refers to the difference of time between the first packet of each flow. Pairs with a higher score than a PT (*passthrough threshold*) will be selected as candidates.
The Correlation Stage processes the set of potentially correlated flows identified in the previous stage. Likewise, its output is also similar to the Ranking stage, but instead of assigning a confidence score that predicts if a given pair is simply correlated, this confidence score (also between 0 and 1) can be read as the probability that *a client-generated flow is correlated to a given OS-generated flow*. It leverages many ideas from DeepCorr [47] and implements a convolutional neural network whose inputs are the time series of inter-packet delays and packet sizes of each pair of flows, obtaining a higher precision than the generic statistical correlation equivalents. The main difference compared to DeepCorr [47] is that Torpedo's CNN has less neurons and kernels in both convolutional and fully connected layers, due to scalability requirements.
The last step of the Matching phase (and also Query Execution Workflow) corresponds to the Session Analysis Stage, in which the flow pairs produced by Torpedo's flow correlator are analysed and then it is decided whether that set of flows is sufficient for identifying a browsing session between a client and an OS. In this phase, the flow pairs that share some resemblance between them (e.g., the timing differences between the first packet sent by the client and the first packet received by the OS are similar) are aggregated into buckets, only remaining those that belong to the bucket containing the largest amount of pairs. After this refined selection, it is counted the *minimum number of estimated requests* (MERS) per session and the *session correlation score* is calculated based on the score of all requests within the largest bucket. Finally, only candidate sessions which have a session correlation score higher than a ST (*session threshold*) will be then defined as correlated.

2.3.5 Countermeasures and Limitations

In this section, we'll discuss some of Torpedo's possible countermeasures as well as a specific limitation of the current version. To prevent the potential success of such an attack, the countermeasures that will be mentioned can assist Tor against OS traffic deanonymization. Regarding the limitation in particular, we are discussing it in order to motivate the future work concerning the introduction of multi-party computation techniques to solve the problem that will be described ahead. A possible solution for this obstacle can enhance the attack, furthering its adoption, so it seems reasonable for us to conceive the worst case scenario, if we also want to achieve the best possible countermeasures.

The precision of the results obtained in the request separator classifier did not achieve their full potential, so one possible defense is for Tor clients to send overlapping requests to other webpages while trying to access an OS.

Another Torpedo's countermeasure, however, is increasing concurrent replies sent by OSes. This can be accomplished if clients and OSes are launched in the same endpoint. In that case, a possible overlapping of requests may occur and, therefore, spoil the flow separation process, as well as the recall itself.

What can also jeopardize Torpedo's correlation effectiveness are traffic obfuscation schemes. Up until now there is yet to exist a feasible and practical system with good network performance that provides anonymity. However, as research in this field advances and costs of deployment are reduced, an adopted Tor traffic obfuscation system may very well be considered a valid countermeasure to Torpedo.

Finally, although it can't be considered a countermeasure, there is the limitation that we are trying to solve, that consists in allowing Torpedo to be run collaboratively by ISPs using multi-party computation techniques. So far, there are no guarantees of data privacy both on the input and the output of the correlation process. This implies that ASes are required to share sensitive data (e.g., flow information) with a third party, which may discourage many of them from participating in Torpedo. In the next section, we will try to study the state-of-the-art of the already existing multi-party computation frameworks that try to overcome this particular limitation.

2.4 Multi-party Computation

In this chapter, we will first define multi-party computation, indicate the objectives and guarantees behind this protocol along with a historical background and some security considerations when building a privacy-preserving model using MPC. Then, we'll divide practical multi-party computation applications into two distinct categories: the general purpose frameworks and the machine learning based ones and summarize the state-of-the-art for each one of them.

2.4.1 Introduction

Multi-party computation (MPC) is a cryptographic protocol that has to ensure that a group of mutually untrusted entities can correctly compute a function on their secret inputs without disclosing them to the other participants of the computation [20]. Let's suppose a basic example, where we want to compute the average salary for a group of 3 people, being them A, B and C. If we employ secret sharing techniques, as we'll further discuss ahead, the salary of A can be split into three shares that add up to the salary

of that person and those shares are then distributed between A, B and C. This process then repeats for these last 2 people. After that, all of them will have 3 shares that, when summed up, will not reveal any confidential information. Finally they can add up all of these totals between them and divide it by 3 to obtain the average salary.
According to Cramer et al. [20], the most basic conditions that a MPC protocol aims to guarantee are:

- **Privacy:** The output of the computed function is the only new information that is released to the participants.
- **Correctness:** The value of this function is distributed according to the prescribed functionality.

It is then evident why Torpedo [23] can become a widely adopted and even more threatening attack if it provides a privacy-preserving solution. The implementation of a MPC protocol in Torpedo may not only allow operators and ISPs to correctly compute their users' data without sharing them to a third party, but also ensure that both entities themselves have very limited knowledge regarding these data.
Before discussing the historical evolution of MPC, we should first expose a useful primitive for these protocols: *secret sharing*, which is a method for distributing a secret between a group of participants, where each one gets a share of the secret. Some sets of participants are qualified, while others are unqualified, which means that not all of them can deduce the shares they are given. A popular and yet simple technique is the additive secret sharing scheme, that essentially consists in adding all the shares in order to recreate the original secret. In 1979, Shamir provided another efficient scheme that was known as Shamir's secret sharing [59]. It is based on polynomial interpolation in a plane of two dimensions. The intuition behind it is that it takes n points to define a polynomial of degree $n - 1$. Shamir suggested to embed the secret s as the first coefficient ($(0, s)$ in the 2D axis) of the polynomial of degree $t - 1$, while all the other coefficients are generated random points that will define the curve in the 2D axis. To unlock the secret, it is just required that t participants get their shares from that curve and that way, they can actually reconstruct the original secret. Both of these schemes are used in several practical MPC implementations, as we'll study in Sections 2.4.2 and 2.4.3.
It was also in 1979 that a specific problem was addressed with a MPC solution in mind for the first time, when Shamir et al. [60] (the creators of RSA) attempted to answer the question "Can two potentially dishonest players play a fair game of poker without using any cards (e.g. over the phone)?". At the time, this problem turned out to be insoluble, since the answer would be paradoxical, but at least the authors managed to provide a fair way to play the game, where the players did not apply secure message transmission, but rather secure computation, employing the use of cryptographic algorithms, that three years later, in 1982, became the foundation of a particular sub-field of MPC: two-party computation (2PC). It was in that year that this term was first introduced in the famous Andrew Yao's Millionaires' Problem. The problem simply consists in two millionaires that are trying to know who is richer without disclosing their wealth [68]. Five years later, Goldreich et al. [28] proposed a general solution for this class of problems, that proved to be the foundation for the next couple of years in the field of multi-party computation. The algorithm developed in that work accomplished polynomial-time complexity for an *independent number of participants*, provided that the majority of them was honest, where the input of a dishonest party either was either revealed or the participant himself would get eliminated. According to this research, we have to consider two types of adversaries:

- **Passive or semi-honest:** These adversaries try to obtain information outside the scope of the protocol, but they also act in accordance to the original protocol, following the due specification. They generally violate the privacy requirements of the computation, trying to extract information about the other participants' inputs, but without tampering the computation output.

- **Active or malicious:** In this case, the adversaries can deviate arbitrarily from the prescribed specification and have not only the chance to disrupt the privacy of the computation, but also the correctness. For instance, the BGW protocol [12], that employs a Shamir's secret sharing scheme, is secure against $a < n/2$ passive adversaries, but for a malicious setting, an adversary can only disrupt the computations or obtain sensitive information if $a < n/3$ [10, 12], being a the number of adversaries and n the total number of participants.

We should then take into consideration that the security guarantees for a malicious model are stronger than for a semi-honest one, but the latter are generally more efficient, since they only concern about avoiding the disclosure of the participants' private information.
It was only several years later, in 2008, that Bogetoft et al. [14] were able to create a large-scale and practical MPC application that could achieve efficient results in the context of a secure auction, dismissing the requirement of a third party. The approach was based on Shamir's secret sharing among 3 servers (3PC) and assumed a passive adversary who could corrupt an arbitrary number of input clients and a minority of the servers. It was regarded as the first step towards a practical commercial MPC application. Currently, MPC frameworks have advanced and are closer and closer to become feasible in practice, since they are getting increasingly more efficient as research in the field evolves. The protocol itself can even be applied to solve various real-life problems, especially those that usually require trusting in a mediator or a third party not to disclose secret values of the participants in a computational process, namely voting [8], auctions [14] and privacy-preserving machine learning [45].

2.4.2 General-purpose frameworks

In this section, we'll discuss some general-purpose MPC frameworks proposed in the literature.

2.4.2.1 SPDZ [22] / SPDZ-2 [36]

In 2012, Damgård et al. [22] proposed a MPC protocol that would later be known as SPDZ, one of the predecessors of many subsequent practical general-purpose MPC frameworks. This protocol assumes an active adversary model that can corrupt up to $n-1$ of the n participants and also presumes synchronous communication and secure point-to-point channels. It can be used to compute arithmetic circuits over finite fields. It has an online phase where the efficient computation takes place, but it is preceded by a preprocessing phase where there are generated triples of the inputs of the computation, via somewhat homomorphic encryption (SHE). Homomorphic encryption denotes the form of encryption that enables operating efficiently on encrypted data without the need to decrypt it first. In a SHE scheme we can't compute anything on encrypted data, because somewhat homomorphic encryption only supports a limited number of homomorphic operations, being able to evaluate two types of gates (e.g., addition and multiplication), but just for some circuits, unlike other stronger HE notions such as full homomorphic

encryption (FHE), that allow the evaluation of arbitrary circuits with several types of gates of unbounded depth. The assumptions for a SHE scheme are, therefore, weaker than for a FHE, but that is what makes it a better practical solution, since the performance is also superior for these types of schemes.

The privacy and correctness requirements of SPDZ are guaranteed by authenticated shared values with information-theoretic MACs on top of them through an additive secret sharing scheme. In a scenario where the majority of the participants is dishonest, we cannot assure a successful termination, so the operations just abort if cheating is detected, in what is called *active-security-with-abort*. Although this only provides security against passive attacks (semi-honest adversaries), the overall protocol can be considered actively secure, since the only way an attacker can inflict some "damage" is by adding an error to the decryption output, which results in wrong secret shares, thus producing a failed MAC check.

In the following year, Keller et al. [36] took the first steps towards a practical solution based on the specification of SPDZ [22], that would later be known as SPDZ-2. This work mainly focused in providing an efficient actively secure protocol against a dishonest majority, something that was lacking in similar works in the research field. The authors made use of SPDZ to achieve the desirable security and efficiency and then built a compiler for a Python-like language on top of that, whose bytecode would be interpreted by a simple virtual machine (VM). The implementation directly tackles general purpose computation, translating the high level function descriptions to low level operations and then executing them in the runtime engine of the VM.

2.4.2.2 MASCOT [35]

In 2016, Keller et al. [35] presented MASCOT. This protocol considered the task of accomplishing secure MPC of arithmetic circuits over a finite field. While the boolean circuits' approach usually results in faster protocols that mostly leverage symmetric cryptography to perform secure computations, the arithmetic circuits' approach is much slower, since it employs asymmetric cryptography, this presuming that the majority of the participants is not honest. Most MPC models that assume malicious adversaries with a dishonest majority try to solve this problem by utilizing this last approach for each multiplication in the pre-processing stage of the protocol. However, this comes with a very high cost in performance, both in throughput and communications. MASCOT instead uses *oblivious transfer* (OT) for efficient and secure multiplications, relying almost solely on symmetric cryptography and therefore avoiding the aforementioned shortcomings. Essentially, oblivious transfer is a protocol where there's a sender and there's a receiver and the messages that are sent to the receiver remain oblivious to the sender, so he does not know what messages he transmitted, if any. The first oblivious transfer method was introduced in 1981 by Rabin [56] and simply consisted in a sender sending a bit to an OT machine. The receiver either gets the same bit or an undefined value (with 50% chance each). Four years later, Evan et al. [25] proposed an approach (1-2 oblivious transfer) that would be eventually generalized to 1-n oblivious transfer, in which the receiver only receives one out of n requested messages and the sender is unaware of which one of those values was sent.

Getting back to MASCOT, the approach followed by Keller et al. [35] was creating multiplication triples, where there are three values: two of them are random (a, b) and the other is a secret-shared global random MAC key (r). The shares are then generated from random pieces of those values (a, b) and also from random pieces of the products $(a \cdot b, a \cdot r, b \cdot r, a \cdot b \cdot r)$. In order to obtain an actively secure

protocol, MASCOT guarantees correctness of the products of the MAC generation and also correctness and privacy of the multiplication triples. To achieve a security model against malicious adversaries, the authors decided to use simple consistency checks and privacy amplification techniques, at a cost of a 6 times less efficient protocol than the semi-honest version, but still 200 times faster than the previous best actively secure implementation at that time (SPDZ-2 [36]).

2.4.2.3 SCALE-MAMBA [9]

SCALE-MAMBA [9] is a fork of SPDZ-2 [36]. It implements arithmetic computation modulo a prime (including support for fixed point and floating point arithmetic), not modulo a power of two, but it also implements binary computation based on secret sharing for both honest and dishonest majorities. All of these computations are achieved with malicious security in mind, providing an active-security-with-abort MPC practical framework, meaning that if a malicious adversary aborts the operation, the deviation is detected but the protocol cannot recover from that. SCALE is the runtime, that executes the protocol based on the bytecodes. MAMBA is the front-end language that is inspired by Python. The compiler, however, is currently being developed on Rust, replacing this implementation.

2.4.2.4 MP-SPDZ [34]

In 2020, Keller developed MP-SPDZ [34], which stands for Multi-Protocol SPDZ and, like SCALE-MAMBA [9], is also a fork of SPDZ-2 [36], an implementation of SPDZ [22]. MP-SPDZ supports other 34 MPC variants and provides a Python interface from where they can be accessed, thus favouring the comparison/benchmarking of the protocols. It can be seen as the ultimate practical general-purpose MPC framework, since it covers all the possible security models (semi-honest/malicious with honest majority/dishonest majority), also using the most common computation primitives (binary circuits or arithmetic circuits) and including some usual support primitives, namely homomorphic encryption, garbled circuits, secret sharing and oblivious transfer. The motivation behind this work lied mainly on the research convenience of assessing the security models of different protocols as well as the need to compare them on similar settings. This way, Keller provides an implementation of 34 MPC protocols within the same virtual machine, allowing a compiler to turn high-level code into bytecode so that the execution takes place in the said virtual machine.

2.4.2.5 Comparison

Table 2.1 summarizes some of the main differences between general-purpose MPC frameworks. "Open-source" denotes if the code is available for everyone to use it. "Supports other protocols" simply refers to whether the system provides support to additional protocols or not. "Semi-honest security" and "Malicious security" distinguish systems who can ensure both correctness and privacy for semi-honest and malicious adversaries, respectively.

Table 2.1: Simple comparison between general-purpose MPC frameworks

Features	SPDZ-2	MASCOT	SCALE-MAMBA	MP-SPDZ
Open-source	✓	✓	✓	✓
Supports other protocols	✗	✗	✗	✓
Semi-honest security	✓	✓	✓	✓
Malicious security	✓	✓	✓	✓

2.4.3 Machine learning based frameworks

In this section, we'll discuss some machine learning based MPC frameworks proposed in the literature that could help us with Torpedo's [23] neural networks in the matching phase of the protocol pipeline.

2.4.3.1 ABY^3 [44]

In 2018, Mohassel and Rindal created a privacy-preserving machine learning (PPML) framework via MPC called ABY^3 [44]. This protocol is characterized by providing a specific kind of multi-party computation (a three-server model), that is also known as 3PC. This, however, does not mean that the computation cannot be done by more than 3 servers. It just means that, at most, it can tolerate one corrupted server, therefore supporting an arbitrary number of clients. The main accomplishment of ABY^3 was being able to switch between three modes: Arithmetic, Binary and Yao's secret sharing in a three-server model (ABY^3), while also ensuring privacy against malicious adversaries (not correctness). Like other PPML frameworks, there is a training and a testing phase. In the former, the servers perform a training procedure on the encrypted data they receive from the clients, while in the latter, they release a prediction based on the encrypted training model. It is worth noting that there is still a huge gap between a plain-text training and a privacy-preserving setting training, but 3PC protocols manage to reduce this. In a training scenario, while executing in semi-honest settings (ABY^3 allows us to run in both semi-honest settings or malicious ones), the protocol accomplishes a running time in order of 270$\times$ faster than the previous best 2PC protocol for a CNN model [57] and it still requires less rounds of communication (Chameleon [57] considers a similar security model with an honest majority like ABY^3 [44], but for 2PC, instead of 3PC).

2.4.3.2 SecureNN [64]

SecureNN [64], like ABY^3 [44], is a protocol that provides 3PC for multiple neural networks, supporting some building blocks, such as matrix multiplication, convolution, ReLU and Maxpool. This framework is capable of not only achieving semi-honest security (satisfying both correctness and privacy requirements) against a passive adversary, but also guaranteeing privacy against a malicious corruption, ensuring that this server cannot learn anything from the legitimate clients, even if it diverges arbitrarily from the protocol specification. However, considering the fact that there are 3 servers who are trying to securely compute a function over the clients' inputs while keeping them private, if one of those servers is corrupt, then there are only 2 honest servers left (honest majority) and, in practice, that leads to secure point-to-point channels between all parties and other expensive computational assumptions under this scenario. Firstly, the data owners send secret shares of their input to the 3 servers. Then, these servers run an algorithm to train the neural network over the aggregated data and produce a trained model that will be applied in the subsequent inference phase. Training-wise, Wagh et al. [64] were able to attain $> 99\%$

prediction precision with CNNs in the MNIST dataset [39], that is commonly used to train NNs, especially for image recognition systems. Regarding secure inference, they also compared their protocol to prior work in 2PC [33, 40, 45, 57] and concluded a large margin of improvement both on the throughput and on the communication in LAN and WAN settings. According to the authors, this is probably due to the exclusion of oblivious transfer and garbled circuits in this implementation.

2.4.3.3 FLASH [15]

FLASH [15] is another solution for PPML via MPC, that actually acts as a robust and efficient 4PC protocol, since it allows 4 parties to tolerate one malicious adversary (honest majority), thus providing the strongest security assumption (Guaranteed Output Delivery), which means that at the end of a computation all parties obtain the output regardless of the behaviour of a particular adversary. If we consider a scenario where machine learning acts as a service (MLaaS) and we want to provide efficient queries for a specific client, then we shall ensure that in the presence of a malicious adversary as one of the parties, the protocol must not abort for all the legitimate client queries, otherwise this will result in both financial and client trust loss. There are some advantages in moving from a 3PC setting to a 4PC one. Firstly, there is no need for an expensive secure channel, unlike ABY^3 and SecureNN [44, 64]. Besides, with 4 parties it is easier to reach an agreement with each message that is exchanged, since it only takes a majority rule over the 3 assumed honest parties, something that we can't easily satisfy in 3PC. Considering that one of the goals of the authors was to create a robust and efficient 4PC protocol with Guaranteed Output Delivery, they had to avoid asymmetric cryptography primitives and focused their efforts on developing a new secret sharing scheme that they called *mirrored-sharing*, a scheme that basically allows two sets of disjoint parties (evaluators and verifiers) to cooperatively compute and execute the due arithmetic or binary operations in a single execution. Byali et al. [15] also knew that a framework developed over fields would lead to increased latency, so they chose to build their protocol over rings, which are able to support fast and efficient arithmetic operations, therefore achieving a better performance. The authors employed several techniques in order to attain good efficiency for machine learning, namely: dot product, truncation, MSB-extraction, bit conversion and insertion. Due to these primitives, FLASH accomplishes substantially better throughput results compared to the previous state-of-the-art, ABY^3 [44], for Deep Neural Networks (DNN) and Binarized Neural Networks (BNN), both in LAN and WAN settings.

2.4.3.4 TF Encrypted [21]

TF Encrypted [21] is an open-source framework built on top of TensorFlow [4] and Keras [2] that implements multi-party computation over neural networks. First, the neural networks can be trained via these frameworks either on a plain-text training dataset or a private one. Following that, they run private inferences on the remaining testing dataset, where the prediction inputs and the model weights are kept private. One aspect that distinguishes TF Encrypted from most ML based MPC protocols is that it provides integration with other machine learning frameworks, such as TensorFlow and Keras, which are widely adopted and allow TF Encrypted to be easily integrated into them, supporting high level APIs, that soften the user experience, and abstract most of the underlying complexity. Security-wise, TF Encrypted assumes a semi-honest adversary model for 3PC and relies on additive secret sharing primitives and secure point-to-point channels to employ that desired security.

2.4.3.5 CrypTen [38]

CrypTen [38] is a PPML framework built on top of PyTorch [3] that focuses on implementing MPC over neural networks and, like TF Encrypted [21], tries to encourage both ML practitioners and researchers to adopt it by providing a high level API with user-friendly and readable operations. It first leverages a Tensor object from PyTorch and makes it a CrypTensor, by encrypting it. This is, incidentally, a design principle of CrypTen, since it acts as a machine learning first API and therefore it is closely linked to ML features, such as automatic differentiation. Unlike other protocols that we've reviewed so far, it actually supports computations in the GPU, thus relieving the CPU load and making private training and inference more efficient. It functions under a semi-honest threat model and provides MPC for an *arbitrary number of parties*, implementing arithmetic and binary secret sharing. While the former is more commonly used in matrix multiplications and convolutions, the latter is generally applied in the evaluation of popular activation functions, such as ReLU. So far, the framework results are also promising, e.g., the authors showed that two parties were able to privately classify an image in 2-3 seconds and were also able to make a secure phoneme prediction for 16kHz speech recordings faster than real-time.

2.4.3.6 Comparison and Discussion

Table 2.2 summarizes some of the main differences between machine learning based MPC frameworks. "Open-source" denotes if the code is available for everyone to use it. "Supports ML frameworks" alludes to the integration with machine learning frameworks, e.g., TensorFlow and PyTorch. "Supports CNNs" simply refers to whether the system provides support to Convolutional Neural Networks or not. "Supports GPUs" indicates if the system is able to off-load computations to the GPU. "Semi-honest security" and "Malicious security" distinguish systems who can ensure both correctness and privacy for semi-honest and malicious adversaries, respectively and, finally, "Parties supported" specifies the type of MPC and server-model of the frameworks.

After studying some of the most relevant ML based MPC frameworks in the literature, we've come to the conclusion that we should employ either TF Encrypted [21] or CrypTen [38] in the context of our work, mainly due to their usability, high-level abstractions and their integration with other machine learning frameworks, such as TensorFlow and PyTorch, that could also lead us to a more readable and accessible code. In our case, we'll have to cope with private training and inference on a CNN similar to DeepCorr [47], so by choosing TF Encrypted, it becomes desirable to inherit some of the same libraries used to develop DeepCorr (TensorFlow). In the future, assuming the development using PyTorch with CrypTen, we can also probably guarantee other interesting properties such as hardware acceleration (with GPU computations), as well as MPC for an arbitrary number of parties.

Table 2.2: Simple comparison between machine learning based MPC frameworks

Features	ABY^3	SecureNN	FLASH	TF Encrypted	CrypTen
Open-source	✓	✓	✓	✓	✓
Supports ML frameworks	✗	✗	✗	✓	✓
Supports CNNs	✓	✓	✓	✓	✓
Supports GPUs	✗	✗	✗	✗	✓
Semi-honest security	✓	✓	✓	✓	✓
Malicious security	✗	✗	✓	✗	✗
Parties supported	3PC	3PC	4PC	3PC	Arbitrary

2.5 Differential Privacy

In this last section of the Background and Related Work chapter, we will discuss another privacy-preserving technique: differential privacy. Since we had not initially planned to explore this topic, its exposition will most likely not be as thorough and complete as the ones of previous topics. Nevertheless, we will try to convey a general idea of the algorithm, as well as its current application in machine learning.

2.5.1 Definition

Differential privacy (DP) is a strong mathematical enforcement of privacy [67]. An algorithm is considered differentially private if by looking at the output, no one is able to tell whether any individual sample was included or removed from the original dataset. Essentially, the guarantee of a differentially private operation is that its behaviour does not or hardly changes when a single or small set of samples joins or leaves the training set. Therefore, the main goal of these algorithms is to ensure that sensitive information in the dataset is not leaked. This can only be accomplished by the introduction of a thoroughly tuned value of noise during the computation, which will hamper any attempt from an attacker to expose the private information.
There is, however, a trade-off between the privacy obtained and accuracy, meaning that the addition of noise erodes the accuracy of the computation. It is then convenient that we find an admissible middle ground, where the solution preserves enough data privacy without compromising too much the accuracy of the model.

Two values that must always be taken into account when computing the privacy guarantees of a DP algorithm are:

- ***Epsilon*** **(ϵ):** It measures the strength of the privacy guarantees and essentially represents the probability of change in the model output by including or removing a single training sample from the dataset. Hence, we want this value to be small, so the DP algorithm can be regarded as privately strong. Generally, higher noise produces a smaller ϵ.

- ***Delta*** **(δ):** It bounds the probability of an arbitrary change in the behaviour of the model, i.e., of the privacy guarantee not persisting. It is usually set to less than the inverse of the dataset size.

2.5.2 Machine Learning Practical Application

In the 21st century, large data breaches are becoming a norm. Most of them occur due to negligent security practices of some IT companies, which, occasionally, poorly protect their customers' private data. It is extremely relevant that machine learning and security researchers alike combine their efforts so that these data breaches become less frequent, since the significance of big data requires a reconsideration of privacy. An emerging and viable answer to this issue is differential privacy.

While machine learning generally deals with composing suitable information from raw data, differential privacy, on the other hand, is concerned with protecting and preserving sensitive details. In 2014, Ji et. al [32] tried to understand how it would be possible to foment a mutual and favourable relation between these two areas. They specifically focused on approaches that could convert machine learning operations into their differentially private variants and suggested that the objectives of differential privacy and model generalization could indeed be compatible, i.e., training a model with DP could still guarantee a reasonable validation accuracy when evaluating new unseen data. This idea is established as one of the first steps towards feasible machine learning frameworks that support differential privacy primitives.

For instance, Private Aggregation of Teacher Ensembles or PATE [52] is a DP approach for ML, that consists in splitting sensitive data into disjoint training sets, where a "teacher" classifier independently trains each of these data partitions, without any guarantees of privacy. In order to make inferences that respect privacy, PATE adds random noise while aggregating the predictions made individually by each teacher to form a single common prediction. Then, there is a "student" entity, which is trained by transferring knowledge acquired by the teacher ensemble in a privacy-preserving way. The noisy aggregation mechanism is a critical step for this, since it transforms the unlabeled data the student receives into private labels, which will be subsequently used to train a model, that can be deployed to respond to any inference queries from users.

TensorFlow Privacy [5], on the other hand, is a framework that implements a distinct differentially private algorithm (DP-SGD) [6]. DP-SGD, contrary to PATE, achieves PPML by assuming less about the machine learning task and instead introduces additional exclusive DP parameters to the training process. In TF Privacy, DP is integrated by the application of differentially private optimizers, since this is when the majority of the computations occur. In the first stage, the gradients are calculated and then they are clipped according to the `l2_norm_clip` parameter, which will be further explained in Section 4.3.1. In the second stage, the random `noise_multiplier`, also detailed in Section 4.3.1, is sampled and then added to the gradients that were clipped in the last phase. These two modifications constitute the most significant difference between the vanilla SGD optimizer and its differentially private variant (DP-SGD). Fortunately, TF Privacy also supports a DP version of Adam, which is quite useful, since the standard version of the optimizer was applied in the training algorithm for the neural networks of Torpedo's code. The intuition, however, is similar to the one described above for DP-SGD. After training the model, we can measure the achieved privacy guarantee. A small privacy budget is typically desired, once that corresponds to stronger privacy properties. The framework includes the libraries that allow the computation of this value, which can be expressed by two parameters that were already explained before: *epsilon* and *delta*. Further details about this specific step will be discussed in Section 4.6.

Chapter 3

Design of the Solution

In this chapter, we will explore the various steps that led to the conception of our planned solution.

3.1 General Approach

To develop the solution we have come up with, we analysed some steps that would be mandatory for us in order to keep the data private during Torpedo's matching phase. Therefore, we start by listing some fundamental design goals that we took into consideration (Section 3.1.1), then we represent our ideas into atomic operations, firstly considering the plain-text use-case (Section 3.1.2) and the private one afterwards (Section 3.1.3) and finally we also present an architectural overview (Section 3.1.4).

3.1.1 Design Goals

While designing a privacy-preserving version of Torpedo, we had some of the following main design goals in mind:

- Guarantee that the ISPs never possess the entirety of the model, but only shares of it.
- Ensure that the operator is able to perform private queries on encrypted data without having the model.
- Develop a modular approach between the "server" (master and ISPs) and the "client" (operator).
- Prioritize a low false positive rate for the correlated pairs.
- Achieve a reasonable performance for both the training and inference phases.

3.1.2 Plain-text Operations

The first phase is the data pre-processing phase. This operation can be seen as the filtering and standardization of the `data` that comes from the network traffic captures in *.csv* files (the `datasets` themselves) according to user-defined `parameters`.

```
preProcessing(datasets,parameters)->data
```

Subsequently, we have the training phase, the previously pre-processed `data` is trained by a user-defined training `algorithm` (which can vary in the `parameters` it receives). The `model` is the resulting output from the training phase.

```
training(data,algorithm,parameters)->model
```

Finally, we have the inference or testing phase. The objective here is to obtain an output `result` from the inference `query` based on the `model` that was generated in the last step.

```
testing(model,query)->result
```

3.1.3 Private Operations

All the previous operations were applicable for a non-secure version of Torpedo. However, in our work proposal, the objective was to guarantee that the matching phase of the attack preserves data privacy. As a result, in this section, we will specify the subtle changes that have to be made so that the attack can function in a privacy-preserving setting.

Before exposing the new modifications, we'll define two primitives that will be essential for the comprehension of private operations. Those primitives are directly related to secret sharing and without its application, it would be impossible for us to translate and convert between shares that can be computed by distinct intervening parties, thus preserving data privacy.

The first primitive is `share` and it can be seen as the the application of a secret sharing `scheme` to `data`, resulting in the distribution of `secret_shares` between multiple parties.

```
share(data,scheme)->secret_shares
```

The reverse operation is `reconstruct` and corresponds to the secret reconstruction of `data` between a majority of parties that follow a certain secret sharing `scheme` upon combining their `secret_shares`.

```
reconstruct(secret_shares,scheme)->data
```

With the definition of these primitives, the training phase has to be adjusted. Although the atomic operations are similar to the non-private counterparts, there has to be added an intermediate step between the end of training and the start of testing. That is the inclusion of a secret sharing `scheme` that secures the `model` and distributes `shared_model` weights between the different participants in the computations (the ISPs in our particular case).

```
training(data,algorithm,parameters)->model

share(model,scheme)->shared_model
```

In addition to the secret shared model that precedes the inference phase, we now have to consider the secret sharing of the `query`, that will break the query into `shared_query`.

```
share(query,scheme)->shared_query
```

After this step, we can safely process the `shared_query` in the `shared_model` and produce the `shared_result`.

```
testing(shared_model,shared_query)->shared_result
```

Finally, we just have to recover the `result`, by combining the majority of the shares of `shared_result` according to a certain secret sharing `scheme`.

```
reconstruct(shared_result,scheme)->result
```

3.1.4 Architecture Overview

As we can observe in Figure 3.1, we established a system architecture where the master is the model owner, meaning that it is the agent responsible for pre-processing the datasets and training the model. This master can represent an abstraction for a more powerful entity, such as an AS that controls the ISP nodes in the network. After this initial step, the model is secret shared between the three servers according to the specification of the MPC protocol we choose. The model weights previously trained by the master are broken into shares and distributed by the three ISPs. This ensures that none of the ISPs is able to recover the model by itself. After this distribution, the servers can process inference queries sent by an external client (the operator, in our case), that also has to pre-process the datasets before secret sharing its inference query and submitting it directly to the ISP servers. Finally, the servers can safely run the shares of the query in the shares of the model they have, producing a result that will be recovered between the majority of the parties before being sent back to the operator. This output corresponds to values that sit between 0 and 1. While values close to 0 resemble no correlation to an OS session, values near 1 indicate that the session may indeed be correlated.

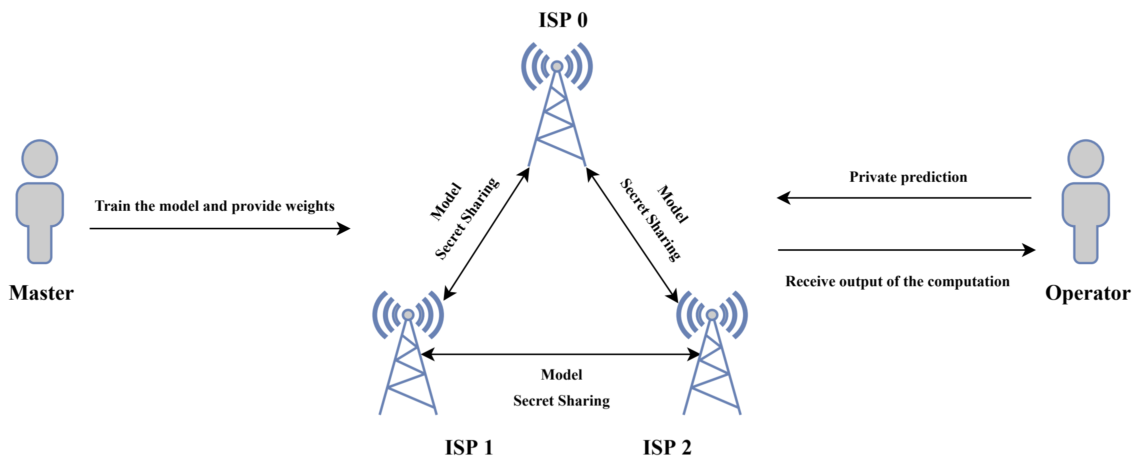


Figure 3.1: General architecture of the privacy-preserving Torpedo solution

3.2 Security and Trade-offs

In this section, we'll discuss the theoretical impact of adding multi-party computation (Section 3.2.1) and differential privacy (Section 3.2.2) operations to the domain of our problem, as well as some of the resulting trade-offs.

3.2.1 Multi-Party Computation Impact

As we had already previously defined, ensuring a privacy-preserving solution for Torpedo would require the application of a secure multi-party computation framework. TF Encrypted was the chosen option, not only because it could be easily integrated with TensorFlow, having several high-level abstractions that make the programmer's job more accessible, but also because it appeared to be quite well documented. Furthermore, we could also take advantage of the fact that Torpedo's code leveraged the TensorFlow library, so TF Encrypted looked like a legitimate choice, all the more so because of its easy integration and machine learning oriented design principle. It essentially acts as an API that incorporates TensorFlow primitives and allows the employment of a supported back-end MPC protocol, responsible for keeping the sensitive data private during our predefined computations. For this instance, we chose a protocol that we had also previously reviewed: SecureNN [64].

As a MPC protocol, SecureNN works well with simple additions. However, multiplications require more computational power and consequently increase both training and classification times themselves. As we have studied in Section 2.2, these operations represent most of the computations involved in a neural network.

In addition to that, we also have to consider the geographical locations of the different parties present in the computation (master, ISPs and operator). As such, the latency of these computations in a realistic scenario is a factor that we cannot certainly undervalue.

An approach to mitigate this inconvenience is to increase the amount of data to be processed in the computations, e.g., applying batching techniques. In that way, instead of processing one pair in each individual inference, we can process a larger number, sacrificing the CPU resources, but mitigating the impact of message delays in the network.

The existing overhead of MPC can also be attenuated by building a model that is able to achieve a reasonable performance at the cost of less precision. This is not the optimal solution, but instead an acceptable trade-off given our project requirements. However, we can always assume that an operator is able to perform multiple correlations over time to ensure more confidence in the obtained results.

3.2.2 Differential Privacy Impact

On top of MPC, our solution also contemplates a preliminary extension for a Differential Privacy or DP setting, that we had not initially planned. DP is basically a framework that measures the privacy guarantees that a certain algorithm offers. Learning with differential privacy provides provable guarantees of privacy, which mitigates the risk of sensitive training data exposure. Essentially, by training our model with differential privacy, we can ensure that the model is not memorizing sensitive information about the training set. Otherwise, an adversary could infer private information by just querying the deployed model [62].

Although our attack does not necessarily require the application of DP techniques in order to succeed, this can be seen as a potential topic of further investigation in the field of privacy-preserving machine learning and could also be useful for a scenario where Torpedo acts as a deanonymization framework aids criminal investigations instead of a traffic correlation attack against the Tor network. Regardless of that, if we intend to conceive an attack with higher capabilities, we can also assume that the master entity,

responsible for managing the ISPs present in the computations, preserves data privacy by responsibly training the models on private data, thus providing stronger privacy guarantees, i.e., using differential privacy.

With that in mind, we tried to develop a solution for that use case. Our implementation was far from perfect, as we we will describe in Section 5.5, but we still managed to reach some significant conclusions regarding the impact of adding this module to our solution.

Although the training time of the model for this setting did not increase that much, it was still higher compared to the MPC solution. The most notable drawback, however, was the model accuracy that we obtained, which was impressively lower than we expected. As a result, so were the predictions that we got. In Section 5.5, we will attest these statements with a performance and precision evaluation.

Chapter 4

Implementation

In this chapter, we will cover the steps that led to the implementation of our solution. We will first expose the code structure of the project (Section 4.1). Then, we will explain the data pre-processing phase (Section 4.2). Only afterwards, we will discuss the training phase (Sections 4.3 - 4.6) as well as the secret sharing of the model and the inference phase (Section 4.7). Finally, we will summarize the workflow of the project (Section 4.8) and detail some frameworks that were used in those programs (Section 4.9).

4.1 Project Structure

Firstly, there is a main program for the ranking stage, which is `server-ranking.py` and another main program for the correlation stage, this one being called `server-correlation.py`. The operator has `client.py`. These programs have 305, 317 and 145 lines of code, respectively and the operations that will be described in the following sections translate into their code. What is also essential to the functioning of the solution are the server scripts, that allow the ISPs to participate in the distributed computations and act as TF Encrypted parties. Each script can be represented as follows, where `player` corresponds to the name of the party (e.g., "server0") and `config` is a file that has the hostmap configuration detailed.

```
python3 -m tf_encrypted.player {player} --config {config}
```

As a configuration file, we can specify the IP addresses of the participants and the ports from where they communicate with each other.

```
{
    "server0": "10.154.0.3:4440",
    "server1": "10.164.0.4:4440",
    "server2": "10.132.0.10:4440",
    "operator": "10.186.0.2:4440"
}
```

4.2 Data Pre-Processing

Before we train the model that will be responsible for running the operator's private inference queries, we have to load the data that comes from *.csv* files and pre-process it. The `generateDataset` function

is, to a large extent, extremely identical to the non-private counterpart of Torpedo, but we will still briefly describe how it works and detail the subtle tweaks that we made. Firstly, we have to create two *numpy* arrays (`l2s` and `labels`) filled with zeros, which, depending on the datasets that we are loading, will correspond to the training data and training labels or testing data and testing labels, respectively.

```
Shape of l2s: len(pairsFoldersInput)*(negative_samples+1),8,flow_size,1
Shape of labels: len(pairsFoldersInput)*(negative_samples+1),1
```

While the shape of `l2s` matches, at least for the most part, the input shape of which the neural networks process the data, the shape of `labels` partially matches the output layers of the neural network. `pairsFoldersInput` is the dataset that we load into memory and the length of that dataset represents its number of samples. `negative_samples` is a user-defined parameter that was originally 2, but then we changed to 1, which means that we chose to work with a perfectly balanced dataset, that starts by having as many negative samples as positive samples. If this value was, for instance, 2, then we would have 2 non-correlated pairs for each correlated pair based on the combination of different flow pairs of size `flow_size`. The value `8` that appears in the shape of `l2s` had been previously defined as a good value for the first convolution layer of the neural network and its meaning will be discussed ahead, in Section 4.5.
Each flow pair gets assigned with a certain ingress and egress traffic times and sizes, both for the client and server sides. This information is then filled in the first 8 indexes of the `l2s` array according to the description of Section 4.5, while the `labels` array is assigned with 1 (correlated pair). We repeat the exact same process for the non-correlated pairs, the only difference being that this time the `labels` array gets assigned with the value 0. Finally, we return both `l2s` and `labels` as *numpy* arrays.
We decided to generate these arrays for 3200 training samples and 2144 testing samples (60% training and 40% testing). The greatest common divisor (GDC) of these values is 32, which seemed like an appropriate number in case we wanted to apply batching with powers of two while training the data, without having to resort to padding techniques.

4.3 Training Parameters

In this short section, we will introduce some training parameters that are crucial to comprehend the work that is done in the neural networks for both the ranking and correlation stages.

- **Epochs:** An epoch is essentially a cycle in the training of a neural network. The data is used exactly once, both for the forward and backpropagation. Each epoch can have several iterations, depending on the number of training samples and the batch size we are using (e.g, if we have 1000 training samples and the batch size is 100, then each epoch is composed by 10 iterations). At the end of the training process, after a certain number of epochs, we expect a neural network to converge.

- **Batch Size:** Unlike the number of epochs, that define the amount of times the training algorithm goes through the entire dataset, the batch size is a training parameter that controls the number of training samples to work through before the model gets updated. Generally, $1 \leq$ *batch size* $\leq$ *total*

training samples. The dataset has to divide evenly by this number in order for the training to function properly. We can either remove some samples from the dataset (our solution) or change the batch size in such a way that the number of samples in the dataset divides evenly by it. Typically, the higher the batch size, the faster the training process, since each epoch has less iterations.

- **Loss Function:** The loss function is used to calculate the absolute value of the difference between the expected output and the actual prediction. Then, this result is applied to derive the gradients in order to update the weights of the model. In general, the higher the loss, the worse our predictions, since this value should diminish at the end of each epoch, while the model accuracy increases. A low loss means that the expected values and actual predictions are identical, which is one of the main goals when training a model.

- **Learning Rate:** The learning rate defines how much the model changes every time the weights are updated. It can be represented as $0 < \textit{learning rate} < 1$. A small value results in a longer training time, whereas a higher value may not ensure a smooth training process, since the weights may be extremely irregular due to the faster training. Therefore, to balance this limitation, we generally compensate with a higher number of epochs for a small learning rate (so we can notice the small changes that are made in each update) and fewer for a higher learning rate (so the weights of the model do not become unstable as quickly).

- **Optimizer:** The optimizer is an algorithm that is responsible for updating the weights of the model at a certain learning rate. Its function is to find the right value for the weights when minimizing the loss and improving the overall accuracy. Therefore, the choice of the algorithm is one of the most relevant factors to consider before training the model.

- **Dropout Probability:** This parameter represents the probability of a given layer output neuron being excluded or dropped during the training stage. It is generally applied if we intend to reduce overfitting and improve the generalization of the obtained model. Overfitting can occur when the neural network memorizes the training dataset, but it is not able to generalize those same results if it tries to evaluate new data. The main idea is to randomly drop neurons from the neural network, thus preventing the network from co-adapting too much and making the model more robust.

- **Activation Function:** These functions determine how the sum of the weights is transformed into an output node at the several layers of the network. The choice of these functions has a great impact in the performance of the neural network as well as the values that will appear in the output nodes at each layer. Typically, all the hidden layers apply the same activation function for all the hidden nodes, but the output layer can use a different function for their nodes in specific. This last choice is especially important, since the chosen function will define the type of prediction expected by the model.

4.3.1 DP-Exclusive Training Parameters

Now, we will briefly expose and describe some training parameters, that are exclusive to the MPC+DP setting.

- **L2 Norm Clip:** It is the maximum Euclidean norm for each individual gradient that is calculated on an individual training sample from a batch. This parameter is used to restrict the optimizer's sensitivity to individual samples.

- **Noise Multiplier:** It is used to bound the amount of noise that is sampled and added to gradients before they are applied to the optimizer. Generally, more noise means better privacy, although this also affects the performance of the model to some extent. It is then useful for studying the effect of adding differential privacy to the model.

- **Microbatch Size:** The microbatches are fundamentally a subset of batches. This new level of granularity is needed, since clipping gradients per sample may deteriorate the performance in a MPC+DP setting, so they have to be instead clipped on microbatches. For instance, let's suppose we have a batch of 128 training samples. Rather than clipping all these 128 samples individually, we can split them by 32 microbatches (e.g.) of 4 training samples. This ensures a higher level of parallelism, so this parameter allows us to achieve a better performance when its value is smaller than the batch size. Typically, $1 \leq \textit{microbatch size} \leq \textit{batch size} \wedge \textit{batch size} \mod \textit{microbatch size} = 0$.

4.4 Neural Network of the Privacy-Preserving Ranking Stage

This stage is constituted by a simple neural network whose values are presented in Table 4.1. Similarly to the non-private counterpart of Torpedo, this neural network is also fed with three inputs: *the packets in forward direction* (the difference between the number of packets sent by the first flow and received by the second), *the packets in reverse direction* (the difference between the number of packets sent by the second flow and received by the first) and *the time of capture* (the time difference between the first packet of each flow).

It is only composed by three fully connected hidden layers due to efficiency requirements, since this stage is responsible for performing a fast scan of the flows before reaching the next stage, where convolution comes in and the combinatorial nature of the pairs is a relevant factor to take into account if we intend to ensure an acceptable performance. The output is a value between 0 and 1, that represents the probability of the flows being correlated. Higher values than a user defined threshold constitute pairs that are most likely to be correlated, which means that these pairs can then be selected as candidates and be fed into the next stage.

Table 4.1: Neural network architecture of the privacy-preserving ranking stage

Layer	Number of neurons	Activation function
Fully Connected Layer 1	300	ReLU
Fully Connected Layer 2	80	ReLU
Fully Connected Layer 3	10	ReLU
Output Layer	1	Sigmoid

4.5 Neural Network of the Privacy-Preserving Correlation Stage

When we define the neural network, we start by specifying the number of epochs we want to train our model. 15 epochs seemed like an appropriate number, since a higher value did not contribute to a better testing accuracy, thus only increasing the training time. The main difference regarding the model of the ranking stage is the fact that we include two convolution and max pooling layers.

Therefore, the neural network of this stage is substantially more complex than in the previous stage, as it shows in Table 4.2. The input layer received by the network is composed by 8 vectors with the specified training dataset size. 4 vectors are tied to the first flow, while the other 4 are related to the second one. Half of these vectors account for temporal features of the flows and the other half for size features (both for ingress and egress traffic). In this process, the purpose of the convolution layers is to essentially recognize the observed patterns of the input data that is fed to the network. Subsequently, the fully connected layers, similarly to the last stage, are responsible for creating relations and reaching conclusions between the observed data.

One of our main limitations is the fact that TF Encrypted does not allow the use of rectangular shapes in the kernel, only square ones. For example, in the non-private counterpart of Torpedo, the first convolution layer had a kernel of size (2,30), meaning that there were being processed 2 input vectors at a time in the kernel, each one observing 30 packets at a time (in both directions). With TF Encrypted, however, we had to either increase the first value or decrease the second. If we increased the first value to a shape of (30, 30), our network would try to analyse 30 features, when we are just interested in timing and size features. Therefore, it is preferable to decrease the second value to a shape of (2,2), even though the accuracy may worsen, since we are only processing two packets from both flows at a time. We also repeat this process for the second convolution layer, but with a kernel of shape (4,4).

The second convolution layer processes mixes the inter-packet arrival times and size features of the packets hence the stride of (4,4), opposed to the previous (2,2) stride, that only analysed one type of features at each window. Thus, this layer is even more complex than the first one because it is trying to identify patterns in both features of the packets. However, by increasing the stride, we are also decreasing the amount of times we perform the expensive operation of convolution, which leads to better performance.

Only after this, we flatten the network and proceed to a stage of three fully connected layers similar to the ranking stage, that attempt to reach some reasoning about the observed patterns. Finally, like the previous stage, we end up returning a number between 0 and 1, due to the binary nature of the intended result (correlated or non-correlated), which we can obtain with the Sigmoid function in the output layer.

Table 4.2: Neural network architecture of the privacy-preserving correlation stage

Layer	Kernel number	Kernel size	Stride	Window Size	Number of neurons	Activation function
CL 1	512	[2,2]	[2,2]	—	—	ReLU
MPL 1	—	—	[1,1]	[1,5]	—	—
CL 2	256	[4,4]	[4,4]	—	—	ReLU
MPL 2	—	—	[1,1]	[1,5]	—	—
FCL 1	—	—	—	—	1024	ReLU
FCL 2	—	—	—	—	128	ReLU
FCL 3	—	—	—	—	32	ReLU
OL	—	—	—	—	1	Sigmoid

CL - Convolution Layer
MPL - Max Pooling Layer
FCL - Fully Connected Layer
OL - Output Layer

4.6 Training without DP *vs* Training with DP

There are some subtle changes that we have to apply depending on the setting we are training the model on.

For a MPC setting, we just train the network in a particularly effortless way. We assign an `Adam` optimizer with a `learning_rate` of `0.0001` and compile the model with that same optimizer for a `sigmoid_cross_entropy_with_logits` loss function, which is suitable for the classification of soft binary labels (probabilities between 0 and 1). We resort to the `Adam` optimizer, as it is known to scale well for problems that presume large amounts of data and it generally converges faster than other optimizers with little tuning involved [37]. We chose a `learning_rate` of `0.0001`, since higher values may have converged faster, but were not able to find the best results we were looking for, so we traded-off a longer training time for a more accurate process.

For a MPC+DP setting, we employ a `DPKerasAdamOptimizer`, which is the equivalent of the `Adam` optimizer but as a differentially private variant. This optimizer requires more parameters than the vanilla `Adam` one. Some of these parameters include `l2_norm_clip`, `noise_multiplier` and `num_microbatches`, besides the standard `learning_rate`. According to the TensorFlow Privacy GitHub repository [5], the `l2_norm_clip` should represent the percentile of the gradient we expect at each microbatch, so we considered `0.5` to be appropriate. The `noise_multiplier` is what determines the proportion of noise we want to add to the dataset. More noise achieves better privacy guarantees, but it also decreases the utility of the model. So, we generally chose the recommended minimum value of `0.3`. Unfortunately, in our tests, the `num_microbatches` parameter could not be higher than 1, otherwise it would result in a `ValueError`, even though we tested it with values that were evenly divisible by the batch size. As a consequence, we had to keep this parameter assigned to `1` when training the model with DP. The loss function, on the other hand, was similar to the first normal public training one, but instead of computing the vector of a mean over the batch, like we did before with a `reduce_mean` over the `sigmoid_cross_entropy_with_logits` function, we compute the vector per sample (as a tensor), thus taking out the mean.

Depending on the setting, we train the training input data (`l2s_train`) against the training target data (`labels_train`), as well as the validation data, composed by `l2s_test` and `labels_test` with variable batch sizes and for a certain number of epochs, which ends up generating a model. In our

experiments, the number of epochs for the correlation stage was 15 (with or without DP), but it had to be increased from 10 to 30 in the ranking stage when including differential privacy. At the end of each epoch, we save the weights of the model at a checkpoint in persistent memory, so that we can efficiently load these weights in the future, therefore skipping the training process.

Additionally, if we are using DP, we also have to compute `epsilon` and `delta`, which are values that measure the privacy guarantees of our model. `epsilon` determines the probability of a given model output change based on including or removing a single training sample. Therefore, we want this value to be as small as possible. `delta`, on the other hand, bounds the probability of arbitrary change in the behaviour of our model, i.e., the probability of the privacy guarantee not holding up. Generally, we set this to a very small number, at least less than the inverse of the training dataset. The intuition behind measuring the privacy guarantees of a model is to ensure that if a single training sample does not affect the outcome of the learning process, then the information contained in that training sample cannot be memorized, thus respecting the privacy of that individual point. Besides, getting to know what level of differential privacy was achieved also allows us to consider the drop in utility that is often observed when switching from a standard to a differentially private setting. In order to compute `epsilon`, we first have to identify all the parameters that are relevant to measuring the potential privacy loss induced by the training process. These are the `noise_multiplier`, the `sampling_probability` (i.e., the probability of an individual training point being included in a batch) and the number of `steps` the optimizer takes over the training data.

TF Privacy provides two methods (`compute_rdp` and `get_privacy_spent`) to derive the privacy guarantees achieved from these last three parameters. First, we define a list of `orders`, at which the Rényi divergence will be computed. `compute_rdp` then returns the Rényi Differential Privacy (RDP) [43] achieved by the sampling and subsequent Gaussian noise addition applied to the gradients in DP-SGD, for each of these orders. Finally, the method `get_privacy_spent` computes the best `epsilon` for a given `target_delta` by taking the minimum value over the orders.

4.7 Secure Model Serving and Private Inferences

After the training process, the master entity has to connect itself to the three ISP servers and secret share the model weights between them. The first step is the setup of a `tfe.serving.QueueServer` object, which will launch a FIFO serving queue, allowing the TFE servers to accept prediction requests on the secured secret shared model from an external entity (the operator). This queue has to know the `input_shape` and `output_shape` it will process, i.e., the shapes of the input and output data, respectively. We then define a maximum number of requests (`num_steps`) that can be accepted by the TFE workers and finally run the server.

At the operator side, we start by pre-processing the data of the captures in the same way it was described in Section 4.2. The operator, however, does not possess the model, so it just has to connect itself to it before submitting private inferences. Therefore, it has to create a `tfe.serving.QueueClient` with the same attributes of the queue created at the master, i.e, equal input and output shapes. This queue will be in charge of secret sharing the plaintext input data before submitting the shares as a prediction request. When we run it, we insert the data into the queue, secret share it locally and finally submit these

shares to the ISPs (one query at a time) until it reaches the limit of requests. The ISPs respond to these inferences according to the shares they have as well as the MPC protocol specification. Ultimately, these shares are reconstructed at the operator itself and presented as the plaintext output.

4.8 Workflow Summary

To summarize the workflow of the implementation for both ranking and correlation stages:

1. Launch three ISP servers as three distinct TF Encrypted parties.
2. Load the training sets from persistent memory and pre-process the data as the master entity.
3. Depending on the settings:
    - 3.1 For a MPC setting, train the model according to the architecture of our neural network layers and public training parameters.
    - 3.2 For a MPC+DP setting, train the model according to the architecture of our neural network layers and DP training parameters.
4. Save the weights of the model in persistent memory, e.g., a checkpoint.
    - 4.1 Additionally, for a MPC+DP setting, compute the privacy spent of the model training.
5. Create a logical "server" that connects the master to the three ISPs and secret shares the model between them.
6. As the operator, the data also has to be pre-processed, but not trained, since this entity is oblivious to the model.
7. Create in the operator side an analogous "client" that connects itself to the already defined "server".
8. Submit a limited number of inference queries.
9. The ISPs process the queries based on the model shares they have.
10. At the end of each iteration, the operator reconstructs the final result.
11. It gradually receives the results of the queries until it reaches the maximum number of requests.

4.9 Frameworks

In this short section, we will list the main technologies that we used in order to develop this project as well as the corresponding versions for each one of them.

- Python 3.6.9
- TensorFlow 1.15.5
- Keras 2.2.4-tf

- NumPy 1.18.5
- Matplotlib 3.3.4
- TensorFlow Encrypted 0.7.0
- TensorFlow Privacy 0.0.1

Chapter 5

Results and Evaluation

In this chapter, we will expose the obtained results and evaluate the developed solution. Our main goal is to provide a privacy-preserving implementation for Torpedo that performs well and achieves acceptable precision with all the privacy constraints in mind, meeting the design goals we enumerated previously in Section 3.1.1. We will start by defining our methodology in Section 5.1, present the hardware and geographical settings of the machines in Section 5.2 and only after that we will expose the results for the ranking (Section 5.3) and correlation (Section 5.4) stages, respectively. Additionally, we will briefly analyse the performance for the DP extension in Section 5.5 and finally highlight some security key points for MPC and DP (Section 5.6).

5.1 Methodology

In this section, we will explain some of the metrics and methods that will be used to evaluate the solution we implemented.

Precision: To clear up any possible misinterpretation, it is already worth mentioning the difference between accuracy and precision. The former is a parameter that can be measured as soon as the model is being trained. It reflects how many times the model is correct. That is, a model with 80% accuracy gets, on average, 80 out of 100 predictions right. Precision, on the other hand, is associated with specific classes, i.e., true positives, false positives, true negatives and false negatives. We will apply two useful metrics to evaluate this parameter: *true positive rate (TPR)* and *false positive rate (FPR)*. While the first one alludes to the number of correlated pairs that are correctly labeled (*true positives*) over the total number of correlated pairs in the dataset (*true positives* + *false negatives*), the latter refers to the number of uncorrelated pairs that have been inaccurately placed as correlated (*false positives*) over the all the uncorrelated pairs present in the whole dataset (*false positives* + *true negatives*). One of our design goals was to minimize *FPR*, while maximizing *TPR*. The following formulas represent the relations described above.

$$TPR = \frac{TP}{TP + FN} \qquad\qquad FPR = \frac{FP}{FP + TN}$$

Performance: Performance measures the amount of time a system takes to process a certain task. In our given scenario, we will evaluate the performance of the privacy-preserving variant for Torpedo by measuring the amount of time it takes to train the model (*training time*) and also identify the correlated flow pairs from a specific batch of a testing dataset (*testing time*). Even though these training and testing times are highly influenced by the hardware capabilities, TF Encrypted does not support the use of a GPU for the training process, therefore our experiments will be bound by the CPU power and will not be able to speedup the training phase even when using a resourceful GPU. Besides, a realistic scenario simulating a safe environment will require TF Encrypted to be run in a distributed setting, invoking a `RemoteConfig` object. Accordingly, another relevant metric is to assess the impact of both latency and the geographical location among processes that participate in the secure computations. The round-trip time (RTT) of the messages exchanged between the parties will definitely vary depending on the location of these servers, so we can't ignore it in order to evaluate the performance of our solution.

Data partitioning and uncorrelated pairs: The data was divided into training and testing datasets. Approximately 60% of that data was used for training the model and the remaining 40% for testing it. The `negative_samples`, like previously mentioned, define the number of generated uncorrelated pairs per an actual correlated pair. By default we assigned this value to `1` for both ranking and correlation stages, since we wanted to have a perfectly balanced dataset (1 negative or uncorrelated sample for each positive one).

5.2 Benchmarking

In this short section, we will present two tables that detail the hardware specifications and geographical locations for each party that will be present in the experiments. It is also worth noting that these tests were conducted by deploying and running virtual machines in Google Compute Engine, an IaaS offered by Google Cloud Platform [1].

5.2.1 Hardware Specifications

Table 5.1: Settings for the hardware specifications of the machines

Setting	Machine	CPU	Number of VCPUs	Memory (RAM)
1 (Default)	Master	Intel Skylake	4	15 GB
	Operator	Intel Broadwell	2	7,5 GB
	ISPs	Intel Skylake	2	7,5 GB
2	Master	Intel Skylake	1	7,5 GB
	Operator	Intel Broadwell	2	7,5 GB
	ISPs	Intel Skylake	2	7,5 GB
3	Master	Intel Skylake	4	15 GB
	Operator	Intel Broadwell	2	7,5 GB
	ISPs	Intel Skylake	1	7,5 GB
4	Master	Intel Skylake	4	15 GB
	Operator	Intel Broadwell	1	7,5 GB
	ISPs	Intel Skylake	1	7,5 GB

5.2.2 Geographical locations

Table 5.2: Settings for the geographical locations of the machines

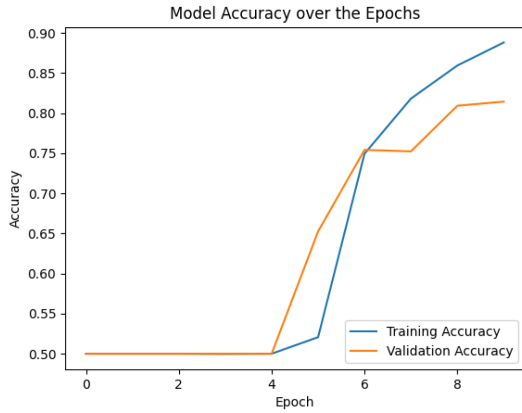
Setting	Machine	Location
1	Master	Localhost
	Operator	Localhost
	ISPs	Localhost
2 (Default)	Master	Finland
	Operator	Poland
	ISPs	Finland, Netherlands and Belgium
3	Master	Taiwan
	Operator	New Dehli
	ISPs	Singapura, Seoul and Sydney
4	Master	Iowa
	Operator	Northern Virginia
	ISPs	Montreal, São Paulo and Oregon
5	Master	Belgium
	Operator	Iowa
	ISPs	São Paulo, Zurich and Melbourne

5.3 Evaluation of the Privacy-Preserving Ranking Stage

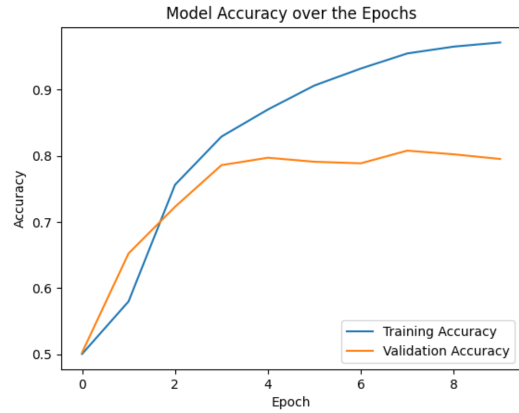
In this section, we will analyse and expose the results of some experiments for both the performance and precision metrics on the privacy-preserving ranking stage.

5.3.1 Accuracy

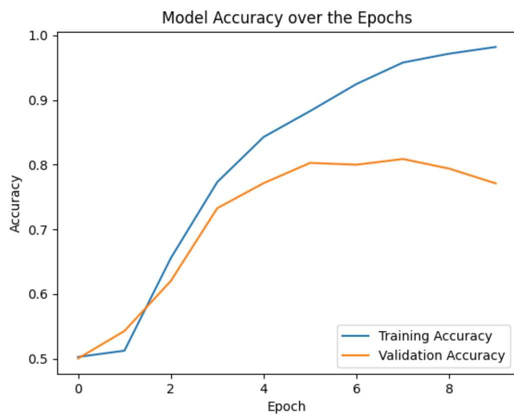
Plots from Figure 5.1 compare the training and validation model accuracy for different batch sizes, in a setting that does not contemplate differential privacy. As we can observe, the training accuracy generally reaches values close to 95% to 100%, while the validation accuracy sits around the 80% mark. Although this is still generally a good value, the difference between the training and validation accuracy can be explained due to a slight model overfitting, that was deemed mandatory in our case, because with less than 10 epochs the model could not learn the necessary features to provide an appropriate subsequent classification phase. A good solution would be to include Dropout layers, since it would diminish overfitting. However, this was not supported by TF Encrypted, when cloning the TensorFlow model to the TF Encrypted counterpart, so we had to accept this limitation.



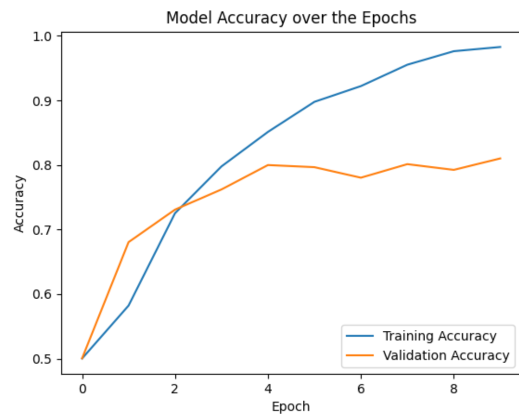
(a) Model accuracy over the epochs in the privacy-preserving ranking stage for a batch size = 2



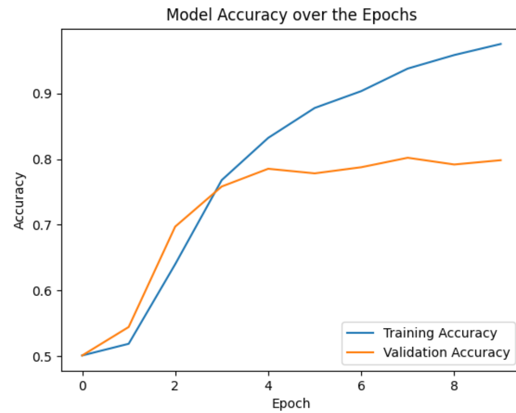
(b) Model accuracy over the epochs in the privacy-preserving ranking stage for a batch size = 4



(c) Model accuracy over the epochs in the privacy-preserving ranking stage for a batch size = 8



(d) Model accuracy over the epochs in the privacy-preserving ranking stage for a batch size = 16

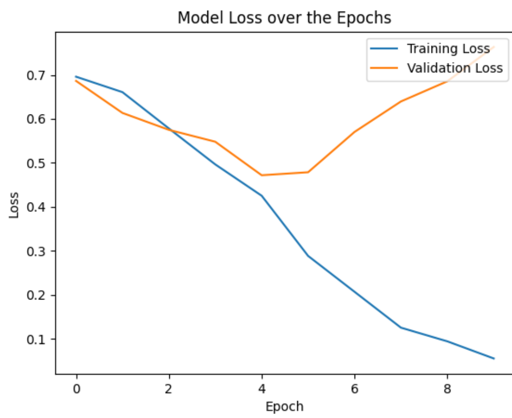


(e) Model accuracy over the epochs in the privacy-preserving ranking stage for a batch size = 32

Figure 5.1: Model accuracy plots for the privacy-preserving ranking stage

(a) Model loss over the epochs in the privacy-preserving ranking stage for a batch size = 2

(b) Model loss over the epochs in the privacy-preserving ranking stage for a batch size = 4



(c) Model loss over the epochs in the privacy-preserving ranking stage for a batch size = 8

(d) Model loss over the epochs in the privacy-preserving ranking stage for a batch size = 16

(e) Model loss over the epochs in the privacy-preserving ranking stage for a batch size = 32

Figure 5.2: Model loss plots for the privacy-preserving ranking stage

5.3.2 Loss

Like it was explained in the previous section, the network had to slightly overfit so that the private inferences were able to achieve better results. This can be demonstrated by the increase of the validation loss observed in the plots from Figure 5.2. When this happens, it can possibly mean that the model is starting to learn patterns only relevant for the training set and not great for generalization, which leads to some wrong predictions in the validation set. However, we knew that, at the same time, it was still learning some patterns which would turn out to be useful for model generalization, since the validation accuracy did not get worse. Similarly to the accuracy plots, we also concluded that the variation of the batch sizes does not result in any meaningful changes to these values across the different figures.

5.3.3 Evaluating the performance for the standard settings

For the default settings described in Sections 5.2.1 and 5.2.2, the training and validation accuracy remain approximately the same regardless of the batch sizes, as we can observe in Figure 5.3.
One factor that can determine a modification of these values is the number of epochs, but attending to Figures 5.1 and 5.2, we also inferred that if we had increased this value beyond 10, we would not obtain significant changes in the validation accuracy and would only be increasing the training time unjustifiably. Meanwhile, a number of epochs below 10 would not provide the necessary time for the model to learn the packets' size and timing features, which would result in less accurate inferences.
As we are able to observe, the batch size only influences the training process. The higher it is, the less it takes for the model to complete the training process, since each epoch aggregates more training samples, decreasing the total epochs' iterations.
The testing time will not vary much unless the intervening parties are either geographically distant or have more/less computational resources, as we will conclude in the following sections.

Table 5.3: General performance metrics evaluated for different numbers of epochs and batch sizes in the privacy-preserving ranking stage default setting

Number of Epochs	Batch Size	Training Time	Training Accuracy	Validation Accuracy	Testing Time (5 inferences)
10	2	38.6 s	89%	81%	4.1 s
	4	19.1 s	97%	80%	3.8 s
	8	10.9 s	97%	78%	3.8 s
	16	6.0 s	98%	80%	3.8 s
	32	3.7 s	98%	80%	3.8 s

5.3.4 Evaluating the performance for different hardware specifications

Table 5.4 shows the results of the performed tests when we modified the hardware specifications for each machine, according to the settings that were defined in Section 5.2.1. In this particular experiment, we were trying to assess the impact that the hardware we were using could have on the training and testing times of the privacy-preserving ranking stage.
The values that appear in this table represent multiple runs, which provides higher confidence and consistency. Surprisingly, setting 2 (the master machine with lower capabilities) showed little to no change in

the training times compared to other settings, where its resources were increased. This could be the consequence of small computational power involved in this specific phase, due to the simple neural network that is used to train the model.

The testing times, however, were slightly more consistent to what we were expecting. Despite being just moderately higher than the values shown in settings 1 and 2, the increase that can be observed in settings 3 and 4 is the outcome of the reduced capabilities across the three ISPs and the operator, respectively. This is not only affecting one particular machine, but all the machines that are performing MPC, therefore they take slightly more time to perform each inference.

Table 5.4: Evaluation of the training and testing times for different hardware specifications in the privacy-preserving ranking stage

Hardware Setting	Batch Size	Training Time	Testing Time (5 inferences)
1	2	38.6 s	4.1 s
	4	19.1 s	3.8 s
	8	10.9 s	3.8 s
	16	6.0 s	3.8 s
	32	3.7 s	3.8 s
2	2	39.1 s	3.6 s
	4	19.9 s	3.9 s
	8	10.9 s	4.2 s
	16	6.0 s	4.4 s
	32	3.7 s	4.5 s
3	2	39.6 s	4.9 s
	4	20.4 s	3.7 s
	8	11.4 s	4.1 s
	16	6.2 s	4.0 s
	32	3.8 s	3.9 s
4	2	40.9 s	4.6 s
	4	21.2 s	4.0 s
	8	11.7 s	4.3 s
	16	6.5 s	4.2 s
	32	3.8 s	4.0 s

5.3.5 Evaluating the performance for different geographical locations

In the last performance test for this phase, we decided to evaluate the testing time of the privacy-preserving ranking stage when the machines are disposed in several different geographical locations, like detailed in Section 5.2.2.

Table 5.5 presents the values that we obtained across these locations. The default setting that was used across multiple experiments turned out to be $\approx 4\times$ slower than the local setting, in which all the entities could be instantiated in five different terminals and the servers could communicate using different ports. This setting, however, must just be considered for testing purposes, since a solution like this runs on the same machine, thus not providing the necessary security requirement of hardware isolation for a realistic MPC setting.

Expectedly, settings 3, 4 and 5 demonstrated that the geographical location of the parties represents a major factor that influences the testing time of the solution. While settings 3 and 4 simulate the hypothesis where all the parties are still located within the same continent, the last setting emulates a "worst case scenario", where almost all the members are extremely distant from each other (across multiple continents), which will increase the round-trip time (RTT) of the messages exchanged as well as the communication delays between the parties compared to the other settings. Due to these reasons, the testing time was $\approx 3\times$ and $\approx 2\times$ higher in comparison to settings 3 and 4, respectively.

Table 5.5: Evaluation of the testing time for different geographical locations in the privacy-preserving ranking stage

Location Setting	Testing Time (5 inferences)
1	0.73 s
2	3.8 s
3	13.4 s
4	19.4 s
5	39.8 s

5.3.6 Precision

For this experiment, we kept the default batch size of 32, since we would get no benefits in taking more time in the training process. The key parameter of this test, however, is the correlation threshold that is defined by the operator. Pairs with values that are equal or greater than this threshold represent a correlated pair, therefore the adjustment of this variable is extremely important in order to produce desirable precision results (high TPR and low FPR).

The number of private inferences has to be previously established, since we won't indefinitely run the entire testing dataset. Hence, these results only constitute a partial approximation of the precision values that can be obtained when privately evaluating the flow pairs. Nevertheless, there were several iterations over these experiments, so the results expressed in Table 5.6 are able to provide a relatively high level of confidence.

As we were expecting, higher thresholds lead to a lower TPR, since the amount of false negatives undoubtedly increases. Analogously, on the opposite side of the spectrum, lower thresholds allow the number of false positives to be greater, thus increasing the FPR. A good threshold tries to maximize the TPR while keeping the FPR low. Although we did not find an optimal solution for this scenario, the threshold of `0.75` was the one that expressed this relation the best.

Table 5.6: Privacy-preserving ranking stage precision metrics evaluated for 20 private inferences varying the correlation threshold

Inferences	Batch Size	Threshold	TP	TN	FP	FN	TPR	FPR
20	32	0.65	9	6	4	1	90%	40%
		0.70	9	8	2	1	90%	20%
		0.75	9	9	1	1	90%	10%
		0.80	9	8/9	1/2	1	90%	10-20%
		0.85	7	9	1	3	70%	10%
		0.90	7	9	1	3	70%	10%
		0.95	4	9	1	6	40%	10%

TP - True Positives
TN - True Negatives
FP - False Positives
FN - False Negatives
TPR - True Positive Rate
FPR - False Positive Rate

5.4 Evaluation of the Privacy-Preserving Correlation Stage

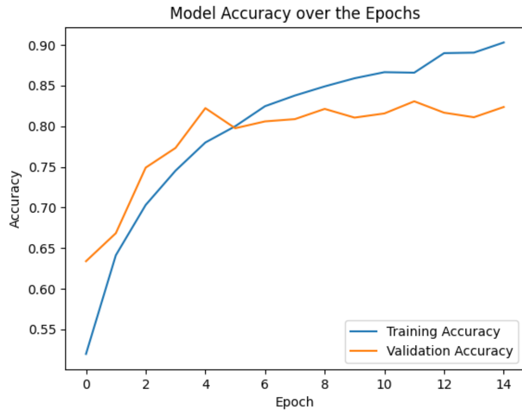
Similarly to the previous section, we will also present the obtained results for the same metrics that the past experiments evaluated, but this time for the privacy-preserving correlation stage.

5.4.1 Accuracy

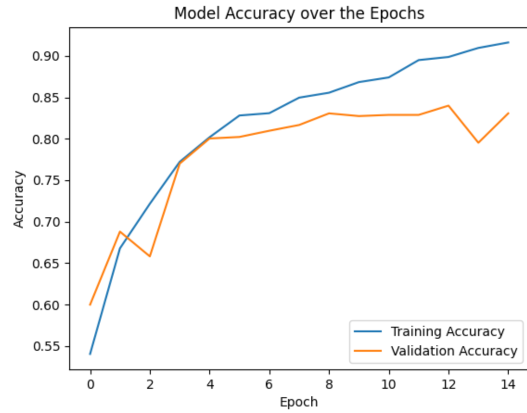
Like in the privacy-preserving ranking stage, we had to tune the number of epochs in order to prevent the model to overfit or underfit.
In this phase, however, we managed to properly balance the difference between the training accuracy and validation accuracy, so the plots from Figure 5.3 demonstrate slightly lower values of training accuracy when compared to Figure 5.1, but instead are closely followed by the validation accuracy.
As a result, the model does not overfit as much as before, since it does not memorize too much training samples, thus performing better when confronted with new data.

5.4.2 Loss

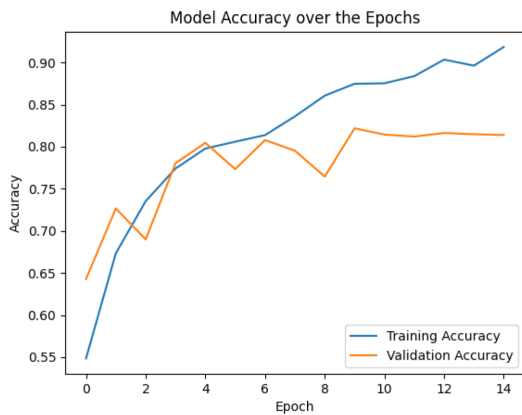
Unlike the previous phase, here the validation loss stayed considerably closer to the training loss, despite some occasional spikes, as seen in the plots from Figure 5.4.
It probably would still be advantageous if the training process was, to a slight extent, shorter and more regularized, since we can observe some irregularities in the validation loss from 10 epochs onwards.
However, similarly to the privacy-preserving ranking stage, we found out that if we adjusted this value to 10, the private inferences would not be as accurate as desired, so we chose to allow this less noticeable model overfitting once again, just to provide a smoother classification phase.



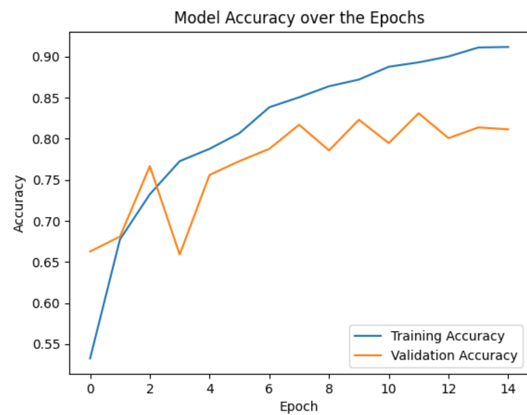
(a) Model accuracy over the epochs in the privacy-preserving correlation stage for a batch size = 2



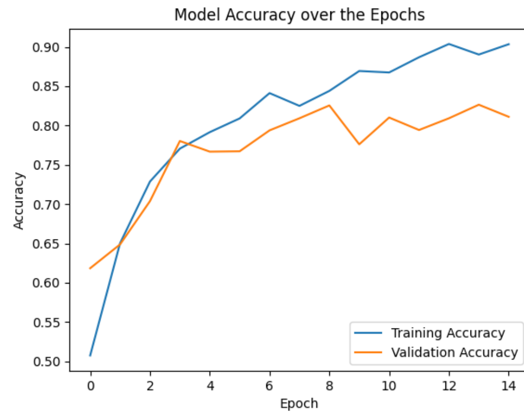
(b) Model accuracy over the epochs in the privacy-preserving correlation stage for a batch size = 4



(c) Model accuracy over the epochs in the privacy-preserving correlation stage for a batch size = 8

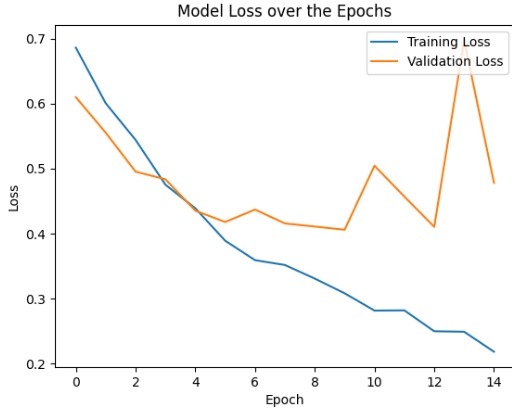


(d) Model accuracy over the epochs in the privacy-preserving correlation stage for a batch size = 16



(e) Model accuracy over the epochs in the privacy-preserving correlation stage for a batch size = 32

Figure 5.3: Model accuracy plots for the privacy-preserving correlation stage



(a) Model loss over the epochs in the privacy-preserving correlation stage for a batch size = 2

(b) Model loss over the epochs in the privacy-preserving correlation stage for a batch size = 4

(c) Model loss over the epochs in the privacy-preserving correlation stage for a batch size = 8

(d) Model loss over the epochs in the privacy-preserving correlation stage for a batch size = 16

(e) Model loss over the epochs in the privacy-preserving correlation stage for a batch size = 32

Figure 5.4: Model loss plots for the privacy-preserving correlation stage

5.4.3 Evaluating the performance for the standard settings

For the default settings established in Sections 5.2.1 and 5.2.2, we analysed the performance, evaluating the training time, training accuracy, validation accuracy and testing time for different batch sizes, as presented in Table 5.7. In this occasion, although the training accuracy was not as higher as in the

privacy-preserving ranking stage, we managed to obtain validation accuracy values closer to the training accuracy ones, like it was detailed in the previous sections. As expected, the training and testing times here are significantly greater than in the last phase due to the convolution operations, but, once again, only the training time gets impacted by the batch size. The testing time itself, despite being slightly irregular, is just affected by the hardware specifications or the geographical locations of the computation participants. The following sections will support this statement.

Table 5.7: General performance metrics evaluated for different numbers of epochs and batch sizes in the privacy-preserving correlation stage default setting

Number of Epochs	Batch Size	Training Time	Training Accuracy	Validation Accuracy	Testing Time (5 inferences)
15	2	680.4 s	90%	82%	86.7 s
	4	454.3 s	92%	83%	85.1 s
	8	347.7 s	92%	81%	81.1 s
	16	285.2 s	91%	81%	81.9 s
	32	268.5 s	90%	81%	87.9 s

5.4.4 Evaluating the performance for different hardware specifications

Contrary to the previous phase, in the privacy-preserving correlation stage it is clearly perceptible the impact that the hardware can have both in the training times (demonstrated by the results obtained in setting 2, when the master machine is downgraded) as well as in the testing times (when we allocate less resources for the servers and operator in settings 3 and 4, respectively).

The training times in Table 5.8 are approximately similar across the different settings except for setting 2, as we'd expect. Our results show that training the model with a master that only has 1 vCPU instead of the default 4 vCPUs originates a discrepancy of $\approx 2 - 2.5\times$ slower training times.

Moreover, the testing times are also influenced by the hardware capabilities of the servers and the operator. Settings 3 and 4 present values that deviate $\approx 15 - 20$ seconds from the ones in previous settings. None of these variations are surprising, since the neural network that is used for this particular phase has additional computational complexity due to the convolution process. Therefore, modifying the hardware specifications can clearly provoke an effect on these values.

Table 5.8: Evaluation of the training and testing times for different hardware specifications in the privacy-preserving correlation stage

Hardware Setting	Batch Size	Training Time	Testing Time (5 inferences)
1	2	680.4 s	86.7 s
	4	454.3 s	85.1 s
	8	347.7 s	81.1 s
	16	285.2 s	81.9 s
	32	268.5 s	87.9 s
2	2	1615.9 s	80.7 s
	4	1025.5 s	80.2 s
	8	736.4 s	81.7 s
	16	618.6 s	82.9 s
	32	565.9 s	85.3 s
3	2	673.6 s	97.1 s
	4	454.1 s	103.8 s
	8	349.9 s	97.8 s
	16	283.2 s	102.8 s
	32	257.7 s	97.2 s
4	2	665.5 s	100.0 s
	4	448.7 s	108.9 s
	8	343.6 s	99.8 s
	16	278.3 s	100.9 s
	32	256.0 s	103.6 s

5.4.5 Evaluating the performance for different geographical locations

Similarly to the last phase, we also experimented to run inferences in this stage while varying the geographical locations of the machines. The conclusions we extracted were also the ones we were anticipating: high latency significantly increases the testing time of the privacy-preserving correlation stage.

Table 5.9 expresses the contrast of these values. Running the inferences locally is twice as fast as running them across some machines spread in Europe (according to the locations in setting 2). However, if we try to reproduce a scenario where these computers are located across different continents (setting 5), we found out that the testing time increases $\approx 6\times$ when compared to our default setting and $\approx 12\times$ in comparison to the local environment.

Table 5.9: Evaluation of the testing time for different geographical locations in the privacy-preserving correlation stage

Location Setting	Testing Time (5 inferences)
1	42.2 s
2	87.9 s
3	229.3 s
4	258.4 s
5	521.5 s

5.4.6 Precision

When measuring the precision for this stage, we had to account for the high and inescapable testing times. Due to that reason, and also considering the fact that this evaluation is only an approximation of the actual precision of the system, we decided to decrease the number of inferences from 20 (of the last phase) to 10.

Nevertheless, we still repeated these experiments over several occasions, just to ensure that we had a certain level of confidence when registering the results on Table 5.10. There are, however, some additional differences that are worth mentioning when comparing these values to the ones that appear in Table 5.6.

Firstly, the correlation thresholds did not get as high as the ones observed in the other phase, since the TPR was already greatly reduced when the threshold was at 0.85. So, we considered unproductive trying to experiment with even higher thresholds and instead focused on the lower values.

Surprisingly, we noticed that, for most of the tests that were performed, a value that ranged from `0.60` $\leq$ `threshold` $\leq$ `0.75` was good enough to achieve optimal results (100% TPR and 0% FPR).

Table 5.10: Privacy-preserving correlation stage precision metrics evaluated for 10 private inferences varying the correlation threshold

Inferences	Batch Size	Threshold	TP	TN	FP	FN	TPR	FPR
10	32	0.55	5	4	1	0	100%	20%
		0.60	5	5	0	0	100%	0%
		0.65	5	5	0	0	100%	0%
		0.70	5	5	0	0	100%	0%
		0.75	5	5	0	0	100%	0%
		0.80	4	5	0	1	80%	0%
		0.85	3	5	0	2	60%	0%

TP - True Positives
TN - True Negatives
FP - False Positives
FN - False Negatives
TPR - True Positive Rate
FPR - False Positive Rate

5.5 Preliminary Performance Evaluation for the DP extension

This section will expose the preliminary findings in regards to the MPC+DP alternative privacy-preserving setting that we started to implement. The results themselves do not meet the requirements of our system, but we find relevant to present them nonetheless, so they can be further improved.

There was a restriction for these experiments: the number of `microbatches` had to be 1, like we had already previously explained. As far as we know, this was due to a bug in the specific TF Privacy version that we used, which was the only one compatible with the libraries that had to be imported in our TF Encrypted version.

Although this prevented us from splitting and training the samples into several microbatches, we were still able to modify other hyper-parameters, such as the `l2_norm_clip` (which did not produce ob-

servable changes in the results) and the `noise_multiplier` (the most relevant hyper-parameter in this experiment). These general performance results can be observed in Tables 5.11 and 5.12 for the privacy-preserving ranking and correlation stages, respectively.

We had to increase the number of epochs for the first phase from 10 to 30, since with 10 epochs the model barely learned anything. Nevertheless, the training and validation accuracy values are still generally low and this occurs in both stages, meaning that the datasets are not that resilient against the introduction of noise properties.

Another evident deduction is the increase of the training and testing times mainly in the correlation stage, when compared to the obtained results in Table 5.7 for the same batch size and number of epochs. This is expected, since the noise and microbatches combined will elevate these times.

Finally, there is also a noteworthy conclusion that has to be taken into consideration, which is the relation between the noise multiplier and the *epsilon* (ϵ) that is computed at the end of the training process. Considering the fact that ϵ represents the probability of the output changing with the addition or removal of a single training sample, then we seek a small value, so that the privacy guarantees of the model are stronger. Therefore, a higher noise multiplier means a lower ϵ. However, the increase of the noise multiplier also comes at the expense of weaker training and validation model accuracy. As we can observe in both tables, these values are extremely low when compared to the normal public training of previous experiments.

A further analysis of this setting is something that should be taken into consideration, but we already demonstrated that there will always be trade-offs when choosing an adequate noise multiplier value that manages to ensure good privacy guarantees without compromising too much the utility of the system.

Table 5.11: General performance metrics evaluated for different noise multipliers in the privacy-preserving ranking stage for the MPC+DP setting

Number of Epochs	Batch Size	Noise Multiplier	Training Time	Training Accuracy	Validation Accuracy	Testing Time (5 inferences)	Epsilon (ϵ)
30	32	0.05	20.2 s	79%	69%	3.9 s	508121.49
		0.10	20.0 s	60%	56%	4.1 s	27973.14
		0.20	20.1 s	53%	52%	4.1 s	515.54
		0.30	19.6 s	51%	51%	4.1 s	134.20

Table 5.12: General performance metrics evaluated for different noise multipliers in the privacy-preserving correlation stage for the MPC+DP setting

Number of Epochs	Batch Size	Noise Multiplier	Training Time	Training Accuracy	Validation Accuracy	Testing Time (5 inferences)	Epsilon (ϵ)
15	32	0.10	310.2 s	65%	62%	103.7 s	14032.62
		0.20	307.6 s	57%	54%	98.6 s	303.82
		0.30	306.1 s	55%	53%	96.8 s	90.13
		0.40	310.9 s	54%	52%	99.9 s	35.16
		0.50	317.1 s	50%	50%	102.2 s	17.10

5.6 General security key-points

By including a setting that supports DP training, we are able to mitigate the risk of exposing sensitive information about the training set. If this was not the case, then a potential attacker could reveal some private information by just querying the deployed model. Like exposed before in Section 3.2.2, this attack does not require the application of DP techniques in order to succeed, but its inclusion can potentially be meaningful if eventually Torpedo starts acting as a deanonymization framework, supporting criminal investigations instead of indiscriminately performing a traffic correlation attack against the Tor network. As it was detailed in the previous section, we did not manage to achieve great preliminary results, but a model trained with DP that provides strong privacy properties (low ϵ) should not be affected by a single or small set of training samples in its dataset.

Regarding MPC, there are some relevant security key-points that have to be mentioned. The introduction of MPC in this work occurs after training the model, when we split the model weights into shares and then send a share of each value to the different TF Encrypted servers (ISPs). This ensures a key property: if an attacker looks at the share on one ISP, it reveals nothing about the original value (model weights). These weights can just be possessed if a majority of the shares in the ISPs are reconstructed.

Another relevant characteristic is the fact that throughout the entire inference phase all the data is constantly encrypted. The operator never shares raw data, only secret shared inferences, since the queries are initially secret shared before being submitted.

Moreover, the operator does not have to possess the model in order for this attack to be successful. It just pre-processes the data the same way the master does, connects its client queue to the server queue deployed in the master machine, secret shares the data locally and then it is finally ready to send this data as private inference shares. The master never sees the predictions of the operator and the operator never downloads the model, being essentially able to get private predictions on encrypted data with an encrypted model.

Chapter 6

Conclusion

Finally, we will reflect on the achievements that were accomplished by this thesis and suggest some future improvements and lines of research that can be studied in order to improve or expand the developed solution.

6.1 Contributions

Perhaps the most significant achievement of this thesis is precisely the development of a Torpedo variant that contemplates the inclusion of privacy-preserving techniques, which mainly consist in MPC but also incorporate preliminary operations for DP training. As far as we know, this constitutes a novelty for a large-scale deanonymization attack against the Tor network.

We followed a modular approach that allowed us to separate the entities involved in the computations of Torpedo, these entities being the master, the ISPs/servers and the operator. We had to train the data on different neural networks depending on the requirements of each phase, but always prioritizing both precision and performance.

Our results show that these privacy-preserving Torpedo variants are able to process a large quantity of flow pairs, while maintaining an acceptable performance (reasonable training and testing times) as well as precision (generally high TPR and low FPR), at least in the Multi-Party Computation setting.

Additionally, we also started the development of a preliminary privacy-preserving extension (Differential Privacy), that hampers the memorization of the sensitive training data, reducing the chances of the model weights being leaked.

6.2 Future Work

A perceptible future line of research would be the expansion of the Differential Privacy setting, in order to improve the obtained results, not only performance-wise, but mainly in regards to precision. A thorough analysis and extension of this setup would encompass a valid direction of future work by itself.

This TensorFlow version of Torpedo could also be adapted or redesigned to a PyTorch counterpart, so that it would be feasible to implement Torpedo under a different MPC framework (e.g., CrypTen instead of TF Encrypted) and compare the obtained results for both solutions.

Finally, another possible adaptation that could be made would be a change of paradigm in the attack

architecture, by allowing the data to be jointly trained between all the ISPs, contrary to the training process in the master machine and subsequent distribution of model shares between the participants. This, however, would require a considerable amount of modifications in the data pre-processing phase, since each ISP would be responsible for different datasets (e.g., ISP 1 for the entry point data and ISP 2 for the exit point data).

Bibliography

[1] Compute Engine: Virtual Machines (VMs) — Google Cloud. https://cloud.google.com/compute.

[2] Keras: The Python Deep Learning API. https://keras.io/.

[3] PyTorch. https://pytorch.org/.

[4] TensorFlow. https://www.tensorflow.org/.

[5] TensorFlow Privacy GitHub. https://github.com/tensorflow/privacy.

[6] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, oct 2016.

[7] Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU). *CoRR*, abs/1803.08375, 2018.

[8] Joël Alwen, Rafail Ostrovsky, Hong-Sheng Zhou, and Vassilis Zikas. "incoercible multi-party computation and universally composable receipt-free voting". In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, pages 763–780, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[9] A Aly, K Cong, D Cozzo, M Keller, E Orsini, D Rotaru, O Scherer, P Scholl, N P Smart, T Tanguy, and T Wood. SCALE-MAMBA v1.14: Documentation, 2021.

[10] Gilad Asharov and Yehuda Lindell. A Full Proof of the BGW Protocol for Perfectly Secure Multiparty Computation. *Journal of Cryptology*, 30(1):58–151, Jan 2017.

[11] Marco Barbera, Vasileios Kemerlis, Vasilis Pappas, and Angelos Keromytis. CellFlood: Attacking Tor Onion Routers on the Cheap. volume 8134, pages 664–681, 09 2013.

[12] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 1–10, New York, NY, USA, 1988. Association for Computing Machinery.

[13] Stevens Le Blond, Pere Manils, Chaabane Abdelberi, Mohamed Ali Kâafar, Claude Castelluccia, Arnaud Legout, and Walid Dabbous. One Bad Apple Spoils the Bunch: Exploiting P2P Applications to Trace and Profile Tor Users. *CoRR*, abs/1103.1518, 2011.

[14] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure Multiparty Computation Goes Live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security*, pages 325–343, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[15] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning. Cryptology ePrint Archive, Report 2019/1365, 2019.

[16] Enrico Cambiaso, Ivan Vaccari, Luca Patti, and Maurizio Aiello. Darknet Security: A Categorization of Attacks to the Tor Network. 03 2019.

[17] Sambuddho Chakravarty, Angelos Stavrou, and Angelos D. Keromytis. Traffic Analysis against Low-Latency Anonymity Networks Using Available Bandwidth Estimation. In *Proceedings of the 15th European Conference on Research in Computer Security*, ESORICS'10, page 249–267, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] Rahul Chauhan, Kamal Kumar Ghanshala, and R.C Joshi. Convolutional Neural Network (CNN) for Image Detection and Recognition. In *2018 First International Conference on Secure Cyber Computing and Communication (ICSCCC)*, pages 278–282, 2018.

[19] Nicolas Christin. Traveling the Silk Road: A Measurement Analysis of a Large Anonymous Online Marketplace. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 213–224, New York, NY, USA, 2013. Association for Computing Machinery.

[20] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 7 2015.

[21] 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. *CoRR*, abs/1810.08130, 2018.

[22] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 643–662, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[23] Pedro Manuel Torres Pires de Medeiros. Distributed System for Cooperative Deanonymization of Tor Circuits, 2021.

[24] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[25] Shimon Even, Oded Goldreich, and Abraham Lempel. A Randomized Protocol for Signing Contracts. *Commun. ACM*, 28(6):637–647, June 1985.

[26] Andrea Forte, Nazanin Andalibi, and Rachel Greenstadt. Privacy, Anonymity, and Perceived Risk in Open Collaboration: A Study of Tor Users and Wikipedians. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '17, page 1800–1811, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Information Geographies. The anonymous internet.

[28] O. Goldreich, S. Micali, and A. Wigderson. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 218–229, New York, NY, USA, 1987. Association for Computing Machinery.

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press.

[30] Kevin Gurney. *An Introduction to Neural Networks*. Taylor & Francis, Inc., USA, 1997.

[31] Rob Jansen, Florian Tschorsch, Aaron Johnson, and Björn Scheuermann. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. 01 2014.

[32] Zhanglong Ji, Zachary Chase Lipton, and Charles Elkan. Differential Privacy and Machine Learning: a Survey and Review. *CoRR*, abs/1412.7584, 2014.

[33] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, Baltimore, MD, August 2018. USENIX Association.

[34] Marcel Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1575–1590, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 830–842, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Marcel Keller, Peter Scholl, and Nigel P. Smart. An Architecture for Practical Actively Secure MPC with Dishonest Majority. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 549–560, New York, NY, USA, 2013. Association for Computing Machinery.

[37] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[38] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. CrypTen: Secure Multi-Party Computation Meets Machine Learning, 2021.

[39] Yann LeCun, Corinna Cortes, and Chris Burges. MNIST Handwritten Digit Database.

[40] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 619–631, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Machine Learning Mastery. A Gentle Introduction to the Rectified Linear Unit (ReLU), 2020.

[42] Damon McCoy, Kevin Bauer, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. "shining light in dark places: Understanding the tor network". In Nikita Borisov and Ian Goldberg, editors, *Privacy Enhancing Technologies*, pages 63–76, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[43] Ilya Mironov. Rényi Differential Privacy. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, aug 2017.

[44] Payman Mohassel and Peter Rindal. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 35–52, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, 2017.

[46] S.J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *2005 IEEE Symposium on Security and Privacy (S P'05)*, pages 183–195, 2005.

[47] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. DeepCorr: Strong Flow Correlation Attacks on Tor Using Deep Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1962–1976, New York, NY, USA, 2018. Association for Computing Machinery.

[48] Milad Nasr, Amir Houmansadr, and Arya Mazumdar. Compressive Traffic Analysis: A New Paradigm for Scalable Traffic Analysis. pages 2053–2069. Association for Computing Machinery, 10 2017.

[49] Rebekah Overdorf, Mark Juárez, Gunes Acar, Rachel Greenstadt, and Claudia Díaz. How Unique is Your .onion? An Analysis of the Fingerprintability of Tor Onion Services. *CoRR*, abs/1708.08475, 2017.

[50] Andriy Panchenko, Asya Mitseva, Martin Henze, Fabian Lanze, Klaus Wehrle, and Thomas Engel. Analysis of Fingerprinting Techniques for Tor Hidden Services. In *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society*, WPES '17, page 165–175, New York, NY, USA, 2017. Association for Computing Machinery.

[51] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, WPES '11, page 103–114, New York, NY, USA, 2011. Association for Computing Machinery.

[52] Nicolas Papernot, Martín Abadi, Úlfar Erlingsson, Ian Goodfellow, and Kunal Talwar. Semi-supervised Knowledge Transfer for Deep Learning from Private Training Data, 2016.

[53] Tor Project. About – Tor Metrics (Consensus).

[54] Tor Project. Servers – Tor Metrics.

[55] Tor Project. Who uses Tor?, 2019.

[56] Michael O. Rabin. How To Exchange Secrets with Oblivious Transfer, 2005. Harvard University Technical Report 81 talr@watson.ibm.com 12955 received 21 Jun 2005.

[57] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. *CoRR*, abs/1801.03239, 2018.

[58] Lamyaa Sadouk. *Time Series Analysis*. IntechOpen, 11 2019.

[59] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[60] Adi Shamir, Ronald L Rivest, and Leonard M Adleman. Mental Poker, 1 1979.

[61] Yi Shi and Kanta Matsuura. Fingerprinting Attack on the Tor Anonymity System. In *Proceedings of the 11th International Conference on Information and Communications Security*, ICICS'09, page 425–438, Berlin, Heidelberg, 2009. Springer-Verlag.

[62] Reza Shokri, Marco Stronati, and Vitaly Shmatikov. Membership Inference Attacks against Machine Learning Models. *CoRR*, abs/1610.05820, 2016.

[63] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. RAPTOR: Routing Attacks on Privacy in Tor. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 271–286, Washington, D.C., August 2015. USENIX Association.

[64] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: Efficient and private neural network training. Cryptology ePrint Archive, Report 2018/442, 2018.

[65] Gabriel Weimann. Going dark: Terrorism on the dark web. *Studies in Conflict & Terrorism*, 39(3):195–206, 2016.

[66] Philipp Winter, Roya Ensafi, Karsten Loesing, and Nick Feamster. Identifying and Characterizing Sybils in the Tor Network. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1169–1185, Austin, TX, August 2016. USENIX Association.

[67] Alexandra Wood, Micah Altman, Aaron Bembenek, Mark Bun, Marco Gaboardi, James Honaker, Kobbi Nissim, David R. OBrien, Thomas Steinke, and Salil Vadhan. Differential privacy: A primer for a non-technical audience. *Vanderbilt Journal of Entertainment & Technology Law*, 21(1):209–275, 2018.

[68] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.