Learning cryptography using adversarial neural networks

メラウーシュ, イザーク

Learning cryptography using adversarial neural networks

Ishak Meraouche

July 2023

Department of Informatics
Graduate School of Information Science and
Electrical Engineering, Kyushu University

Contents

List of Symbols

C	Ciphertext
P	Plaintext
K	Symmetric secret key
K_{pub}	Public Key
K_{priv}	Private Key
L_1	L1 Distance
$d(x, y)$	L1 Distance between x and y
L_{NN}	Loss of the neural network NN
W_{NN}	Weights vector of the neural network NN
Θ_{NN}, θ_{NN}	Weights vector of the neural network NN
π_{nn}	Probability produced by the neural network NN
RN	Random Noise
XOR	Exclusive OR operation
TPM	Tree Parity Machine
q	A prime number
r	A random number
G	A cyclic group

g	Generator of a cyclic group
CR	Concatenation result used in TPMs

List of Figures

List of Tables

Abstract

The field of Artificial Intelligence (AI) has seen an outstanding growth in the recent years in response to the increase of the needs of society in different areas, such as healthcare, finance, information processing, and cybersecurity. Still, on the subject of cybersecurity, one area that remained unexplored for decades is cryptography. In the early days of AI, neural networks struggled in learning the simplest forms of cryptography, such as the exclusive or operation, which discouraged researchers to further explore this area.

This situation changed with the seminal work by Abadi and Andersen, who took initial steps in 2016 to build neural network models that can learn secure forms of Cryptography using adversarial training. They propose a Generative Adversarial Networks model that can learn encryption. In their proposal, two neural networks train to protect their communication from a third neural network which is the attacker playing the role of an eavesdropper. The results were successful, and the two parties have been able to protect their communication from the attacker. The two parties are not trained to learn any specific encryption algorithm but rather train to achieve a specific goal. Their assigned goal is to generate ciphertexts that can be decrypted by Bob using a symmetric key but cannot be decrypted by the attacker without the key. The attacker is given the goal of decrypting the ciphertexts without the key. By continuously training against each other, the two parties end up coming with a way to protect their communication from this attacker.

This seminal work has also been used as a foundation for other cryptology solutions. For example, Hayes and Danezis show that it is possible to use the same model setup with a few changes to the neural network structure to make the neural networks learn Steganography. In their model, the sender learns to hide messages inside an image and the receiver learns to extract these messages properly. The attacker is a classifier that receives the original image and the steganographic image as input and tries to tell which one contains a hidden message.

Despite the positive feedback and follow up contributions that the Abadi and Andersen model has seen, it has been shown to have multiple flaws that need to be addressed, some of which were yet to be resolved in the state of the art. For example, Lu Zhou et al show that the ciphertexts generated failed the National Institute of Standards and Technology (NIST) statistical test and are weak against multiple statistical attacks like the χ^2 attack. This means that the ciphertexts might contain sensitive information about the key and/or the plaintext.

Another important challenge was uncovered by Jamie Hayes and George Danezis, who show that even if two separate neural networks train on the same data, they are not guaranteed to learn the same encryption/decryption technique. This is known as the problem of non-convexity where deep learning models that train with the same data are not guaranteed to reach the same results. The reason is the big number of possible outcomes makes it almost impossible for the models to converge to the same outcome. As two machines that train on the same dataset of plaintexts and keys are not guaranteed to learn the same encryption algorithm, communicating among more than two parties becomes a difficult problem that needs to be addressed.

Finally, Abadi and Andersen themselves also point out that the neural networks cannot learn asymmetric encryption. In all their experiments, they are only able to protect their communication using symmetric keys. Giving a pair of public and private keys to the two parties makes the receiver not able to use the private key to decrypt the messages that have been encrypted with the public key.

This thesis explores the flaws and challenging issues stated above – weak ciphertexts, problem of non-convexity, inability to learn asymmetric encryption – and proposes solutions to address these issues, advancing the state of the art in this area.

To address the problem of non-convexity and weak ciphertexts, we propose a neural networks model that allows multiple parties to synchronize together and learn the same encryption/decryption technique. We also add more attackers that apply different cryptanalysis attacks which pushes the sender to generate stronger ciphertexts. The ciphertexts generated become resistant to the cryptanalysis attacks employed by the attackers and their encryption is equivalent to the One Time Pad encryption. Our model is a combination of different contributions that improve the security of the ciphertexts. We also show how this model can be used to obtain secret sharing schemes realizing any general access structure.

Jamie Hayes and George Danezis stress out that their model is also subject to the problem of non-convexity and that it is not guaranteed that two parties training on different machines can learn the same steganography algorithm. We address this issue by showing how more than two parties can learn the same steganography technique with the focus on a 3-party case.

Lastly, we address the issue of the model not being able to learn asymmetric encryption. We propose a model, the first of its kind, which learns to encrypt and decrypt data using asymmetric information. This removes the overhead of using a key exchange scheme to agree on a common state beforehand. Our model can encrypt data using a public key that can be known to anyone and decrypt data using a private key that is only known to the receiver.

Acknowledgment

In this part of the thesis, I would like to express my gratitude and appreciation to all those who have supported me throughout my journey to accomplish this thesis and acquire my PhD Degree.

First and foremost, I am immensely grateful to my supervisor Kouichi SAKURAI without whom, I would not have had the chance to come to Japan to pursue my PhD. Professor Sakurai has always provided me with invaluable guidance, support and advice. His patience and expertise have been the key in shaping this thesis. I am really grateful for having him as my supervisor throughout my journey.

I am also immensely indebted to Dr. Sabysachi Dutta who has co-supervised my work and has provided me with an immense amount of support, guidance, and constructive feedback at every stage of my journey.

I would also like to express my gratitude to Dr. Rodrigo Roman for his constructive feedback on my thesis defense presentation slides and abstract. Without his valuable comments, the thesis defense presentation quality would never be as it is now.

I would like to also express my gratitude to Dr. Haowen Tan, Dr. Isaac Agudo and Prof. Sraban Kumar Mohanty for co-authoring a few of my published papers and giving valuable feedback that helped these paper get accepted during the peer review process.

I would like to also express my gratitude to Dr. Danilo V. Vargas, Dr. Yujie Gu for their support at the early stages of my journey.

I would like to also express my gratitude to Prof. Chen-Mou Cheng, Prof. Chunhua Su and Dr. Toru Nakamura for being my external advisors and giving me feedback on my research.

I would like to extend my gratitude to the members of my thesis committee, Prof. Koji Nuida and Prof. Ikeda Daisuke for their insightful comments and suggestions during the thesis defense. Their valuable input has undoubtedly enhanced the quality of this research work.

Furthermore, I am deeply grateful for Oiwa Yasura and Kayo Yamamoto for their immense support in all the administrative procedures that I had to go through. Their support has helped me do these procedures with ease and peace of mind.

I am also deeply grateful for my family for their unwavering belief in my abilities, and continuous encouragement. Their constant support and sacrifices have been the cornerstone of my achievements, and I am forever indebted to them. My gratefulness also goes to my close friends especially Yasser, Hakou, Ramzi, Samir, Khirou and Mourad for their always being there for me when I needed them. Last but not least, no amount of gratefulness would be enough to thank Achouak for always being there for me and giving me the support and attention I needed.

Lastly, I would like to express my gratitude to the staff of the Embassy of Japan in Algeria for the trust they put in me by recommending me for a fully funded scholarship. My gratitude especially goes to the MEXT (Ministry of Education, Culture, Sports, Science and Technology) for sponsoring my PhD studies between April 2019 and March 2023. This work was also partially supported by the Telecommunications Advancement Foundation (TAF) of Japan.

To everyone who has played a part, big or small, in the completion of this thesis, I offer my sincere thanks. Your support has been invaluable, and I am grateful for the contributions each one of you has made.

Preface

Artificial Intelligence and Machine Learning techniques are continuously evolving in response to the demand for obtaining models that can handle complex tasks in high-dimensional domains which include image processing, realistic image generation, speech recognition, facial recognition etc. Although these fields have seen significant innovation and improvement in the technologies during the last two decades, the area of neural networks based cryptology remains relatively unexplored. The primary reason for this lag may be attributed to the inability of neural networks to learn computing simple functions like the exclusive-or (xor) which is one of the basic functions used in cryptography. However, the interest in pursuing neural networks based cryptography got rekindled after the work by I. Kanter, W. Kinzel and E. Kanter [6] who successfully showed how to establish common secret keys between two neural networks. Although their construction was proven insecure soon afterwards, later several follow up works showed how to increase the security-level of the key exchange using suitable parameters that makes it computationally difficult to calculate the keys.

Abadi and Andersen from Google Brain [1] took another direction in late 2016. Instead of experimenting with key exchange, they wanted to know if neural networks can learn to protect communications. They used Generative Adversarial Networks (GANs) for these experiments. In their setup, two parties (Alice and Bob) share a secret key and are confronted to an eavesdropper (Eve). Alice encrypts messages and sends the ciphertexts to Bob which he will try to decrypt. Eve intercepts the encrypted messages and tries to decrypt them without the Key. Alice is penalized via her loss function if Eve's accuracy is too high, or Bob's accuracy is too low. Their experiments show that after some training iterations, Alice and Bob beat Eve and protect their communication from her. In this novel way of doing neural networks based cryptography, all parties that are part of the system including adversaries are considered neural networks. Each neural network has a goal expressed in the form of a loss function that it must optimize. The communicating parties (Such as Alice and Bob) are penalized via their loss function if the adversaries are getting too accurate in their decryption which pushes them to generate stronger ciphertexts that are harder for the adversaries to decrypt. This "game" goes on until it reaches a state where either the adversary or the communicating parties have won. Therefore, Abadi and Andersen's approach does not try to make the neural networks learn a specific encryption algorithm but rather sets a goal to each of them via a loss function and trains them to optimize these loss functions. It has also been shown by researchers such as Jamie Hayes and George Danezis that it is possible to use the same model setup with a few changes to the neural network structure in order to make the neural networks learn Steganography.

While the work by Abadi and Andersen has seen a lot of positive feedback and follow up contributions, it has multiple flaws that need to be addressed. The most important flaw is regarding the security of the ciphertexts generated. It has been shown by Zhou *et al.* [7] that the ciphertexts generated are weak against multiple statistical attacks such as the χ^2 attack. The ciphertexts also failed the NIST statistical test. Another problem with this proposal is related to the non-convexity of neural network models in general. It is known that the loss function, which is the objective that the model tries to minimize during training is not convex. A function is said to be convex if it has a unique global minimum point. However, in deep learning and in Abadi and Andersen's model [1], the loss functions are highly complex and non-convex, meaning that they can have multiple local minima. This problem of non-convexity means that even two separate neural networks train on the same data, they are not guaranteed to learn the same encryption/decryption technique as there is a high chance that each of them converged to different local minima. This makes communicating among more than two parties a difficult problem that needs to be addressed as point out by Jamie Hayes and George Danezis [8]. Another flow that has been shown by Abadi and Andersen [1] is that the neural networks can only use symmetric keys for encryption and decryption. Abadi and Andersen's attempt to make the neural networks learn to use asymmetric keys has failed and the receiving party was not able to decrypt the message sent to it. In their approach, they kept the exact same setup and model but instead of giving Alice and Bob the same symmetric key, they generated a pair of public/private keys and gave Alice the public key for encryption and Bob the private key for decryption. One last flaw that is not as critical as the previous ones is the time required to train the neural networks. Training GANs is known to be time and power consuming making it difficult for power-limited devices to use the model.

This thesis explores the flaws and challenging issues stated above and proposes solutions to these issues.

To address the issue of weak ciphertexts and the difficulty to communicate among multiple parties caused by the problem of non-convexity, this thesis proposes a neural network model that allows multiple parties (neural networks) to learn perfectly secure symmetric encryption (the One Time Pad encryption) and that allows multiple randomly initialized neural networks to synchronize and learn the same encryption technique which will be used to protect their communication from attackers. These attackers apply different known cryptanalysis attacks such as the chosen plaintext attack and the communicating parties are penalized every time the attack is successful pushing them to produce stronger ciphertexts that are resistant to these attacks. Our perfectly secure encryption neural network model is a combination of different contributions [7, 9, 4] that improve the security of the ciphertexts generated. We also show how a subgroup of the parties can communicate privately or how a group communication can be performed without initializing a new training session. Furthermore, we used this model to obtain secret sharing schemes realizing any general access structure.

The steganography neural network model proposed by Jamie Hayes and George Danezis [8] is also subject to the problem of non-convexity described earlier. Jamie Hayes and George Danezis have stressed out this point and mention in their paper that due to the problem of non-convexity, it is not guaranteed that two neural networks training on different machines can learn the same steganography algorithm. To address this issue, this thesis shows how more than two neural networks can learn the same steganography algorithm. The approach is inspired from our multi-party adversarial encryption algorithm and shows how more than two neural networks can learn the same steganography algorithm and extract the messages hidden inside images correctly focusing on a 3-party case.

To address the issue of the neural networks not being able to encrypt and decrypt data using asymmetric keys, this thesis proposes a neural networks model, the first of its kind, which learns encryption using asymmetric information therefore removing the overhead of having to share a common random state before initiating a symmetric encryption communication session. Asymmetric encryption allows the neural networks to communicate directly using a pair of keys. The first key (the public key) is known to anyone and is used to encrypt the message or document to be sent. The second key (the private key), only known to the receiver, allows decryption of the message or document. Existing contributions required the neural networks to encrypt and decrypt data with a key that has been shared using a separate cryptographic protocol beforehand. Our technique is also shown to be secure against multiple cryptanalysis attacks.

To address the issue of the long training time that might not be optimal for some performance-limited devices such as IoT devices, the appendix of this thesis proposes an alternative lightweight neural networks model that is suitable for such devices. This model learns a secure encryption scheme based on the Tree Parity Machines (TPMs) key exchange protocol proposed by I. Kanter, W. Kinzel and E. Kanter [6]. This model learns to produce ciphertexts that are random enough to pass the NIST statistical test. The model is lighter than Abadi and Andersen's model and training is significantly faster. In contrast to Abadi and Andersen's model, the TPM key exchange protocol uses a simple neural network model with only one hidden layer making its training faster. This model trains using the Hebbian learning technique and does not rely on adversarial training in a GANs setup. Knowing that the TPMs key exchange has been subject to multiple attacks in the past that rendered it insecure, we performed the key exchange process based on the latest contributions to the TPM based key exchange protocol that makes the problem of guessing the keys difficult [10].

Chapter 1

Introduction

With the rapid expansion of communication through networks among multiple terminals (computers, smartphones etc.), it is stringent to develop technologies to protect the information exchanged in those networks. Often, when one device communicates with one or more devices, a cryptographic protocol is applied to encrypt all the transmitted data in order to protect the communication(s). Two kinds of cryptographic protocols are typically considered in the literature: one to establish a common secret key, and the other to encrypt the messages exchanged. In the practical applications, the lightweight and secure protocols are highly desired especially for some terminals with low performance, such as devices with limited battery-lives. To meet the application requirements, cryptography is consistently evolving through time according to the extensive development and improvement on the security of cryptographic protocols. Among others, the RSA cryptosystem [11] is widely used as a standard for public-key encryption and digital signature; and the Rijndael algorithm (also known as AES) [12] for symmetric encryption.

Machine learning plays a major role in cryptanalysis, a sub-domain of cryptology [13, 14, 15]. Roughly speaking, cryptanalysis aims to test and analyze the security of cryptographic protocols by feeding different inputs to the cryptographic algorithm and analyzing the outputs in order to find a common or repetitive pattern in the outputs that might help find the secret key or even decrypt the ciphertext without access to the key. Machine learning can help learn from the data generated by the cryptographic algorithm and detect significant patterns [16, 17].

In late 90's and early 2000's, several cryptographic protocols using machine learning and deep learning models were proposed such as [6], but were deemed insecure and even some concrete attacks [18] were shown subsequently. The interest in neural network based cryptography took a dip because of the fact that simple computations, even as basic as exclusive-or (XOR) operation could not be computed by simple neural networks.

However recently, Abadi and Andersen [1] initiated a research direction on learning to protect communications with adversarial neural cryptography. Specifically, it aims to create neural networks that can learn to encrypt a communication without being taught any specific encryption algorithm. This technique is based on generative adversarial networks (GANs), in which neural networks try to achieve a goal in the presence of an adversary (i.e. another neural network) by pitting against each other [19]. The main idea behind GANs is to have two neural networks competing in order to generate a new set of data that can be taken as the real data. GANs are powerful in their ability of mimicking various types of data, and hence broadly used especially in image and voice generation [20, 21, 22] to generate synthetic data which are indistinguishable from the true data distribution. Following Abadi and Andersen's work [1], a flow of research appeared in order to study the security of their model (e.g. [7]), as well as extend it to an assumed perfectly secure protocol [9], and many more [23, 8, 3, 24, 25].

In this thesis, we present a multi-party encryption scheme based on GANs [19] and previous contributions in adversarial cryptography [1, 4, 9]. We also show how it can be used to build a secure secret sharing scheme that does not rely on any primitives. Next, we show how neural networks can learn adversarial steganography among multiple parties. Lastly, we show how neural networks can learn asymmetric encryption in a GANs setup and remove the overhead of exchanging secret keys as in the models in [1, 9, 4]. In the appendix of the thesis, we show how to build a symmetric encryption scheme based on a more secure version of the original Tree Parity Machines model proposed in [6].

Chapter 2

Background

In this section we introduce the background material that will be used in the later sections. We begin with some basic terminologies.

2.1 Cryptography

Cryptography is the practice of securing communication and information through the use of codes, ciphers, and other techniques to protect the confidentiality, integrity, and authenticity of the information. It involves the transformation of plaintext (readable information) into ciphertext (unreadable information) through the use of mathematical algorithms and protocols. The process of converting ciphertext back to plaintext is known as decryption.

Cryptography has a wide range of applications, such as secure communication, data encryption, digital signature, and secure key exchange. It provides a way to protect the information from unauthorized access, tampering, and eavesdropping. Cryptography has become increasingly important in today's digital age, as more and more information is stored, transmitted, and processed electronically.

In terms of security, cryptography can be broadly divided into two main models – information theoretic security and computational security. In the former model the adversary, against whom a cryptographic protocol is supposed to ensure security, is taken to be computationally unbounded and in the latter one the adversary is assumed to be bounded with respect to its computational power. We make a note of the fact that any cryptographic primitive providing information theoretic security does not depend on any kind of hardness assumption and hence cannot be broken (in a provable manner) even with unlimited computing power. On the other hand, computationally secure primitives are based on *hardness* assumptions e.g. *integer factorization, discrete log computation* where the security is based on the infeasiblility of obtaining any "practical" algorithm to break the hardness problem(s).

2.1.1 Encryption & Decryption

Encryption is the process of converting plaintext (readable information) into ciphertext (unreadable information) through the use of mathematical algorithms and protocols. The process of converting ciphertext back to plaintext is known as decryption.

Encryption is used to protect the confidentiality of information by making it unreadable to anyone who does not have the proper decryption key. Encryption algorithms take the plaintext and a key as input, and output the ciphertext. The key is used to control the encryption process, and it is critical that the key be kept secret.

Decryption is the process of converting the ciphertext back to its original plaintext form. It uses the same key that was used during the encryption process, and the process is reversed. Decryption algorithms take the ciphertext and the key as input, and output the original plaintext.

2.1.2 Key Exchange

Secure key exchange is a cryptographic process that allows two or more parties to establish a shared secret key over an insecure communication channel. The shared secret key can then be used to secure subsequent communications between the parties using symmetric-key cryptography.

The goal of secure key exchange is to ensure that the shared secret key is only known to the parties that are intended to use it and to prevent it from being intercepted by an eavesdropper. There are various key exchange protocols that have been developed to achieve this goal, such as Diffie-Hellman [26] and RSA [11].

2.1.3 Digital Signature

Digital signature is a mathematical scheme that is used to verify the authenticity and integrity of a digital message or document. It is a way to ensure that the message or document has not been tampered with and that it was actually sent by the person or entity that it claims to be from.

2.1.4 One Time Pad

One time pad (OTP) is a symmetric key encryption technique which requires an n bit message to be xor-ed with a uniform n bit key to compute the ciphertext. The recipient who is already in possession of the n bit key can recover the message. It can be observed that this primitive is an information theoretically secure encryption scheme. However, OTP suffers from some serious drawbacks – size of the secret key has to be same as the message as well as the key has to be uniformly distributed over the key space and that the key cannot be reused. For every message an independently chosen key has to be used and this makes the scheme impractical to implement.

2.1.5 Steganography

Steganography is the practice of hiding secret information within a cover message or digital file in such a way that it is not noticeable to anyone except the intended recipient. The objective of steganography is to conceal the very existence of the message, making it difficult for an eavesdropper to detect the presence of the hidden information. This technique involves embedding secret information within a cover message, digital image, audio or video file, or other digital medium, without altering the original cover medium in a noticeable way.

Steganography can be seen as form of covert communication that enables individuals to exchange information securely and discreetly, making it a valuable tool for privacy and security purposes. While similar to cryptography, which involves encrypting messages to make them unreadable to anyone except the intended recipient, steganography focuses on hiding the existence of the message, rather than just making it unreadable.

2.2 Artificial Intelligence

Artificial Intelligence (AI) is the simulation of human intelligence processes by machines, especially computer systems. These processes include learning (the acquisition of information and rules for using the information), reasoning (using the rules to reach approximate or definite conclusions), and self-correction.

AI can be used to perform a wide range of tasks, such as image and speech recognition, decision making, natural language processing, and even autonomous vehicles. The field of AI is constantly evolving and advancing, with researchers and engineers working to develop more advanced and sophisticated AI systems.

There are several subfields in AI, such as Machine Learning which is a subset of AI, it is the study of algorithms and statistical models that computer systems use to perform a specific task without being explicitly programmed.

2.2.1 Machine Learning

Machine Learning (ML) is a subset of artificial intelligence that deals with the development of algorithms and statistical models that enable computer systems to learn and improve their performance on a specific task, without being explicitly programmed.

In machine learning, a model is trained on a dataset, which is a collection of examples used to learn the relationships and patterns in the data. The model can then be applied to new, unseen data to make predictions or decisions. There are several types of machine learning, such as supervised learning and unsupervised learning.

2.2.2 Neural Networks

A neural network is a type of machine learning model that is inspired by the structure and function of the human brain. It is a network of interconnected "neurons" that are organized into layers. These layers are connected by pathways called "synapses", which allow information to flow through the network.

The basic building block of a neural network is the "neuron", which is a simple mathematical function that receives input, processes it, and produces an output. The inputs are passed through the layers of the network, and at each layer, the neurons process the input and produce output that is passed on to the next layer. The final output of the network is a prediction, classification, or a value.

Neural networks can be used for a wide range of tasks such as image classification, speech recognition, natural language processing, and many more. They can be used in supervised, unsupervised, and reinforcement learning tasks.

There are different types of neural networks, such as feedforward neural networks, recurrent neural networks and convolutional neural networks. Each of them have different architectures and are used for different types of tasks.

In summary, neural networks are a type of machine learning model that is inspired by the structure and function of the human brain. They consist of interconnected "neurons" that are organized into layers, and can be used for a wide range of tasks such as image classification, speech recognition, natural language processing, and many more. They can be used in supervised, unsupervised, and reinforcement learning tasks, and there are different types of neural networks with different architectures and are used for different types of tasks.

2.2.3 Supervised Learning and Unsupervised Learning

Supervised learning and unsupervised learning are two main categories of machine learning.

Supervised learning is a type of machine learning where the model is trained on labeled data, meaning that the desired output is already known for each input. The goal of supervised learning is to learn a function that can predict the output based on the input. Examples of supervised learning tasks include image classification, speech recognition, and linear regression.

Unsupervised learning, on the other hand, is a type of machine learning where the model is trained on unlabeled data, meaning that the desired output is not known for each input. The goal of unsupervised learning is to uncover hidden patterns or structures in the data without the guidance of labeled outcomes. Examples of unsupervised learning tasks include clustering, anomaly detection, and dimensionality reduction.

In summary, the main difference between supervised and unsupervised learning is that in supervised learning the model is trained on labeled data and the goal is to predict the output based on the input, while in unsupervised learning the model is trained on unlabeled data, and the goal is to uncover hidden patterns or structures in the data without the guidance of labeled outcomes.

An example of supervised learning is Classification. Classification is a type of machine learning task that involves predicting a categorical label for a given input data. The goal is to assign the input data to one of a predefined set of classes or categories, using a set of numerical or categorical attributes that describe the input. it's a supervised learning task and there are different types of classification algorithms that can be used depending on the characteristics of the data and the specific requirements of the task. Examples of classification tasks include image classification, sentiment analysis, and spam detection.

An example of unsupervised learning is anomaly detection. Anomaly detection is used to identify unusual or abnormal patterns in the data. In this task, the algorithm is trained on normal data, and it learns to identify patterns in the data. Then it can be used to detect any data points that deviate from this data.

2.2.4 Types of neural networks

There are several different types of neural networks and each are designed for a specific target. We discuss the most used ones in the following.

Feedforward Neural Networks (FFNNs)

A Feedforward Neural Network (FFNN) is a type of neural network that consists of an input layer, one or multiple hidden layers, and an output layer. The key characteristic of FFNNs is that the information flows in one direction, from the input layer to the output layer, without looping back.

The input layer takes in the input data, and each subsequent layer processes the data using a set of computations (also called activation functions) to produce output data. The output data from one layer is then passed as input to the next layer, until the output layer is reached. The output layer produces the final output of the network, which can be a prediction, classification, or a value.

FFNNs can be used for a wide range of tasks such as image classification, speech recognition, natural language processing, and many more. They are commonly used in supervised learning tasks, where the network is trained with labeled data, to learn the mapping between inputs and outputs.

In summary, Feedforward Neural Networks (FFNN) are a type of neural network that consist of layers of neurons, where information flows in one direction, from input layer to output layer. They are commonly used for a wide range of tasks such as image classification, speech recognition, natural language processing and many more, and are commonly used in supervised learning tasks.

Convolutional Neural Networks (CNNs)

A Convolutional Neural Network (CNN) is a type of neural network that is primarily used for image and video recognition tasks. The main feature of CNNs is the use of convolutional layers, which scan the input image with a small matrix called a filter or kernel and extracts features such as edges, textures, and shapes. These filters are then used to create feature maps, which are then processed by multiple layers to extract higher-level features.

One of the key advantages of CNNs is that they are able to effectively handle the high dimensionality of image data by using a technique called pooling, which reduces the spatial size of the feature maps while retaining the most important information. This helps to reduce the number of parameters in the network and make it more computationally efficient.

Another important aspect of CNNs is that they are designed to be translation invariant, meaning that the features they learn are not affected by the position of the object in the image. This allows CNNs to recognize objects even when they are not perfectly centered in the image.

In summary, CNNs are neural networks that are primarily used for image and video recognition tasks, they use convolutional layers to extract features from images, pooling to reduce the spatial size of the feature maps, and are designed to be translation invariant which makes them more robust to the position of the object in the image.

Recurrent Neural Networks (RNNs)

A Recurrent Neural Network (RNN) is a type of neural network that is designed to process sequential data, such as time series data or natural language. The key feature of RNNs is that they have a "memory" component, which allows the network to remember previous inputs and use them to inform the processing of future inputs. This memory component is implemented using a set of "recurrent" connections between the neurons in the network, which allow information to flow through the network over multiple time steps.

RNNs can be used for a variety of tasks such as text generation, speech recognition, language translation and many more. They are popular in natural language processing tasks because they can process sequential data such as sentences and paragraphs where the meaning of a word is dependent on the context and previous words.

In summary, RNNs are neural networks that are designed to process sequential data by allowing information to flow through the network over multiple time steps, allowing the network to maintain an internal "memory" of previous inputs.

Generative Adversarial Neural Networks (GANs)

Generative Adversarial Networks (GANs) [19] are a class of deep learning models that are designed to generate new, previously unseen data that is similar to a given training set. They consist of two main parts: a generator network and a discriminator network. The generator network is trained to create new data samples that are similar to the training set, while the discriminator network is trained to distinguish the generated samples from the real samples in the training set.

During training, the generator produces new data samples, and the discriminator evaluates them to determine whether they are real or fake. The generator's objective is to create samples that are realistic enough to fool the discriminator, while the discriminator's objective is to correctly identify the generated samples as fake.

The two networks are trained together in an adversarial manner, where the generator improves by trying to generate samples that can fool the discriminator, and the discriminator improves by trying to correctly identify the generated samples. This process continues until the generator is able to produce samples that are indistinguishable from real samples.

GANs have been used to generate new images, videos, audio, and other types of data. Their ability to generate new data that is similar to a given training set makes them a useful tool in applications such as image synthesis, data augmentation, and style transfer.

In summary, GANs are deep learning models that are designed to generate new, previously unseen data that is similar to a given training set. They consist of two main parts, a generator network that is trained to create new data samples, and a discriminator network that is trained to distinguish the generated samples from the real samples in the training set. They are trained together in an adversarial manner, where the generator improves by trying to generate samples that can fool the discriminator, and the discriminator improves by trying to correctly identify the generated samples. GANs have been used to generate new images, videos, audio, and other types of data, and their ability to generate new data that is similar to a given training set makes them a useful tool in various applications such as image synthesis, data augmentation and style transfer.

GAN has proven to be a useful approach to build cryptographic tools in presence of a neural network considered as an adversary.

One should note however that while adversarial learning using GANs is generally used for image processing, it is not limited to that. GANs-Based cryptography is an example of that. Another example is by Dash [27] where the authors investigate whether it is possible to apply adversarial neural networks for playing the popular hide-and-search board game called Scotland Yard. The authors show that neural networks can indeed learn to assess the game like humans and find the hider.

2.3 Neural Networks Based Cryptography

Neural Networks based cryptography is a relatively new approach to the field of cryptography. The first attempts to design cryptographic protocols using machine learning were implemented in the late 90s, however, the security of these methods was not sufficient. The main idea behind neural networks based cryptography is to leverage the powerful representation and generalization capabilities of neural networks to learn and perform specific cryptographic tasks. This is in contrast to traditional cryptographic methods, where algorithms are explicitly designed and implemented to perform specific tasks such as key exchange or data encryption/decryption.

As mentioned above, neural networks-based cryptography is a relatively new field, with few contributions prior to the advent of GANs and the advancement of deep learning. One of the earliest papers on this topic, "Secure Key Exchange Using Synchronized Neural Networks" [6], proposed a method for secure key exchange, but was later found to be vulnerable [18]. Following the development of GANs, a seminal paper was published in late 2016 [1], which proposed a symmetric key encryption system using two neural networks and an adversary. Subsequent research has further studied the security of this method [7], applied it to steganography [8, 23], and improved its security [9, 4].

In a traditional symmetric two-way communication between two parties such as Alice and Bob, the parties are assumed to share a common randomness (the secret key K) as well as an encryption/decrytpion algorithm to use to encrypt and decrypt the data. The encryption/decryption algorithm is assumed to be known to everyone including attackers and eavesdropper. Assuming that we have an eavesdropper Eve that is listening to the communication, she is deemed to be aware of the algorithm Alice and Bob are using to encrypt and decrypt messages but the secret key K is unknown to her. When Alice wants to send a message P, she feeds it in the algorithm along the secret key K in order to encrypt it. The ciphertext C is the output of the algorithm and will be sent publicly to Bob who will use the decryption algorithm in order to decrypt the ciphertext C using the same key K that have been used during encryption by Alice.

A common challenge in this kind of communications is how to share the secret key K without having to meet physically.

There are many classical cryptography methods to share a secret key between two parties e.g. the Diffie-Hellman Key Agreement Protocol [26]. One can also use public key encryption protocols such as RSA [11] to encrypt a secret key and send it to the recepient. However our focus will be on neural networks based protocols in this thesis.

One of the most basic and early forms of cryptography was the key exchange scheme proposed in Kanter [6] during early 2000s and showed how two neural networks can learn to exchange a secret key without using any sort of known cryptography methods. The mechanism will be discussed in Section 2.3.1 below.

2.3.1 Secure exchange of information by synchronization of neural networks

The researchers in [6] pioneered the use of machine learning in cryptography by building neural networks that can learn key agreement. They used this technique to establish a shared, random state from which a secret key could be generated.

The idea consists of having two neural networks called Tree Parity Machines (TPMs) and synchronize them to convey on a key securely in the presence of passive eavesdroppers that have access to the communication but cannot alter or replay messages.

The structure of the two neural networks considered in [6] was a Feed Forward Neural Network consisting of three layers. A single-neuron output layer, K hidden neurons and $K \cdot N$ input neurons as shown in Fig. 2.1.

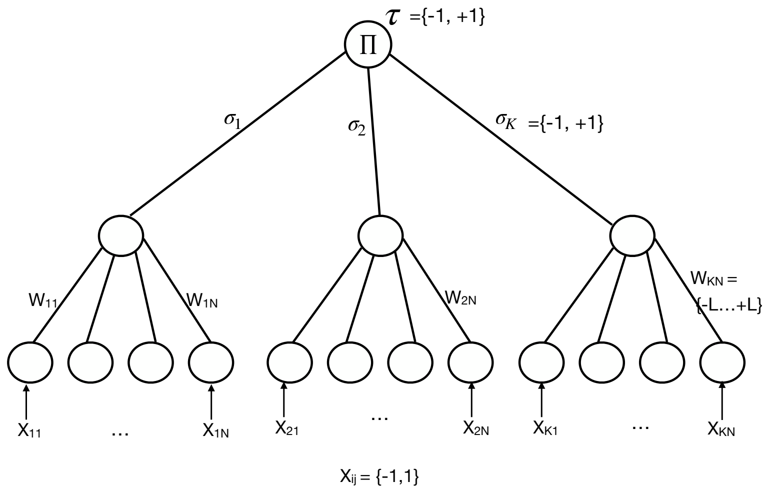

Figure 2.1: Neural Network structure of the parity tree machine.

The leading party (Alice) starts with generating a random input of size N and shares it publicly with the other party (Bob). They both pass them through their neural network and get the output O. They compare their outputs and if they are equal then the two neural networks are shown by the authors to be synchronized (have equal weights vector W) and can use their weights vector W as a secret key [6].

However shortly after this proposal, Klimov [18] three working methods that can break the protocol. We describe the three attacks in the following.

However, this scheme has been shown to be vulnerable against multiple attacks as shown in [18]. The three attacks are summed up below.

Attacks performed on the TPM key exchange protocol

The authors in [18] analysed and showed that the TPMs based key exchange [6] is vulnerable against multiple attacks.

The Genetic Attack This attack uses a biological perspective and genetic algorithms to target two neural networks, Alice and Bob. The method simulates a large population of neural networks that have the same structure as Alice and Bob, and trains them with the same public inputs. The neural networks from the population whose outputs are similar to the two targeted neural networks are then synchronized with the targets, allowing the attacker to read the communication between them.

The Geometric Attack In this attack, the authors simulate each input of the two target neural networks as a K random hyperplanes $X_1, \ldots, X_K$ corresponding to K perceptrons and the weights of each neural network as K points $W_1 ... W_K$ in the N-discrete space $U = \{-L, \ldots, L\}^N$ where $W_i = (w_{i1}, \ldots, w_{iK})$. Concretely, an attacker constructs a neural network with random weights but with the same neural network structure as the target and at each step of training, the weights are updated according to these rules:

- If the two target neural networks have different outputs, the attacker does not update his weights.
- If the two target neural networks have the same outputs and the attacker also has the same output, the neural network's weights will be updated in the normal way.
- If the two target neural networks have the same outputs but not the attacker, then the attacker should find an i_0 that minimises this formula: $\left|\sum_{j=0}^{N} w_{ij}^{C} \cdot x_{ij}\right|$ and updates the weights assuming the hidden bits and the target's outputs.

The authors of [6] however conducted a study on the geometric attack in [28] to prevent it. In their study, they deducted that neural networks with a larger value N of the hidden units will increase the complexity of the geometric attack exponentially and therefore render it difficult to conduct. Brute force attacks and similar attacks are also affected by the size of N.

The probabilistic attack In the probabilistic attack, the attacking Tree Parity Machine is actually a probabilistic Tree Parity Machine this means that the weights are actually probabilistic weights $p_{i,j}(l) = l \in [-L, +L]$ where each probabilistic weight is a probabilistic distribution that represents the probability of the Tree Parity Machine A taking l as a parameter. Then, by passively eavesdropping the inputs $x_{i,j}$ the attacker can use either the Hebbian learning rule or the Monte-Carlo method to update $p_{i,j}(l)$ and end up with identical weights to the parties communicating. This is mainly due to the limited possible values in $[-L, +L]$ which makes them easy to simulate.

The work done in [6] was improved by Prabakaran [29]. The authors worked on a solution for the probabilistic attacks that were used before to break [6]. In order to improve the security and remove the possibility of an attacker passively synchronizing, they introduced queries: instead of generating random inputs, Alice and Bob generate (in turn) at every iteration a set of inputs that is correlated to their respective weights, by doing this, the probability of an attacker passively synchronizing is low because the input is either linked to Alice's weights or Bob's weights. The inputs are generated using a specific algorithm and do not reveal much information about the weights of the neural network and allow to have a mutual influence between A and B which highly reduces the probability of a successful passive attack.

Improvements to the Tree Parity Machine model

The tree parity machine [6] has seen several improvements and attacks since it was first introduced.

One of those improvements is the work done by Reyes [30] where the authors transformed the Tree Parity Machine into a Permutation Parity Machine (PPM) to improve the security.

A Permutation Parity Machine has the same overall neural network structure as a Tree Parity Machine; however the number of parameters and their values are different from the TPM.

A Permutation Parity Machine is defined as a neural network with K hidden units just like the Tree Parity Machine. These units are simple perceptrons (neurons) each having its own input. There are N units with N inputs that take binary values (either 0 or 1).

As for the weights W, they are drawn from a state vector $S \in \{0, 1\}^G$ where G must be greater than $K \cdot N$.

The i^{th} hidden units are calculated using an exclusive or between the weight w and the input x. The final output is either 1 or 0.

Reyes [30] conduct comparative attacks on the TPM and the PPM. The results show that the Permutation Parity Machine performs better against the attacks proposed by Klimov [18] compared to the Tree Parity Machine. The authors then demonstrate that the probability order of a successful attack on the Permutation Parity Machine can be as low as 10^{-20} when the value of N is equal to 16 and the value of G is equal to 128.

The probability of a successful attack is demonstrated to be dependent on the value of G by the following formula: $P_E = \frac{1}{2^{G-1}}$.

We can see that with $G = 128$ we have $P_E = 10^{-20}$.

The result is therefore lighter than the method proposed in [28] as they use a value of 1000 for N which will significantly increase the synchronization time and resources usage compared to this method.

However Seoane [31] demonstrate a successful probabilistic attack on the Permutation Parity Machine which therefore renders the PPM discussed by Reyes [30] non-secure.

Another improvement to the Tree Parity Machine has been done by Salguero [10]. They studied the original TPM and proposed an optimal structure that generated a 512 bits key. This was done by doing over 10 million simulations with different parameters and neural network sizes. All of these simulations were accompanied by a passive adversary trying to synchronize in a passive way along Alice and Bob. In their simulations, the authors showed a case where the neural networks synchronize in a maximum of 6 seconds with a 0% success rate for the attacker. This was done by using the values $K = 8$, $N = 16$ and $L = 2^3$ for the structure of the TPM. The authors finally validate their results with the heuristic rule and the results show that a small change in the parameters would lead to a polynomial increase of the synchronization time and therefore the authors presume that their method is secure enough.

2.3.2 Adversarial Cryptography

Abadi and Anderson [1] were the researchers that spurred the research in the adversarial cryptography topic. Their model shows how to train two neural networks in a GAN setup to learn symmetric encryption without being shown any specific encryption algorithm.

The model consists of two neural networks (Alice and Bob) sharing a common randomness (a secret key K) and their goal is to establish a secure communication in the presence of Eve which will play the role of the eavesdropper and will try to decrypt without the key.

Alice and Bob's goal is to communicate securely by minimizing the error between the original plaintext and Bob's deciphered output text. Eve's goal is to reconstruct the plaintext using the cipher text only i.e. without knowing the secret key.

While in the setup of a GAN, Eve's goal would normally be to distinguish between the ciphertext C and a random value from a certain distribution. However, her goal here is the reconstruction of the plaintext from the ciphertext only. It does not matter if the cipher text contains some meta data that proves that it comes from a certain plaintext.

Neural Networks Structure used

The neural network structure used by Alice, Bob and Eve is described in Table 2.1 below. The neural networks need to have the same structure in order to be able to synchronize their weights and obtain the same weights matrix after training.

Training Process

To train Alice and Bob to communicate securely in the presence of Eve, they need to train their neural networks and reach a common state. When they reach the common state, they are said to be synchronized and Alice's encryptions can be decrypted by Bob. However, as Eve is also in the setup, she will force Alice to generate strong ciphertexts that are tied to the secret key to prevent her from decrypting without the key.

Table 2.1: Summary of the neural network structure proposed in [1] for Alice, Bob and Eve. © 2022 IEEE.

Layer	FC Layer Type	Activation	Filters	Kernel Size	Strides	Padding
1	FC Layer (Dense)	Relu	-	-	-	-
2	Conv1D	Sigmoid	2	4	1	same
3	Conv1D	Sigmoid	4	2	2	Valid for Alice and Bob. Same for Eve.
4	Conv1D	Sigmoid	4	1	1	same
5	Conv1D	Tanh	1	1	1	same

Training will go through multiple iterations and will continue until the neural networks are synchronised. In each iteration, Alice will generate a random plaintext P and a random secret key K. The key is assumed to be shared with Bob with a secure key exchange technique such as the Diffie-Hellman key exchange procotol [26]. The secret key K and the plaintext P are concatenated and fed into Alice's neural network where it will be processed. The output is the ciphertext C and is sent to Bob and intercepted by Eve. Bob will concatenate the ciphertext with the secret key K and feed it into his neural network in order to produce his tentative decryption P_{Bob}. Eve will perform the same process as Bob but as she does not have access to the secret key, her neural network's input will be the ciphertext C only. Her output is her tentative decryption P_{Eve}. The neural networks are trained until Bob's accuracy is as close as possible to 100% and Eve's accuracy is as close as possible to 50%. Eve is not trained to be 100% wrong because in that case, she can just flip the bits and become 100% right in the next iteration. Being 50% wrong is the worst case scenario for an attacker making random guesses in terms of probabilities. Figure 2.3 summarizes the training process.

Loss Functions

The accuracy of the neural networks are assessed through the following loss functions:

- We use $A(\Theta_A, P, K)$ to denote Alice's output on input P, K.
- We use $B(\Theta_B, C, K)$ to denote Bob's output on input C, K.
- We use $E(\Theta_E, C)$ to denote Eve's output on input C.

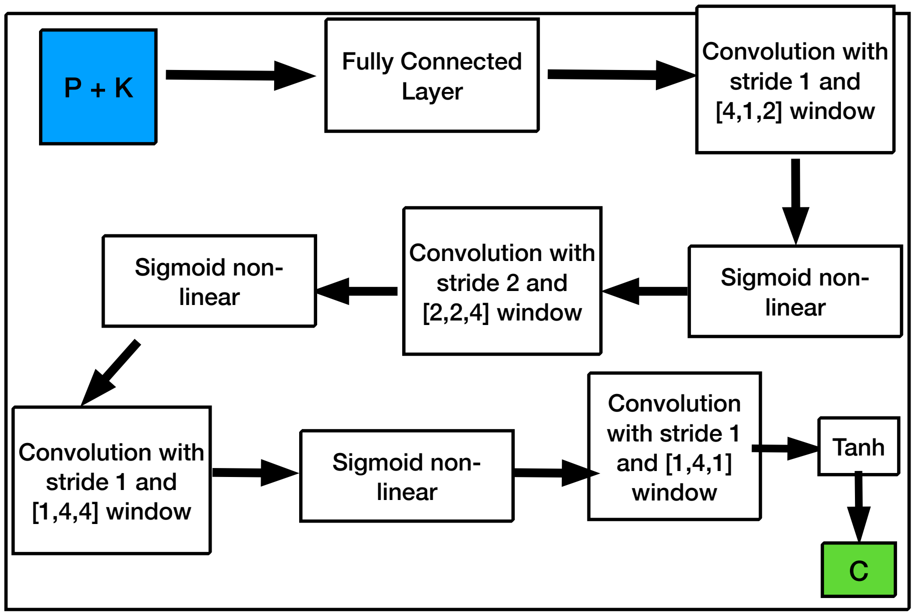
P + K
Fully Connected Layer
Convolution with stride 1 and [4,1,2] window
Sigmoid non-linear
Convolution with stride 2 and [2,2,4] window
Sigmoid non-linear
Convolution with stride 1 and [1,4,4] window
Sigmoid non-linear
Convolution with stride 1 and [1,4,1] window
Tanh
C

Figure 2.2: Encryption flow through Alice's neural network [1]

Where $\theta_{\{A,B,E\}}$ represent the neural networks parameters of Alice, Bob and Eve respectively. We also denote the function $d(x, y)$ to be the $L1$ distance between x and y.

The loss function for Bob is defined below in Equation 2.1.

$$L_B(\Theta_A, \Theta_B, P) = d(P, B(\Theta_B, C, K)) \tag{2.1}$$

Intuitively, the loss function L_B determines how wrong Bob is in his decryption.

Similarly, we define the loss function for Eve in Equation 2.2 below.

$$L_E(\Theta_A, \Theta_E, P) = d(P, E(\Theta_E, C)) \tag{2.2}$$

Intuitively, L_E determines how wrong Eve is when decrypting the ciphertext without the key.

Lastly, we define the loss function for Alice in equation 2.3 below. The loss function is composed of two parts. The first part is Bob's loss and the second part is $(1 - L_E^2)$. The choice behind this loss function is that we want Alice to generate ciphertexts that are easily decrypted by Bob but at the same time difficult to decrypt for Eve.

$$L_A = L_B + (1 - L_E^2) \tag{2.3}$$

After training is done, the parameters/weight that define the state of the neural networks can be saved locally and loaded in order to be used for future secure communications.

As stated before, this model is very interesting but one might ask "how does it differ from classical cryptography protocols?" or "what advantages does it provide?". The answer is that

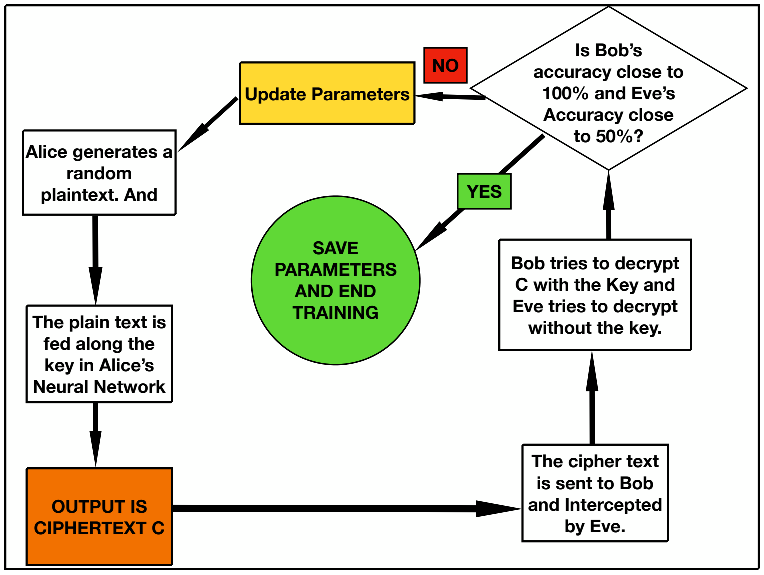
Update Parameters
NO
Is Bob's accuracy close to 100% and Eve's Accuracy close to 50%?
Alice generates a random plaintext. And
YES
SAVE PARAMETERS AND END TRAINING
Bob tries to decrypt C with the Key and Eve tries to decrypt without the key.
The plain text is fed along the key in Alice's Neural Network
OUTPUT IS CIPHERTEXT C
The cipher text is sent to Bob and Intercepted by Eve.

Figure 2.3: Diagram showing the training process of the neural networks [1]

there is no need to build a specific algorithm with detailed steps which is a big difference and an advantage at the same time. The neural networks will work on learning a method on their own without being taught or shown any specific encryption method such as AES. The only drawback is that it takes a considerable amount of time to synchronize two parties for the first time as they do not have pre-saved parameters. Also, as adversarial cryptography models do not rely on any mathematical primitve to get their security, they can be good post quantum cryptography candidates.

2.3.3 Encryption formula and/or Algorithm

Adversarial cryptography models are neural networks constituted of convolutions mostly. These convolutions process and transform the data that has been received by the first fully connected layer. Creating an exact encryption formula is a quite difficult and complicated task as it requires creating a formula for the convolutions which is theoretically possible but will take a considerable amount of time, efforts and resources. As for the algorithm, the neural networks are learning a black-boxed algorithm that depends on the final weights and it would be difficult to get one.

2.3.4 Perfectly Secure Adversarial Encryption

The model described before has been shown to produce ciphertexts that contain information about the plaintext and/or the secret key [7]. Therefore, the authors in [4] modified the neural network structure and the training process with the aim to produce ciphertexts that are secure and do not leak information about the plaintext and/or the key. The key modifications shown in [4] to improve the security are described below.

In addition to Eve, a neural network modeling the threat that an attacker could decrypt the ciphertexts without the secret key, two more neural networks have been introduced. The first one corresponds to an attacker that receives the ciphertext and the secret key and therefore can easily decrypt the ciphertexts generated by Alice. Alice's neural network in return will be forced to generate more complicated ciphertexts that do not entirely rely on the secret key but also the neural network structure of Alice and Bob. The authors conclude that adding an aggressive attacker that has access to leaked secret keys pushes Alice to learn a mapping that does not entirely rely on the secret key but also on the parameters of their neural networks and therefore produce stronger ciphertexts. We could then assume that the neural network parameters contribute to the mapping from plaintext to ciphertext and vice versa. The other additional threats refers to an attacker that tries to tell fake and real ciphertexts apart. This neural network receives a plaintext, its corresponding ciphertext and a randomly generated ciphertext and tries to tell which ciphertext corresponds to the plaintext. This pushes Alice to generate ciphertexts that are indistinguishable from randomly generated ones and therefore makes sure that no information can be extracted from these ciphertexts that are related to the plaintext and/or the secret key.

Apart from more aggressive attackers which seem to be pushing Alice to generate better ciphetexts as shown in [4, 7], The authors in [4] also modified the structure of each neural network. The new neural network structure they used is shown in Table 2.2

Table 2.2: Neural network structure proposed in [4]. The Resblocks [5] in their model contain two identical convolutional layers. © 2022 IEEE.

Layer#	Layer Type	Activation	Filters	Kernel Size	Strides
1	FC Layer (Dense)	ReLU	-	-	-
2	Resblock*	Sigmoid	2	2	1
3	Conv1D	Sigmoid	4	4	2
4	Resblock*	Sigmoid	4	4	1
5	Conv1D	Tanh	1	1	1

The ciphertexts generated by the neural networks are information theoretic secure as they are shown in [4] to be the result of the XOR operation between the plaintext and the secret key. Equation 2.4 taken from [4] shows a sample XOR operation between the plaintext P and the secret key K performed by the neural network (NN). Both the plaintext and ciphertext have a size of 42 bits. In their example, we can see that the first bit p_1 from the plaintext has been XOR-ed with the second bit of the secret key k_2, the second bit of the plaintext with the seventh bit of the key, etc.

$$NN\left(P, K\right) = \begin{bmatrix} p_1 \oplus k_2 \\ p_2 \oplus k_7 \\ p_3 \oplus k_9 \\ p_4 \oplus k_{14} \\ p_5 \oplus k_{25} \\ p_6 \oplus k_{21} \\ p_7 \oplus k_{19} \\ \cdots \\ p_{41} \oplus k_{39} \\ p_{42} \oplus k_{11} \end{bmatrix} \tag{2.4}$$

The authors in [9] took a different approach by removing all the attackers and keeping only one Attacker that receives two plaintexts and one ciphertexts in order to differentiate between the two plaintexts and tell which one has been encrypted to the given ciphertexts. While their method has been shown to be effective at learning the OTP, the results of the model given in [4] are slightly better in terms of successful communications.

2.3.5 Secret Sharing

In a secret sharing system, a secret is distributed among a user set U such that authorized subsets of users can reconstruct the secret, and unauthorized set will not learn anything. Let Γ be a subset of the power set, 2^U , that specifies the subsets of users that form an authorized set; *i.e.*, the set of their shares can recover the secret. A subset $F \subset U$ which is not in Γ, *i.e.* $F \notin \Gamma$, is called an unauthorized set and the set of shares $(S_u)_{u \in F}$ will be independent of secret S. The collection of unauthorized sets is denoted by $\mathcal{F}$. Note that, in our model $\Gamma \cap \mathcal{F} = \emptyset$ and $\Gamma \cup \mathcal{F} = 2^U$. A formal definition of secret sharing [32] is as follows.

Definition 1 (Secret Sharing Scheme)**.** Let U be a set of n users labeled by $[n] = \{1, 2, \ldots, n\}$. Let $(\Gamma, \mathcal{F})$ denote an access structure on these n users with $\mathcal{F} = 2^U \backslash \Gamma$. A secret sharing scheme Π for an access structure $(\Gamma, \mathcal{F})$ consists of a pair of algorithms $(Share, Rec)$. $Share$ is a randomized algorithm that gets as input a secret S (from a domain of secrets $\mathcal{S}$ with at least two elements), Γ and the number of parties n, and generates n shares $(S_1, \ldots, S_n) \longleftarrow Share(S)$. Rec is a deterministic algorithm that gets as input the shares of a subset B of parties and outputs a string. The requirements for defining a secret sharing scheme are as follows:

- *Correctness:* If $\{S_u\}_u \leftarrow Share(S)$ for some secret $S \in \mathcal{S}$, then for any $B \in \Gamma$, we always have $Pr[Rec(\{S_u\}_{u\in B}) = S] = 1$.

- *Secrecy:* Let $\{S_u\}_u = Share(S)$. For $F \in \mathcal{F}$, let $S_F = \{S_u\}_{u\in F}$. Then, for any $s_0, s_1 \in \mathcal{S}$ and for any distinguisher D with output in $\{0, 1\}$, it must hold that

$$|Pr[D(Share(s_0)_F) = 1] - Pr[D(Share(s_1)_F) = 1]| \leq \epsilon.$$

First information theoretic secret sharing for threshold access structures was proposed by Shamir [33] and Blakley [34]. Later, threshold secret sharing was extended to the case of general access structure in [35] and also to different types of important access structures like hierarchical [36, 37, 38], compartmented [36] etc.. It is well known that for information theoretic secret sharing the share size is at least the secret size. Krawczyk [39] proposed a computationally secure secret sharing to reduce the share size. Secret sharing for image data was introduced in [40].

The authors in [41] modeled the secret sharing problem as a classification problem and built GANs based secret sharing scheme. Their model contains a Generator and a Discriminator that compete against each other. The Generator takes as input a secret S and outputs m shares. The discriminator is fed m real shares and m fake random shares and has to tell which ones are real and which ones are not. The training continues until the generator is producing shares that are indistinguishable from random ones and the discriminator is not able to differentiate between them.

A very recent work [42] addresses the construction of progressive secret sharing. Their technique assigns multiple weights to model parameters for progressive recovery. Actually, they encode their model parameters using polynomial based threshold secret sharing such that a hierarchy is achieved among the set of shareholders. The sum of the weights needs to be higher than a threshold value to recover the secret.

2.3.6 Adversarial Steganography

In addition to adversarial encryption, other researchers [23, 8] pushed the idea of the model proposed in [1] in order to build a neural networks model that can learn steganography.

In their models and similarly to the work by Abadi [1], Alice will use an image and a secret text as input to her neural network. Alice's output will be the steganographic image that Bob is going to try to extract the secret texts from. A different image/secret-text combination is used at every training iteration in order to prevent Alice and Bob from learning an algorithm specific to one particular image or secret text.

2.3.7 Adversarial Cryptography Based on the Topology Evolving Neural Networks

The authors of this work [24] wanted to build a model based on a new topology called Spectrum-Diverse Neuroevolution with Unified Neural Models [43] which is basically a type of neural network structure that can evolve by adding or removing neurons to/from its structure. So concretely, in [24] they do not use a fixed neural network structure but a structure that can evolve on the go. however the training process and the concept is the same as the original adversarial cryptography model proposed by Abadi [1]. The results from [24] show that it is possible to implement such neural networks and they can evolve and learn a symmetric encryption protocol.

2.3.8 GAN-Based Key Secret-Sharing Scheme in Blockchain

The authors of this work [44] implement a secure key sharing scheme based on GANs. The idea consists of transforming the text of a private key into an image which will be the original image for the GAN. The original image is then divided into several sub-images and each of them is encoded using DNA coding. Finally, the proposed scheme is trained to extract the secret key using the encoded sub-images. This scheme helps lower the hardness of recovering a lost private key in block chain.

2.3.9 Generative Adversarial Privacy

Training neural network models requires having on hand a lot of data. This data is generally is difficult to acquire due to privacy problems. A solution that is often used is to anonymize the data by removing any identifying details like names, unique identifying numbers, etc. However recent attacks such as in [45, 46] show that it is possible to deanonymize the data and link it to its original holders.

This is where the work Chong [47] comes into play, through what they called Generative Adversarial Privacy (GAP) the authors built a model that can protect the data and anonymize it properly while preserving its utility.

The model is composed of two learning blocks: A privatizer that learns to process the public data in order to output a sanitized version of it and an adversary that tries to learn private data from the public data. This is done through competing in a constrained minimax zero-sum game. The privatizer trains on minimizing the adversary's performance and the adversary tries to find the best strategy to maximize its performance. A loss function is used to measure the efficiency of the adversary.

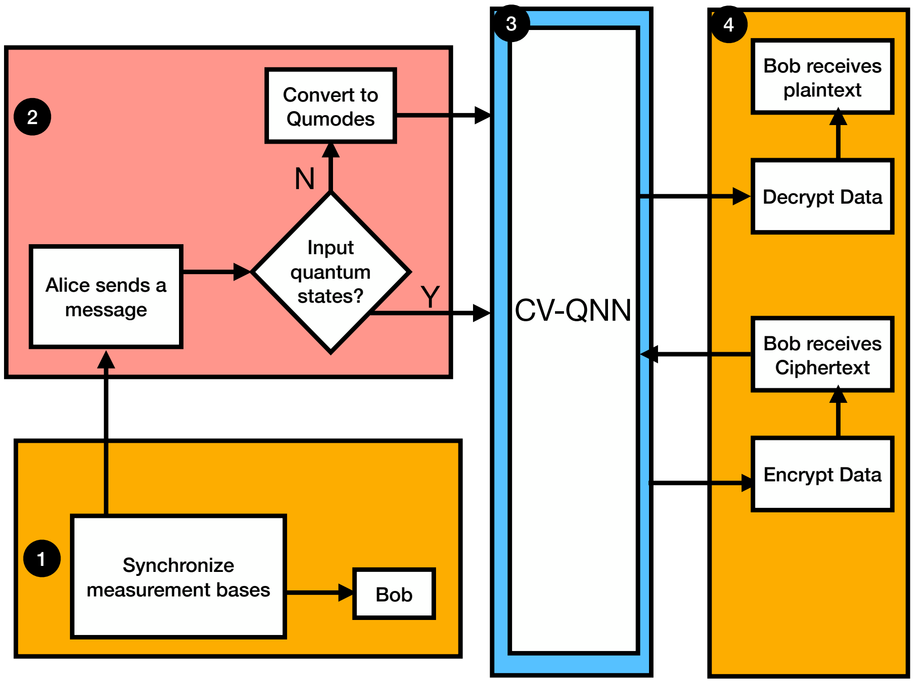

Figure 2.4: Diagram showing the training process of the neural networks [2]

2.3.10 An Approach to Cryptography Based on Continuous-Variable Quantum Neural Network

While Abadi [1] used classic neural networks for their setup and training, Shi [2] did a similar work but using another approach based on Quantum Neural Networks. The neural networks learn to encrypt plaintexts in an adversarial setup. The training starts with creating a classical neural network that can theoretically do the specified task (Encryption, Classification, etc). The model is optimized with the Adam algorithm [48] and the authors perform their experiments using the Strawberry Fields32 tool. There are two neural networks with the same structure, and the authors adopted a 3-layer (Input Layer, Hidden Layer, Output layer) structure.

The communication is between Alice and Bob and consists of four stages as illustrated in Figure 2.4:

- The first stage is to obtain Legitimate Measurement bases for Alice and Bob.
- The second stage preprocesses and transforms the data into quomodes.
- The third stages handles the key preparations.
- The last stages is the communication stage where data is encrypted and decrypted.

Chapter 3

Multi-Party Adversarial Encryption

3.1 3-Party Adversarial Cryptography

3.1.1 Proposed model, Training, Results

Our motivation behind experimenting with a 3-party communication is to see if Alice can learn to communicate with more than one neural network or not. Communicating with more one party is very common in real life situations and there can be cases where Alice will need to synchronize with more than one party.

The approach we took to test a 3-party communication is straight forward. We added a third neural network to the setup as shown in Figure 3.1. Then, in order to make Alice take Charlie into consideration, we need to add Charlie's loss to Alice's overall loss so that she is penalised when Charlie's decryption is bad. As for Charlie's loss, it is the same as Bob's and is shown in Equation 3.1 below.

$$L_{Ch}(\Theta_A, \Theta_{Ch}, P) = d(P, Ch(\Theta_{Ch}, C, K)) \tag{3.1}$$

Where Θ_{Ch} represents Charlie's neural network parameters and $Ch(\Theta_{Ch}, C, K)$ represents Charlie's output on ciphertext C and key K using the parameters Θ_{Ch}. As for Alice's, her new loss L_{A_2} it is defined in Equation 3.2 below.

$$L_{A_2} = L_B + L_{Ch} + (1 - L_E^2) \tag{3.2}$$

The training setup is the same done in [1], explained in Section 2.3.2 and shown in Figure 2.3 with the Addition of Charlie. Concretely, Alice is assumed to have already shared a secret key with Bob and Charlie. The key K and the plaintext P are fed into Alice's neural network in order to produce C, the ciphertext which is sent to Bob and Charlie. Eve will also intercept the Ciphertext. Bob and Charlie will input K and C to their neural network and produce $P_{Bob}, P_{Charlie}$ respectively

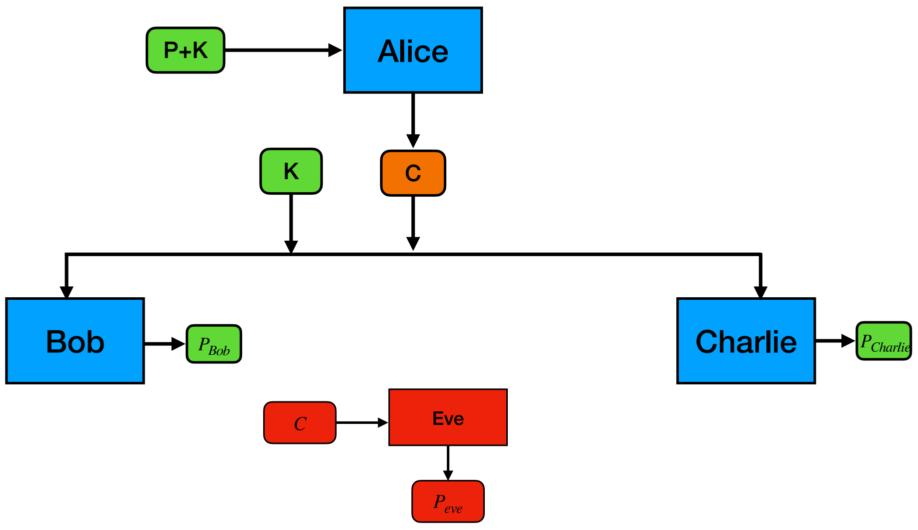

Figure 3.1: Setup of a 3-Party Adversarial communication between Alice, Bob and Charlie. Alice generates a random plaintext P and a random key K which is assumed to be shared with a key exchange scheme with Charlie and Bob. Alice receives P, K as input and produces the ciphertext C. C, K are fed to Bob's and Charlie's neural networks so that they produce P_{Bob}, $P_{Charlie}$ respectively. Eve receives the ciphertext C and produces P_{Eve}.

which is their tentative decryption using the key. Eve will input C to her neural network and produce her tentative decrytpion P_{Eve}. The decryption accuracy is shown in Figure 3.2.

3.1.2 Communication scenarios

In a three party communication, multiple scenarios are possible. In our original work, we have considered three different scenarios which are defined below.

First Scenario

This is the basic scenario where all the communicating parties (Bob, Charlie) are synchronized with Alice in order to build a group communication. In this scenario, encrypted messages sent to/from any party can be decrypted and read by the others. In this scenario, Bob can send messages directly to Charlie and vice-versa.

Second Scenario

In the second scenario, Alice synchronises with Bob and Charlie separately in order to gain secrecy between Bob and Charlie. Alice will need to use a unique set of parameters to synchronise with Bob and another unique set of parameters to synchronise with Charlie. In this scenario, Bob and Charlie cannot communicate with each other directly.

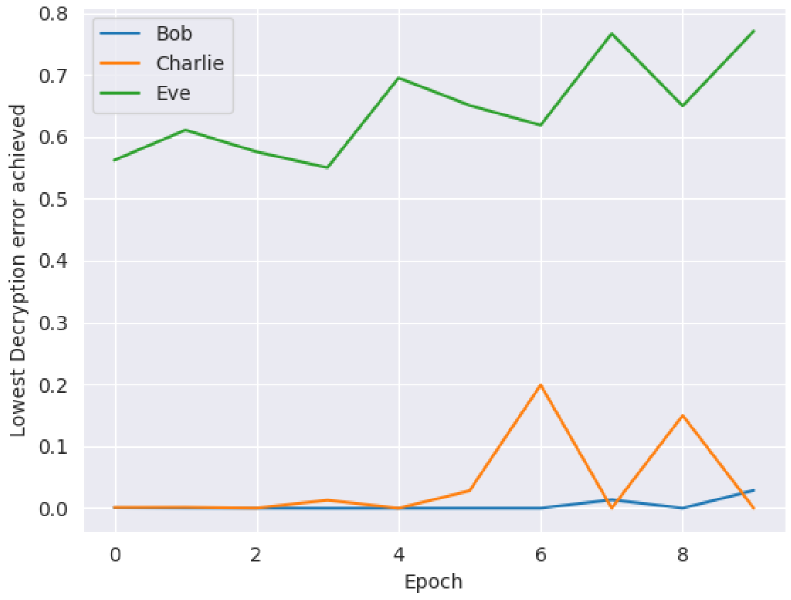

Figure 3.2: Results of Training Alice to train with Bob and Charlie [3]

Third Scenario

The third scenario is useful when there is no physical or wireless communication possible between Alice and Bob. To solve this issue, Charlie is assumed to have a connection with both Alice and Bob and will act as a bridge between them. Charlie will then synchronise his neural network with Alice and Bob. When Charlie receives a ciphertext from Bob or Alice, he will decrypt it using the parameters used to synchronise with the sender, encrypt it again using the parameters used to synchronise with the receiver and forward the result to the receiver. This scenario can be useful with IoT devices for example where devices are often too far from each other to communicate together directly.

The three scenarios are illustrated in Figure 3.3

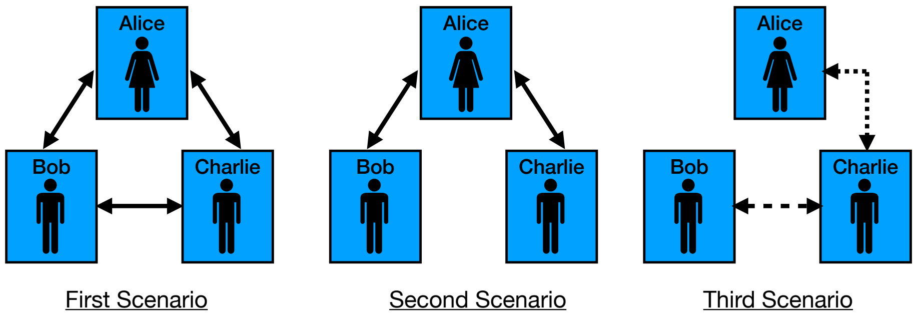

Figure 3.3: Thre three communication scenarios.

3.2 Multi Party Adversarial Cryptography

Our Multi-Party adversarial cryptography scheme is an extension of our original 3-party model presented in Section 3.1. As the original model [1] which our 3-party model [3] is based on has been shown to be weak against a few statistical attacks [7], we updated the model to produce perfectly secure ciphertexts based on the methods shown in [9, 4]. Our new proposed model shows how to communicate with more than two parties and introduces more attackers that push Alice to generate strong ciphertexts that are not prone to the statistical attacks shown in [7]. The attackers are inspired from the ones used in [9, 4].

In our multi-party communication, we do not use the model and training process proposed in [1] but the one proposed in [4] as it provides neural networks that are able to generate perfectly secure ciphertexts.

3.2.1 Training Process

Alice's goal is to synchronise with N parties that have the same structure as her. Alice also needs to protect her communication from different attackers. These attackers all share the same neural network as Alice. Each neural network's loss function L_{NN} is defined in Equation 3.3 below.

$$L_{NN}(W_{NN}, P, K) = d(P, NN(W_{NN}, C, K)) \tag{3.3}$$

Where P is the plaintext, K is the secret key, C is the ciphertext, W_{NN} are the neural network parameters of the neural network NN, d is the L_1 distance and $NN(W_{NN}, C, K)$ is the neural network's output on input C and K using the parameters W_{NN}.

Alice has to take into consideration every neural network in the setup and therefore Alice's loss function is going to be the sum of the losses of all the neural networks in the setup. Alice's loss function is shown in Equation 3.4.

$$L_{Alice} = \sum_{i=1}^{N} L_{NN_i} \tag{3.4}$$

Equation 3.4 contains the sum of the losses of every neural network in the setup and allows Alice to generate ciphertexts that can be decrypted by them at the end of training. However, without any attacker to compete against, the plaintext-ciphertext mapping is going to be weak as shown in [9] despite the communication being successfully established. We tackle this issue by adding multiple attackers to the setup similarly to the setups in [4, 9]. These attackers will to push Alice to generate perfectly secure ciphertexts that are resistant to different cryptographic attacks.

3.2.2 Choice of symmetric keys

Similarly to the model in [1], the keys are taken for granted and assumed to have been exchanged via a secure channel or key exchange protocol. The choice of the method to exchange keys is left to the user. During training, the neural networks are trained with random keys and a new random key is generated during each training iteration. This allows the neural networks to learn an encryption scheme that does not depend on a specific key but works with any type of key. In a real world scenario, the networks can use a secure channel or a key exchange scheme to agree on a key and use it for as long as they wish. In the event that the key is leaked or needs to be changed, they need to exchange a new key. Another solution is to exchange the key once and have a hash function $H(K, C)$ that transforms the key K based on a common counter C that is updated every time the neural networks need to update their key. This will allow them to have a new key without having to rely on an algorithm such as the diffie hellman protocol [26] to generate a new key.

3.2.3 Attackers used in the setup and Overall Architecture

The four attackers used in our model and the types of attacks that they perform are described as follows:

- Attacker 1: Has access to the ciphertext only and tries to decrypt without the key as proposed in [1]. This is the most basic attacker which pushes Alice to generate ciphertexts that rely on the secret key and prevent Alice from learning a plaintext-to-ciphertext mapping that is too simple. This neural network has the same structure as Alice.

- Attacker 2: Has access to the ciphertext and the secret key and learns to decrypt with the key. This attacker pushes Alice to generate ciphertexts that rely not only the secret key but also the neural network parameters as shown in [4]. This neural network has the same structure as Alice.

- Attacker 3: Receives a plaintext P, its corresponding ciphertext C and a random ciphertext C' and has to determine which ciphertext belongs to P. This attacker outputs two probabilities: π_1, the probability that C is a ciphertext for P and π_2 the probability that C' is a ciphertext for P. This attacker pushes Alice to generate ciphertexts that are indistinguishable from randomly generated ones as shown in [4].

- Attacker 4: This attacker receives two plaintexts P_1, P_2 and a ciphertext C and has to tell which plaintext has been encrypted to C as proposed in [9]. This attacker outputs two probabilities: π_1, the probability that P_1 is the plaintext that corresponds to C and π_2, the probability that P_2 corresponds to C. This Attacker pushes Alice to generate ciphertexts that are secure against chosen plaintext attacks as shown in [9].

These attackers are neural networks and are trained to perform different attacks. Even if one Attacker might look stronger than another one, it can only perform the attack it has been trained for. As a result, we use different attackers so that Alice can learn to produce ciphertexts that are resistant to multiple Attacks. Figure 3.4 below shows the overall multi party model including the four attackers.

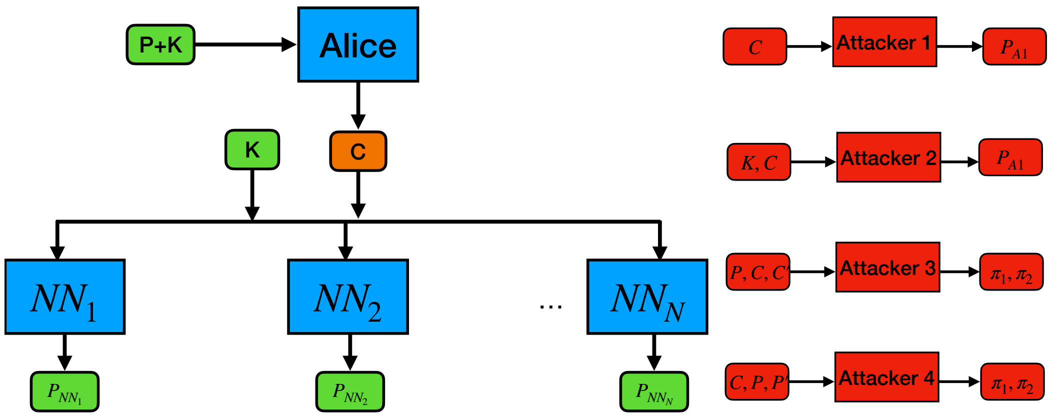

Figure 3.4: Overall architecture of the multi-party adversarial encryption model

We can see in Figure 3.4 that Alice receives as input a plaintext P and the secret key K and outputs C, the ciphertext. All the neural networks that are communicating with Alice $(NN_1 \cdots NN_N)$ receive as input the ciphertext C and the key K and produce their decrypted text $P_{NN_1} \cdots P_{NN_N}$. Attacker 1 receives the ciphertext and tries to decrypt it and output P_{A1}. Attacker 2 receives C, K and tries to decrypt with the key outputting P_{A2}. Attacker 3 receives a plaintext P, its corresponding ciphertext C and a random ciphertext C' and output two probabilities: π_1 the probability that C is a ciphertext for P and π_2 the probability that C' is a ciphertext for P. Lastly, Attacker 4 receives a ciphertext C its corresponding plaintext P_1 and a random plaintext P_2 and outputs two probabilities: π_1 the probability that P_1 is the plaintext that was encrypted to C and π_2 the probability that P_2 is the plaintext that was encrypted to C.

3.2.4 Loss functions used for the attackers

$Attacker_1$'s loss function is defined in Equation 3.5 and similarly to the other neural networks, it is the L_1 distance between the plaintext and $Attacker_1$'s output.

$$L_{A1}(W_{A1}, P) = d(P, A1(W_{A1}, C)) \tag{3.5}$$

Table 3.1: Neural network structure used for *Attacker*$_3$ and *Attacker*$_4$ in order to produce probabilities.

Layer#	Layer Type	Activation	Filters	Kernel Size	Strides
1	FC Layer (Dense)	ReLU	/	/	/
2	Resblock	Sigmoid	2	2	1
3	Conv1D	Sigmoid	4	4	2
4	Resblock	Sigmoid	4	4	1
5	Conv1D	ReLU	1	1	1
6	FC Layer (Dense)	Softmax	/	/	/

Attacker$_2$'s loss function is defined in Equation 3.6 and similarly to the other neural networks, it is the L_1 distance between the plaintext and *Attacker*$_2$'s output. The difference from *Attacker*$_1$ is that *Attacker*$_2$ has also access to the secret key that we suppose that it was leaked to him.

$$L_{A2}(W_{A2}, P) = d(P, A2(W_{A2}, C, K)) \tag{3.6}$$

As for *Attacker*$_3$ and *Attacker*$_4$, these two neural networks are making classifications and therefore needs to have some changes in their neural network structure in order to output probabilities. Basically, we keep the same neural network structure as Alice but add an additional softmax-activated fully connected layer as a last layer in order to output probabilities instead of a bistream. The new neural network structure for *Attacker*$_3$ and *Attacker*$_4$ is shown in Table 3.1.

We notice that the only change is an additional softmax-activated fully connected layer at the end of the neural network structure.

As for the loss function for *Attacker*$_3$, it is the binary cross-entropy. Given N plaintexts $\left[P_{(0)}, P_{(1)}, ...P_{(N-1)}\right]$, and two sets of N ciphertexts $\left[C^1_{(0)}, C^1_{(1)}, \cdots C^1_{(N-1)}\right]$, $\left[C^2_{(0)}, C^2_{(1)}, \cdots C^2_{(N-1)}\right]$ we define the loss function L_{A3} for *Attacker*$_2$ in Equation 3.7 below.

$$L_{A3} = -\frac{1}{N}\sum_{i=0}^{N-1}\sum_{j=1}^{2} y^j_{(i)} \log\left(\pi^j_{(i)}\right) \tag{3.7}$$

Where $y^j_{(i)} = 1$ if $P_{(i)}$ is the plaintext of $C^j_{(i)}$ and 0 otherwise. Intuitively, $\pi^j_{(i)}$ is the probability that $C^j_{(i)}$ is the ciphertext corresponding to the plaintext $P_{(i)}$. Therefore, *Attacker*$_3$ learns by minimizing L_{A3}.

The loss function of *Attacker*$_4$ is similar to the one of *Attacker*$_3$ i.e. the binary cross-entropy. Given N ciphertexts $\left[C_{(0)}, C_{(1)}, \ldots, C_{(N-1)}\right]$, and two sets of N plaintexts $\left[P^1_{(0)}, P^1_{(1)}, \cdots P^1_{(N-1)}\right]$,

$\left[P^2_{(0)}, P^2_{(1)}, \cdots P^2_{(N-1)}\right]$ we define the loss function L_{A4} for $Attacker_4$ in Equation 3.8 below.

$$L_{A4} = -\frac{1}{N}\sum_{i=0}^{N-1}\sum_{j=1}^{2} y^j_{(i)} \log\left(\pi^j_{(i)}\right) \tag{3.8}$$

Where $y^j_{(i)} = 1$ if $C_{(i)}$ is the ciphertext of $P^j_{(i)}$ and 0 otherwise. Intuitively, $\pi^j_{(i)}$ is the probability that $P^j_{(i)}$ is the plaintext corresponding to the ciphertext $C_{(i)}$.

Therefore, $Attacker_4$ learns by minimizing L_{A4}.

Alice's loss function defined in 3.4 needs to be modified in order to take into consideration the four attackers that we added to the setup. Alice's new loss function is defined in Equation 3.9.

$$\begin{aligned} L_{Alice} = \sum_{i=1}^{N} L_{NN_i} + (1 - L^2_{A_1}) + (1 - L^2_{A_2}) \\ -min(L_{A3}, 0.5) - min(L_{A4}, 0.5) \end{aligned} \tag{3.9}$$

We use $min(L_{A3}, 0.5)$ and $min(L_{A4}, 0.5)$ in Alice's loss function in order to prevent Alice from maximising the loss of $Attacker_3$ and $Attacker_4$. Ideally, we want their loss to be equal to 0.5 which, in probabilities, corresponds to making assumptions that are random.

3.2.5 Synchronizing neural networks with different structures

While the neural network structure is required to be the same in the original adversarial encryption model [1], most related works [4, 9] as well as in our contributions, one might wonder about the possibility of synchronizing neural networks with different structures. To answer this question, we tried synchronizing two parties to learn the same encryption algorithm with the original adversarial model [1] however it seems that the receiving party is not able to decrypt the ciphertexts properly. Given that the encryption algorithm is somehow black-boxed and cannot be expressed formally, it is difficult to given a definite answer to why the neural networks are not able to learn the same encryption algorithm. One theory is the network parameters that need to be equivalent or equal for the networks to encrypt and decrypt properly together. Another theory is that the encryption/decryption algorithm is highly tied to how the neural networks structure and how the data is processed and transforms by each layer. Using different neural networks structures (even as simple as removing one layer) causes an imbalance in how the data is processed/transformed and leads to the inability of the neural networks to communicate properly. It would be interesting to explore in the future how the networks are learning the encryption/decryption and to find a way to synchronize neural networks with different structures.

3.2.6 How we deal with the problem of problem of Non-Convexity

The loss functions used in deep neural networks are known to be non-convex. This means that even if two neural networks train with the same dataset, they are not likely nor guaranteed to end up with the same local minima. While this is not a problem for common deep learning tasks such as classifications and it is what makes the neural networks diverse, it can cause a problem for adversarial cryptography models. For example, in a classification task where we have to classify cats and dogs, two solutions might lead to different accuracy scores of the model but if the solutions have a high accuracy score, they will both classify an image of a dog as a dog and will rarely produce a different solution. However, in adversarial cryptography, two neural networks training on the same set of plaintexts and keys are very likely to end up in different local minimas and therefore learning different encryption/decryption algorithms and will not be able to communicate. We tackle this issue by training all the networks together as shown in our model architecture in Figure 3.4.

3.2.7 Training Results

As a proof of concept, we implement our proposed multi-party perfectly secure encryption model [25].

In the implementation, Alice wants to communicate with three neural networks NN_1, NN_2 and NN_3 in the presence of the four attackers shown in Figure 3.4. We implement the model using Tensorflow and Keras.

The following are the hyperparameters used to train our neural network.

- **Datasize**: 64 bits for the plaintexts, keys and ciphertexts.
- **Batch Size**: 256.
- **Number of epochs**: Up to 200 but the training might stop earlier if the receiver has reached 100% accuracy and the prediction accuracy of attackers is close to random guesses.
- **Training steps per epoch**: 300.
- **Learning Rate**: 0.0008.
- **Optimizer**: Adam's optimizer.

Alice and the three neural networks NN_1, NN_2 and NN_3 as well as $Attacker_1, Attacker_2$ have the neural network structure shown in Table 2.1. $Attacker_3$ and $Attacker_4$ have the neural network structure shown in Table 3.1.

We train the neural networks until NN_1, NN_2 and NN_3 are able to decrypt the ciphertexts sent by Alice and $Attacker_1, Attacker_2$ have a decryption accuracy of around 50% which is equivalent to a random decryption where the attacker does not know which bit is correct and which one is not. If we trained them to reach 100% accuracy, they would be able to become 100% right just by flipping all their bits. Therefore having 50% accuracy when decrypting is the worst case scenario from the point of the of an attacker making random guesses.

Figure 3.5 shows the decryption accuracy of the three neural networks NN_1, NN_2 and NN_3 as well as the decryption accuracy of the two attackers $Attacker_1, Attacker_2$ after 155 epochs.

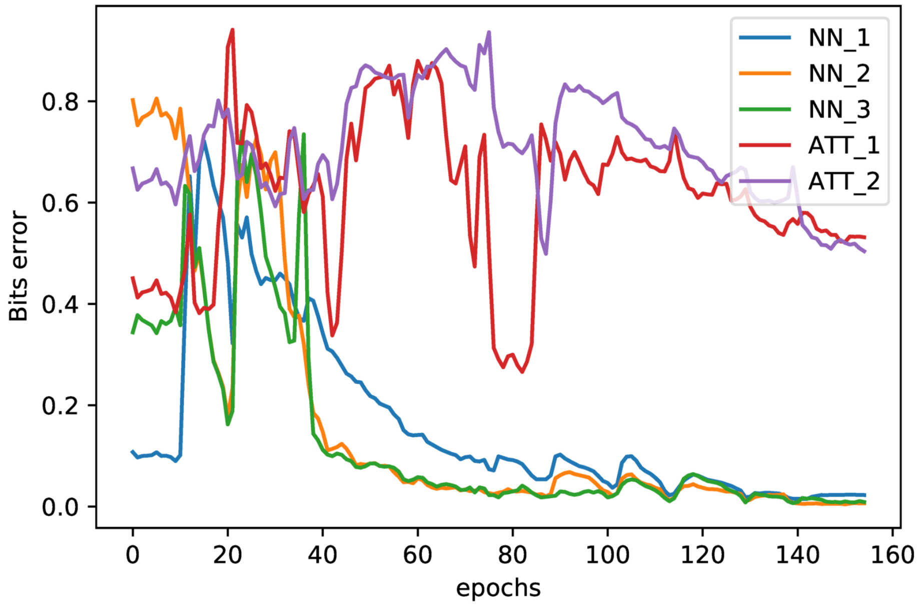

Figure 3.5: Decryption accuracy of the three neural networks NN_1, NN_2 and NN_3 and the two attackers $Attacker_1, Attacker_2$ during training. 0% Bits error means that the neural network produced a plaintext with 100% of the bits correct and 1.0 Bits error means that the neural network produced a plaintext with 0% of the bits correct. © 2022 IEEE.

We can see that the neural networks start with random decryption accuracy at the beginning of training. After 50 epochs, the three neural networks communicating with Alice start getting better at their decryption with around 20% error in their decryption. The two other attackers have a decryption error ranging between 40 and 60%. It is only after 140 epochs that the neural networks finally reach a stable state where the parties communicating with Alice have perfect accuracy while the two attackers $Attacker_1, Attacker_2$ have approximately 50% accuracy which is the training goal for them so that their output is close to random and they cannot tell which bit is wrong and which one isn't.

As for $Attacker_3$ and $Attacker_4$, they were not able to produce correct probabilities from the beginning to the end of training. Figure 3.6 shows the probabilities produced by $Attacker_3$ on real and fake ciphertexts and Figure 3.7 shows the probabilities produced by $Attacker_4$ on real and fake plaintexts. Both of the neural networks are producing 50% probability on real and fake inputs meaning that they are not able to tell which one is real and which one is not.

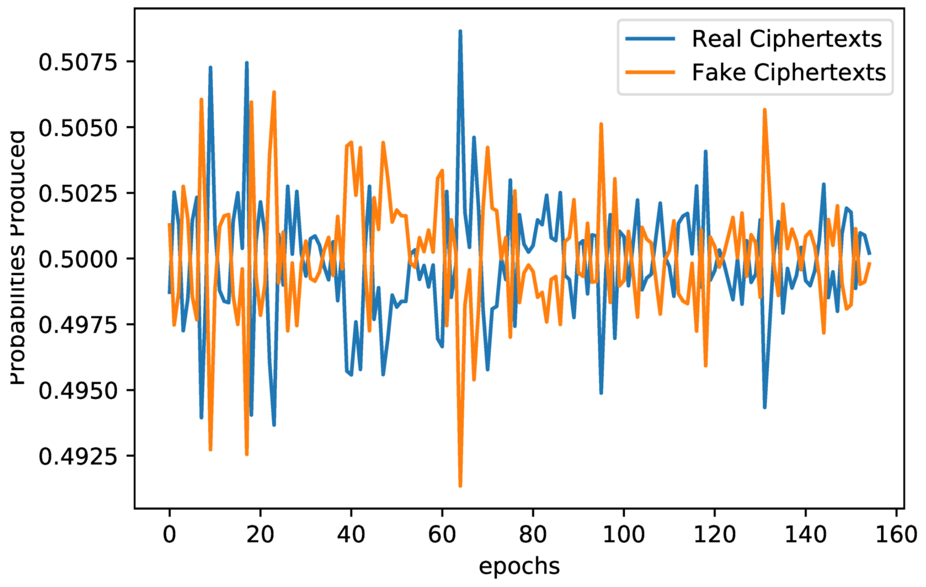

Figure 3.6: Probabilities produced by $Attacker_3$ on real and fake ciphertexts. The attacker is producing probabilities that are close to 0.5 for each of the two inputs meaning that this attacker is not able to distinguish between real and fake ciphertexts to tell which one the original message P has been encrypted to. © 2022 IEEE.

The experimental results show that the ciphertexts generated by Alice are secure against all these attackers as cannot be decrypted without the key and the weights of the neural networks that trained with Alice. Additionally, Figures 3.6 and 3.7 show that the ciphertexts cannot be differentiated from randomly generated ones and contain no information about the plaintexts as $Attacker_4$ has not been able to link the real plaintext to the given ciphertext.

Therefore, we are achieving the same results as the results of the work proposed inresults [4] while allowing more than one party to communicate with Alice. This means that the encryption done by Alice or the Dealer when performing secret sharing will produce outputs that are secure against the aforementioned cryptographic attacks and attackers.

I. Meraouche, S. Dutta, S. K. Mohanty, I. Agudo and K. Sakurai, "Learning Multi-Party Adversarial Encryption and Its Application to Secret Sharing," in IEEE Access, vol. 10, pp. 121329-121339, 2022, doi: 10.1109/ACCESS.2022.3223430 © 2022 IEEE.

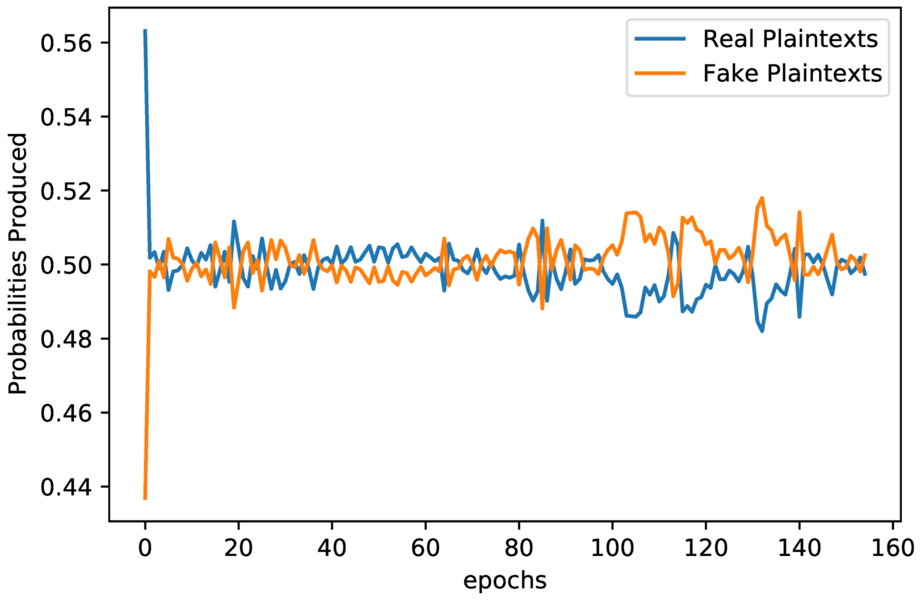

Figure 3.7: Probabilities produced by $Attacker_4$ on real and fake plaintexts. The attacker is producing probabilities that are close to 0.5 for each of the two inputs meaning that this attacker is not able to distinguish between real and fake plaintexts to tell which one has been encrypted to the real ciphertext C. © 2022 IEEE.

3.3 Secret Sharing Scheme based on Multi Party Adversarial Encryption

3.3.1 Our Secret Sharing Scheme

Our proposed secret sharing scheme is based on our model that learns multi-party adversarial encryption proposed in Section 3.2 and uses the second scenario where it is possible to achieve secrecy of the messages exchanged between one or more parties with Alice.

In our secret sharing scheme, we have a Dealer D that has a master secret MS to be divided into N shares $st_1, \cdots, st_N$ and distributed among N shareholders $SH_1, \cdots, SH_N$ such that all the N shareholders are required to reconstruct the master secret MS. That is, we first propose an N-out-of-N secret sharing scheme. Using this construction, we later generalize to propose secret sharing schemes for any general access structure.

The Dealer and the shareholders are all neural networks with the same structure shown in Table 2.1. The Dealer plays the same role of Alice in the proposed multi-party adversarial encryption model and synchronizes with the shareholders as described in the second scenario (*see* Section 3.2). For N shareholders to synchronize with, the Dealer has N sets of parameters $W = \{W_1, \ldots, W_N\}$. Dealer uses one unique set of parameters W_i to synchronize with a unique shareholder SH_i. The Dealer can also be viewed as a server containing N neural networks each synchronizing with one unique shareholder.

Once the synchronization is complete, the Dealer generates N secret keys $K_1, \dots, K_N$ that are going to be used to encrypt/decrypt the data with the N shareholders. We assume that the Dealer has a secure tunnel with every shareholder in order to deliver the secret key to them. The overall setup has the following variables:

- The master secret MS.
- $W_1, \dots, W_N$ denote the parameters which the Dealer has used to synchronize with the N shareholders $SH_1, \dots, SH_N$. Every W_i was used to synchronize with the shareholder SH_i ($i \in [1, N]$).
- We denote the parameters (of the neural network) of the i^{th} shareholder by W_{SH_i}. W_{SH_i} are the result of the synchronization process of the i^{th} shareholder with the Dealer. We note that this W_{SH_i} is equal to W_i after the training.
- W_{SH_i} will be stored by the i^{th} shareholder and W_i will be stored by the dealer.
- $K_1 \cdots K_N$, the secret keys that the Dealer has distributed to the N shareholders $SH_1, \dots, SH_N$. Every K_i is sent to SH_i with $i \in [1, N]$.

Additionally, we define the following functions that we use in the process of creating the shares and the reconstruction of the master secret:

- The function $Enc(W_i, M, K_i)$ denotes the encryption by the Dealer with plaintext input M, key K_i and using the parameters W_i.
- The encryption process consists of passing the message M, the key K_i through the Dealer's neural network and calculating the output of its neural network using the parameters W_i. The output is the encrypted result.
- $Dec(W_{SH_i}, C, K_i)$ denotes the decryption of the input C by the i^{th} shareholder using the key K_i and the parameters W_{SH_i}.
- The decryption process consists of passing the encrypted message C, the key K_i through the i^{th} shareholder's neural network and calculating the output of its neural network using the parameters W_{SH_i}. The output is the decrypted result.

3.3.2 Shares Construction.

In the Setup we have a Dealer (neural network) with parameters $W_1, \dots, W_N$ in synchronization (as described in the second scenario of Section 3.2) with N parties $SH_1, \dots, SH_N$. The Dealer with input master secret MS constructs N shares $st_1, \dots, st_N$ in the following manner.

1. The Dealer generates N random keys $K_1, \ldots, K_N$. Each key K_i is shared with the shareholder SH_i using a secure tunnel.

2. In the first step, the Dealer encrypts MS using W_1, K_1 and computes $S_1 = Enc(W_1, MS, K_1)$. Dealer now proceeds as follows:
   Computes $S_i = Enc(W_i, S_{i-1}, K_i)$ for all $i = 2, \ldots, N$.

3. The dealer sends S_N to SH_N through a secure tunnel and deletes all the information from its own storage.

4. The resulting shares are $st_i = (W_{SH_i}, K_i)$ for $1 \leq i \leq N-1$ and $st_N = (W_{SH_N}, K_N, S_N)$.

3.3.3 Master Secret reconstruction.

The reconstruction procedure is as follows. When all the N shareholders agree to recover the master secret they take the following steps.

The shares construction process is illustrated in Figure 3.8

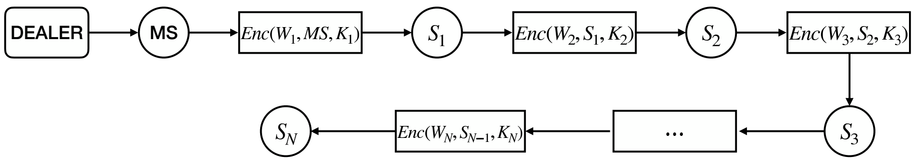

Figure 3.8: The shares construction process

Recall, only the last encryption S_N has been distributed to the shareholder SH_N using the secure tunnel that we assume the Dealer has with all shareholders. The other shareholders have only kept their corresponding neural network parameters and secret keys. The reconstruction of the master secret is done by decrypting S_N in the reverse order:

1. SH_N calculates S_{N-1} by decrypting S_N with his parameters W_{SH_N} and key K_N i.e., $S_{N-1} = Dec(W_{SH_N}, S_N, K_N)$. Then, SH_N forwards S_{N-1} to SH_{N-1}.

2. The process is repeated $N-1$ times where in each step, shareholder SH_i calculates S_{i-1} and forwards it to SH_{i-1} until the first shareholder SH_1 receives S_1 and decrypts it to the master secret MS.

The master secret reconstruction process is illustrated in Figure 3.9

The *correctness* of the recovery of master secret follows immediately from the correctness of synchronization process (i.e., $W_{SH_i} = W_i$ for all i) and the correctness of the decryption algorithms $Dec(W_{SH_i}, S_i, K_i)$ for all i. The *security* property of the above (N, N) scheme follows from the security of the encryptions $Enc(W_{SH_1}, MS, K_1)$ and $Enc(W_{SH_i}, S_{i-1}, K_i)$ for all $i = 2, \ldots, N$. We

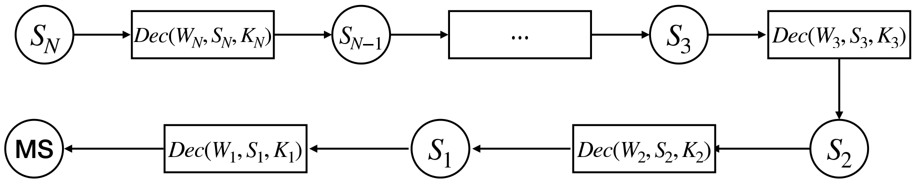

Figure 3.9: The master secret reconstruction process

emphasize that the encryption algorithms are in fact one time pads as described in Section 3.2. Therefore, the N-out-of-N secret sharing scheme we achieve is perfectly secure.

3.3.4 Secret Sharing schemes for General Access structures

We have presented a construction to realize N-out-of-N secret sharing scheme for any value of N. Using this basic construction, we can achieve secret sharing schemes for any general access structure (GAS). Suppose $\Gamma = \{B_1, \ldots, B_r\}$ is a general access structure on a set U of users. The dealer runs a $|B_i|$-out-of-$|B_i|$ secret sharing as described above for every set B_i, $1 \leq i \leq r$. The master secret MS remains the same but the parameters W's and the keys K's are independently chosen for each B_i's. The correctness and secrecy of this construction is evident from the respective properties of the underlying $|B_i|$-out-of-$|B_i|$ schemes.

I. Meraouche, S. Dutta, S. K. Mohanty, I. Agudo and K. Sakurai, "Learning Multi-Party Adversarial Encryption and Its Application to Secret Sharing," in IEEE Access, vol. 10, pp. 121329-121339, 2022, doi: 10.1109/ACCESS.2022.3223430 © 2022 IEEE.

3.4 Limitations of this technique

Neural networks based cryptography is a growing research area and still has some drawbacks. One of these drawbacks is that neural networks are not able to know on their own when it is a good time to stop training and start communicating which causes a problem in real world applications. They need to be monitored during their training and an intelligent entity needs to tell them when they need to stop and start communicating. For GANs-based models that learn cryptography techniques, their training needs to be monitored either by a human that chooses the best state or an automated software that monitors their progress and chooses the best state according to a predefined configuration. This best state is where the Communicating parties are having 100% decryption accuracy and the simulated attackers are no better than attackers making random guesses.

Chapter 4

Learning Adversarial Steganography among more than two parties (3-party case)

4.1 3-Party Adversarial Steganography

While most the contributions that followed up the work by Abadi and Andersen [1] focused on cryptography, some researchers [8, 23] also explored the possibility of enabling Alice and Bob to learn Steganography and the results were successful as shown in Section 2.3.

The authors in [8] show that, due to the problem of non convexity, it is difficult for two neural networks training separately to learn the same steganographic algorithm. Meaning that if Bob trains with a Local Alice to learn steganography and Charlie learns with another Local Alice to learn an algorithm, they are not guaranteed to learn the same steganography algorithm and therefore a communication between them will not be possible. We propose a solution [49] by showing that Bob and Charlie can learn the same steganography algorithm if they learn to communicate with the same Alice. Similarly to our contribution on 3-party adversarial encryption [3], We consider three possible scenarios to train the neural networks. Our approach is also the same used in our 3-party encryption model [3]. We add Charlie's loss to Alice's loss so that Charlie is taken into consideration by Alice.

We show that in our method (presented in Section 4.4.1), two different parties can learn the same algorithm and are able to extract the same plaintext to solve the problem stated in Section 4.4 in [8] where multiple parties are not guaranteed to learn the same algorithm when being on different machines.

The next two scenarios viz. scenario 2 and scenario 3 (presented in Section 4.4.2) show that different 3-party steganographic communications are achievable with similar training time and extraction accuracy as the two-party case. Although, they do not allow all parties to communicate with each other but they can be useful when multi-hop communications are allowed where we allow a party to act as an intermediary.

The content has of this Chapter has been Reproduced with permission from Springer Nature from our original published paper [49].

4.2 Neural Network Structure Used

Similar to the scenarios in [8] and in [1], for Alice we use a random cover image and a random n-bit bit-stream (The secret message) as input. We flatten the cover image and concatenate it with the secret message which is then fed into the neural network. The first layer is a fully connected layer so that the message is mixed with the image. It is then reshaped to a $4 \times 4 \times I * 8$ where I is the width of the cover image. Then we follow with a sequence of four layers which consist of a sequence of convolutions, Batch normalization and ReLU except for the final layer where tanh is used as the activation function. Bob and Charlie use the image output by Alice as their input, their structure consists of four consecutive and fractionally-strided convolutions. All the layers use Leaky ReLU except the final one, which uses *tanh* in order to get a binary output. Eve has the same structure as Bob and Charlie except in the final layer where he has Sigmoid in order to output probabilities whether what he was fed is a normal or steganographic image.

4.3 Training Process & Loss functions.

In order to allow multiple neural networks to learn the same protocol on different machines, we choose one machine that will act as a leader (Alice), and all of the other machines will train with that leader which is be better than training Bob and Charlie with a local Alice running on their own machine. Sharing a leading machine is what allows multiple parties to learn the same algorithm. In our contribution [49], we only focus on communicating among three parties to reduce training time. When training is complete, the three neural network will learn the same embedding and extraction algorithm (i.e the steganography algorithm).

The parameters used to train the neural networks are as follow: θ_{A} for Alice, θ_{B} for Bob, θ_{C} for Charlie and θ_{E} for Eve. The function $E_{\mathbf{A}}(\theta_{\mathbf{A}}, M, P)$ represents the embedding function on the image M and the plaintext P for Alice and with a steganographic image as output and $D_{\mathbf{B}}(\theta_{\mathbf{B}}, S)$ represents the extraction algorithm of Bob on the steganographic image S with output as $\mathbf{P_{Bob}}$. The loss function for Eve is as follow: $D_{\mathbf{E}}(\theta_{\mathbf{E}}, M, S)$ where M is a normal image and S is the steganographic image. Eve's output on an image is the probability of it being a steganographic image.

The distance between the the original plaintext and each extracted text is calculated in order to tell how accurate each neural networks is:

$$d(P, P') = \sum_{i=0}^{N} \mid P_i - P'_i \mid$$

The definition of the loss function for Bob is the euclidean distance d between Bob's output and the Original plaintext:

$$L_B(\theta_A, \theta_B, M, C) = d(P, D_B(\theta_B, E_A(\theta_A, M, C)))$$

When adding Charlie, the third neural network, he will have the same structure as Bob and therefore the same loss function:

$$L_C(\theta_A, \theta_C, M, C) = d(P, D_C(\theta_C, E_A(\theta_A, M, C)))$$

The loss function for Eve is the sigmoid cross entropy loss:

$$L_E(\Theta_A, \Theta_E, M, S) = -y \cdot \log(E(\Theta_E, x)) - (1-y) \cdot \log(E(1-\Theta_E, x))$$

where $y = 0$ if $x = S$ and $y = 1$ if $x = M$.

As in [8]. Alice's loss is the weighted sum of Bob's, Charlie's and Eve's loss on steganographic images, combined with an additional loss term:

$$L_E(\Theta_A, \Theta_E, M, S) = \lambda_A \cdot d(S, M) + \lambda_B \cdot L_B + \lambda_C \cdot L_C + \lambda_E \cdot L_E(\Theta_E, S)$$

where $\Theta_A, \Theta_B, \Theta_C, \Theta_E \in \mathbb{R}$ represent the weight given to each of the respective loss terms. Note that it is mandatory to add the parameters of Charlie or any additional party to the above equation so that they all can learn the same algorithm.

To summarize, the main idea behind hiding and extracting the plain text consists of concatenating the image with the plaintext (secret message of Alice) and passing it through Alice's neural network. The output from Alice's neural network is sent to Bob and Charlie which will again go through their neural networks respectively. The outputs of Bob and Charlie are plaintexts and the neural networks are trained until the plaintexts become equal to Alice's secret message.

4.4 Experiments and Results

Our goal is to train the three neural networks and synchronize them in a way that they learn the same embedding and extraction algorithm. Our three communicating neural networks with the same structure are Alice, Bob and Charlie and Eve will play the role of the eavesdropper. We use the CelebA data set [50] to train the neural networks, a new image and plaintext are used in every step of training.

This section presents the experiments we conducted in three different scenarios to simulate communications between Alice, Bob and Charlie. Our goal is to Bob and Charlie learn the same embedding and extraction algorithm. All the neural networks have the same structure which is defined in section 4.2. Eve receives two images: the real steganographic image and a random one

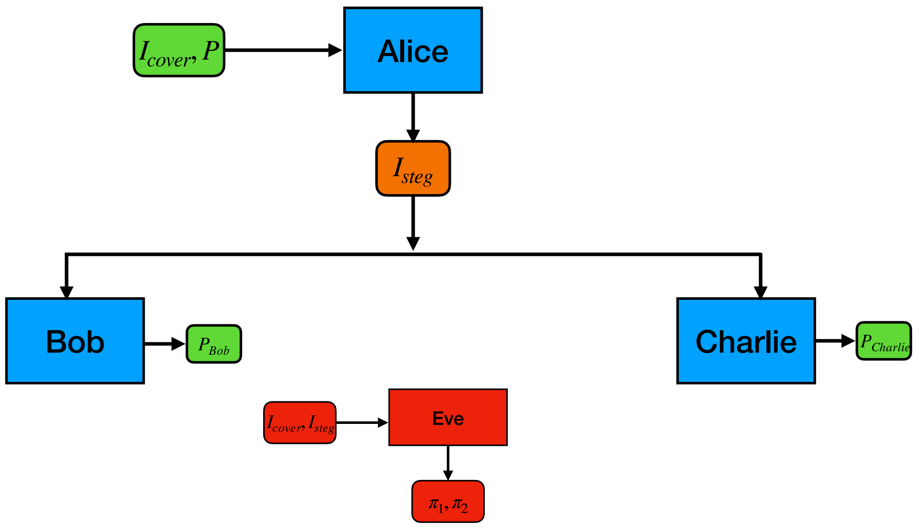

Figure 4.1: Setup of the 3-party steganography model. Alice receives a cover image I_{cover} and a plaintext P to hide inside it. The image is flattened, concatenated with the plaintext and fed into Alice's neural network which will process it in order to produce the steganographic image I_{steg}. The steganographic image is fed into Bob's and Charlie's neural networks and they will try to extract the hidden text by prodycing P_{Bob} and $P_{charlie}$ respecrtively. Eve receives the cover image and the cover image and will output π_1 the probablity that I_{cover} contains the hidden text and $\pi 2$ the probability that I_{steg} contains the hidden text.

and has to tell which one contains the hidden message by outputting two probabilities reflecting which images Eve thinks has the highest probability of containing a hidden text . As for the dataset, we use the CelebA data set [50] to train the neural networks, a new image and plaintext are used in every step of training. The overall model structure is shown in Figure 4.1

4.4.1 First Scenario

In this scenario we solve the issue mentioned in Section 4.4 of [8] to allow multiple parties to learn the same embedding and extraction algorithm simultaneously.

Training process I.

We have Alice, Bob, Charlie in three different machines and Alice will embed random plaintexts inside images from CelebA data set. We see that after around 200 epochs Bob and Charlie will synchronize and extract the same plaintext. The training process is presented in Algorithm 1.

Algorithm 1 Training Process I.

```
1: image_array ← CelebADataSetImages // loads the image data set into an array
       // n refers to the number of images in the data set and synchronized is a Boolean variable
   that indicates if synchronized or not
2: initialize: $i = 0$
3: while i <= n or synchronized do
4:   AliceGetNextImage($i$) //Alice gets the next image from the dataset
5:   AliceGenerateBitstream(100) //Alice generates a random bitstream of 100 bits
6:   AliceGenerateStegoImage($i$) //Alice generates the steganographic image
7:   BobCharlieExtractBitstream(stegoImage) //Bob and Charlie extract the bitstream from the
   steganographic image
8:   EveAnalyzeImages(stegoImage, originalImage) //Eve tries to distinguish between stegano-
   graphic and cover images
9:   UpdateParameters($\Theta_A, \Theta_B, \Theta_C, \Theta_E$) //update the parameters
10: end
```

Algorithm Explanation.

In this algorithm, we load the training data set and we enter a loop that only stops when there is no more images or that the neural networks are synchronized. In the body of the loop, we get the first image, generate a bitstream which will be the plaintext we will hide, feed it into the neural network of Alice in order to generate the steganographic image. Bob and Charlie will then extract the text from the stego image and Eve will analyse the image and output either 1 if she thinks that the image is steganographic or 0 if it is a cover image. After getting the output, the loss function is verified and if it is not satisfied yet, we will jump to the next iteration.

Experiment I.

In the first experiment, Alice generates a random 100 bits plaintext P, and hides it inside a sequentially picked cover image from CelebA data set and sends it to Bob and Charlie.
Bob and Charlie extract the hidden text and output $P_{\text{Bob}}, P_{\text{charlie}}$.As Eve has access to the communication, he intercepts the cover image and tells if it's a cover image or a steganographic image by outputting either 1 or 0 as P_{eve}.
In every iteration, we calculate the loss of each neural network as well as the the minimum distance between the original plaintext and the extracted ones respectively: $P_{\text{Bob}}, P_{\text{Charlie}}$. Figure 4.2 shows the plotted loss made by each neural network during training.

Analysis I.

In Experiment I, we can see in Fig. 4.2 that Bob and Charlie have a loss of about 0.5 at the beginning of the experiment but reach 0 loss after around 175 epochs. We also see that they

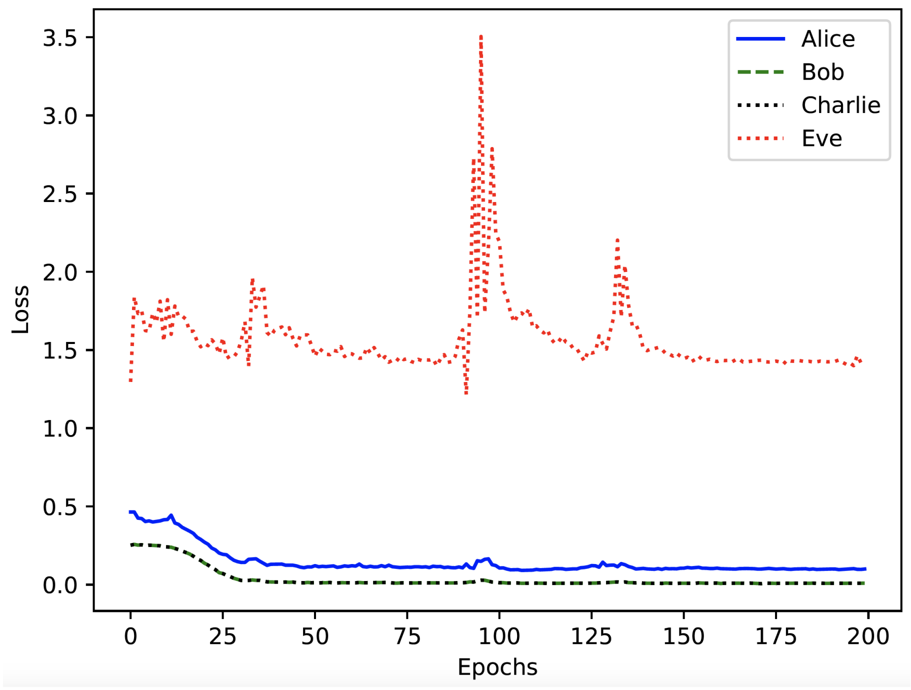

Figure 4.2: Test Results of Experiment 1

learned the same extraction algorithm.

To make sure they learned the same embedding and extraction algorithms, we also make Bob & Charlie generate a random plaintext and hide it inside a random image; Bob will send its image to Charlie for extraction and vice versa. We will not update the parameters in exchanges between Bob and Charlie in order to only learn when exchanging with Alice.

Fig. 4.3 shows the accuracy of Charlie while extracting images received from Bob, and Fig. 4.4 shows the accuracy of Bob when extracting images received from Charlie. We can see that after around 160 epochs, they both have 100% accuracy when extracting messages and this therefore proves that they all learned the same embedding and extraction algorithms.

4.4.2 Second & Third Scenario

Second Scenario

Training process II. In the Second Scenario, we split the neural networks into two communicating pairs. Concretely, Alice will communicate with Bob and Bob will communicate with Charlie. And Alice cannot communicate directly with Charlie. This can be useful in IoT devices where two devices are too far from each other and need another device as a bridge.

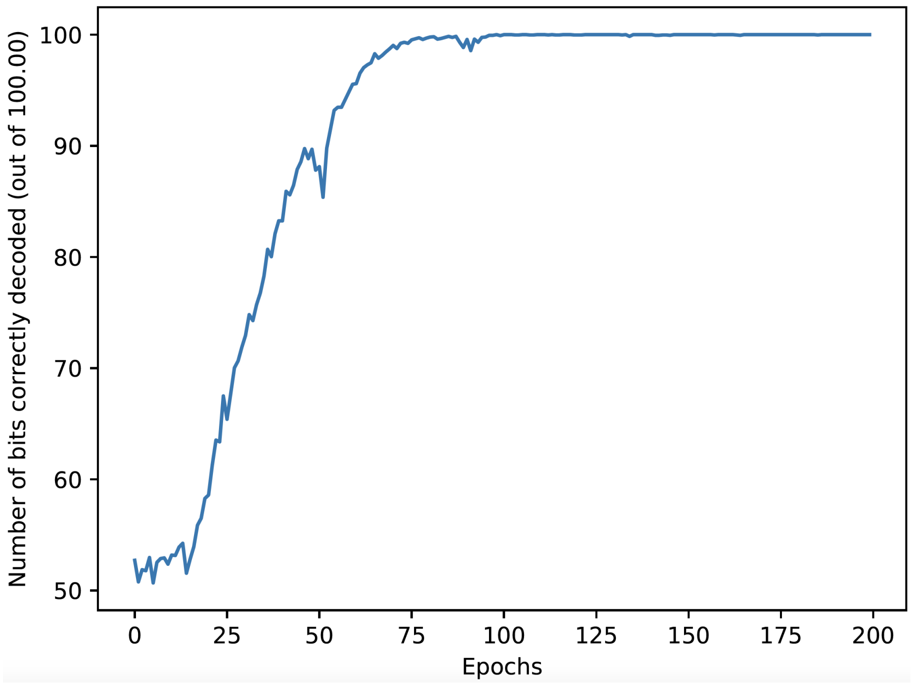

Figure 4.3: Charlie's Accuracy when extracting from Bob in scenario I

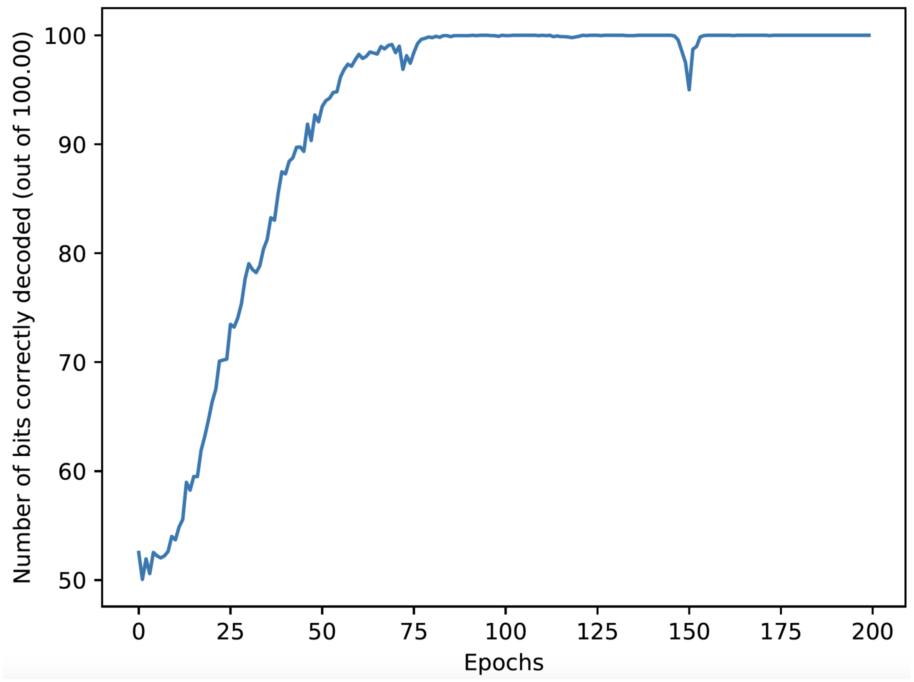

Figure 4.4: Bob's accuracy when extracting from Charlie in scenario I.

Experiment II. In this scenario, Alice generates a steganographic image and sends it to Bob. Bob extracts the hidden text, hides it again in another image through its neural network and the new steganographic image is sent to Charlie. Charlie will extract the hidden text from the image received from Bob in order to output P_{Charlie}. Eve can intercept any of the images exchanged between the neural networks. At every step of the training, we calculate the distance between the original plain texts and the extracted texts, we also calculate the loss of every neural network. The loss of the neural networks are plotted in Fig.s 4.5 and 4.6.

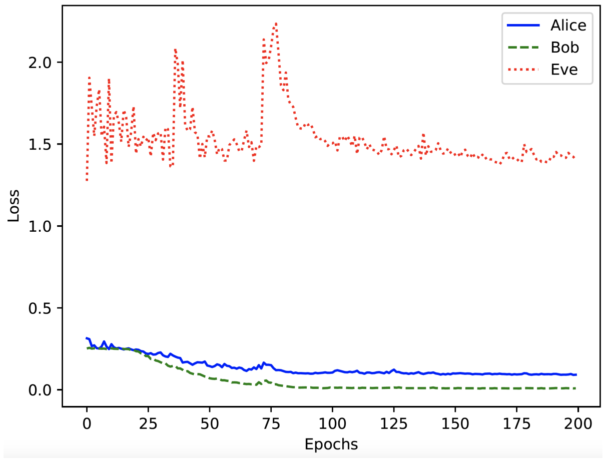

Figure 4.5: Loss in the communication between Alice and Bob in Exp. II

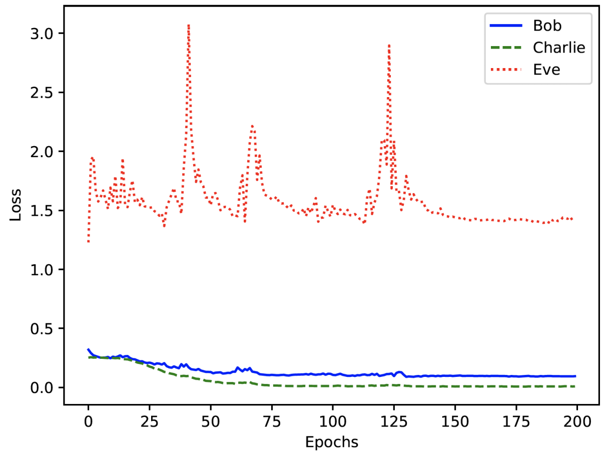

Figure 4.6: Loss in the communication between Bob and Charlie in Exp. II

Analysis II.

Similarly to the first scenario, we can see that Alice, Bob and Charlie converge to 0 loss in around 200 epochs. However the communication between Charlie and Alice is not possible and the accuracy was around 50% during our tests Therefore they will need to use Bob as a bridge to communicate.

Third Scenario

Training process III. In this scenario, Alice's synchronization with Bob and Charlie is done in an independent way; In other words, Alice will use one unique set of parameters for the communication with Bob and another unique set of parameters with Charlie. Therefore, a communication between Bob and Charlie would require the use of Alice as a bridge.

Experiment III.

Two sets of unique parameters are generated by Alice in this scenario; When generating a steganographic image by Alice, if the image is sent to Bob, the first set of parameters is used and when sending the steganographic image to Charlie, the second set of parameters is used. We record the loss of every neural network and we plot it in Fig. 4.7 and Fig. 4.8.

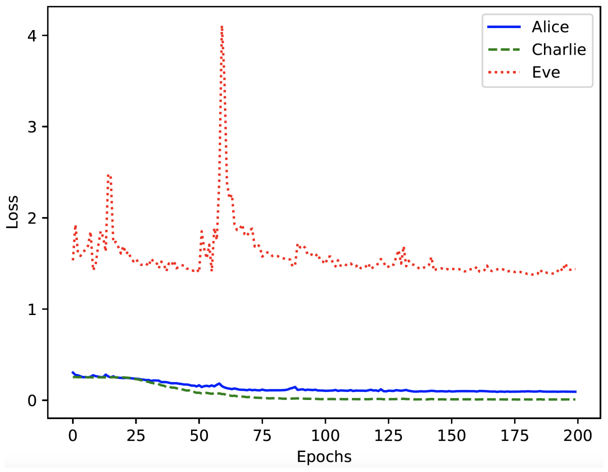

Figure 4.7: Loss in the communication between Alice and Charlie in Exp. III

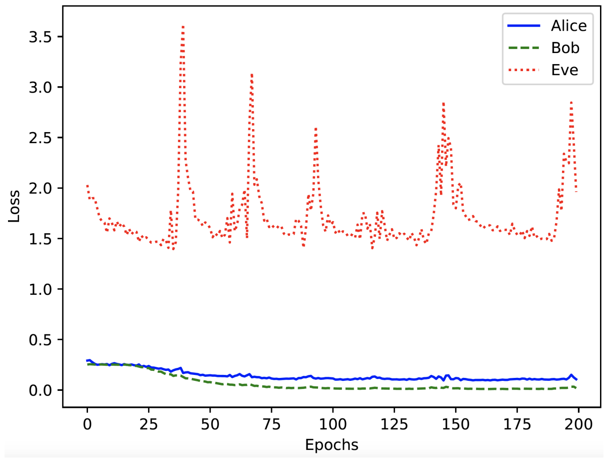

Figure 4.8: Loss in the communication between Alice and Bob in Exp. III

Analysis III.

We can see that Bob and Charlie start with a loss of around 0.5 but the loss quickly decreases and we get a nearly perfect accuracy. Eve's loss was between 2.0 and 3.5. Which means that each pair of neural networks is synchronized. But the communication between Charlie and Bob is not possible and Alice must be used as a bridge.

4.4.3 Discussions

We have shown in the first scenario of the communication that it is possible to add a third party to Alice and Bob and enable them to learn the same algorithm therefore solving the issue in Section 4.4 in [8]. However in the second and third scenario the 3-party communication will need to use one of the parties as a bridge whether the training party uses the same or different parameters for each party due to the problem of non-convexity.
Overall, the usage of cases depends on the scenario. First scenario fits best when having multiple machines communicating as it allows them to learn the same algorithm. Second and third scenarios are best for talking separately. The same applies in the case of multiple parties communication.

After training our neural networks, we tried feeding them the same set of images and see the output. Fig. 4.10 and Fig. 4.11 show the images generated by Alice at the beginning and end of training respectively. Fig. 4.9 shows the set of original images.

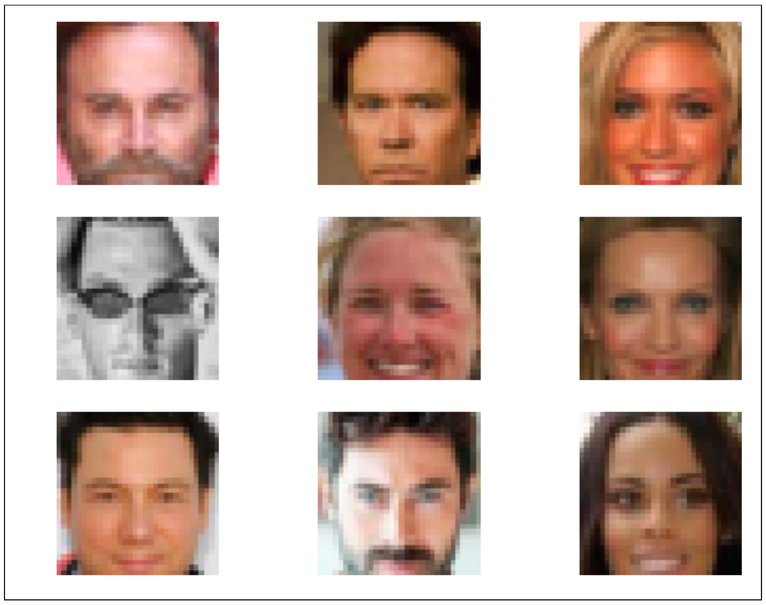

Figure 4.9: Original Image

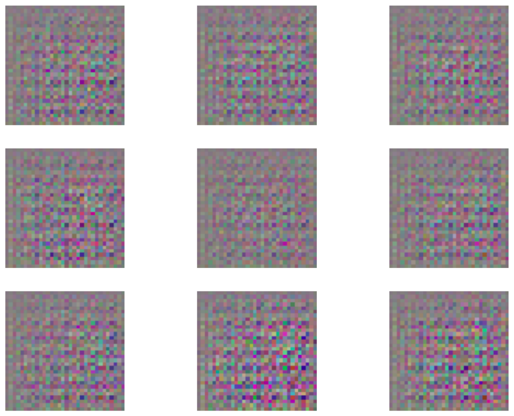

Figure 4.10: Images generated by Alice before training

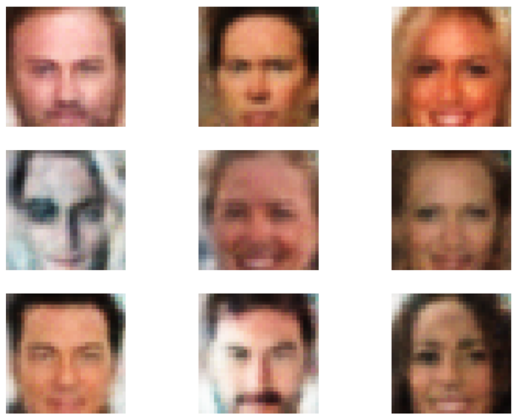

Figure 4.11: Images generated by Alice after training

Chapter 5

Asymmetric encryption using adversarial neural networks

5.1 Learning asymmetric encryption using adversarial neural networks

In this contribution, we present a novel approach to secure communications using adversarial neural networks. Our basic model consists of five agents: sender Alice, receiver Bob, eavesdropper Eve, and two neural networks for generating public and private keys. Unlike previous work in the field [1, 4, 9], our model does not require shared symmetric information between Alice and Bob. Instead, our approach allows Alice and Bob to learn encryption and decryption using asymmetric information.

The public and private keys generated by the neural networks in charge of that operation allow Alice to encrypt a message using the public key and Bob to decrypt the message using the private key, while ensuring that Eve cannot decrypt the ciphertext using the public key. Our experiments show that the neural networks are able to establish secure communication and provide robust security guarantees, even against stronger adversaries such as ones that perform attacks using leaked information or perform chosen plaintext attacks. The last three experiments show that neural networks (with asymmetric information) can secure the communication providing stronger security guarantees and resilience to leakage attacks which may include leakage from the private key.

To the best of our knowledge, this is the first work in which Alice and Bob with asymmetric information can train themselves to protect their communication. Our results demonstrate that multi-agent adversarial neural networks can provide strong security guarantees and resilience to a wide range of attacks, making it a promising approach for securing communication using asymmetric information.

5.2 Our Approach

In this chapter, we propose a multi-agent adversarial neural networks model where the sender and receiver learn to encrypt data using asymmetric information and protect their communication from eavesdroppers. Next, we try resetting Eve's neural network and retraining her to see if she can break Alice and Bob's communication as it happened in [1]. Lastly, we add more attackers to the setup and see how they affect the ability of Alice and Bob in establishing a secure communication from them.

The first of the three attackers is modeled to capture leakage attacks where the leakage is on the private key. Surprisingly enough, we see that even if the adversary is given access to all private keys, the sender (Alice) and receiver (Bob) can learn a mapping that does not entirely rely on the key to encrypt and decrypt the data and therefore prevent the attacker from decrypting the messages. The second of the three attackers is a classifier included in order to model the security against an adversary whose goal is to distinguish between real ciphertexts and randomly generated ones. The last of the three attackers is also a classifier added to prove indistinguishability between adversarially chosen plaintexts and their corresponding ciphertexts. While the two attackers performing classifications are designed to distinguish between plaintexts and ciphertexts respectively, their model can be used for other classification tasks. These last three experiments prove that the neural networks are able to adapt to more aggressive attackers and produce ciphertexts that only Bob can decrypt despite a leakage of the private key. As the security of our model does not reduce to any hardness assumption of mathematical problems, we hope that it can be a potential candidate for post-quantum cryptography [51].

To the best of our knowledge, our model is the first in its kind able to learn encryption using asymmetric information. There is no other publicly available contribution that investigated how neural networks can learn encryption using asymmetric information. All existing models are made to learn encryption using symmetric information which needs to be securely exchanged before the communication starts. Our model removes the need to securely exchange symmetric information before starting the communication.

This chapter is organized as follows: Section 5.3 describes our model setup, neural networks structure, and hyper-parameters used as well as a detailed description of the loss functions used in our model. Section 5.4 shows our different experiments and an analysis of their results. Lastly, we show the advantages of our technique in Section 5.6.

5.3 Our Model

5.3.1 Model Setup

Our initial one-attacker model is structured as follows: We have five neural networks (Alice, Bob, Public Keys Generator, Private Keys Generator, and Eve) all of these neural networks have the same structure but play different roles in the setup. We use P to denote the plaintext message to be sent, C the ciphertext generated by Alice when encrypting P, K_{pub} the public key generated by the public keys generator, K_{priv} the private key generated by the private keys generator and RN the random noise used to generate the public and private keys.

Alice will play the role of the sender. Alice accepts a message P and a public key K_{pub} as input to her neural network in order to produce the ciphertext C. Bob will play the role of the receiver. Bob accepts the ciphertext C generated by Alice and the private key K_{priv} as input to his neural network in order to produce P_{bob}, his decryption attempt of the ciphertext C. Eve will play the role of the eavesdropper. Eve accepts the ciphertext C and the public key K_{pub} as input to her neural network in order to produce P_{eve}, her decryption attempt of the ciphertext C generated by Alice. As for the public key K_{pub}, it is generated by the public keys generator which accepts a random noise RN as input to its neural network. The public keys generator will pass the random noise through its neural network and output the public key K_{pub} which is sent to Alice and intercepted by Eve. The private keys generator accepts the random noise RN and the public key K_{pub} as input to its neural network and outputs the private key K_{priv}. This model with one attacker will be used in experiment 1 in Section 5.4.1.

The model setup with one attacker (Eve) is shown in Figure 5.1. This model will be used as a proof of concept to show that Alice and Bob can learn encryption using asymmetric information.

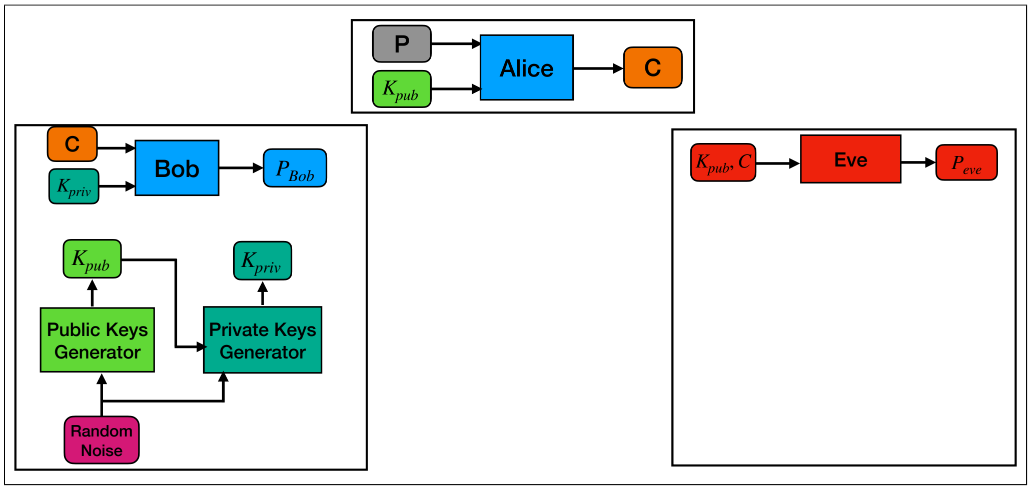

Figure 5.1: Our initial model setup with Eve only as an attacker.

After experiment 1, we will do 3 more experiments each containing the attackers of the previous experiment plus one new attacker. In experiment 2, we will add a new attacker denoted $Attacker_1$ that has access to all the private keys generated by the private keys generator and tries to learn to decrypt the ciphertexts generated by Alice using those Keys. $Attacker_1$ receives as input the ciphertext C and the private key K_{priv} and produces his decrypted text P_{A1}. In experiment 3, we add another attacker denoted $Attacker_2$ that receives two ciphertexts C_1, C_2 and one plaintext P. $Attacker_3$ will output π_1, the probability that C_1 is a ciphertext for P and π_2 the probability that C_2 is a ciphertext for P. Lastly, in experiment 4 we add the last attacker denoted $Attacker_3$ that receives two plaintexts P_1, P_2 and one ciphertext C. $Attacker_4$ will output π_1, the probability that P_1 is the plaintext encrypted to the ciphertext C and π_2, the probability that P_2 is the plaintext encrypted to the ciphertext C. The model with the additional three attackers is shown in Figure 5.2 and the Training Process is shown in Figure 5.3.

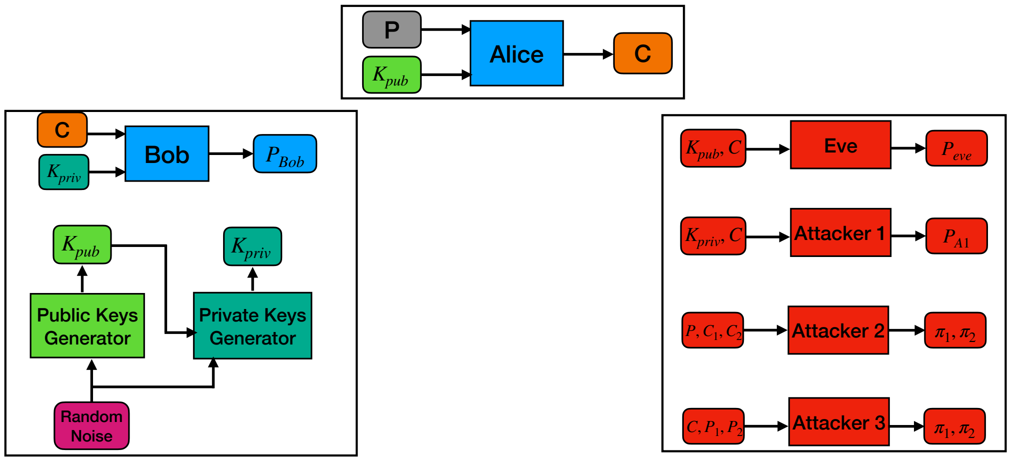

Figure 5.2: Our final model with Eve and 3 additional attackers.

In a real world scenario, Alice and the four attackers would be on different machines (or nodes) in an open network. Bob, the public keys generator and the private keys generators would be on the same machine (or network node). For a machine to play the role of the receiver, it will need to have three neural networks as shown in our setup in Figures 5.1. The first neural network will be in charge of generating public keys from the random noise and distributing it to the nodes in the network. The second neural network will be in charge of generating private keys from the public key and the random noise and distributing them to the third neural network (Bob) when he needs to decrypt a received message.

The random noise RN was introduced not only to serve as a seed for generating the public key by the public keys generator but also to create a common local randomness between the public keys generator and the private keys generator. If the private keys generator does not use the random noise to generate the private key, any attacker with knowledge of the neural network structure might learn to generate the private key from the public key.

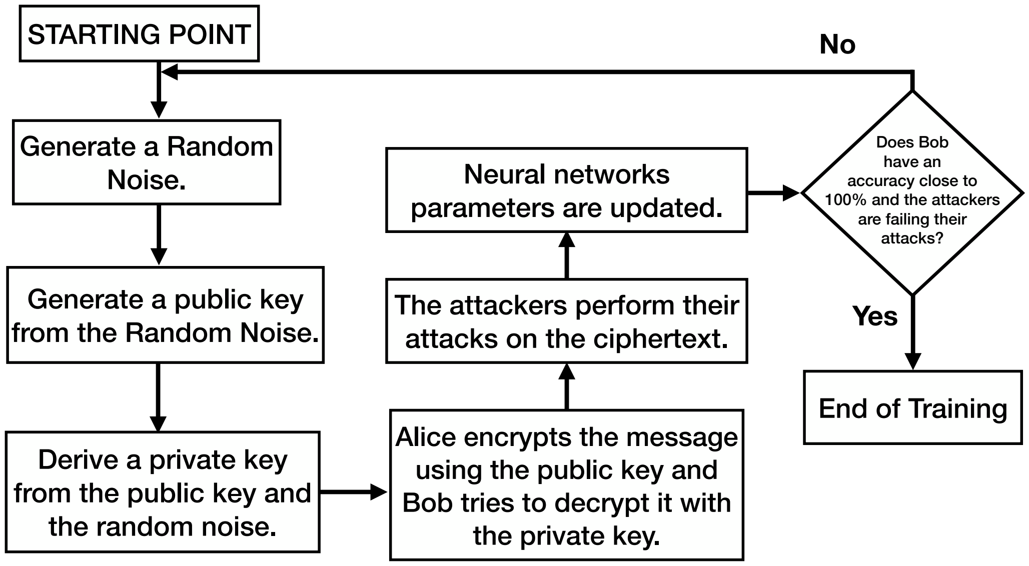
STARTING POINT
Generate a Random Noise.
Generate a public key from the Random Noise.
Derive a private key from the public key and the random noise.
Alice encrypts the message using the public key and Bob tries to decrypt it with the private key.
The attackers perform their attacks on the ciphertext.
Neural networks parameters are updated.
Does Bob have an accuracy close to 100% and the attackers are failing their attacks?
No
Yes
End of Training

Figure 5.3: Training process of our model.

5.3.2 Neural Network Structure

Similarly to Abadi and Andersen [1], we follow the mix and transform architecture and use the same neural network structure they used. All the agents in the model except $Attacker_3$ and $Attacker_4$ share the same structure. As for $Attacker_3$ and $Attacker_4$, they have all the layers as the other agents plus one fully connected layer at the end to make a classification and output probabilities. We also use the same data size for all input and output data (i.e. the input message, ciphertext, public key, private key all have the same size). $Attacker_3$ and $Attacker_4$ will output two floating numbers that sum up to 1.

The neural networks structure starts with a fully connected (FC) layer with an input size equal to the output size. The FC layer will shuffle the input bits hence the *mix* and transform architecture. Afterwards, the output from the FC layer is followed by a series of 1D convolutions that will transform unique groups of their input. The last convolution has an output size equal to the datasize. $Attacker_3$ and $Attacker_4$ however have one final FC layer with an output size of 2. This FC layer is going to output the two probabilities π_1, π_2 that they are going to produce.

Table 5.1 summarizes the input and output sizes for each neural network, with N being the value chosen for the data size. In our experiments, $N = 64$.

As we can see in Table 5.1, Alice, Bob, Eve, Private Keys Generator, $Attacker_2$ have the same input size of $2N$ and output size of N. The public keys generator has an input and output size of N. $Attacker_3$ and $Attacker_4$ have an input size of $3N$ and an output size of 2. Concretely, Alice will accept as input a plaintext of size N and a public key of size N to output a ciphertext of size N. Eve will accept as input a ciphertext of size N and a public key of size N to output a deciphered text of size N. Bob will accept as input a ciphertext of size N and a private key of

Table 5.1: Summary of the input and output layers size for each neural network

Neural Network	Input Layer Size	Output Layer Size
Alice, Bob, Eve, Private Keys Generator, *Attacker*_2	2N	N
Public Keys Generator	N	N
*Attacker*_3, *Attacker*_4	3N	2

size N to output a deciphered text of size N. The private keys generator will accept as input a random noise of size N and a public key of size N to output a private key of size N. $Attacker_3$ will receive as input two ciphertexts and one plaintext of size N each and output two probabilities of size 1 each. $Attacker_4$ will receive as input two plaintexts and one ciphertexts of size N each and output two probabilities of size 1 each.

Table 5.2 summarizes the neural network structure for Alice, Bob, public keys generator, private keys generator, and Eve. The $6th$ layer is only used for $Attacker_2$ and $Attacker_3$ in order to produce probabilities. The other agents have the layers from layer 1 to layer 5 only. The $5th$ layer is sigmoid-activated for Attackers 3 and 4 and Tanh-activated for the other neural networks.

Table 5.2: Summary of the neural network structure used in our model for Alice, Bob, public keys generator, private keys generator, Eve and the three attackers $Attacker_1$, $Attacker_2$, $Attacker_3$.

Layer #	Layer Type	Activation	Filters	Kernel Size	Strides	Padding
1	FC Layer (Dense)	Relu	-	-	-	-
2	Conv1D	Sigmoid	2	4	1	same
3	Conv1D	Sigmoid	4	2	2	valid
4	Conv1D	Sigmoid	4	1	1	same
5	Conv1D	**Sigmoid** for Attackers $3, 4$ **Tanh** for the others.	1	1	1	same
6 ($Attacker_2$ and $Attacker_3$ only.)	FC Layer (Dense)	Softmax	-	-	-	-

The overall architecture for each neural network and the model setup are illustrated in Figure 5.4.

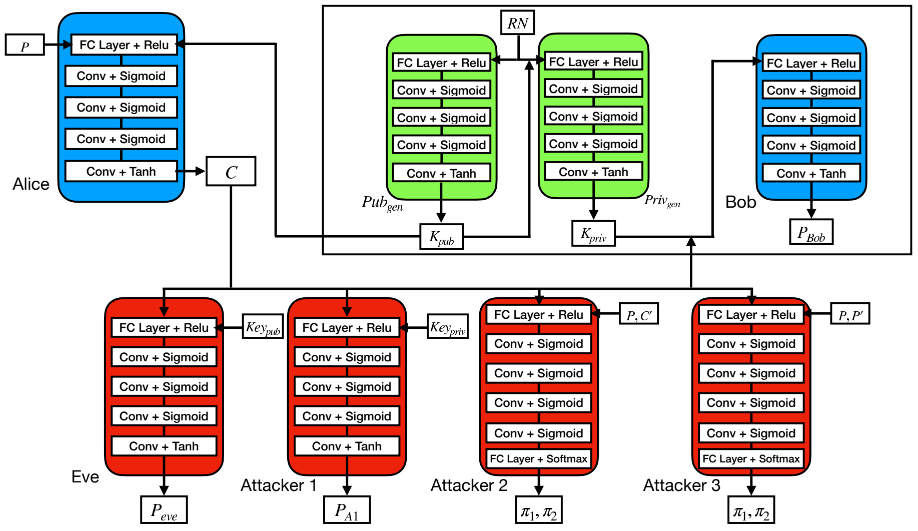

Figure 5.4: Overall architecture of our model including the neural networks structure used.

5.3.3 Hyperparameters Used

The following are the hyperparameters used to train our neural network. Grid search was used to do our hyperparameter turning operation and find the optimal values to use to train our network. The most important hyperparameter is the data size. We noticed that the best results are obtained using a data size of 64. Other data sizes might constitute an advantage for one or more attackers. The other hyperparameters only affected the training time in our experiments.

- **Datasize**: $N = 64$.
- **Batch Size**: 256.
- **Number of epochs**: Up to 200 but the training might stop earlier if the receiver has reached 100% accuracy and the prediction accuracy of attackers is close to random guesses.
- **Training steps per epoch**: 300.
- **Learning Rate**: 0.0008.
- **Optimizer**: Adam's optimizer.

5.3.4 Loss Functions

The loss functions for the neural networks similar to the ones used in [1, 4, 9] and are described below.

- We use $A(\Theta_A, P, K_{pub})$ to denote Alice's output on input P, K_{pub}.

- We use $B(\Theta_B, C, K_{priv})$ to denote Bob's output on input C, K_{priv}.
- We use $E(\Theta_E, C, K_{pub})$ to denote Eve's output on input C, K_{pub}.
- We use $Pub\_Gen(\Theta_{pub}, RN)$ to denote the public key generator's output (K_{pub}) on input RN.
- We use $Priv\_gen(\Theta_{priv}, K_{pub}, RN)$ to denote the public key generator's output (K_{priv}) on input K_{pub}, RN.
- We use $A1(\Theta_{A1}, C, K_{priv})$ to denote $Attacker_1$'s output P_{A1} on input C, K_{priv}.
- We use $A2(\Theta_{A2}, C_1, C_2, P)$ to denote $Attacker_2$'s output π_1, π_2 on input C_1, C_2, P.
- We use $A3(\Theta_{A3}, P_1, P_2, C)$ to denote $Attacker_3$'s output π_1, π_2 on input P_1, P_2, C.

Where $\theta_{\{A,B,E,pub,priv,A1,A2,A3\}}$ represent the parameters of Alice, Bob, Eve, the public keys generator, the private keys generator, $Attacker_1$, $Attacker_2$ and $Attacker_3$ respectively. P represents the plaintext that will be encrypted by Alice and C represents the ciphertext generated from that encryption. K_{pub} and K_{priv} represent the public key and the private key respectively. RN represents the random noise. C_1, C_2 are one real and one fake ciphertext that $Attacker_2$ has to tell apart by producing the probability π_1 that C_1 is the ciphertext of P and π_2 the probability C_2 is the ciphertext of P. P_1, P_2 are one real and one fake plaintext that $Attacker_3$ has to tell apart by producing the probability π_1 that P_1 is the plaintext of C and π_2 the probability P_2 is the plaintext of C.

Lastly, we denote the function $d(x, y)$ to be the $L1$ distance between x and y.

The loss function for Bob is defined below in Equation 5.1.

$$L_B(\Theta_A, \Theta_B, \Theta_{Pub_{gen}}, \Theta_{Priv_{gen}}, P, RN) = d(P, B(\Theta_B, C, K_{priv})) \tag{5.1}$$

Intuitively, the loss function L_B determines how wrong Bob is in his decryption.

The loss function can be expanded to the following to see all the involved variables and where they are generated from:

$$L_B(\Theta_A, \Theta_B, \Theta_{Pub_{gen}}, \Theta_{Priv_{gen}}, P, RN) = \\ d(P, B(\Theta_B, A(\Theta_B, P, Pub_{gen}(\Theta_{pub}, RN)), Priv_{gen}(\Theta_{priv}, Pub_{gen}(\Theta_{pub}, RN), RN)))$$

Similarly, we define the loss function for Eve in Equation 5.2 below.

$$L_E(\Theta_A, \Theta_E, \Theta_{Pub_{gen}}, \Theta_{Priv_{gen}}, P, RN) = d(P, E(\Theta_E, C, K_{pub})) \tag{5.2}$$

Intuitively, L_E determines how wrong Eve is when decrypting the ciphertext using the public key.

The loss function can be expanded to the following to see all the involved variables:

$$L_E(\Theta_A, \Theta_E, \Theta_{Pub_{gen}}, \Theta_{Priv_{gen}}, P, RN) = \\ d(P, E(\Theta_E, A(\Theta_B, P, Pub_{gen}(\Theta_{pub}, RN)), Pub_{gen}(\Theta_{pub}, RN))) \quad (5.3)$$

The loss function for the private keys generator is going to be equal to Bob's loss as we want it to generate a private key that allows Bob to decipher the messages correctly. This will push the private keys generator to adjust its parameters in order to be able to map the public key's and the random noise's values to a value that allows Bob to decrypt correctly. The loss function for the private keys generator $L_{priv_{gen}}$ is defined in Equation 5.4 below.

$$L_{priv_{gen}} = L_B \quad (5.4)$$

Where L_B is defined in equation 5.1. The private keys generator learns by minimizing Bob's loss. From another point of view, the private keys generator is learning to generate private keys that Bob can use to decrypt the ciphertexts.

Lastly, we define the same loss function for Alice and the public keys generator in equation 5.5 below. The loss function is composed of two parts. The first part is Bob's loss and the second part is $(1 - L_E^2)$. The choice behind this loss function is that we want Alice to generate ciphertexts that are easily decrypted by Bob but at the same time difficult to decrypt for Eve.

$$L_{A,pub_{gen}} = L_B + (1 - L_E^2) \quad (5.5)$$

The loss functions of the three attackers $Attacker_{1,2,3}$ will be described in experiments $2, 3, 4$ respectively. The loss function of Alice and the public keys generator will also change in those last three experiments to adapt to the new attacker and resist its attacks.

5.4 Training, Results and Discussion

5.4.1 Training Experiment 1

Goal of the experiment: show that neural networks can learn to encrypt data using asymmetric information and prevent an attacker from learning to decrypt without the private key.
Result of the experiment: Success.

In the first experiment, we simply implement the model shown in Figure 5.1 using Tensorflow with the neural network structure shown in Table 2.1 to see the initial results and confirm that Alice and Bob can learn to protect a communication using asymmetric information. We tried keys and messages of sizes $N = 32, 64, 128$ bits but a size of $N = 64$ seems to give the best results. Therefore, all the input messages, ciphertexts, random noises and public/private keys will have a size of $N = 64$. We operate in batches of size 256.

To train our neural network, we start by generating a batch of random noises and feed it to the public keys generator that is going to output a batch of public keys. We then generate a batch of messages and feed it along the batch of public keys to Alice so that we get a batch of ciphertexts. At the same time, the private keys generator will also receive the same batch of public keys and the batch of random noises in order to output a batch of private keys. Once we have our batches of ciphertexts and private keys, we feed the ciphertexts to Eve and record her correct decryption rate using the L_1 metric. We also feed the batch of ciphertexts and private keys to Bob and record his correct decryption rate using the L_1 metric. We keep training this way until Bob has reached 0% decryption error and Eve has a decryption error of around 50%. As mentioned in [1], we do not want to train Eve to be 100% wrong as she can be 100% correct in the next iteration just by flipping the bits of her output. In terms of probabilities, we want Eve to output decrypted messages that are no better than random guesses and 50% is the probabilistic representation of making random guesses. The training is done in epochs of 300 steps each. We train for around 100 epochs which gives us a total of $30,000$ training steps. The training was conducted on a GPU equipped machine. Once training is finished, we plot the decryption error of Bob and Eve. The results of training the model are shown in Figure 5.5.

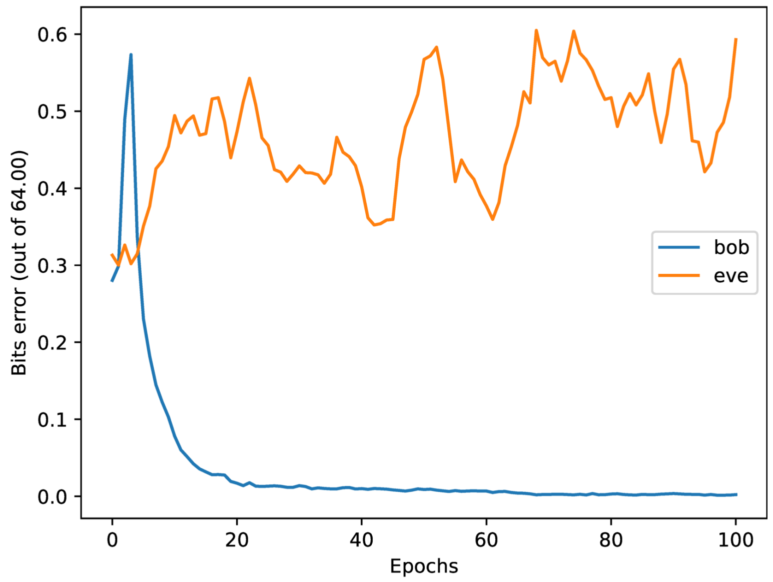

Figure 5.5: Bob's and Eve's bits error during the first experiment.

As we can see in Figure 5.5, Bob converges to 100% accuracy after approximately 80 epochs starting from around 50% accuracy in the first epoch. This means that Bob has learned to decrypt Alice's ciphertexts without error at the end of the training and therefore Alice and Bob have been successful in making a communication. On the other hand, Eve started with approximately 30% bits error but Alice quickly started to make her encryption harder to decrypt by Eve and after approximately 15 epochs we can see that Alice succeeded in making the ciphertexts hard to decrypt for Eve and easy to decrypt for Bob. After the $20th$ epoch and until the $100th$ epoch, we can see that Eve's error is between 35% and 60%. This means that Eve sometimes gets better in her predictions but Alice quickly changes her encryption method to make it harder again for Eve to make good predictions while working on maintaining a zero error for Bob. During all of our experiments, Eve has never done better than 30% bits error at the beginning of the training and always converges to the $40-70\%$ error range which makes her in the best case scenario 20% better than an attacker making random guesses. This means that Alice and Bob were successful in making a communication and protecting it from Eve.

Once we finished the training, we want to verify that resetting Eve's neural network does not make her any better in guessing Alice's messages. We do this because as mentioned previously, Abadi and Andersen [1] noticed that resetting Eve's neural network in their tentative of an asymmetric communication allows her to read all the messages exchanged between Alice and Bob. Therefore, we also reset Eve's neural network and restart the same training process as before for approximately 40 epochs or $12,000$ iterations. The results are shown in Figure 5.6.

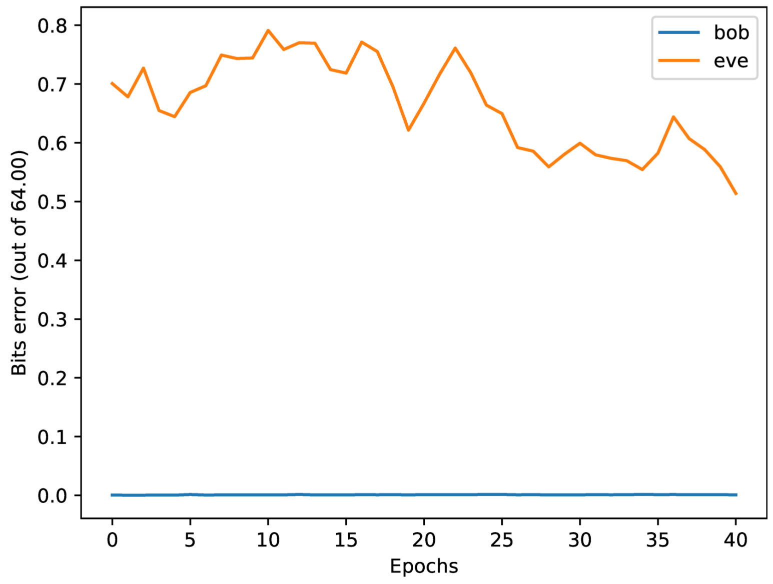

Figure 5.6: Bob's and Eve's bits error after resetting Eve's neural network.

As we can see in Figure 5.6, resetting Eve's neural network does not make her any better in making prediction. We can see that Eve started at 70% bits error but Alice managed to drop Eve's accuracy to approximately 50% after approximately 35 epochs. Alice's changes in her encryption method did not affect Bob's ability to decrypt and we can see that he kept zero decryption error during the whole training process.

5.4.2 Training Experiment 2

Goal of the experiment: show that neural networks can prevent an attacker (who has access to the private key) from decoding the communication, by learning a plaintext-to-ciphertext mapping that does not entirely rely on the private key.
Result of the experiment: Success.

In the second experiment we add a second attacker called $Attacker_1$. We assume that this attacker has somehow gained access to the private keys generated by the private keys generator. Therefore, and at every iteration, this attacker gets the batch of ciphertexts and private keys and outputs his decrypted texts. The model setup of Experiment 2 is shown in Figure 5.7.

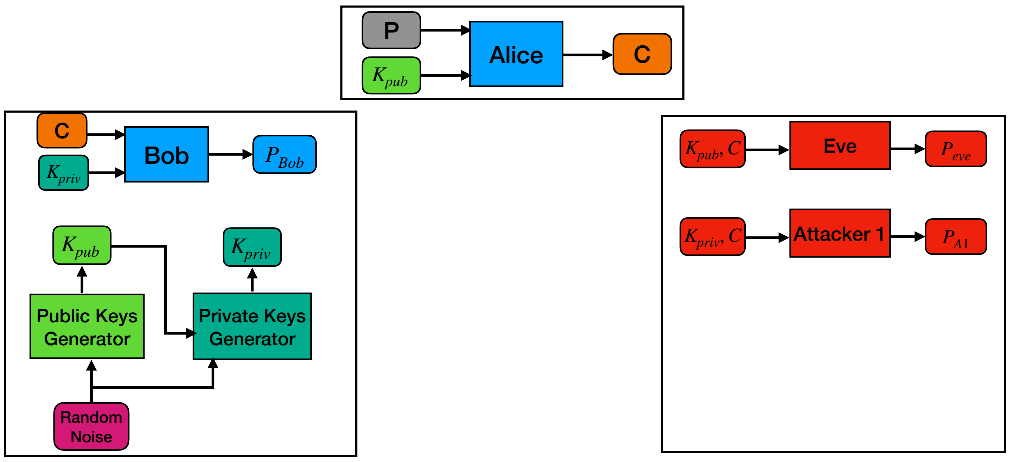

Figure 5.7: Model setup used in Experiment 2.

The loss function L_{A1} for $Attacker_1$ is the same as Bob's loss and is defined in Equation 5.6 below.

$$L_{A1}(\Theta_A, \Theta_{A1}, \Theta_{Pub_{gen}}, \Theta_{Priv_{gen}}, P, RN) = d(P, A1(\Theta_{A1}, C, K_{priv})) \tag{5.6}$$

Intuitively, $Attacker_1$ wants to minimize the distance between the original plaintext and his output. $Attacker_1$ will learn by minimizing L_{A1}.

Alice and the public keys generator now need to resist one more attacker therefore, we need to change their loss function. Their new function L^2_{A,pub_gen} is defined in Equation 5.7 below.

$$L^2_{A,pub\_gen} = L_{A,pub\_gen} + (1 - L^2_{A1}) \tag{5.7}$$

Where L_{A,pub_gen} has been defined in equation 2.3 and L_{A1} is defined in equation 5.6. Intuitively, The higher L_{A1} gets, the lower L^2_{A,pub_gen} will be and therefore Alice and the public keys generator know that they are beating $Attacker_1$.

Similarly to Alice and the public keys generator, and as the private keys generator is involved in the decryption process, we also need to change his loss function as shown in Equation 5.8 below.

$$L_{priv_{gen}} = L_B + (1 - L_{A1}^2) \tag{5.8}$$

Therefore, the private keys generator now learns by minimizing L_B.

$Attacker_1$ is similar to one of the attackers proposed in [4] where the first attacker has access to all the symmetric encryption keys and the sender and receiver were able to prevent him from reading the conversation despite his access to the secret keys. The results of the second experiment are shown in Figure 5.8 below.

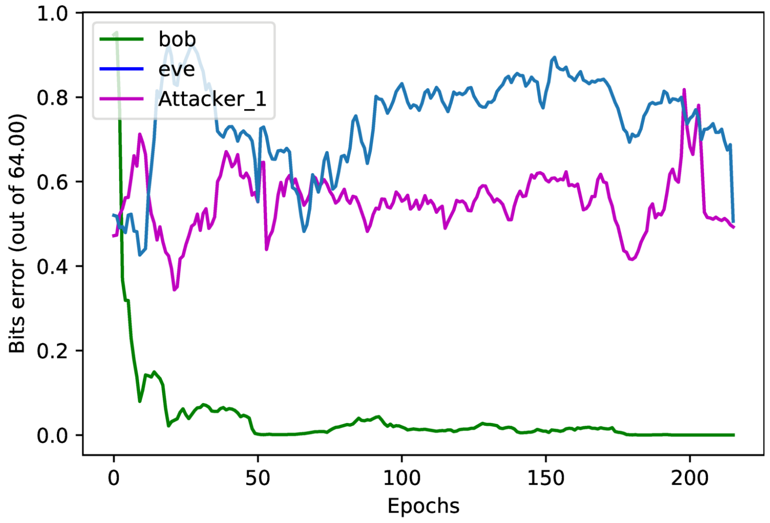

Figure 5.8: Bits decryption error for Bob, Eve and $Attacker_1$ in Experiment 2.

We can see that due to *random* weights initialization, Bob started with almost zero accuracy and $Eve, Attacker_1$ both started at around 50% decryption error. However, this quickly changed after approximately 10 epochs where Bob's error started decreasing to 10% while $Attacker_1$'s error stayed around 50%. Until epoch 200, the decryption error of the three neural networks kept changing but only stabilized at around 50% for Eve and $Attacker_1$ while Bob has finally reached 0 decryption error. At this point, the model training has finally reached a stable point where Bob has no decryption error and the two other attackers are producing outputs that are slightly only better than random guesses (i.e. they have a decryption error that is around 50%). This a strange behavior as Bob and $Attacker_1$ have access to the same information. But similarly to the results in [4] this is explained by the fact that the model is learning a pattern that is not totally dependent on the private key. This mapping is hard for $Attacker_1$ to learn and he ends up producing outputs that are only slightly better than random guesses. As shown in previous works as in [7, 4, 9], training against multiple or more advanced attackers by giving them access to private information such as a secret key in part or in whole allows Alice and Bob to learn a stronger encryption techniques in order to resist the attacker that is breaking the communication. By giving $Attacker_1$ access to the private keys, we forced Alice and Bob to learn a mapping that does not entirely depend on the private key and successfully prevented $Attacker_1$ from learning to decrypt the ciphertexts produced by Alice. The training continues until Bob is not making any error in his decryption while Eve and $Attacker_1$ have around 50% decryption error.

5.4.3 Training Experiment 3

Goal of the experiment: show the indistinguishability between ciphertexts.
Result of the experiment: Success.

In the third experiment, we want to know whether the ciphertexts generated by Alice are indistinguishable from randomly generated ones. Therefore, we add another attacker called $Attacker_2$ to the setup which is similar to the third attacker proposed in [4]. $Attacker_2$ accepts a plaintext P, its corresponding ciphertext C_1 and a randomly generated ciphertext C_2. $Attacker_2$ will analyze the two ciphertexts and output π_1, the probability that C_1 is the ciphertext that corresponds to P and π_2, the probability that C_2 corresponds to P. The model setup of the third experiment is shown in Figure 5.9.

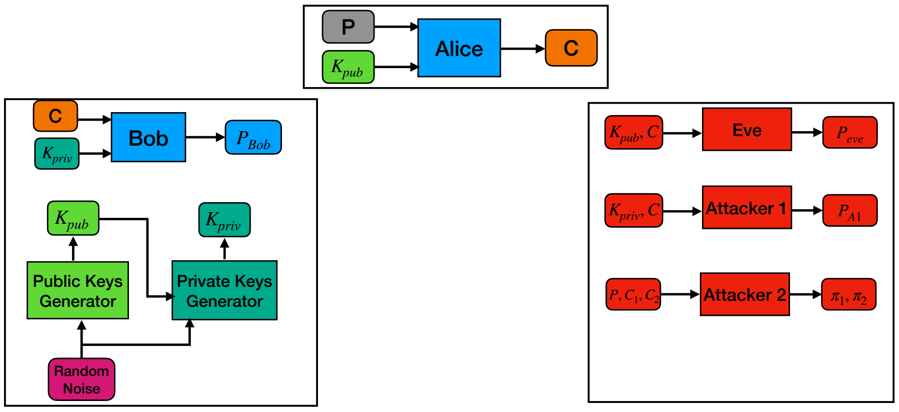

Figure 5.9: Model setup used in Experiment 3.

As this neural network has to make a classification instead of outputting a decrypted message like Eve and $Attacker_1$, some minor changes to its neural network structure have to be made. The new neural network structure for $Attacker_2$ is shown in Table 5.3 below.

As shown in Table 5.3 above, we have added a fully connected (FC) layer with two outputs at the end of the neural network structure. This FC layer is activated through a softmax activation function in order to obtain two probabilities that sum up to 1. Ideally, we want an output of $\pi_{1,2} = 0.5$ which will mean that the neural network is not able to make any difference between the real ciphertext and the one generated randomly.

Table 5.3: Neural network structure used for the $Attacker_2$

Layer #	Layer Type	Activation	Filters	Kernel Size	Strides	Padding
1	FC Layer (Dense)	Relu	-	-	-	-
2	Conv1D	Sigmoid	2	4	1	same
3	Conv1D	Sigmoid	4	3	3	valid
4	Conv1D	Sigmoid	4	1	1	same
5	Conv1D	Sigmoid	1	1	1	same
6	FC Layer (Dense)	Softmax	-	-	-	-

As we are changing the learning goal of this attacker, we also need to define a new loss function for our new optimization problem. We use the binary cross entropy as a loss function for $Attacker_2$.

Given N plaintexts $\left[P_{(0)}, P_{(1)}, ...P_{(N-1)}\right]$, and two sets of N ciphertexts $\left[C^1_{(0)}, C^1_{(1)}, \cdots C^1_{(N-1)}\right]$, $\left[C^2_{(0)}, C^2_{(1)}, \cdots C^2_{(N-1)}\right]$ we define the loss function L_{A3} for $Attacker_2$ in Equation 3.7 below.

$$L_{A3} = -\frac{1}{N}\sum_{i=0}^{N-1}\sum_{j=1}^{2} y^j_{(i)} \log\left(\pi^j_{(i)}\right) \tag{5.9}$$

Where $y^j_{(i)} = 1$ if $P_{(i)}$ is the plaintext of $C^j_{(i)}$ and 0 otherwise. Intuitively, $\pi^j_{(i)}$ is the probability that $C^j_{(i)}$ is the ciphertext corresponding to the plaintext $P_{(i)}$.

Therefore, $Attacker_2$ will learn by minimizing L_{A3}. Alice and the public keys generator now need to resist one more attacker, therefore their loss function will change to $L^3_{A,pub_{gen}}$ in the third experiment. $L^3_{A,pub_{gen}}$ is defined in Equation 5.10 below.

$$L^3_{A,pub_{gen}} = L^2_{A,pub_{gen}} - min(L_{A3}, 0.5) \tag{5.10}$$

Where $L^2_{A,pub_{gen}}$ is defined in Equation 5.7 and L_{A3} is defined in Equation 5.9. The min function is used to guarantee that Alice and the Public keys generator will not try to maximize $Attacker_2$'s classification errors as he can invert his predictions in the next iteration and become 100% correct.

As for the training, it is conducted in the same environment and using the same hyper parameters as in the first experiment. The results are shown in Figure 5.10 and Figure 5.11 below.

As we can see in Figure 5.10, from the beginning to the end of training, $Attacker_2$ is making random predictions and all its probabilities are close to 0.5. Therefore, this attacker is not able to distinguish between the two ciphertexts fed to him in order to tell which one corresponds to the given plaintext. At the same time Bob's decryption accuracy started with a little less than 40% decryption error as shown in Figure 5.11. However the presence of $Attacker_2$ in addition to $Attacker_1$ and Eve resulted in even more perturbation in Bob's decryption accuracy especially

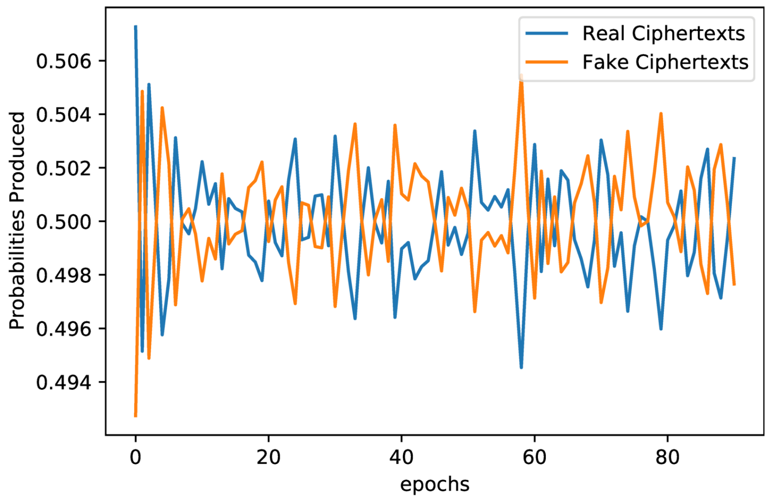

Figure 5.10: Probabilities produced by $Attacker_2$ on real and fake ciphertexts in Experiment 3.

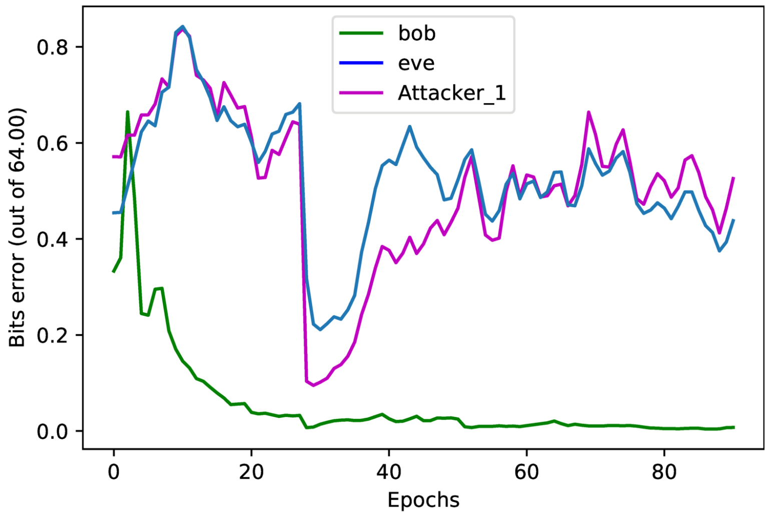

Figure 5.11: Bits decryption error for Bob, Eve and $Attacker_1$ in Experiment 3.

between the $20th$ and the $60th$ epoch. The accuracy of Bob stabilized at 100% after the $80th$ epoch where the accuracy *eve* and $Attacker_1$ also stabilized around 50%. Similarly to the previous experiment, the training stops when Bob is not making errors in his decryption while Eve and $Attacker_1$ have around 50% decryption error and $Attacker_2$ is producing random probabilities.

This experiment shows that, similarly to the results in [4], the ciphertexts generated by Alice are indistinguishable and guarantee indistinguishably between a real ciphertext and a randomly generated one. This means that the ciphertexts contain no information on the plaintexts that can make them differ from randomly generated ones.

Discussion on the modification of the encryption

Public key encryption protocols with deterministic encryption such as textbook RSA [11] are known to have security issues due to the availability of the public key to everyone including malicious users. As in deterministic encryption a plaintext will always be encrypted to the same ciphertext when using the same key, an attacker can build a database of known plaintexts and their corresponding ciphertexts as he has access to the public key and can encrypt any message he wants. Afterwards, after intercepting some ciphertexts, the attacker can query his database and see if the ciphertext exists in the database. If it does, the attacker has now the original plaintext without the need to make any decryption as he knows that the plaintexts are always encrypted in the same way.

One workaround in textbook RSA [11] that is widely used in the implementations of RSA is padding. Padding is a technique that includes a randomness in the plaintext before encrypting it. When the receiver makes the decryption, he will just discard the randomness and keep the plaintext. This way, two plaintexts encrypted with the same public key would have a different ciphertext as they have been padded with a different randomness.

Similarly to textbook RSA [11], our model also learns to perform deterministic encryption and therefore using the model to encrypt data without padding is not secure. In order to solve this issue, a sender can concatenate $32bits$ plaintext and $32bits$ randomness and use the result as input to its neural network along the public key. The randomness will be mixed with the plaintext and the public key before the ciphertext is generated and therefore the same plaintext encrypted twice can now have two different ciphertexts even if it was encrypted with the same public key and the same neural network. As Bob has a 100% decryption accuracy, he can just keep the first $32bits$ and discard the last 32 after decrypting the ciphertext sent by Alice.

5.4.4 Training Experiment 4

Goal of the experiment: show indistinguishability between plaintexts and their corresponding ciphertexts.
Result of the experiment: Success.

The $4th$ and last experiment is similar to the third experiment. We want to add an attacker $Attacker_3$ that receives two plaintexts and one ciphertext in order to tell which plaintext has been encrypted to the ciphertext. Similarly to [9], by preventing this $4th$ attacker from making good classifications of the plaintexts, we are showing security against Chosen Plaintext Attacks (CPA).

The model setup of the fourth experiment is shown in Figure 5.12.

$Attacker_3$ accepts a ciphertext C, its corresponding plaintext P_1 and a randomly generated plaintext P_2. $Attacker_3$ will analyze the two plaintexts and output π_1, the probability that P_1 is the plaintext that corresponds to C and π_2, the probability that P_2 corresponds to C. Just like

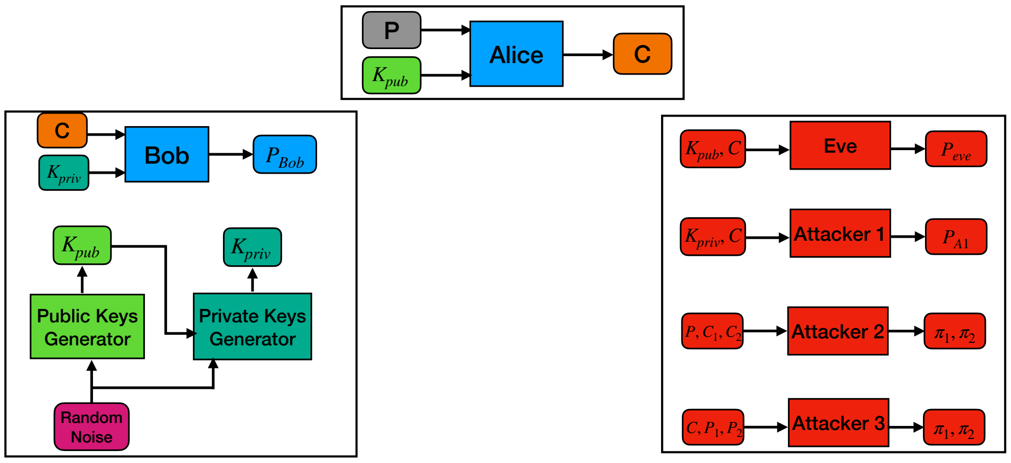

Figure 5.12: Model setup used in Experiment 4.

$Attacker_2$, our optimization problem will be classification and therefore the $Attacker_3$ will use the same neural network structure as $Attacker_2$ shown in Table 5.3.

The loss function of $Attacker_3$ will also be similar to the one of $Attacker_2$ i.e. the binary cross-entropy.

Given N ciphertexts $\left[C_{(0)}, C_{(1)}, ...C_{(N-1)}\right]$, and two sets of N plaintexts $\left[P^1_{(0)}, P^1_{(1)}, \cdots P^1_{(N-1)}\right]$, $\left[P^2_{(0)}, P^2_{(1)}, \cdots P^2_{(N-1)}\right]$ we define the loss function L_{A4} for $Attacker_3$ in Equation 5.11 below.

$$L_{A4} = -\frac{1}{N}\sum_{i=0}^{N-1}\sum_{j=1}^{2} y^j_{(i)} \log\left(\pi^j_{(i)}\right) \tag{5.11}$$

Where $y^j_{(i)} = 1$ if $C_{(i)}$ is the ciphertext of $P^j_{(i)}$ and 0 otherwise. Intuitively, $\pi^j_{(i)}$ is the probability that $P^j_{(i)}$ is the plaintext corresponding to the ciphertext $C_{(i)}$.

Therefore, $Attacker_3$ will learn by minimizing L_{A4}. Alice and the public keys generator now need to resist one more attacker, therefore their loss function will change to $L^4_{A,pub_{gen}}$ in the fourth experiment. $L^4_{A,pub_{gen}}$ is defined in Equation 5.12 below.

$$L^4_{A,pub_{gen}} = L^3_{A,pub_{gen}} - min(L_{A4}, 0.5) \tag{5.12}$$

Where $L^3_{A,pub_{gen}}$ is defined in Equation 5.10 and L_{A4} is defined in Equation 5.11. The min function is used to guarantee that Alice and the Public keys generator will not try to maximize $Attacker_3$'s classification errors as he can just invert his predictions and become 100% correct if he is trained to be 100% wrong.

As for the training, it is conducted in the same environment and using the same hyper parameters as in the third experiment. The results are shown in Figure 5.13 and Figure 5.14 below.

As we can see in Figure 5.13, from the beginning to the end of training, $Attacker_3$ is making

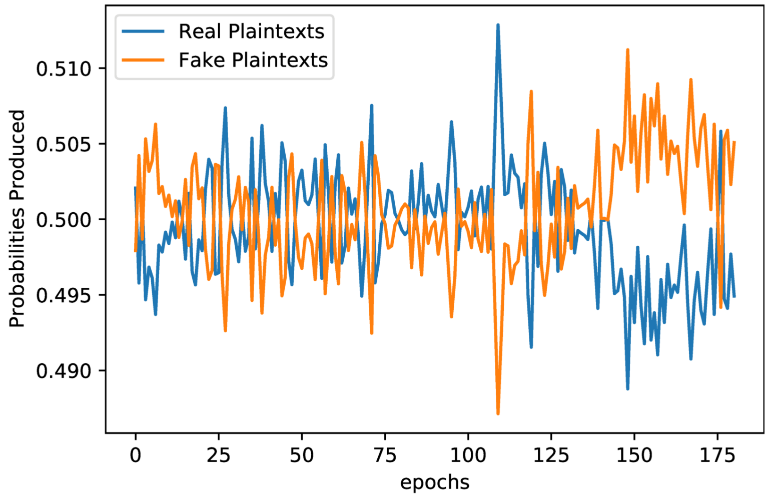

Figure 5.13: Probabilities produced by $Attacker_3$ on real and fake plaintexts in Experiment 4.

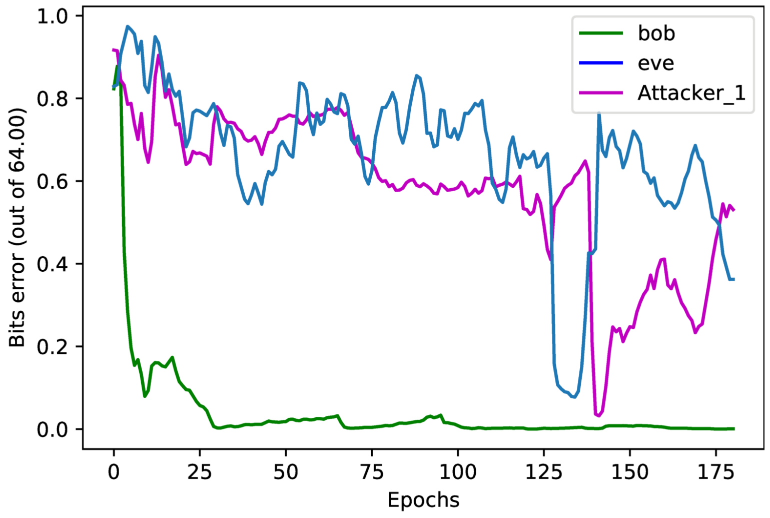

Figure 5.14: Bits decryption error for Bob, Eve and $Attacker_1$ in Experiment 4.

random predictions and all its probabilities are close to 0.5. Therefore, this attacker is not able to distinguish between the two plaintexts fed to him in order to tell which one corresponds to the given ciphertext. At the same time Bob's decryption accuracy started in the same way as in the second experiment as shown in Figure 5.14. Bob's accuracy stabilized at 100% after the $175th$ epoch where the accuracy of *eve* and $Attacker_1$ also stabilized at around 50%. We notice that the training sometimes takes longer to complete as it depends on the parameters obtained at the beginning of training. Similarly to the previous experiments, the training can stop when Bob is not making any error in decryption, $Attacker_2$ and $Attacker_3$ are producing random predictions and $Attacker_1$, Eve have a decryption error rate of around 50%. Table 5.4 shows that out of 3 sets of trials each composed of 15 unique experiment attempts, we have had 100% successful communications while none of the attackers were able to reach their training goal in decrypting the ciphertext or making correct predictions. A successful communication means that Bob has no decryption error. An attacker is considered successful in his attack when he gets over 20% better than a attacker making random guesses.

Table 5.4: Number of successful communications attempted and achieved in Experiment 4.

Trial	Number of Experiments	Successful Communications	Successful attacks by one of the attackers
1	15	15	0
2	15	15	0
3	15	15	0

5.5 How does our model gets its security?

Classic encryption techniques such as RSA [11] rely on some hardness assumptions that prove that the models are secure. However, as our models do not rely on such assumptions, one might wonder how we get our security guarantees. The security of our model is backed up by the attackers used in the model training as they are trained to be the best version of themselves. For example, the attackers performing the CPA and CCA attacks are trained with the binary cross entropy to be the best version of themselves. Alice will have to beat the strongest version of these attackers. The other attackers are also trained to be the best version of themselves with their respective losses.

5.6 Advantages of our technique

Similarly to existing symmetric key adversarial encryption techniques [1, 4, 9], our model does not rely on any hardness assumption of solving a mathematical problem to provide security. Additionally, it does not require any key exchange phase to share a symmetric key that will be used for encryption as in the previously proposed models [1, 4, 9]. It also makes it easy to generate a new pair of keys using the public and private keys generators if the private key gets leaked. As we have shown in the experiments, our model is able to generate its own pairs of asymmetric information (i.e. public and private keys) and perform the encryption with the public key and the decryption with the private key while preventing the eavesdroppers and attackers from reading the communication using public or leaked private keys. Other classic cryptography techniques such as RSA [11] take some time to generate a new pair of keys as the computation is expensive but our model can generate a new pair of keys through a very simple operation. The only limitation is that the neural networks take some time to train for the first time and that they need another training session in case their parameters get leaked. We are thus providing an efficient GANs model able to generate its own keys and learn encryption using asymmetric information to protect communications while preserving the same security and accuracy provided in symmetric key encryption contributions [9, 4].

Table 5.6 shows the key differences between our model and the model proposed in [1].

Model	Encryption Type	Number of Attackers that the model trains against	Training time	Security of Ciphertexts	Pre-shared information required
Our Model	Asymmetric	4	30-120 minutes	High as the model trains against multiple types of attacks.	None
Model in [1]	Symmetric	1	15-30 minutes	Low as the model trains against just one type of attack.	A symmetric key

Table 5.5: Key differences between our model and the model proposed in [1].

5.7 Comparison with existing Techniques

Table 5.6 summarizes the differences between our technique and the most popular similar contributions. We can see that we are achieving encryption using asymmetric information while preserving the same synchronization time which is slower across all contributions compared to the original model. Encryption and Decryption operations are the same across all models. Additionally, and in all the models including ours, the receiver is able to decrypt the received messages with 100% accuracy after training. Performance wise, it is difficult to give a concrete training time because each model has been trained by its authors on different machines. We used a Google Colab pro subscription to train our model and the GPU used in our experiments was an NVIDIA P100. Using this GPU, our model with all the 4 adversaries takes approximately two hours to complete 200 epochs with the Hyperparameters indicated in Section 5.3.3. As a reference, we have also implemented the original model proposed in [1] and ran it with the same GPU as our model. The results show that training the neural networks with the same hyperparameters indicated in Section

5.3.3, the synchronisation is done in approximately 20 minutes which is faster than our model. As for the model proposed in [4], we have implemented their model using Tensorflow V2 and ran it using the same GPU that we used for our model. The results have shown that while it was a little faster than our model, it was still too slow compared to the original model. The authors of [9] did not disclose information about their training time and we did not implement and test it. As for encryption and decryption times, it can be done in a negligible time with all of the models because it is a simple forward pass through the neural network and the structure should not affect the encryption or decryption times.

Table 5.6: Comparison of our model with existing symmetric encryption models.

Contribution	**Type of Encryption**	**Sync. Time**	**Encryption and Decryption time**
Our Model	**Asymmetric**	Slower than the original model.	Fast
Learning to Protect Communications with Adversarial Neural Cryptography [1].	Symmetric (Keys must be exchanged prior to communication)	Fast	Fast
Learning Perfectly Secure Cryptography to Protect Communications with Adversarial Neural Cryptography [9].	Symmetric (Keys must be exchanged prior to communication)	Undisclosed by the authors.	Fast
Information encryption communication system based on the adversarial networks Foundation [4].	Symmetric (Keys must be exchanged prior to communication)	Slower than the original model.	Fast

5.8 Applications Scenarios

Our model can be used to secure communications between two parties. As our model learns asymmetric encryption, it can not only be used to agree on a symmetric key for use with a symmetric encryption key algorithm such as AES [12] but also to communicate directly using the public key.

Figure 5.15 shows the process of establishing a direct communication with the public key and Figure 5.16 shows how our model can be used to agree on a symmetric key for use with a symmetric encryption key algorithm. The scenario in the figures assumes that the neural networks have already been trained.

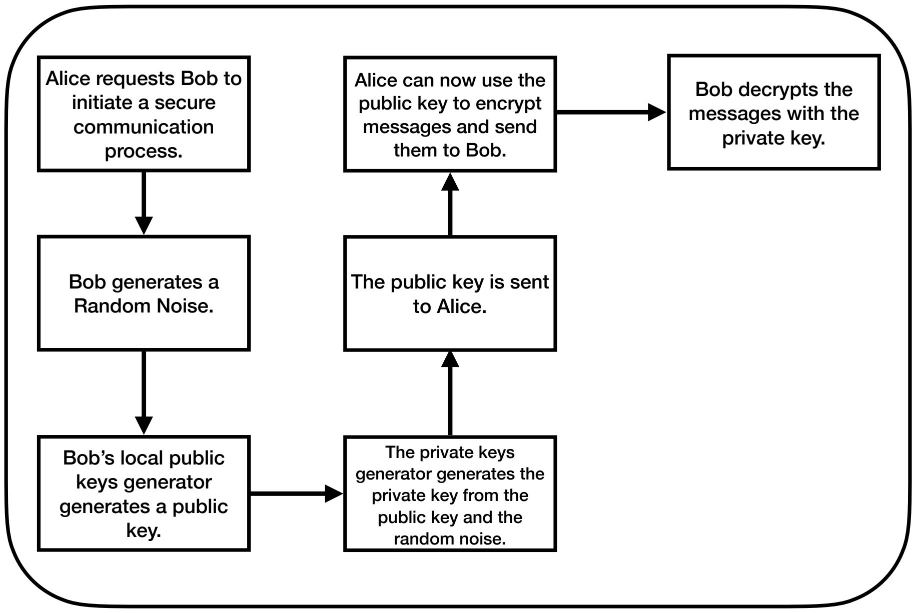
Alice requests Bob to initiate a secure communication process.
Bob generates a Random Noise.
Bob's local public keys generator generates a public key.
The private keys generator generates the private key from the public key and the random noise.
The public key is sent to Alice.
Alice can now use the public key to encrypt messages and send them to Bob.
Bob decrypts the messages with the private key.

Figure 5.15: Process of establishing a secure direct communication using our model. The communication channel between Alice and Bob is authenticated but not confidential. The random noise generated by Bob is done using a pseudo random generator and does not involve Bob's Neural Network.

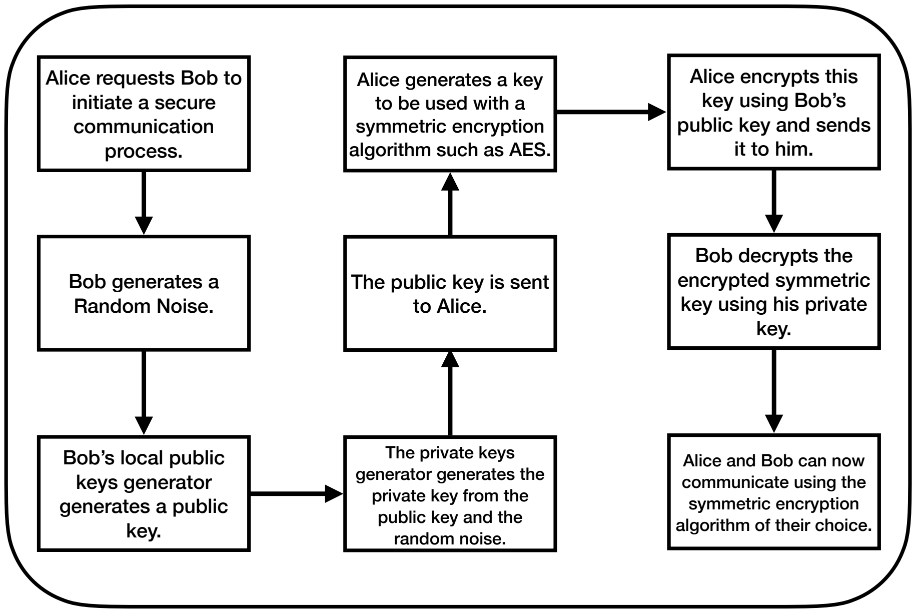
Alice requests Bob to initiate a secure communication process.
Bob generates a Random Noise.
Bob's local public keys generator generates a public key.
The private keys generator generates the private key from the public key and the random noise.
The public key is sent to Alice.
Alice generates a key to be used with a symmetric encryption algorithm such as AES.
Alice encrypts this key using Bob's public key and sends it to him.
Bob decrypts the encrypted symmetric key using his private key.
Alice and Bob can now communicate using the symmetric encryption algorithm of their choice.

Figure 5.16: Process of agreeing on a symmetric key using our model. The communication channel between Alice and Bob is authenticated but not confidential. The random noise generated by Bob is done using a pseudo random generator and does not involve Bob's Neural Network.

Chapter 6

Conclusion and Future work

Adversarial cryptography is a promising research direction that is likely to become a post quantum cryptography primitive due to the fact that it does not get its security from any hardness assumptions. We have introduced adversarial cryptography in this thesis by detailing the most significant contributions and then presented our contributions in the field.

In our most significant contributions, we proposed a neural networks model that can learn multi-party adversarial encryption. The model allows multiple neural networks to synchronize and learn the same encryption/decryption algorithm. This model is also more secure compared to the original model as it trains to produce ciphertexts that are resistant to multiple cryptographic attacks. Our methodology can be a candidate for providing post-quantum security when multiple servers/NNs learn to communicate among themselves as the encryption has been shown to be the same as the one time pad. We also shown how our model can be used for other applications. We have shown how to build an information theoretic secure secret sharing scheme for General Access Structures. In the proposed secret sharing scheme, the Dealer trains and synchronizes with multiple shareholders and then splits a secret into N shares and distributes it among them.

To solve the issue of sharing secret keys in neural networks that learn symmetric encryption, we have presented a multi-agent adversarial neural networks model that is able to learn to protect a communication using asymmetric information. The experiments have shown that our model is able to learn to encrypt a communication using a pair of public and private keys in the presence of an eavesdropper. The experiments have also shown that our model is able to adapt to different attackers that have access to different kinds of information. Experiment 2 showed that the neural networks are able to learn a plaintext-to-ciphertext mapping that does not entirely rely on the private key which allowed them to prevent $Attacker_1$ from decrypting the ciphertexts despite having access to all the private keys. Experiment 3 showed that the ciphertexts generated by Alice are indistinguishable and an attacker having access to two ciphertexts and one plaintext cannot tell which ciphertext belongs to that plaintext. The last experiment, Experiment 4, showed that the ciphertexts do not contain any information related to the original plaintext as $Attacker_3$ failed to differentiate two plaintexts apart and tell which one has been encrypted to the given ciphertext.

Future work directions include testing the ciphertexts produced against quantum computers in order to assess their resistance to quantum attacks as well as brute force attacks. Exploring the possiblity of learning cryptographic techniques such as digital signature is also an interesting research direction.

Bibliography

[1] M. Abadi and D. G. Andersen, "Learning to protect communications with adversarial neural cryptography," *CoRR*, vol. abs/1610.06918, 2016.

[2] J. Shi, S. Chen, Y. Lu, Y. Feng, R. Shi, Y. Yang, and J. Li, "An approach to cryptography based on continuous-variable quantum neural network," *Scientific Reports*, vol. 10, no. 1, p. 2107, 2020.

[3] I. Meraouche, S. Dutta, and K. Sakurai, "3-party adversarial cryptography," in *Advances in Internet, Data and Web Technologies* (L. Barolli, Y. Okada, and F. Amato, eds.), (Cham), pp. 247–258, Springer International Publishing, 2020.

[4] Z. Li, X. Yang, K. Shen, R. Zhu, and J. Jiang, "Information encryption communication system based on the adversarial networks foundation," *Neurocomputing*, vol. 415, pp. 347–357, 2020.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[6] I. Kanter, W. Kinzel, and E. Kanter, "Secure exchange of information by synchronization of neural networks," *EPL (Europhysics Letters)*, vol. 57, 02 2002.

[7] L. Zhou, J. Chen, Y. Zhang, C. Su, and M. A. James, "Security analysis and new models on the intelligent symmetric key encryption," *Computers & Security*, vol. 80, pp. 14 – 24, 2019.

[8] J. Hayes and G. Danezis, "Generating steganographic images via adversarial training," *Advances in Neural Information Processing Systems*, vol. 30, pp. 1954–1963, 2017.

[9] M. Coutinho, R. de Oliveira Albuquerque, F. Borges, L. J. García-Villalba, and T. Kim, "Learning perfectly secure cryptography to protect communications with adversarial neural cryptography," *Sensors*, vol. 18, no. 5, p. 1306, 2018.

[10] E. Salguero, W. Fuertes, and J. Lascano, "On the development of an optimal structure of tree parity machine for the establishment of a cryptographic key," *Security and Communication Networks*, vol. 2019, pp. 1–10, 03 2019.

[11] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, p. 120–126, Feb. 1978.

[12] T. Jamil, "The rijndael algorithm," *IEEE Potentials*, vol. 23, pp. 36–38, April 2004.

[13] S. Prajapat and R. S. Thakur, "Various approaches towards crypt-analysis," *International Journal of Computer Applications*, vol. 127, no. 14, pp. 15–24, 2015.

[14] R. Spillman, M. Janssen, B. Nelson, and M. Kepner, "Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers," *Cryptologia*, vol. 17, no. 1, pp. 31–44, 1993.

[15] N. D. Truong, J. Y. Haw, S. M. Assad, P. K. Lam, and O. Kavehei, "Machine learning cryptanalysis of a quantum random number generator," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 2, pp. 403–414, 2018.

[16] J. So, "Deep learning-based cryptanalysis of lightweight block ciphers," *Security and Communication Networks*, vol. 2020, pp. 1–11, 07 2020.

[17] F. Wang, R. Ni, J. Wang, Z. Zhu, X. Chen, and Y. Hu, "Novel fully convolutional network for cryptanalysis of cryptosystem by equal modulus decomposition," *Laser Physics Letters*, vol. 17, p. 095201, jul 2020.

[18] A. Klimov, A. Mityagin, and A. Shamir, "Analysis of neural cryptography," in *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings* (Y. Zheng, ed.), vol. 2501 of *Lecture Notes in Computer Science*, pp. 288–298, Springer, 2002.

[19] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[20] Y. Gao, R. Singh, and B. Raj, "Voice impersonation using generative adversarial networks," *CoRR*, vol. abs/1802.06840, 2018.

[21] Y. Kataoka, T. Matsubara, and K. Uehara, "Image generation using generative adversarial networks and attention mechanism," in *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pp. 1–6, 2016.

[22] A. Siarohin, S. Lathuilière, E. Sangineto, and N. Sebe, "Appearance and pose-conditioned human image generation using deformable gans," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2019.

[23] M. Yedroudj, F. Comby, and M. Chaumont, "Steganography using a 3-player game," *Journal of Visual Communication and Image Representation*, vol. 72, p. 102910, 2020.

[24] Y. Zhu, D. V. Vargas, and K. Sakurai, "Neural cryptography based on the topology evolving neural networks," in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pp. 472–478, 2018.

[25] I. Meraouche, S. Dutta, S. K. Mohanty, I. Agudo, and K. Sakurai, "Learning multi-party adversarial encryption and its application to secret sharing," *IEEE Access*, vol. 10, pp. 121329–121339, 2022.

[26] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[27] T. Dash, S. N. Dambekodi, P. N. Reddy, and A. Abraham, "Adversarial neural networks for playing hide-and-search board game scotland yard," *Neural Computing and Applications*, vol. 32, pp. 3149–3164, 2018.

[28] A. Ruttor, W. Kinzel, and I. Kanter, "Dynamics of neural cryptography," *Physical Review E*, vol. 75, no. 5, p. 056104, 2007.

[29] N. Prabakaran and P. Vivekanandan, "A new security on neural cryptography with queries," 2008.

[30] O. M. Reyes and K.-H. Zimmermann, "Permutation parity machines for neural cryptography," *Physical Review E*, vol. 81, no. 6, p. 066117, 2010.

[31] L. F. Seoane and A. Ruttor, "Successful attack on permutation-parity-machine-based neural cryptography," *Physical Review E*, vol. 85, no. 2, p. 025101, 2012.

[32] A. Beimel, "Secret-sharing schemes: A survey," in *Coding and Cryptology*, (Berlin, Heidelberg), pp. 11–46, Springer Berlin Heidelberg, 2011.

[33] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[34] G. R. Blakley, "Safeguarding cryptographic keys," in *AFIPS 1979*, pp. 313–317, 1997.

[35] M. Ito, A. Saito, and T. Nishizeki, "Multiple assignment scheme for sharing secret," *J. Cryptology*, vol. 6, no. 1, pp. 15–20, 1993.

[36] E. F. Brickell, "Some ideal secret sharing schemes," in *Workshop on the Theory and Application of of Cryptographic Techniques*, pp. 468–475, Springer, 1989.

[37] M. Nojoumian and D. R. Stinson, "Sequential secret sharing as a new hierarchical access structure," *J. Internet Serv. Inf. Secur.*, vol. 5, no. 2, pp. 24–32, 2015.

[38] T. Tassa, "Hierarchical threshold secret sharing," *Journal of cryptology*, vol. 20, no. 2, pp. 237–264, 2007.

[39] H. Krawczyk, "Secret sharing made short," in *Annual international cryptology conference*, pp. 136–146, Springer, 1993.

[40] M. Naor and A. Shamir, "Visual cryptography," in *Workshop on the Theory and Application of of Cryptographic Techniques*, pp. 1–12, Springer, 1994.

[41] W. Zheng, K. Wang, and F.-Y. Wang, "Gan-based key secret-sharing scheme in blockchain," *IEEE Transactions on Cybernetics*, vol. 51, no. 1, pp. 393–404, 2021.

[42] X. Wang, H. Shan, X. Yan, L. Yu, and Y. Yu, "A neural network model secret-sharing scheme with multiple weights for progressive recovery," *Mathematics*, vol. 10, no. 13, p. 2231, 2022.

[43] D. V. Vargas and J. Murata, "Spectrum-diverse neuroevolution with unified neural models," *CoRR*, vol. abs/1902.06703, 2019.

[44] W. Zheng, K. Wang, and F. Wang, "Gan-based key secret-sharing scheme in blockchain," *IEEE Transactions on Cybernetics*, pp. 1–12, 2020.

[45] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *Proceedings - 2008 IEEE Symposium on Security and Privacy, SP*, Proceedings - IEEE Symposium on Security and Privacy, pp. 111–125, Sept. 2008. 2008 IEEE Symposium on Security and Privacy, SP ; Conference date: 18-05-2008 Through 21-05-2008.

[46] E. Finn, X. Shen, D. Scheinost, M. Rosenberg, H. Jessica, M. Chun, X. Papademetris, and R. Constable, "Functional connectome fingerprinting: Identifying individuals using patterns of brain connectivity," *Nature neuroscience*, vol. 18, 10 2015.

[47] C. Huang, P. Kairouz, X. Chen, L. Sankar, and R. Rajagopal, "Context-aware generative adversarial privacy," *CoRR*, vol. abs/1710.09549, 2017.

[48] D. P. Kingma and J. A. Ba, "A method for stochastic optimization. arxiv 2014," *arXiv preprint arXiv:1412.6980*, vol. 434, 2019.

[49] I. Meraouche, S. Dutta, and K. Sakurai, "3-party adversarial steganography," in *Information Security Applications* (I. You, ed.), (Cham), pp. 89–100, Springer International Publishing, 2020.

[50] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[51] S. Tarasenko, N. Andriyanov, and A. Gladkikh, "Analysis of the applicability of artificial neural networks for the post-quantum cryptography algorithms development," *Journal of Physics: Conference Series*, vol. 2032, p. 012026, 10 2021.

[52] I. Meraouche, S. Dutta, and K. Sakurai, "Tree parity machine based symmetric encryption: a hybrid approach," in *Mathematics and Computing* (R. K. et al (eds.), ed.), (Singapore), pp. 1–11, Springer Singapore, 2023.

[53] T. Godhavari, N. Alamelu, and R. Soundararajan, "Cryptography using neural network," in *2005 Annual IEEE India Conference-Indicon*, pp. 258–261, IEEE, 2005.

[54] M. Sýs, Z. Riha, V. Matyas, K. Marton, and A. Suciu, "On the interpretation of results from the nist statistical test suite," vol. 18, pp. 18–32, 01 2015.

Appendix A

Tree Parity Machines based Encryption scheme

A.1 Introduction

In this chapter, we put forward a symmetric key encryption technique that does not require any common pre-shared "knowledge" between the parties. More specifically, we use a type of neural networks called Tree Parity Machines (TPMs) which when synchronized, enable two parties to reach a common state. The common state can be used to establish a common secret key. Our method makes use of the Tree Parity Machines to reach a common state between the parties communicating and encrypt the communications with an ElGamal-type encryption methodology. The advantage of our implementation is that the initial key exchange method is fast, lightweight and believed to become a post-quantum candidate. We have analyzed the randomness of the produced ciphertexts from our system using NIST randomness tests and the results are included. We also demonstrate security against chosen plaintext attacks.

In a symmetric key encryption setup, the sender and the receiver must share a common randomness that allows them to produce a secret key that can be used to encryption and decryption. Exchanging symmetric keys is a challenging issue in cryptography. There are multiple techniques that allow two or more parties to remotely perform a key exchange operation. One of the most common techniques is the Diffie-Hellman protocol [26]. While this technique is known to be secure, it is vulnerable against Quantum Attacks and researchers are working on building new techniques that are resistant against Quantum Attacks. One of the previously discussed techniques involve neural networks.

In this chapter, we propose a neural networks based symmetric encryption scheme that makes use of Tree Parity Machines [6] to build a secure symmetric encryption scheme.

Tree Parity Machines (TPMs) enable two parties to reach a common state. The common state can be used to derive a secret key. Our method [52] makes use of the Tree Parity Machines to reach a common state between the parties communicating and encrypt the communications with an ElGamal-type encryption methodology. The advantage of our implementation is that the initial key exchange method is fast, lightweight and believed to become a post-quantum candidate. We have analyzed the randomness of the produced ciphertexts from our system using NIST randomness tests and the results are included in Section A.3. We also demonstrate security against chosen plaintext attacks.

With this work, we aim to take initial steps into hybridising existing cryptography techniques with recent neural networks based cryptography techniques. Concretely, we aim to realize a symmetric-key encryption scheme based on the hardness of discrete logarithm problem and using the Tree Parity Machine proposed by Kanter, Kinzel and Kanter [6]. Tree Parity Machines are neural networks composed of three layers: `Input layer, Hidden layer and Output layer`. Kanter et al. [6] show that two Tree Parity Machines can be synchronized and obtain the same state that can be later used to generate a common secret key.

We choose the Tree Parity Machines to establish key(s) in order to gain more speed and flexibility in the key generation process. Additionally, the Tree Parity Machines exchange does not rely on a problem which is hard to solve by probabilistic polynomial time adversaries and therefore can be seen a potential quantum-safe candidate. To the best of our knowledge, such a hybrid approach towards constructing a symmetric-key encryption has not been considered in the literature before.

The content has of this Appendix has been Reproduced with permission from Springer Nature from our original published paper [52].

A.2 A Hybrid Method for Symmetric Encryption

Existing neural networks based symmetric key encryption schemes arising from TPMs [6, 53] or from adversarial neural networks [1, 3] do not provide any provable security. The security of the aforementioned proposals were not based on any hard problem like discrete log or factorization problem. In fact, [18, 7] analyzed the shortcomings of the proposals which imply vulnerability against multiple attacks including chosen plaintext attacks.

In this section, we present a hybrid approach to develop a symmetric key encryption which uses a TPMs to generate common randomness between two parties and a set up for ElGamal type encryption. The main advantage of our technique is that there is no need to share a common (secret) state in advance like the work in [1]. The synchronization of the Tree Parity Machines of the two parties will allow them to reach a common secret state that can be used to generate secret keys.

For the sake of completeness we summarize the Diffie-Hellman key exchange protocol below (between Alice and Bob):

1. q is a prime, G is a cyclic group of order q in which DDH is hard and g is a generator of G. The information (G, g, q) is made public.
2. Alice uniformly selects $r_A \in \mathbb{Z}_q - \{0\}$; computes $K_A = g^{r_A} \mod q$; sends K_A to Bob.
3. Bob uniformly selects $r_B \in \mathbb{Z}_q - \{0\}$; computes $K_B = g^{r_B} \mod q$; sends K_B to Alice.
4. Alice computes $K_B^{r_A}$ and Bob computes $K_A^{r_B}$ locally.
5. Both output $g^{r_A \cdot r_B}$ as their common secret key.

In our proposal, we let the parties Alice and Bob use TPMs to generate the common randomness which serves as the exponent of g to establish the secret key. The details are given below.

1. (G, g, q) are public values such that G is a DDH group.
2. Alice and Bob synchronize their TPMs as shown before in Section 2.3.1 and Figure A.1 to obtain same vector of parameters W.
3. Alice and Bob generate a common secret Key from the weight-vector W (Details in Section A.2.1).

Figure A.1 gives a pictorial depiction of the steps during the synchronization process of TPMs.

As we can see in Figure A.1, the two TPMs initialise their neural network randomly and then keep updating it during training until they have equal parameters. The Hebbian Learning technique is shown [6] to be able to synchronize two TPMs to have equal parameters W starting from a random non-commmon state.

Each training iteration to update the parameters $W_{i,j}$ for each neural network is performed as follows:

- The TPMs reshape the input vector to have the same dimensions as the parameters matrix W.
- The TPMs calculate the product of the input R and the parameters W.
- The TPMs sum up the rows of the product and put each result in a new vector.
- The TPMs create a new vector containing the sign of each element in the previous vector.
- The product of the elements in this new vector is the output of their neural network τ.

We notice that for a small example with an *output* $= 1$ and a signs vector of size 3. There are four possible signs vectors: (1, 1, 1), (1, -1, -1), (-1, 1, -1), (-1, -1, 1) which means if the two

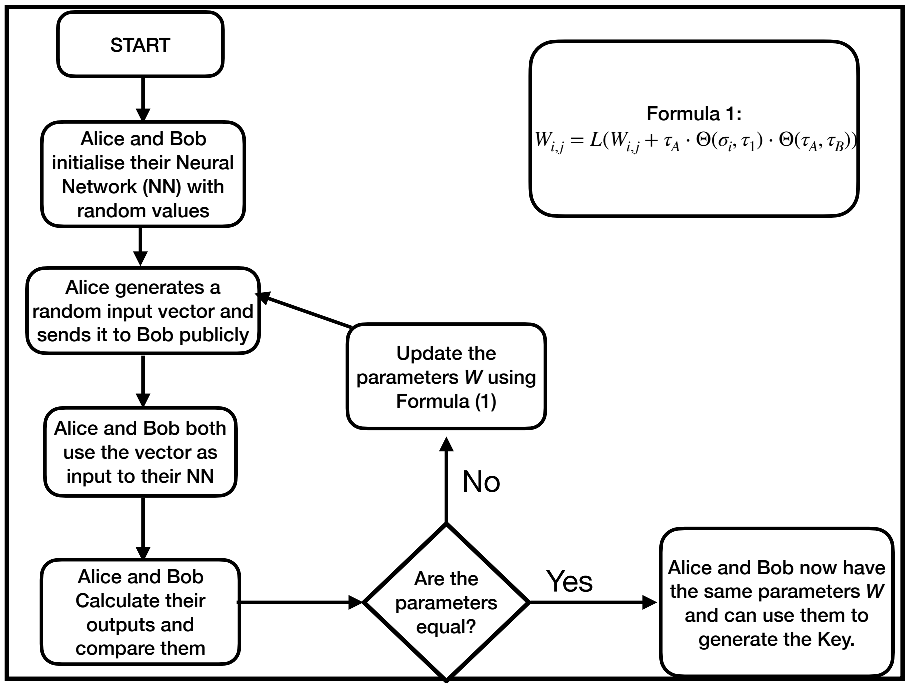
START
Alice and Bob initialise their Neural Network (NN) with random values
Alice generates a random input vector and sends it to Bob publicly
Alice and Bob both use the vector as input to their NN
Alice and Bob Calculate their outputs and compare them
Are the parameters equal?
No
Update the parameters W using Formula (1)
Yes
Alice and Bob now have the same parameters W and can use them to generate the Key.
Formula 1:
$W_{i,j} = L(W_{i,j} + \tau_A \cdot \Theta(\sigma_i, \tau_1) \cdot \Theta(\tau_A, \tau_B))$

Figure A.1: Synchronization process between Alice and Bob.

TPMs got the output of 1, there is one chance out of 4 that they had the same signs vector and three chances out of 4 that they had a different one. As a general rule, there are $p - 1$ out of p possibilities that the outputs of the TPMs are equal but their weights are different (p is the size of the signs vector). However, this is an expected behavior by the Hebbian learning rule update steps above are used to guide the parameters to the same values.

As there is a clear possibility that the TPMs will have similar outputs but different weights, the TPMs need a way to check that their weights are equal after some training iterations. One can use hash functions or other privacy preserving techniques to allow the TPMs to compare their weights while keeping them private.

In the event where they have different weights, they update their parameters with the Hebbian learning rule which is shown as an equation in Formula A.1

$$W_{i,j} = W_{i,j} + (R_{i,j} \cdot \tau_A \cdot \Theta(\sigma_i, \tau_A) \cdot \Theta(\tau_A, \tau_B)) \tag{A.1}$$

Where W is the parameters matrix, R is the input vector, $\tau_{A,B}$ are the outputs of the TPMs A and B respectively, σ_i is the ith value of the signs vector calculated in step 4 before and the Θ function is defined in equation A.2.

$$\Theta(x, y) = \begin{cases} 1 & if x = y \\ 0 & otherwise \end{cases} \tag{A.2}$$

A.2.1 Key Generation

The generation of Key from the weight-vector W and the public generator g are as follows. When the neural networks are synchronized, both the parties have the same parameters W which is an array of integers $W = [a_0, a_1, \cdots, a_{K \cdot N}]$ with a taking values between $-L$ and $+L$.

All the elements of W are converted to binary and concatenated together transforming negative numbers into positive numbers. The result of the concatenation CR is then used to calculate the secret key in binary.

The common secret key is calculated as: $\mathsf{Key} = \widetilde{\mathsf{CR}}$, a conversion of CR into an element of $\mathbb{Z}_q$. In the following we describe the encryption scheme.

Method 1. In the first method, once the key is obtained by Alice and Bob, encryption and decryption algorithms are similar to the ElGamal.

1. Alice (or Bob) randomly chooses $r \leftarrow_{\$} \mathbb{Z}_q$.
2. Alice (or Bob) computes:
   - g^r and outputs $(C_1, C_2) \longleftarrow (g^r, (g^r)^{\mathsf{Key}} \cdot M)$ as ciphertext.
3. Bob (or Alice) receives (C_1, C_2)
4. Bob (or Alice) Computes $M \leftarrow \frac{C_2}{C_1^{\mathsf{Key}}}$

Security against chosen plaintext attacks. It is well-known that over a DDH group the ElGamal encryption provides IND-CPA (Indistinguishability under Chosen-Plaintext Attack) security. The IND-CPA security of the above mentioned scheme can be proved using the same proof strategy and is omitted from this draft.

Method 2. The second method generates a mutual random vector between Alice and Bob and uses it to transform the secret key every time a communication is needed.

The encryption and decryption works as follows for the second method:

1. Alice (or Bob) chooses a random vector r as long as the input of their Neural Network and sends it to the other party.
2. Alice and Bob both pass the vector r through their neural network and calculate the output without the activation (in order to get a vector of integers).
3. Now they both have the same output vector R
4. In order to encrypt a message, Alice (or Bob) does the following:
   (a) Choose the next unused element e with index i inside the vector R.
   (b) Calculate $Key' = Key * e$. i.e. multiplies the key by the element e.
   (c) To encrypt a message M as C, the sender calculates: $C = M * Key' \mod q$.
   (d) The sender sends the pair (C, i) where i is the index of the element e used to transform the key.
5. Bob (or Alice) receives (C, i).
6. Bob (or Alice) gets the ith element e from the vector R.
7. Bob (or Alice) calculates $Key' = Key * e$.
8. Bob (or Alice) can now decrypt C using Key' with the following formula: $M = C * Key'^{-1} \mod q$.

Security against chosen plaintext attacks Similarly to method one, a randomness is introduced to the key at every exchange. The only inconvenient of this method is that the two parties will need a new random vector R everytime they have used all the elements of the current vector R. However this method provides slightly faster encryption as it is only a multiplication whereas in the first method calculating $(g^r)^{\mathsf{Key}}$ is required and is more expensive to calculate. This method is more appropriate for IoT devices such as sensors that do not need to communicate too often with a server. For example sensors that send average weather during the day every 24 hours.

A.3 Results, Training Time and Encryption Time

We conducted hundreds of simulations training Alice and Bob to perform the synchronization of TPMs and get a common secret key to use it for encryption as defined in the previous section. The table A.3 below summarizes our results over 3 sets of 200 training simulations. Each simulation contains an adversary Eve with the same structure as Bob. Eve will try to get the secret key by mimicking Bob's behavior to synchronize with Alice and Bob without their knowledge. Technically speaking, Eve updates her parameters when her output is the same as Bob's and Alice's.

Table A.1: Table summarizing the time and number of exchanges required to synchronize

Sim.	Number of Simulations	Average Number of exchanges	Minimum Sync Time	Maximum Sync Time	Average Sync Time
1	300	222	124ms	610ms	266ms
2	300	229	110ms	606ms	278ms
3	300	225	73ms	580ms	273ms

As shown in Table A.3, we performed 900 simulations each divided in groups of 300. We can see that the average number of exchanges required to synchronize is around 230 exchanges. The synchronization takes on average around 300 milliseconds with a minimum of $73ms$ and a maximum of $610ms$ synchronisation time recorded. During all of our 900 simulations, there has been no simulation where Eve has been able to secretly synchronize with Alice and Bob and end up with the same key as them. We have used $K = 8$, $N = 16$, $L = 8$ as parameters for the Tree Parity Machine. This means that the Tree Parity Machine has $K \cdot N = 128$ inputs neurons, $K = 8$ hidden neurons that can take a value between $-L$ and $+L$ (i.e. Between -8 and $+8$).

We have chosen these values as they have been found to be optimal in the experiments conducted in [10]. The authors that among all the different structures they used, the structure that uses the values $K = 8$, $N = 16$, $L = 8$ is the most secure and out of the 1 million simulations they have performed, only 1 successful synchronization by Eve has been recorded. This is the reason why have used this technique as it is the safest according to the work done in [10].

A.4 Security Analysis

The security of the encryption algorithm is dependent on the randomness of the key generated using the Tree Parity Machine. If the generated key is close to uniformly random then the security of the protocol can be reduced to the security of the key exchange.

A.4.1 NIST Statistical test results

To test the randomness of the keys generated and therefore the security of our proposed method, we have conducted the NIST statistical test on a series of ciphertexts generated by our implementation of the proposed model using the two methods.

We have generated approximately two sets of 500 ciphertexts each with a unique key. The first set being generated with the first method and the second set with the second method. All the ciphertexts had a size of 2048 bits.

Both the two methods got approximately the same results which are detailed in Table A.4.1 below.

In order to compare our results, we also conducted the NIST randomness test on ElGamal with the same amount of sample ciphertexts as our first encryption method is similar to that of ElGamal.

Test Name	Result with Method 1	Result with Method 2	Result with Original ElGamal Scheme
`Frequency`	SUCCESS	SUCCESS	SUCCESS
`BlockFrequency`	SUCCESS	SUCCESS	SUCCESS
`CumulativeSums`	SUCCESS	SUCCESS	SUCCESS
Runs	SUCCESS	SUCCESS	SUCCESS
LongestRun	SUCCESS	SUCCESS	SUCCESS
`Rank`	SUCCESS	SUCCESS	SUCCESS
`FFT`	SUCCESS	SUCCESS	SUCCESS
`Non Overlapping Template`	FAILED	FAILED	FAILED
`OverlappingTemplate`	SUCCESS	SUCCESS	SUCCESS
`Universal`	FAILED	FAILED	FAILED
`ApproximateEntropy`	FAILED	FAILED	FAILED
Serial	SUCCESS	SUCCESS	SUCCESS
`LinearComplexity`	FAILED	FAILED	FAILED

Table A.2: NIST randomness test results on our model and ElGamal.

We can see that the method has passed most of the tests performed by the NIST statistical test. According to [54], the samples are considered random and therefore secure if they pass at least 7 NIST statistical tests which is our case.

As we can see, The NIST randomness test shows that our ciphertexts are uniformly random which implies that our method can be used at a production level securely.

A.4.2 Security against Chosen Plaintext Attacks

We have already mentioned the IND-CPA security of the encryption scheme described in this chapter. Due to lack of space we do not give the details. However, in order to prove that the scheme is secure against chosen plaintext attack we must analyze the randomness/unpredictability of the key $\widetilde{\mathsf{CR}}$ in $\mathbb{Z}_q$ generated by the TPMs.

Technique	Encryption Type	Relies on hardness assumptions?	Needs pre-shared information?	Passes the NIST Randomness test?
Encryption Method 1	Symmetric	Yes	No	Yes
Encryption Method 2	Symmetric	No	No	Yes
ElGamel Encryption Scheme	Asymmetric	Yes	No	Yes

Table A.3: Table showing they key differences between the two methods of our proposed encryption technique as well as the original elgamel protocol.

A.4.3 Other attacks

Additionally to the NIST Statistical test, we have noticed that the key exchange using the Tree Parity Machines has been proven to be vulnerable against multiple attacks as shown in [18]. The authors in [18] show three different attacks that can be applied in order to allow a third party Eve to simulate the exchange between Alice and Bob and end up with the same weights array W. However as shown in [10, 28], this can be avoided by increasing the size of the neural networks (i.e. number of input neurons and neurons in the hidden layer).

A.5 Comparison between the two methods and ElGamel

Table A.5 shows a comparison between our two methods and the original ElGamel public key encryption scheme.

A.6 Comparison with existing works

We compare our proposed implementation with the model of Abadi and Andersen [1] in terms of multiple factors as shown in Table A.4.

We can see that our model outperforms the model proposed by Abadi and Andersen [1] in terms of synchronization time. This is mainly due to the large CNN (Convolutional Neural Network) structure used by Abadi and Andersen versus a relatively smaller unique hidden layer neural network structure used in our model. Our model also has the advantage of not relying on any initial common state or pre-shared secret such as a secret key. As for the key length, ElGamal encryption needs large keys therefore the key generated with our model is quite large versus a small 32 or 64 bits key in the model by Abadi and Andersen [1]. The encryption time is roughly

Model	Min & Max Sync. Time Recorded*	Pre-shared info.	Key Length	Enc. Type	Cipher-text size
Our Model	73 - 10^3ms	No	500 bits	ElGamal based affine cipher.	Large
Model in [1]	15 to 30 minutes.	Yes	32 bits	Blackboxed	Small

Table A.4: Comparison of our work with the work by Abadi and Andersen [1]. *The encryption process in our model does not require multiple iterations as opposed to [1].

the same for both the techniques as it is a simple neural network feed forward operation in the model used by Abadi and Andersen [1] and a simple mathematical multiplication in our proposed technique. However the encryption technique learned by the neural networks in [1] is blackboxed and cannot be known. The messages in our model do not need to be as long as the key in contrary to the model in [1] but this comes for the price of larger ciphertexts in our model versus ciphertexts as long as the message in the model in [1]. Lastly, it has not been verified that the Tree Parity Machines can be synchronized with multiple parties simultaneously in contrary to the model in [1] where it is possible as shown in [3].

Additionally, the authors in [7] have shown that the original model by Abadi and Andersen [1] has only passed the `BlockFrequency` and the `NonOverlapping` Template and failed the rest. However the improved versions in [4] and [9] achieve better results.

A.7 Limitations of this technique

In contrary to classic techniques such as the Diffie-Hellman key exchange protocol [26], the TPM based key exchange is not a guaranteed to reach a state where the weights are equal with any initial random state. While the Hebbian learning rule and the synchronisation process proposed in the original TPM exchange model [6] is shown to always reach a state where the parameters are equal, the TPMs need a secure mechanism to verify that they have reached such a state. The TPMs are therefore obliged to use some privacy preserving techniques or hash functions to compare the values of their weights in order to make sure that they are equal.

Appendix B

Published Papers

1. 3-Party Adversarial Cryptography
   Proceedings of the 8-th International Conference on Emerging Internet, Data & Web Technologies (EIDWT-2020), Springer Lecture Notes on Data Engineering and Communications Technologies (LNDECT) 2367-4520, pp. 621-626

   January 2020

   Co-Authors:: Sabysachi Dutta and Kouichi Sakurai

2. 3-Party Adversarial Steganography
   Proceedings of the 21st World Conference on Information Security Applications (WISA-2020), Springer Lecture Notes on Computer Science (LNCS) Volume 12583, pp. 89-100

   December 2020

   Co-Authors:: Sabysachi Dutta and Kouichi Sakurai

3. Neural Networks-Based Cryptography: A Survey
   IEEE Access, Volume 9, pp. 124727–124740

   September 2021

   Co-Authors:: Sabysachi Dutta, Haowen Tan and Kouichi Sakurai

4. Key Exchange Using Tree Parity Machines: A Survey
   Proceedings of the 2nd International Conference on Artificial Intelligence: Advances and Applications (ICAIAA-2021), Springer Algorithms for Intelligent Systems (AIS) 2524-7565, pp. 363-372

   February 2022

   Co-Authors:: Kouichi Sakurai

5. Learning Multi-Party Adversarial Encryption and Its Application to Secret Sharing
   IEEE Access, Volume 10, pp. 121329–121339

   November 2022

   Co-Authors:: Sabyasachi Dutta, Sraban Kumar Mohanty, Isaac Agudo and Kouichi Sakurai

6. Tree Parity Machine based Symmetric Encryption: a Hybrid Approach
   Proceedings of the 8th International Conference on Mathematics and Computing (ICMC-2022), Springer Proceedings in Mathematics & Statistics (PROMS) 2194-1009 pp 61—73

   March 2023

   Co-Authors:: Sabysachi Dutta and Kouichi Sakurai

7. Learning asymmetric encryption using adversarial neural networks
   Engineering Applications of Artificial Intelligence, Volume 123, Part B, 2023, 106220, ISSN 0952-1976.

   May 2023

   Co-Authors:: Sabysachi Dutta, Haowen Tan and Kouichi Sakurai

Index