New Directions in Efficient Privacy-Preserving Machine Learning

Sameer Narahari Wagh
May 12th, 2020

A Dissertation Presented to the Faculty of
Princeton University in Candidacy for the Degree of
Doctor of Philosophy

Thesis Committee:

Prof. Prateek Mittal (Advisor) . . . . . Princeton University
Prof. Vincent Poor . . . . . Princeton University
Prof. Peter Ramadge . . . . . Princeton University
Prof. Niraj Jha . . . . . Princeton University
Dr. Nishanth Chandran . . . . . Microsoft Research, India

To my Mother,
for her unconditional love and unwavering belief in me.

Abstract

Applications of machine learning have become increasingly common in recent years. For instance, navigation systems like Google Maps use machine learning to better predict traffic patterns; Facebook, LinkedIn, and other social media platforms use machine learning to customize user's news feeds. Central to all these systems is user data. However, the sensitive nature of the collected data has also led to a number of privacy concerns. Privacy-preserving machine learning enables systems that can perform such computation over sensitive data while protecting its privacy.

In this dissertation, we focus on developing efficient protocols for machine learning as a target analytics application. To incorporate privacy, we use a multi-party computation-based approach. In multi-party computation, a number of non-colluding entities jointly perform computation over the data and privacy stems from no party having any information about the data being computed on. At the heart of this dissertation are three frameworks – SecureNN, Falcon, and Ponytail – each pushing the frontiers of privacy-preserving machine learning and propose novel approaches to protocol design. SecureNN and Falcon introduce, for the first time, highly efficient protocols for computation of non-linear functions (such as rectified linear unit, maxpool, batch-normalization) using purely modular arithmetic. Ponytail demonstrates the use of homomorphic encryption to significantly improve over prior art in private matrix multiplication. Each framework provides both significant asymptotic as well as concrete efficiency gains over prior work by improving computation as well as communication performance by an order of magnitude.

These building blocks – matrix multiplication, rectified linear unit, maxpool, batch-normalization – are central to machine learning and improvements to these significantly improve upon prior art in private machine learning. Furthermore, each of these systems is implemented and benchmarked to reduce the barrier of deployment. Uniquely positioned at the intersection of both theory and practice, these frameworks bridge the gap between plaintext and privacy-preserving computation while contributing new directions for research to the community.

Acknowledgments

"Alone we can do so little; together we can do so much."
- Helen Keller

Like any worthwhile undertaking, there is an entire universe of people that assist in the making of one PhD and this section is a tribute to my universe of people. I do not believe acknowledgments can be written in hindsight and true to this spirit, this part of the thesis was written over the entire course of my PhD. I see this as a reflection not of my PhD but about the people I shared it with and hence, to all of you, a big thank you for sharing parts of this journey with me!

I strongly believe that researchers are made and not born and no sentiment could better acknowledge my advisor Prateek's contribution in my life. I admire Prateek for his tremendous dedication, enthusiasm, and relentless optimism. I am grateful for his encouragement, for believing in me, and for all the creative freedom given to me during the course of my PhD. Working with Prateek has been one of the best decisions of my life and if I ever were to relive these years, I would make this decision every single time!

I would also like to acknowledge Paul Cuff for his instrumental role in shaping me as a student, scholar, and as an individual. Paul has been like a second advisor to me, a close mentor for life, as well as a wonderful basketball colleague. A special acknowledgment to Prof. Rajesh Narayanan, Prof. Nic Shannon, Prof David Dorfan, and M Prakash Sir, my previous advisors and mentors, who introduced me to the path of research, provided invaluable mentorship, and without whom I would never have chosen this path in life. I would like to thank my thesis committee – Prof. Niraj Jha and Nishanth Chandran as readers, Prof. Vince Poor and Prof. Peter Ramadge as the examiners – as well as other department faculties that I had the opportunity to interact with during my time at Princeton – Prof. Paul Prucnal and Prof. Alejandro Rodriguez. A special acknowledgment to my mentors Nishanth Chandran (doubly!), Hao Chen, Ilya Razenshteyn, Divya Gupta, Tal Rabin, Olya Ohrimenko, Benny Pinkas, and Assi Barak. They have been instrumental in my professional development and without whom I would not be where I am today. I would also like to thank all my collaborators Miran Kim, Yongsoo Song, Fabrice Benhamouda, Eyal Kushilevitz, Xi He, Ashwin Machanavajjhala, Aaron Johnson, Ryan Wails, Lawrence Esswood, Felix Schuster, Manuel Costa, Yanqi Zhou, and David Wentzlaff. Dragoş and Saeed, a very special thanks to both of you, Ponytail would not be half as fun without you. Likewise, Hans and Gerry, it has been a pleasure working with you and I wish you the very best of luck in your journey at Grad school. I would also like to thank all the awards and grants that have supported the research efforts of this dissertation.

Lisa, fortunately or unfortunately, you had an office right opposite Prateek's and had to see me often. No amount of words can thank you – from simple favors, to elaborate requests, you have helped me through it all and I cannot imagine my Princeton journey without you. No matter where I am, you will keep receiving post-cards from around the world. A big thank you to all the incredible ELE departmental

staff Colleen, Roelie, Dorothy, Heather, Stacey, Kate, Jean, Lidia, and Abby who truly form the backbone of the department. A special thank you to Lori, Katie, and Muhamed for all the support in making my graduate life easier and to Cecilia for all her support in organizing all kinds of departmental events – MelodEE, the departmental t-shirt and more. I sincerely applaud your dedication and efforts to make our department such a fun place.

I would like to express my deepest appreciation for some people who have not only shared this journey with me but also made this journey what it is. First and foremost, my deepest gratitude to Rutwik, a dear friend and long-time roommate; I truly cannot imagine my journey at Princeton without you. I have learned so much from you and the countless memories we have together shall be cherished forever. I look forward to more wonderful times together. To Shruti Tople, a dear friend, a trustworthy guide, and a fantastic collaborator. To Sergiy for being a humble link to JG and ELDOA but more importantly for just being yourself. To John-Garret a.k.a JG, for being the best body mechanic on this planet, and for showing me the way to living my life more fully. To Luciano, for all the fun times we shared together and many more to come. To Yeohee, for the fun jamming for MelodEE and for all that I have learned from you. To Tri and Mayank for all the crazy conversations and more than making up for the Ph in a PhD. A special appreciation for my Yoga instructor David as well as all my yoga friends. To my host family Rona, Rob, Kate and Matt, Sara and Bernat who graciously welcomed me not just into the United States but also into their lives. To Brett Bishop, for the enjoyable workouts together and for all your honest inputs on life. To Yan, for being a phenomenal mentor as well as a wonderful book guru. To Lanqing and Jonathan for all the fun puzzles over lunch and otherwise and to Prakhar, Vikash, Shashank, Siddharth, Shaurya, Pranav, Chris, Uno, Wen, Theresa, and Vitor for all the cooking and meals we shared. Finally, Sandra, Hamid, Mina, and Sepehr, you guys have been an instrumental part of my Princeton lives and I will cherish every moment we spent together!

I would also like to thank all my friends across the world, especially from India; you made this journey a real pleasure and you're all fondly remembered during the writing of this acknowledgment. A special and heartfelt appreciation to Harshad Chandak for everything (in particular for all the "making memories"). My sincere gratitude to all my family in India as well as in the States – without your support, this journey would not be easy. I would like to acknowledge a few of my friends, in and outside Princeton for being a part of this journey: Changchang, Peng, Yixin, Yushan, Thee, Liwei, Arjun, Hooman, Tong, Shivam, Irineo, Vera, Nadeen, Molly, Quinn, Abhishek, Shweta, and Sukrit, Sapad, Daniel and Christina, Anders, Harsh, Rahul, Shir, Eysa, and Eduardo, Mohammad Samragh, Sadegh (MSR), Ilia, Lynn, and Pardis. I would also like to thank Anita (and Zalalem) for being one of the best hosts on this planet. I thank numerous people I have played badminton, basketball, ultimate, and soccer with, in particular, my folks and coaches at CFN and the FCRangers crew for the wonderful soccer time in Seattle. I would like to thank all the organizations and grants that have supported my work. Finally, I would like to thank Princeton for giving me this wonderful environment to thrive to the best of my abilities, an entire ecosystem of talented individuals to learn from.

Contents

List of Tables

List of Figures

List of Acronyms

ML . . . . . Machine Learning
AI . . . . . Artificial Intelligence (synonymous with ML in this dissertation)
MPC . . . . . Multi-Party Computation
SMC . . . . . Secure Multi-Party Computation (used synonymously with MPC)
HE/FHE . . Homomorphic Encryption/Fully Homomorphic Encryption
MLaaS . . . . Machine Learning as a Service
IoT . . . . . Internet of Things
CEI . . . . . Child Exploitative Imagery
NN . . . . . Neural Network
CNN . . . . . Convolutional Neural Network
DNN . . . . . Deep Neural Network
DP . . . . . Differential Privacy
GC . . . . . Garbled Circuit
OT . . . . . Oblivious Transfer
FV/BFV . . Fan-Vercauteren (Brakerski/Fan-Vercauteren)
RLWE . . . . Ring Learning With Errors
SIMD . . . . Single Instruction Multiple Data
DFT . . . . . Discrete Fourier Transform
LAN . . . . . Local Area Network
WAN . . . . . Wide Area Network
ReLU . . . . . Rectified Linear Unit
PRNG . . . . Pseudo-Random Number Generator
ZK/ZKP . . . Zero-knowledge/Zero-knowledge Proof
PoPK . . . . Proof of Plaintext Knowledge
RNS . . . . . Residual Number System
NTT . . . . . Number Theoretic Transform
SGD . . . . . Stochastic Gradient Descent

Chapter 1

Introduction

"I really believe that we don't have to make a trade-off between security and privacy. I think technology gives us the ability to have both."

– John Poindexter

Machine Learning (ML), a specialization of a broader scientific field called artificial intelligence (AI), has transformed our technological landscape in the last decade. While science fiction often portrays it as a dystopian world ruled by robots, in reality, ML encompasses a wide range of systems from Google's search algorithm to self-driving cars. ML is actively used in many sectors of society – technology, healthcare, and commerce – due to its potential in transforming services, improving scalability of businesses, and impressive technological breakthroughs. However, most applications of ML require huge amounts of data in order to learn and provide "intelligent" services.

Not independent of this machine learning revolution is the advent of massive amounts of data collection – in particular about users' behavior which is frequently the target of ML services. The ubiquity of personal data in the digital world has rendered user information accessible as never before. This proliferation also poses significant privacy risks due to the lack of transparency on the use of such data as well as the lack of mechanisms to provide users with control over their data. Health records, banking details, and login credentials are just some examples of such sensitive data. The use of location and sensor data over smartphones, search queries, and even social media posts in targeted services have led to a new wave of privacy awareness. Finally, the rise of IoT devices brings about its own set of privacy concerns. The possibility of voice assistants such as Alexa, Google Home, or Siri constantly listening in has further expanded the surface area of privacy attacks.

Our society sits at the nexus of developments in technology that enables us to transition into a better digital ecosystem. We face a range of legal and ethical dilemmas in the quest for a balance between societal advances and fundamental privacy rights. Restricting the collection of such private data can limit the societal benefits we can obtain from analyzing this data. It can also hinder major technological advances – the recent revolution in ML would arguably not have happened as it thrives on data.

Systems, such as those developed in the course of this dissertation, revolve around the paradigm of *private computation* – a method of computation where the data on which the computation is being performed can be kept private or confidential. These technological solutions provide a solution to the conundrum of enabling new applications that rely on sensitive data while at the same time ensuring appropriate levels of privacy for the data. While there exists a number of paradigms to achieve the goals of confidential computing, the naive use of these existing paradigms suffers from significant overheads in computation and communication. Thus, to truly transition into a privacy-conscious world, reducing these overheads is imperative.

1.1 A Vision for a Privacy-Conscious World

I envision a world where our digital infrastructure is re-designed with a privacy-first approach. The current digital norms implicitly require users to surrender their privacy in exchange for services. Such a re-design would allow users to subscribe to complex machine learning services such as location-based services and health trackers without the complete surrender of their sensitive data; allow organizations such as hospitals and banks to run collaborative services such as rare-disease research and anti-money laundering without disclosing their proprietary data; and would also enable new applications such as outsourcing of sensitive data for storage such as cryptographic keys in a manner that preserves privacy.

Achieving these goals requires advances on many fronts: more efficient cryptographic protocols, better system design, "privacy-friendly" hardware, as well as better policy design and adoption. Other than the tools to work over sensitive data, such an effort would more broadly require regulations enforcing such privacy protection and transparent usage of data. It is important to remember that a private computation is typically more expensive in terms of computation or the run-time, as compared to a plaintext (or non-private) computation. For instance, a state-of-the-art work [93] using homomorphic encryption (one technique for private computation) is several orders of magnitude slower as well as computationally resource-intensive compared to the same computation without privacy. Similarly, a state-of-the-art work [111] using multi-party computation (another technique for private computation) is more promising but still about 2 orders of magnitude slower than plaintext computation (non-private computation). For this reason, efforts aimed at designing efficient protocols for fundamental privacy-preserving building blocks are critical to the foundation of such an ecosystem.

To this end, this dissertation focuses on designing novel, efficient protocols to enable privacy-preserving computations. For instance, Chapters 3, 4 contain the state-of-the-art protocols for performing a comparison of two numbers privately. Similarly, in Chapter 5, we will see the most efficient protocol for performing matrix multiplication privately. These building blocks – matrix multiplication, comparison – among others, are central to machine learning and more generally to our entire digital ecosystem and would be critical to a privacy-conscious vision of the world. Any improvements to these protocols are further amplified in practice due to the

repetitive dependence of higher-level functionality on these basic primitives. This dissertation proposes novel ways of performing these computations and eliminating expensive cryptographic primitives, ideas which have transformative capabilities for other lines of work in privacy-preserving technologies. In this way, this dissertation significantly reduces the performance gap between privacy-preserving and plaintext computations.

1.2 Privacy Advocate's Toolkit

In designing privacy-focused systems, it is imperative to have a clear understanding of "What assumptions underlie the privacy of the system?" as well as "What are the privacy guarantees provided by the system?". The assumptions are what allow one to rigorously argue for the privacy of the system, and the guarantees are what provide semantic meaning to the notion of privacy.

In this frame of reference, there are a number of mathematical techniques that allow the design of privacy systems – multi-party computation (MPC) [115, 116, 54, 12], homomorphic encryption (HE) [52, 47, 17], differential privacy (DP) [43, 42, 67, 51], and trusted hardware based solutions [100, 2]. Each technique has its own set of assumptions and privacy guarantees from which the security/privacy of the end-to-end system is argued. Each technique also has its own overhead for achieving privacy. Research progress on reducing this overhead has significantly improved the performance of these techniques since their inception.

For instance, HE (described further in Section 2.3) requires standard cryptographic assumptions but suffers from high computation overhead. MPC (described further in Section 1.3 and Section 2.2) also relies on standard cryptographic assumptions and in addition requires non-collusion assumptions. These techniques provide standard cryptographic notions of privacy – provide "encryption-like" privacy assuming the adversary is computationally bound. Differential privacy on the other hand provides its own privacy guarantee – the presence or absence of any individuals' data in a database has limited effect on the queries performed on the database, providing an intuitive notion of privacy. Similarly, trusted hardware based techniques require, as the name suggests, a trust assumption on the hardware. Each technique has its pros and cons and in this dissertation, we focus on solutions based on MPC and HE.

1.3 A Brief History of Multi-Party Computation

MPC, also known as Secure Multi-Party Computation (SMC), is a sub-discipline of cryptography that allows a set of parties to compute a function of their inputs while keeping these inputs private. The classic example of this is called the Yao's millionaire problem introduced in a seminal work by Andrew Yao in 1982 [116] – "Two millionaires wish to know who is richer; however, they do not want to find out inadvertently any additional information about each other's wealth." Assuming the two millionaires do not have a common trusted friend, i.e., a *trusted third party*, we fall in the realm

of MPC. The two millionaires can run a protocol among themselves, at the end of which both of them will learn a single bit of information: which millionaire is richer. More generally, a MPC set-up assumes that there are n parties $P_1, P_2, \cdots, P_n$ each having an input $x_1, x_2, \cdots, x_n$. The parties would like to jointly compute a pre-determined function $f(x_1, x_2, \cdots, x_n)$ of their inputs in a manner that preserves the privacy of each input. In the above example of the millionaire's problem, $n = 2$, the private inputs x_1, x_2 are the salaries of the two millionaires and the function $f(x_1, x_2) = x_1 \geq x_2$ is simply a comparison function.

Soon after laying the foundations of MPC in his seminal paper in 1982, Andrew Yao introduced the first set of protocols for MPC called *Garbled Circuits* [115, 116]. However, for the next couple of decades, secure computation remained of theoretical interest with few efforts focused on practical deployments. This changed around the turn of the millennium when the algorithmic improvements coupled with the rise of computing infrastructure and improved communication led to efforts at building practical systems for general-purpose MPC. Fairplay [79] was the first demonstration of a generic secure function evaluation system. It compiled a desired two-party privacy-preserving program from a high-level language to an executable that can be run by the two parties as a MPC protocol. However, there were systemic limitations to the performance and scalability of such a system that restricted its use to toy programs. For example, a single instance of the classic millionaire problem can be solved[1] in about $1.25s$ (this is about 7 orders of slower than non-private comparisons). However, since then, advances in protocol design, computation and communication infrastructure, and hardware support have improved the speed of MPC protocols by over 6 orders of magnitude. For example, Falcon, a state-of-the-art protocol described in Chapter 4 takes about $1.78\mu s$ to perform the same single comparison. This tremendous progress has enabled MPC to scale to much larger and interesting applications.

This dissertation is focused on enabling private computation of neural networks (NNs) as the target application. This focus allows one to design specialized protocols that are more efficient than generic MPC protocols which can be used to compute arbitrary functions. In the next section, we will see some concrete use cases of MPC techniques applied to the domain of ML.

1.4 Target Applications of MPC

NNs have proven to be a very effective tool for producing predictive models that are widely used in applications such as healthcare, image classification, finance, and so on. Private computation can assist both private training as well as private inference in ML. It is well known that the accuracy of ML models gets better as the amount of training data increases [118]. Large amounts of training data can be obtained by pooling in data from multiple contributors, but this data is sensitive and cannot be revealed in the clear due to proprietary reasons or compliance requirements [23, 92]. To enable training of NN models with good accuracy, it is highly desirable to securely

[1] The quoted numbers are for a billionaires' problem, where the compared values are 32-bit integers in the local area network (LAN) setting.

train over data from multiple contributors such that plaintext data is kept hidden from the training entities. MPC provides an elegant and rigorous solution to the above class of problems.

At the same time, in scenarios where a service provider has a pre-trained model on sensitive data and would like to deploy machine learning as a service (MLaaS), MPC can also provide a promising solution. The service provider and the client can run a 2-party computation where the client can learning the classification/prediction on its input without ever revealing the input to the service provider. Simultaneously, the service provider does not have to reveal its model, which could be proprietary and can contain valuable trade secrets. Below, we will describe two concrete applications that can be enabled by techniques developed in this thesis:

Predictive Healthcare Application. Consider a pool of M users (data owners) who wish to subscribe to a service that monitors their sleep data. Training a ML model on each user locally is difficult due to the limited diversity of the data available on each user's device. On the other hand, such data can be highly sensitive and reveal the timings, the quality of sleep and other information considered private by the user. This hinders collection of such data and in turn disables ML approaches which crucially rely on data. Using MPC, we demonstrate an approach where the users can execute training over their joint data using N servers in a manner that preserves privacy. First, these M users send "secret shares"[2] of their data to the N servers. The servers collectively run an interactive protocol (developed in this thesis) to train an NN over the joint data to produce a trained model that can be used for inference. The privacy requirement is that no individual party or server learns any information about any other party's training data. We call this the *N-server model*. The trained model can be kept hidden from any single server/party and retained as secret shares between the servers (or reconstructed to obtain the model in the clear). Furthermore, even if the model is retained as secret shares between the N servers, the inference/prediction can still be executed using the trained model on any new test input – keeping the model, the new test input, and the predicted output private from the other parties as well as the servers.

Another target application is the following: A group of M hospitals, each having sensitive patient data (such as heart rate readings, blood group, sugar levels, etc.) can use the above architecture to train a model on their joint data to run MLaaS and help predict some disease or irregular health behavior. The system can be set up such that the patient's sensitive input and predicted output are only revealed to the patient, and remains hidden from everyone else.

Detecting Online Child Abuse. In recent years, the distribution of child exploitative imagery (CEI) has proliferated with the rise of social media platforms – from half a million reported in between 1998-2008 to around 12 million reports in 2017 to about 45 million in 2018 [18, 31]. US law prohibits the production, possession,

[2]Secret shares are parts of the secret that individually do not reveal anything about the secret. For more details refer to Section 2.2.2.

receipt, mailing, sale, distribution, shipment, or transportation of any such material. Furthermore, there has been tremendous effort in blocking access, distribution, and generation of such content [106, 7, 80, 18]. Despite such massive efforts to curb online child abuse, the problem remains largely unsolved. Given the severity of the problem and stringent laws around the handling of such incidents (18 U.S. Code §2251, 2252), it is important to develop solutions that comply with these stringent (privacy) regulations while enabling efficient detection and handling of such data. Given the success of ML in image classification tasks, it is highly desirable to reap the success of this technology to better tackle the problem of CEI. However, the inability to generate a database of the original images (due to legal regulations) leads to a problem of lack of training data in ML. The work in this thesis provides a cryptographically secure framework for this conundrum, where data is split into unrecognizable parts among a number of non-colluding entities and privacy-preserving analytics can then be run over such split data. In this way, MPC provides a two-fold solution, it enables accumulation of good quality training data and at the same time can enable MLaaS for automated detection of CEIs. In this manner, MPC can enable an end-to-end solution for the problem of CEIs in social media with strong privacy guarantees on the underlying data.

1.5 Hybrid Protocols: The Way Forward

Despite considerable efforts from the research community, there is a lack of low-overhead protocols for MPC. How do we design more efficient MPC protocols? How do we design approaches that reduce the gap between plaintext and private computation? This dissertation proposes the use of *hybrid approaches* – co-designing solutions using a spectrum of techniques from the privacy toolkit – as the way forward and supports this proposition through 3 systems: SecureNN, Falcon, and Ponytail. Each of these systems operates in a strictly stronger adversarial model – SecureNN in a semi-honest model, Falcon in a malicious security with honest majority model, and Ponytail in a malicious security with dishonest majority model.

Proposing novel and more efficient protocol constructions, this dissertation challenges a number of unspoken rules in privacy-preserving machine learning. SecureNN pushes the frontiers of privacy-preserving ML by demonstrating faster primitives for basic building blocks of ML algorithms such as non-linear functions: Rectified Linear Unit (ReLU) and Maxpool. We achieve this using simple modular arithmetic and avoid the use of garbled circuit (GC) and oblivious transfer (OT). These techniques (GC or OT) require the use of inter-conversion protocols, i.e., protocols that switch between different types of secret sharing schemes as they rely on Boolean shares. Thus, in demonstrating faster primitives, SecureNN challenges the status quo in that inter-conversion protocols are the most efficient approach for non-linear operations in private ML. Similarly, there is an implicit understanding that MPC protocols are communication-bound, i.e., the communication overhead of protocols are the bottleneck for practical deployment. Falcon and Ponytail demonstrate for the first time that MPC protocols can be compute-bound. These protocols fur-

ther introduce improvements to ML building blocks by operating on smaller ring sizes, proposing new constructions for comparisons (ReLU), and using HE to improve matrix multiplication by orders of magnitude. These have tremendous implications – the gap in performance between private ML vs. plaintext ML can be reduced to acceptable values for practical deployment.

Through a brief summary of the three main chapters of this dissertation, we will look at how hybrid approaches – co-designing solutions using a spectrum of techniques – can bridge this performance gap between plaintext and private computations, thus enabling a plethora of privacy-preserving applications.

(A) SecureNN introduces a cross-layer approach to designing protocols where we integrate the private data layer (i.e., the secret sharing layer) with the private computation layer. This enables us to develop protocols that are much more efficient than state-of-the-art protocols which do not benefit from this coupling of the two layers of abstraction. Furthermore, in a 3-party computation model, these specialized protocols are tailored to vital components such as ReLU and Maxpool in NN training and inference. Secondly, the use of simple modular arithmetic avoids expensive conventional techniques such as garbled circuits and oblivious transfers. These together demonstrate the practicality of such an approach while introducing a new line of cryptographic protocols for MPC.

(B) Falcon demonstrates a hybrid integration of techniques from SecureNN [109] and ABY^3 [84] along with newer protocol constructions for privacy-preserving deep learning. Specifically, Falcon enables efficient protocols for functions such as ReLU and Batch-normalization which are critical to the ML infrastructure. Furthermore, these protocols are developed in a maliciously secure model with honest majority using the redundancy of a replicated secret sharing (2-out-of-3 secret sharing) scheme. Note that protocols in SecureNN are semi-honest secure protocols and do not provide correctness under malicious corruptions. This hybrid integration thus enables the protocols to maintain efficient performance while also operating under stronger security models. Through techniques such as symmetrization and optimized subroutines, the protocols in Falcon further push the frontiers of efficiency in private machine learning.

(C) In Ponytail, a hybrid combination of MPC and HE has been explored in the dishonest majority literature for the first time. Specifically, Ponytail demonstrates the use of HE to improve the performance of privacy-preserving matrix multiplication – both concretely and asymptotically. Matrix multiplication is once again a critical component of ML algorithms today and improvements to this have significant implications for privacy-preserving ML. More generally, this work shows that bilinear operations are most efficiently computed in a privacy-preserving manner using a hybrid combination of MPC and HE. This work also incorporates advances from HE literature to improve the performance of protocols in the MPC literature.

1.6 Summary of Contributions

This thesis altogether aims at pushing the efficiency frontiers of MPC-based private deep learning. It proposes new protocols that enable deployment of privacy technologies at a lower overhead, thus inching these technologies closer to practice. This thesis is primarily based on the following works completed during the course of my PhD:

[109] Sameer Wagh, Divya Gupta, and Nishanth Chandran. "SecureNN: 3-Party secure computation for neural network training." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2019

[111] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. "FALCON: Honest-majority maliciously secure framework for private deep learning." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2021

[29] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragoş Rotaru, Yongsoo Song, and Sameer Wagh. "Maliciously secure matrix multiplication with applications to private deep learning." In: *Advances in Cryptology—ASIACRYPT*. *Author order alphabetical. 2020

Each of these systems is implemented and demonstrates both the asymptotic (in algorithmic complexity) as well as concrete efficiency (practical improvements including the constants involved in algorithmic complexity) improvements. These works strictly improve upon each other in the adversarial setting and contribute uniquely to the vision of making private machine learning practical. Specifically, this thesis makes the following contributions:

(A) SecureNN: A novel cross-layer protocol design paradigm for efficient non-linear operations in MPC. The approach avoids the use of expensive cryptographic techniques such as OTs and GCs in favor of simple modular arithmetic. This improves upon the prior art in private ML by about an order of magnitude. The simplicity of this approach has already led to its early adoption in the industry [4, 97, 89].

(B) Falcon: This work improves upon SecureNN to further improve the performance of non-linear operations. Protocols in Falcon are secure against malicious adversaries (honest majority) and it is the first secure framework to support high capacity networks with over a hundred million parameters such as VGG16 as well as the first to support batch normalization, a critical component of deep learning that enables training of complex network architectures such as AlexNet. This improves upon prior art by upto two orders of magnitude.

(C) Ponytail: This work is the first demonstration of an $O(n^2)$ communication overhead matrix-multiplication of two $n \times n$ matrices in dishonest majority MPC setting; the prior best known algorithm used an $O(n^3)$ communication (or $O(n^{2.8})$ with Strassen's algorithm [66]). The work also makes a compelling

case for the use of a hybrid approach – combining HE with MPC – for efficient privacy-preserving computation.

In the course of improving the performance of private computation techniques, this dissertation also challenges research dogmas such as the use of inter-conversion protocols or the communication-bound nature of MPC protocols (detailed description in Section 1.5) and provides new directions for efficient protocol design. I have also worked on a number of other projects in the space of privacy-enhancing technologies, which are listed here but which will be omitted from the rest of this dissertation for coherency:

[103] David Marco Sommer, Liwei Song, Sameer Wagh, and Prateek Mittal. *Towards probabilistic verification of machine unlearning.* https://arxiv.org/pdf/2003.04247.pdf. 2020

[110] Sameer Wagh, Xi He, Ashwin Machanavajjhala, and Prateek Mittal. "DP-Cryptography:Marrying differential privacy and cryptography in emerging applications." In: *Communications of the ACM*. 2020

[113] Gerry Wan, Aaron Johnson, Ryan Wails, Sameer Wagh, and Prateek Mittal. "Guard placement attacks on path selection algorithms for Tor." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2019

[58] Hans Hanley, Yixin Sun, Sameer Wagh, and Prateek Mittal. "DPSelect: A differential privacy based guard relay selection algorithm for Tor." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2019

[108] Sameer Wagh, Paul Cuff, and Prateek Mittal. "Differentially private oblivious RAM." in: *Privacy Enhancing Technologies Symposium (PETS)*. 2018

[33] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. "The Pyramid scheme: Oblivious RAM for trusted processors." In: *Tech Report*. https://arxiv.org/abs/1712.07882. 2017

[117] Yanqi Zhou, Sameer Wagh, Prateek Mittal, and David Wentzlaff. "Camouflage: Memory traffic shaping to mitigate timing attacks." In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017

Chapter 2

Background and Related Work

In this section, we introduce some of the basic concepts required for a deeper understanding of the contributions of this dissertation. Two-party computation and three-party computation, two special cases of MPC we will consider in this dissertation, will be denoted by 2PC and 3PC respectively. We will introduce notation along with the basic primitives used throughout this thesis.

2.1 Basic Concepts

We now introduce the concepts of statistical distance between two distributions, a concept central to proofs in MPC. Rigorous quantification of the indistinguishability of the transcripts (cf. Section 2.2.3) is done using this metric. We also introduce some basic complexity theory, and the notions of cryptographic security vs. information-theoretic security.

2.1.1 Statistical Distance

Let X, Y be two random variables taking values in some finite set A. The statistical distance between X and Y is

$$\Delta(X, Y) = \max_{S \subset A} |\Pr[X \in S] - \Pr[Y \in S]| \tag{2.1}$$

We say that X and Y are (statistically) ϵ-close if $\Delta(X, Y) \leq \epsilon$. This distance is also known as total variation distance between X and Y. This metric has a few important corollaries:

Theorem 2.1. *The statistical distance is equal to half the ℓ_1 distance, i.e.,*

$$\Delta(X, Y) = \frac{1}{2} \sum_{a \in A} |\Pr[X = a] - \Pr[Y = a]|$$

Theorem 2.2. *Suppose X and Y are (statistically) ϵ-close. Alice and Bob play the following game: Bob flips a fair coin and then sends Alice a sample from either X or*

Y depending on the result of the coin flip (say, X if it lands 'heads' and Y otherwise). Alice is asked to guess from what random variable the sample was taken. Show that Alice cannot guess correctly the coin toss with probability greater than $1/2 + \epsilon/2$. On the contrary, if $\Delta(X, Y) \geq \epsilon$ then there exists a strategy for Alice that will enable her to correctly guess the coin toss with probability at least $1/2 + \epsilon/2$.

Theorem 2.1 shows the connection between the statistical distance and the ℓ_1 distance. Theorem 2.2 on the other hand goes to the heart of the indistinguishability game formulation of cryptography. For instance, "how well can an adversary distinguish between the encryptions of two messages?" is formulated very similarly to the set-up in corollary 2.2.

2.1.2 Complexity Theory

We first define the concept of a negligible function, that will allow us to easily formalize the concepts of cryptographic vs. information-theoretic security.

Definition 1 (Negligible Function)**.** *We say that a function $f : \mathbb{N} \to \mathbb{R}$ is a negligible function, i.e., $f \in \mathsf{negl}(n)$ if for every polynomial $p : \mathbb{N} \to \mathbb{R}$ there exists an integer n_p such that for all $n > n_p$*

$$|f(n)| < \frac{1}{p(n)} \tag{2.2}$$

In other words, a negligible function is any function that asymptotically approaches zero faster than any polynomial function.

2.1.3 Privacy – Cryptographic vs. Information-Theoretic

While there are different notions of quantifying privacy, in this dissertation, we look at systems that provide two notions of privacy – cryptographic privacy or information-theoretic privacy – the latter being a special case of statistical privacy. Here, we use the notions of security and privacy synonymously.

Definition 2 (Statistical Indistinguishability)**.** *Let $\{X\}_n$ and $\{Y\}_n$ be two sets of distributions. We say that these two distributions are statistically indistinguishable if $\Delta(X, Y) \in \mathsf{negl}(n)$.*

To define computational indistinguishability, we consider a probabilistic polynomial time adversary (PPT) $\mathcal{A}$. This adversary tries to distinguish between the two distributions, i.e., samples a polynomial number of data points from either distribution and tries to guess the distribution from which the samples were obtained. Inability to distinguish these with high probability is quantified using computational indistinguishability.

Definition 3 (Computational Indistinguishability)**.** *Let $\{X\}_n$ and $\{Y\}_n$ be two sets of distributions. Let $\mathcal{A}$ denote any probabilistic polynomial-time algorithm that takes in oracle access (can ask for samples) to a distribution and produces an output in $\{0, 1\}$. We say that these two distributions are computationally indistinguishable if $\Delta(\mathcal{A}(X), \mathcal{A}(Y)) \in \mathsf{negl}(n)$ for probabilistic polynomial-time algorithms $\mathcal{A}$.*

Systems that provide computational indistinguishability, i.e., systems where computational assumptions are required to argue the security, are said to provide cryptographic security. If a system provides perfect statistical indistinguishability, i.e., $\Delta = 0$, then the system provides information-theoretic security (akin to Shannon's definition of perfect secrecy [99] achieved using a one-time pad). Finally, if the system provides statistical indistinguishability which is not perfect, then we say that the system provides statistical security (with the appropriate security parameter).

2.1.4 Groups, Rings, and Fields

Groups, *Rings*, and *Fields* are sets of elements with some basic structure. Informally, groups are a set of abstract objects that are closed under a binary operation. Rings are groups under the binary operation addition and also satisfy some properties under the binary operation multiplication. Finally, Fields are groups under both addition and multiplication. More formally,

Definition 4 (Groups)**.** *A Group is a set G which is closed under a binary operator $*$, i.e., for $x \in G, y \in G, x * y \in G$. Furthermore, the set has the following three properties:*

- *Existence of Identity: There exists an $e \in G$ such that $x * e = e * x = x$ for all $x \in G$*
- *Existence of Inverse: For every $x \in G$, there exists $x' \in G$ such that $x * x' = x' * x = e$*
- *Associativity: For all $x, y, z \in G$,*

$$x * (y * z) = (x * y) * z$$

A group is said to be "abelian" if for all $x, y \in G, x * y = y * x$. A Ring is more structured than a group and a Field is even more structured than a Ring. In fact, every Field is a Ring and every Ring is a Group.

Definition 5 (Rings)**.** *A Ring is a set R which is closed under two binary operators $(+, \times)$ with the following three properties:*

- *R is an abelian group under $+$*
- *Associativity of $\times$: For all $x, y, z \in G$,*

$$x \times (y \times z) = (x \times y) \times z$$

- *Distributive Property: For all $x, y, z \in G$,*

$$x \times (y + z) = (x \times y) + (x \times z)$$
$$(y + z) \times x = (y \times x) + (z \times x)$$

Definition 6 (Fields)**.** *A Field is a set F which is closed under two binary operators $(+, \times)$ with the following properties:*

- *F is an abelian group under $+$*
- *$F - \{0\}$ (where 0 is the additive identity of F) is an abelian group under $\times$*

The set of numbers modulo n for a given natural number n, denoted by $\mathbb{Z}/n\mathbb{Z}$, and the set of all polynomials with integer coefficients, denoted by $\mathbb{Z}[x]$, are some examples of Rings. The set $\mathbb{Z}/p\mathbb{Z}$ where p is a prime is an example of a Field. Fields are more fundamental structures to cryptography due to their additional properties such as the existence of multiplicative inverse. However, for the remainder of the thesis, a basic understanding of these concepts should be sufficient.

2.2 Multi-Party Computation

MPC enables a set of parties, each with some private input, to compute a function of these inputs without revealing anything about the inputs other than what is already revealed by the output. Thus it is important to note that MPC does not deal with the question "How much information is revealed by the function output?". This is implicit in the fact that the parties decide to participate in the computation.

In order to compute the functionality, the parties run a *protocol* amongst themselves, at the end of which, based on the protocol description, one or more of the parties can end up with the function computation. The protocols usually run in rounds, a computation phase where each party performs as much computation as possible, followed by a communication phase where each party communicates as much data as possible with other parties before proceeding to the next round. The adversarial model describes the various choices under which a given protocol is shown to be secure. Informally, MPC protocols need to have the following two properties:

- No additional information about the inputs of any party can be inferred from the messages sent during the protocol execution (this is known as input privacy). In other words, the only information revealed is that which is revealed by the output of the computation.

- No proper subset of the parties (in some cases a threshold of parties), colluding with each other (either sharing data between themselves or deviating from the protocol), should be able to force an incorrect output result (this is known as correctness).

Section 2.2.1, described below, discusses the various types of adversaries (which correspondingly modify the above two properties).

2.2.1 Adversarial Models

The security properties of an MPC protocol are quantified by formulating the adversary. Such an adversary is defined by a number of parameters – the adversarial power, the network model, the threshold of corruptions just to name a few. We will introduce the different terminologies required to succinctly define an MPC adversary below.

Synchronous vs. Asynchronous Network. In a synchronous adversarial model, we assume that the parties are connected by a synchronous network with access to a global clock. In such a system, we assume a strict upper bound on message delivery times and hence protocols can proceed in rounds with strong delivery guarantees. On the other hand, in an asynchronous adversarial model, messages can be delivered in an arbitrary order and have arbitrary delays in between. In particular, a standard assumption is that of assuming a rushing adversary, who may not delay or block messages from honest parties but the adversary sees all the messages sent by honest parties to corrupted parties at any given round before sending its own messages for that round.

Semi-honest vs. Malicious Corruptions. A semi-honest adversary or honest-but-curious adversary is one that follows the protocol specification but may try to infer as much as possible while abiding by the protocol description. In other words, if the protocol description states that a party generate a random number, a semi-honest adversary corrupting that party will truly have to generate a random number. In contrast, a malicious or byzantine adversary may arbitrarily deviate from the protocol specification. Semi-honest, malicious corruptions are also known in the literature alternatively as passive, active corruptions respectively. Note that a relaxed version of malicious corruption is the notion of malicious security with abort (security with abort), which ensures that either the protocol succeeds and each party receives its outputs or the protocol aborts, in which case all the honest parties learn that the protocol aborted.

Corruption Threshold. Another adversarial parameter in MPC protocols is the number of adversarial corruptions (parameter t in a t-out-of-n secret sharing). $t < n/2$ is an important threshold value and is called honest majority due to the presence of a majority of honest parties. The strongest model of corruption $t < n$ (in particular $t = n-1$) is referred to as the dishonest majority, as the name indicates it has a majority of parties that behave dishonestly. Finally, there are other important thresholds such as $t < n/3$ for which we know that there exist information-theoretic secure protocols. In general, there are a number of other results in the literature regarding the possibility and impossibility of certain combinations of adversarial models.

Static vs. Adaptive Corruptions. Adaptive security refers to a scenario when the adversary can corrupt parties during the execution of the protocol, in particular, after seeing some of the communication exchanges (transcripts). In contrast, a static adversary must choose the parties to corrupt before the protocol begins.

Computational Power. If the adversary is computationally bounded, i.e., if the adversary is assumed to run in probabilistic polynomial-time, then we call it a computational adversarial setting. The security of protocols in such a setting typically relies on the assumed hardness of some problem (like factoring a large number into prime factors). On the other hand, if the adversary is computationally unbounded,

then we are in an information-theoretic setting. There are two levels of security here – perfect security and statistical security (briefly described in Section 2.1.3). In the context of Section 2.2.3, these two cases correspond to the result of a real execution of the protocol with a real adversary to be exactly the *same* and *statistically close* to the result of an ideal execution with a trusted party and an ideal-world adversary (also known as the simulator).

There are a number of other properties formalized in the literature such as fairness where all parties receive their outputs if any one party receives its output and guaranteed output delivery (G.O.D) where the honest parties are guaranteed to successfully execute the computation. There is substantial literature regarding various impossibility results in achieving fairness, security with abort, G.O.D for various threshold regimes and adversarial computational powers [54, 12, 90]. Security requirements can be split into *privacy* of the input data and *correctness* of the computation, a notion formalized by Araki *et al.* [6].

2.2.2 Secret Sharing

Secret sharing, as the name suggests is a technique used to distribute a secret among a number of participants (each party has a share of the secret). This distribution is such that (1) each individual share does not reveal any information about the secret (secure secret sharing scheme), and (2) the secret can be reconstructed only when a sufficient number of parties combine their shares together (threshold scheme). Secret sharing was invented independently by Adi Shamir [98] and George Blakely [15] in 1979 with different approaches but same properties.

Threshold secret sharing schemes are a generalization of the basic concept of secret sharing. Threshold schemes have a defined number of parties that are necessary to reconstruct the original secret. They are usually denoted as t-out-of-n secret sharing or (t, n)-secret sharing scheme, where $1 < t \leq n$ is the threshold and n is the total number of parties. Throughout this dissertation, we will focus on secret sharing over various Rings and Fields.

2PC examples. The most basic example of secret sharing is in a 2PC set-up where the only non-trivial sharing scheme is a 2-out-of-2 secret sharing scheme. Under such a scheme, given a ring say $\mathbb{Z}_L$, and a secret s, shares can be defined as s_1, s_2 such that $s_1 + s_2 \equiv s \pmod{L}$.

3PC examples. The second interesting example for secret sharing is in a 3PC set-up. Here we can have two possible schemes, 3-out-of-3, where all 3 shares are necessary to reconstruct the underlying secret and 2-out-of-3, where 2 shares can potentially reconstruct the underlying secret. Simple examples of such schemes are as follows: shares s_1, s_2, s_3 such that $s_1 + s_2 + s_3 \equiv s \pmod{L}$ is an example of a 3-out-of-3 sharing scheme. On the other hand, distributing shares $(s_1, s_2), (s_2, s_3)$, and (s_3, s_1) among the three parties forms a 2-out-of-3 secret sharing scheme. When the secret sharing scheme is understood from the context, we will use $[s]_L$ to denote

the sharing of a secret s in $\mathbb{Z}_L$, i.e., s_1, s_2, s_3 with the three parties respectively in a 3-out-of-3 secret sharing scheme.

n-Party setting. Another interesting example of a secret sharing scheme is an n-out-of-n secret sharing. In this setting, commonly known as dishonest majority, each party assumes that potentially all the other parties could be colluding and proceed to securely compute functions in such a setting. In such a scheme, a secret s is shared as $s_1, s_2, \ldots s_n$ with each party respectively, such that

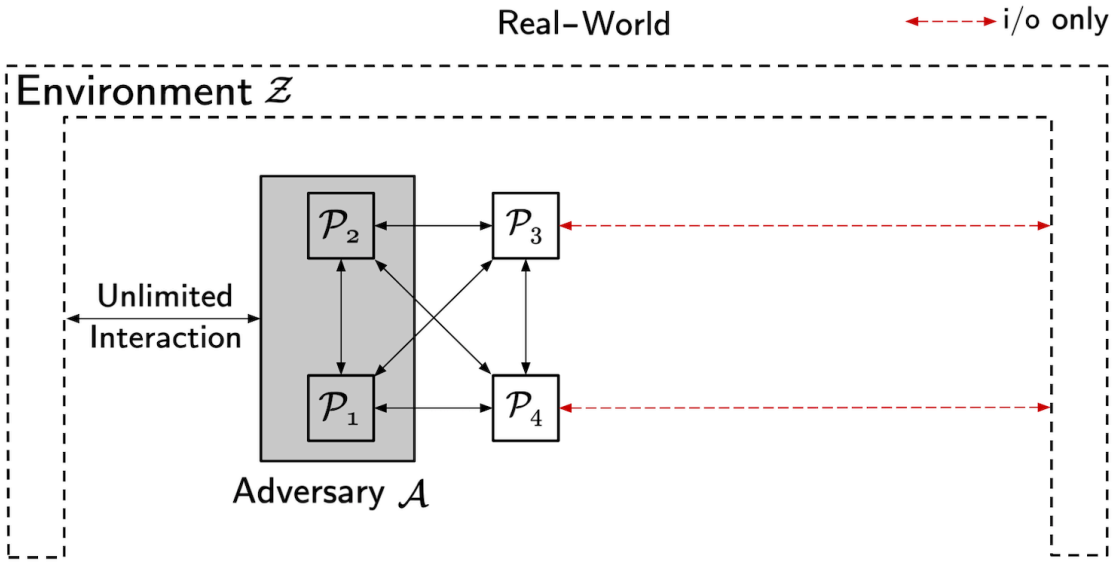
$$s \equiv s_1 + s_2 + \ldots + s_n \pmod{L}$$

for some value of the modulus L.

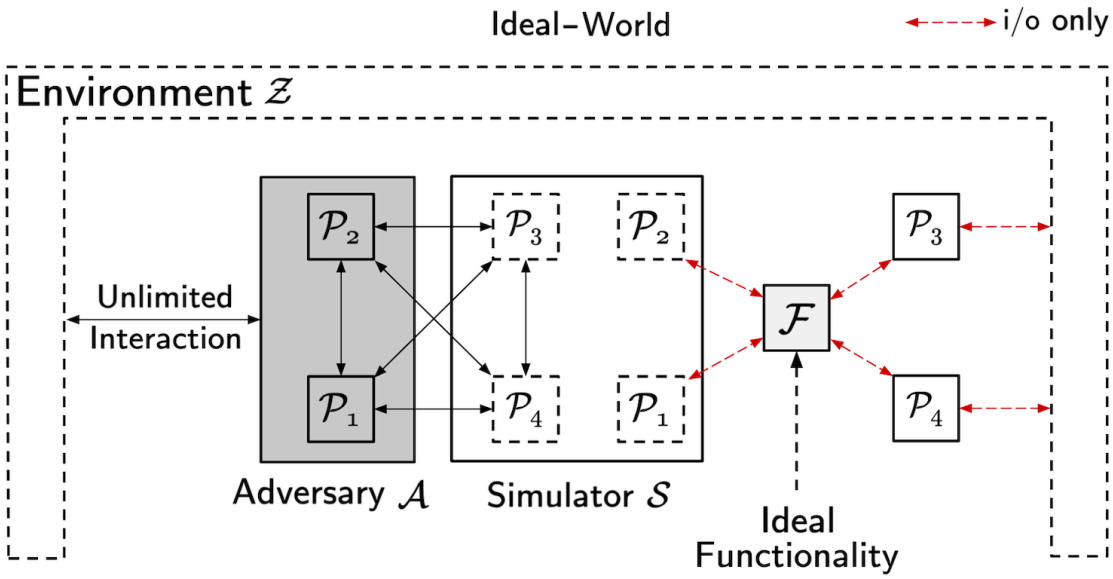
2.2.3 Simulation Based Proofs and UC Security

The security of MPC protocols are usually proven using a real-world/ideal-world simulation paradigm [21, 20]. In this paradigm, there is an environment $\mathcal{E}$, the purpose of which is to model "everything else" besides the protocol execution. This everything else could potentially contain other executions of the same protocol and it is imperative for a strong security formulation to account for that. The spirit of writing a simulation based proofs is as follows: given an adversary $\mathcal{A}$ who can extract any additional information from the protocol execution than the output, we can construct another adversary, which will be called a simulator $\mathcal{S}$, operating in a slightly "different scenario," that can extract the same information. However, by construction, the simulator $\mathcal{S}$ in the "different scenario" will be secure by definition as it will not have access to any private data. With this brief introduction, we will look at the two paradigms in more detail.

Let us consider a simple example of a 4PC set up with 2 adversarial corruptions. Let us assume x_i for $i \in \{1, 2, 3, 4\}$ is the private input of each party and that the function computation is $f(x_1, x_2, x_3, x_4)$. Furthermore, let π be the proposed MPC protocol. In order to prove the security of π, we first formulate an ideal functionality, in this case would simply be $\mathcal{F}$ specified by 4 inputs x_i for $i \in \{1, 2, 3, 4\}$ and 4 outputs all equal to $f(x_1, x_2, x_3, x_4)$. The ideal functionality in general will be more nuanced and has to be carefully defined as it contains the essence of the security guarantees. In this backdrop, the real interaction is defined as the scenario where the parties execute the protocol π in the presence of an adversary $\mathcal{A}$ and the environment $\mathcal{Z}$. This is shown in Figure 2.1a. On the other hand, in the ideal interaction, the parties send their inputs to a trusted external party which truthfully executes the above defined ideal functionality $|F$. This is shown in Figure 2.1b. To prove the security of π, for every adversary $\mathcal{A}$ in the real interaction, there exists an adversary in the ideal interaction called a simulator $\mathcal{S}$ such that no environment $\mathcal{E}$ can distinguish between these two scenarios. In other words, whatever information the adversary extracts in the real interaction, the simulator can extract it in the ideal world as well. This is shown in Figure 2.1, where we demonstrate the non-existence of any environment that can distinguish between the above two scenarios.



(a) Real-world interaction



(b) Ideal-world interaction

Figure 2.1: Examples of real-world and ideal-world interactions in a 4-party MPC protocol with 2 adversarial corruptions (P_1 and P_2 corrupt, P_3 and P_4 honest).

In general, for more complex protocols, we set up hybrid interactions where the sub-protocols used in a given protocol are replaced by their corresponding ideal functionalities and then prove that the interactions can be simulated. This hybrid argument in effect sets up a series of interactions $I_0, I_1, \ldots, I_k$ for some k where I_0 corresponds to the real interaction and I_k corresponds to the ideal interaction. Each neighboring interaction, i.e., I_i, I_{i+1} for $i \in \{0, \ldots, k-1\}$, is then shown to be indistinguishable from each other, in effect showing that the real and ideal interactions are indistinguishable.

The usual technique to construct a simulator is to have the simulator run a simulated version of the protocol internally, i.e., emulating the roles of the honest parties and interacting with the adversary. This is what we call an *internal run*. This internal run can then be used to extract the inputs of the adversarial party (which can then be forwarded to the functionality in the ideal interaction). Note that in the hybrid argument, since the subroutines used in the protocol can be replaced by their corresponding ideal interactions, the simulator can emulate the roles of these trusted functionalities in its internal run. Informally, the simulator needs to show that:

- All the transcripts from the real interactions can be simulated (else the adversary and environment can distinguish between the real and ideal interactions)
- The honest parties receive their outputs correctly (once again, since the adversary and environment can distinguish the real and ideal interaction simply based on the outputs)

2.3 Homomorphic Encryption

Homomorphic Encryption (HE) is a special class of encryption schemes that allows for computation over the encrypted data. Given any two values $\mathsf{Enc}_{\mathsf{pk}}(a)$ and $\mathsf{Enc}_{\mathsf{pk}}(b)$, we can compute both $\mathsf{Enc}_{\mathsf{pk}}(ab)$ and $\mathsf{Enc}_{\mathsf{pk}}(a+b)$ without knowing the decryption key sk. Such an encryption scheme is known as a fully homomorphic encryption (FHE) scheme as opposed to a partial homomorphic encryption scheme that supports one of these operations. Craig Gentry in 2009 [53, 52], was the first to demonstrate that such fully homomorphic encryption schemes are possible. Since then, there has been an explosion of work in faster and improved FHE schemes [47, 17, 30].

More generally, given a set of messages $m_1, m_2, \ldots, m_n$, it is possible to compute an encrypted value which decrypts to $f(m_1, m_2, \ldots, m_n)$ for a given function $f(\cdot)$ using only $\mathfrak{c}_{m_1}, \mathfrak{c}_{m_2}, \ldots, \mathfrak{c}_{m_n}$. A homomorphic encryption scheme has the following properties stated informally:

- The decryption of the evaluation $f(\cdot)$ using the encrypted values $\mathfrak{c}_{m_1}, \mathfrak{c}_{m_2}, \ldots, \mathfrak{c}_{m_n}$ should result in $f(m_1, m_2, \ldots, m_n)$ (this is known as the correctness property).
- The evaluation of the $f(\cdot)$ using $\mathfrak{c}_{m_1}, \mathfrak{c}_{m_2}, \ldots, \mathfrak{c}_{m_n}$ is indistinguishable from an encryption of $f(m_1, m_2, \ldots, m_n)$ (this is known as the circuit privacy property).
- The decryption circuit is "small" (this is known as the compactness property).

Somewhat homomorphic encryption or leveled homomorphic encryption is a relaxed variant of a fully homomorphic encryption scheme where a finite depth circuit can be computed, i.e., the function $f(\cdot)$ has a finite depth[1]. This usually comes at a significant improvement in performance of the algorithm as it eliminates the expensive bootstrapping required for known FHE schemes. At the same time, larger depth schemes require larger encryption parameters which in turn reduce the efficiency. Finally, there are parameter regimes that are known to be insecure and where efficient attacks exist. Hence setting the parameters of HE schemes is a complex interplay of security, efficiency as well as the computation required. In the next section, a quick introduction to the BFV scheme is presented which will be required for the purposes of this dissertation.

2.3.1 BFV Scheme

We use the Fan-Vercauteren variant of Brakerski's scale-invariant HE scheme [16, 47], which we shall refer to as the BFV scheme. The BFV scheme, similar to some of the most efficient FHE schemes, is based on the Ring Learning With Errors (RLWE) assumption [78]. In Chapter 5, we will use this HE scheme to improve matrix multiplication in MPC. Below, we briefly describe the set-up and notation followed by a description of the encryption and decryption algorithms.

Notation. For a positive integer q, let $\mathbb{Z}_q = \mathbb{Z} \cap (-q/2, q/2]$. For a finite set S, $U(S)$ denotes a uniform distribution over S. The basic algebraic structure is a polynomial ring $R = \mathbb{Z}[X]/(X^N + 1)$ and $R_q = \mathbb{Z}_q[X]/(X^N + 1)$ where N is a power of 2. These are the ring of integers of $(2N)$-th cyclotomic field and its residue ring modulo q. We define $\|a\|_\infty$ of an element $a \in R_q$ as the infinite norm of its coefficient vector in $\mathbb{Z}_q^N$. A secret key $\mathsf{sk} = s \in R$ is sampled uniformly from the set R_3 of ternary polynomials with coefficients in $\{0, \pm 1\}$. We define the *normalized norm* of randomness r_m by $\|r_m\| = \max\{\|u\|_\infty, \rho^{-1} \cdot \|e_0\|_\infty, \rho^{-1} \cdot \|e_0\|_\infty\}$. For $B > 0$, we call $\mathfrak{c}$ a *B-ciphertext* if there exists $m \in R_p$ and $r_m = (u, e_0, e_1) \in R^3$ such that $\|r_m\| \leq B$ and $\mathfrak{c} = \mathsf{Enc}_{\mathsf{pk}}(x, r_x)$. We also use U_B to denote a uniform distribution over the set of triples $r = (u, e_0, e_1) \in R^3$ such that $\|u\| \leq B$. Finally, $\vec{x}$ denotes vectors, i.e., $\vec{x} = (x_1, \ldots, x_k)$ for some k specified in the context. We also use the notation $[k]$ to denote the set $\{1, 2, \ldots, k\}$.

BFV Scheme. A public key of BFV is generated by

$$\mathsf{pk} = (-a \cdot s + e, a) \in R_q^2, \tag{2.3}$$

for $a \leftarrow U(R_q)$ and $e \leftarrow \chi$ from the error distribution χ over R. We set as χ the discrete Gaussian with small variance and let ρ be an upper bound of χ, i.e., $|e| \leq \rho$ holds with an overwhelming probability where $e \leftarrow \chi$. The BFV encryption,

[1]The depth of a circuit is defined as the length of the longest path from input to output, which frequently is the number of dependent multiplications in the circuit.

decryption is given by the following equations:

$$\begin{aligned} \mathsf{Enc} : m \mapsto \mathfrak{c}_m &= u \cdot \mathsf{pk} + (\Delta \cdot m + e_0, e_1) \pmod{q} \\ \mathsf{Dec} : \mathfrak{c}_m \mapsto m &= \lfloor \Delta^{-1} \cdot (\mathfrak{c}_0 + \mathfrak{c}_1 \cdot s) \rceil \pmod{p} \end{aligned} \tag{2.4}$$

where $\mathfrak{c}_m = (\mathfrak{c}_0, \mathfrak{c}_1)$, $m \in R_p$ is the message to be encrypted, $\Delta = q/p$, $u \leftarrow U(R_3)$, $e_0, e_1 \leftarrow \chi$ are small polynomials, and $\lfloor \cdot \rceil$ denotes the nearest integer function. For the remainder of the paper, we use the shorthand $r_m = (u, e_0, e_1) \in R^3$ to denote the randomness used for encrypting a plaintext m. We write $\mathfrak{c}_m = \mathsf{Enc}(m, r_m)$ when the randomness is taken as input of encryption.

Single Instruction Multiple Data (SIMD) Optimization. An optimization that significantly improved the performance of HE schemes is the technique of packing multiple values into a single ciphertext. The native plaintext space of BFV is R_p, but using the Discrete Fourier Transform (DFT) over $\mathbb{Z}_p$, one can pack multiple values in a single ciphertext and thereby support parallel computation in a SIMD manner. We choose a plaintext modulus satisfying $p = 1 \pmod{2N}$ so that $X^N + 1 = \prod_{i \in \mathbb{Z}_{2N}^\times} (X - \alpha^i)$ factors into a product of N linear polynomials for a $(2N)$-th root of unity α modulo p. Hence, we can use the packing technique via ring isomorphism $R_p \to \mathbb{Z}_p^N$, $m(X) \mapsto (m(\alpha^i))_{i \in \mathbb{Z}_{2N}^\times}$. Recall that the multiplicative group $\mathbb{Z}_{2N}^\times$ is isomorphic to $\mathbb{Z}_{N/2} \times \mathbb{Z}_2$. In our implementation, we utilize $N/2$ slots corresponding to the cyclic subgroup $\langle 5 \rangle$ of order $N/2$. It allows us to rotate an encrypted vector by evaluating the automorphism $X \mapsto X^5$ homomorphically. More generally, we can perform an arbitrary linear transformation on these two vectors by combining homomorphic rotation and plaintext-ciphertext multiplication in BFV. The complexity of a linear transformation is mainly dominated by k rotations where $k \leq N/2$ is the number of nonzero diagonals $(A_{0,i}, A_{1,i+1} \ldots, A_{N/2-1,i-1})$ of its matrix representation $A \in \mathbb{Z}_p^{N/2 \times N/2}$. Please refer to [56] for details.

2.4 Zero-Knowledge Proofs

A zero-knowledge proof is a protocol by which one party (called the prover) can prove to another party (called the verifier), the correctness of some statement without revealing anything about the statement. Consider, for example, a prover who wishes to prove to a verifier that he knows the password corresponding to an email ID: swag@princeton.edu. A naive solution would be for the prover to hand over the password to the verifier. This is clearly not zero-knowledge as the prover has to give the private knowledge (called the witness) to the verifier. A potential zero-knowledge protocol would be to have the prover send an email to the verifier from the ID with a given challenge text. This protocol, within the limits of secure email, can then be said to be zero-knowledge.

More formally, a zero knowledge proof must have the following three properties:

(A) Soundness: If the statement is false, then no prover should be able to successfully convince the verifier that it is true, except with some small probability.

(B) Completeness: If the statement is true then a honest verifier (one that follows the protocol) should be convinced by an honest prover.

(C) Zero-knowledge: If the statement is true, the verifier learns nothing other than the fact that the statement is true.

2.5 Machine Learning Algorithms

Machine learning is a class of computer algorithms that can perform specific tasks without explicit instructions. An example of this is feeding the algorithms a large corpus of data and programming the algorithm to "learn" the patterns and inferences necessary for automatically performing the task. Such algorithms are widely used in computer vision, spam filtering, natural language processing, etc. For the purpose of this dissertation, we will focus on a class of learning algorithms called supervised learning, applied specifically to computer vision.

In supervised learning, the algorithm is provided with a large corpus of data called training data as well as the appropriate classification of each data point, called the label. Refer to Section 2.5.2 for example datasets considered in this work. Another smaller dataset, or a fraction of the training dataset, is kept aside for testing purposes and is called test data. A supervised learning algorithm then iterates over the training dataset, using the provided label as feedback, to modify its parameters. Eventually, after a large number of iterations, the algorithm converges on a state of parameters that would usually work with high accuracy on the test dataset and at this point the model is referred to as trained.

2.5.1 Neural Networks

The focus of this dissertation is on Deep and Convolutional Neural Network (DNN and CNN) training algorithms. These are a class of ML algorithms that have shown tremendous promise on computer vision applications. The network, which consists of a series of transformation of the input, consists of different types of "layers," the output of the previous one being fed into the next layer. Each layer consists of a number of neurons, a strategy that aims to mimic the human brain in terms of its processing. Each neuron receives a combination of some set of neurons from the previous layer and the firing of neurons is typically mimicked using a thresholding function. The output of the final layer indicates the appropriate label corresponding to the input data (this is called the forward pass). In the training phase, the "predicted output" is compared with the ground truth and the difference is used as a feedback to improve the learnable parameters of the network (this is called the backward pass). This process repeated many times, also known as (stochastic) gradient descent, results in a trained model. A number of different types of network architectures exist that differ in the types of layers used, the sizes/parameters of these layers, the number of

layers, etc. We will take a quick look at the general structure of such networks and then briefly describe a few network architectures relevant to this work.

The most common layers in this class of NN architectures are linear layers – fully-connected and convolutional. Fully-connected layers connect every neuron of the previous layer to every neuron of the following layer. A set of "weights", corresponding to a connection between an input neuron and an output neuron (on the next layer), controls the strength or importance of that input neuron to the output neuron. These comprise the learnable parameters of that layer. A closely related layer is the convolution where the weight matrix is convoluted with the input to result in the output. Such an operation preserves local structures, an important factor in enabling its success in computer vision. In terms of their functionality, these two layers correspond to matrix multiplication and a convolution. In a broad range of NN architectures, every layer in the forward propagation contains a linear operation followed by a (non-linear) activation function f. One of the most popular activation functions is the Rectified Linear Unit (ReLU) defined as $\mathsf{ReLU}(x) = \mathsf{max}(0, x)$. The back-propagation updates the weights appropriately making use of derivative of the activation function (in this case $\mathsf{ReLU}'(x)$, which is defined to be 1 if $x > 0$ and 0 otherwise) and matrix multiplication.

A large class of networks can be represented using the following functions: matrix multiplication, convolution, ReLU, Maxpool (defined as the maximum of a set of values, usually in a sub-matrix), batch-normalization (defined as $\frac{x_i}{\sum x_i}$ for a given set of values $\{x_1, \cdots, x_n\}$) and their derivatives. It is important to note that in MPC, the non-linear activation functions are the dominant cost in stark contrast to plaintext machine learning where matrix multiplications are the dominant cost. This is because non-linear functions, in particular in conjunction with linear layers are expensive to evaluate privately. We consider the following NN architectures in this work:

(A) **Network-A:** This is a 3-layered fully-connected network with ReLU activation after each layer considered in SecureML [85] (see Figure D.1). This is the smallest network with around 118K parameters.

(B) **Network-B:** This network again is a 3-layered network with the first layer as convolution followed by 2 fully-connected layers using ReLU activation after each layer. This architecture is chosen from Chameleon [95] with approximately 100K parameters (see Figure D.2).

(C) **Network-C:** This is a 4-layered network with 2 convolutional and 2 fully-connected layers selected from prior work MiniONN [77]. This network uses Max Pooling in addition to ReLU layer and has around 10,500 parameters in total (shown in Figure D.3).

(D) **LeNet:** This network, first proposed by LeCun et al. [76], was used in automated detection of zip codes and digit recognition [75]. The network contains 2 convolutional layers and 2 fully connected layers with 431K parameters (shown in Figure D.4).

(E) **AlexNet:** AlexNet is the famous winner of the 2012 ImageNet ILSVRC-2012 competition [69]. It has 5 convolutional layers and 3 fully connected layers and

uses Batch-Normalization layer for stability, efficient training, and has about 60 Million parameters (see Figure D.5).

(F) **VGG16:** Another network which we test on is called VGG16, the runner-up of the ILSVRC-2014 competition [102]. VGG16 has 16 layers and has about 138 Million parameters (see Figure D.6).

(G) **ResNet-50:** We also evaluate our framework on ResNet-50, a residual network (RNN, a subclass of CNNs) with 50 layers, the winner of the ILSVRC-2015 competition [59]. VGG16 has about 23 Million parameters (refer to [86] for details).

More details on these networks as well as the governing equations are deferred to Appendix B.

2.5.2 Datasets

This dissertation uses 3 datasets popularly used for training image classification models — MNIST [83], CIFAR-10 [68], and Tiny ImageNet [114]. These are briefly described below:

(A) **MNIST [83]:** MNIST is a collection of handwritten digits dataset. It consists of 60,000 images in the training set and 10,000 in the test set. Each image is a 28×28 pixel image of a handwritten digit along with a label between 0 and 9. We evaluate Network-A, B, C, and the LeNet network on this dataset in both the semi-honest and maliciously secure variants.

(B) **CIFAR-10 [68]:** CIFAR-10 consists of 60,000 images (50,000 training and 10,000 test images) of 10 different classes (such as airplanes, dogs, horses, etc.). There are 6,000 images of each class with each image consisting of a colored 32×32 image. We perform private training and inference of AlexNet and VGG16 on this dataset.

(C) **Tiny ImageNet [114]:** Tiny ImageNet dataset consists of 100,000 training samples and 10,000 test samples with 200 different classes [114]. Each sample is cropped to a size of $64 \times 64 \times 3$. We perform private training and inference of AlexNet and VGG16 on this dataset.

Chapter 3

SecureNN: Efficient 3-Party Computation Protocols

Neural Networks (NN) provide a powerful method for machine learning training and inference. To effectively train, it is desirable for multiple parties to combine their data – however, doing so conflicts with data privacy. This work proposes SecureNN that provides novel three-party secure computation protocols for various NN building blocks such as matrix multiplication, convolutions, Rectified Linear Units, Maxpool, normalization, and so on. This enables us to construct three-party secure protocols for training and inference of several NN architectures such that no single party learns any information about the data. Experimentally, we implement SecureNN and deploy it over Amazon EC2 servers in different settings (LAN/WAN). SecureNN advances the state-of-the-art of secure computation for neural networks in three ways:

(A) Scalability: SecureNN is the first work to provide neural network training on Convolutional Neural Networks (CNNs) that have an accuracy of $> 99\%$ on the MNIST dataset.

(B) Performance: For secure inference, SecureNN outperforms prior 2- and 3-server works (SecureML, MiniONN, Chameleon, Gazelle) by 6×-113× (with larger gains obtained in more complex networks). Total execution times for SecureNN are 2×-4× lower than even just the online times of these works. For secure training, compared to the only prior work (SecureML) that considered a much smaller fully-connected network, SecureNN protocols are 79× and 7× faster, respectively, than their 2- and 3-server protocols. In the WAN setting, these improvements are more dramatic and we obtain an improvement of 553×!

(C) Security: Our protocols provide two kinds of security: full security (privacy and correctness) against one semi-honest corruption and the notion of privacy against one malicious corruption [Araki *et al.* CCS'16]. All prior works only provide semi-honest security and ours is the first system to provide any security against malicious adversaries for the secure computation of complex algorithms such as NN inference and training.

These gains come from a significant improvement in communication through the elimination of expensive garbled circuits and oblivious transfer protocols.

3.1 SecureNN Overview

Secure protocols for NN algorithms generally follow the paradigm of executing arithmetic computation, such as matrix multiplication and convolutions, using Beaver triplets or homomorphic encryption and executing Boolean computation, such as ReLU, Maxpool, and their derivatives, using Yao's garbled circuits. In order to make these protocols compatible with each other, share conversion protocols are also used to move from an arithmetic encoding (either arithmetic sharing or homomorphic encryption ciphertext) to a Boolean encoding (garbled encoding) and vice-versa. The communication cost of securely evaluating Boolean computations is prohibitive due to the use of Yao's garbled circuits that incur a multiplicative factor overhead of 128 (the security parameter, κ). This is precisely where our new protocols come to the rescue. We develop new protocols for Boolean computation (such as ReLU, Maxpool, and their derivatives) that have much lesser communication overhead (at least $8\times$ better) than the cost of converting to a Yao encoding and executing a garbled circuit. We now present our techniques in more detail.

We denote the three servers by P_0, P_1, and P_2. At the start of any protocol, parties P_0 and P_1 hold 2-out-of-2 additive secret shares of the inputs to the protocol. All our protocols maintain the invariant that at the end of the protocol P_0 and P_1 hold 2-out-of-2 shares of the output. We stress that even though for all our protocols only P_0 and P_1 hold the shares of the input and the output, P_2 also crucially takes part in the real computation during the protocol. That is, P_2 is not a dummy party that only assists in the two-party protocol between P_0 and P_1 by providing relevant randomness.

Non-linear activations. We first describe our main ideas for computing the derivative of ReLU function, that is ReLU'.

Function ReLU'. Note that $\mathsf{ReLU}'(x)$ is 1 if $x \geq 0$ and 0 otherwise. First, we note that $\mathsf{ReLU}'(x)$ is closely related to the most-significant bit (MSB)[1] of x in our representation of values in $\mathbb{Z}_{2^{64}}$. That is, $\mathsf{ReLU}'(x)$ is 1 iff $\mathsf{MSB}(x) = 0$. Hence, it suffices to compute the $\mathsf{MSB}(x)$. Next, since computing the LSB of a number is much easier than computing the MSB (as it does not require bit extraction), we flip the problem to computing the LSB as follows: $\mathsf{MSB}(a) = \mathsf{LSB}(2a)$ if we are working over an odd ring[2]. For now, let us assume that we are working over an odd ring and we later describe how we go from even ring $\mathbb{Z}_{2^{64}}$ to an odd ring $\mathbb{Z}_{2^{64}-1}$.

At the start of the protocol, P_0 and P_1 hold shares of a (over $\mathbb{Z}_{2^{64}-1}$), using which they locally compute shares of $y = 2a$. Now, P_2 would assist in computing the LSB of y as follows: From now on, we denote $\mathsf{LSB}(y) = y[0]$. The first observation is that for three numbers u, v, w such that $u = v + w$, $u[0] = v[0] \oplus w[0]$ if the addition does not "wrap around" the ring and $u[0] = u[0] \oplus v[0] \oplus 1$ if addition wraps around. The second observation is that if x is a random number chosen by P_2 and is unknown to P_0 and P_1, then it is okay for P_0, P_1 to learn $r = y + x$. This is because the secret y is masked by random x. Hence, P_2 gives secret shares of x as well as shares of

[1] $\mathsf{MSB}(x)$ is the leftmost bit in the 64-bit representation of x.

[2] In a group of order n, we have $\mathsf{MSB}(x) = 1$ iff $x > n/2$ iff $n > 2x - n > 0$; if n is odd, then so is $2x - n$ and it follows that $\mathsf{MSB}(x) = 1$ iff $\mathsf{LSB}(2x) = 1$.

$x[0]$ to P_0, P_1 and they reconstruct r. Now, all that is left is to figure out whether the addition $y + x$ wraps around the ring or not. For this, the third observation is that this addition wraps around if and only if the final sum is less than one of the individual values – that is, it wraps around iff $x > r$. Thus, if we can compute shares of $x > r$ between P_0 and P_1, then we are done.

Next, we construct a protocol for comparison. We first define a functionality called *private compare* (denoted by $\mathcal{F}_{\mathsf{PC}}$). This three-party functionality assumes that P_0 and P_1 each have a share of the bits of ℓ-bit value x (over field $\mathbb{Z}_p$) as well as a common number r and a random bit β as input. It computes the bit $(x > r)$ (which is 1 if $x > r$ and 0 otherwise) and XOR masks it with the bit β. This output $\beta \oplus (x > r)$ is given to P_2. We implement this functionality by building on the techniques of [37, 87] and provide a more efficient variant. Note that this protocol requires parties to have shares of bits of x over field $\mathbb{Z}_p$. These are provided to P_0, P_1 by P_2. With these protocols, we are ready to compute the $\mathsf{ReLU}'(\cdot)$ function if P_0 and P_1 began with shares of the input over an odd ring.

Now, we revisit the requirement of an odd ring. As we explained above, all of this works, if we had shares of a over an odd ring. Now, we could execute our protocol over the ring $\mathbb{Z}_N$ with N being odd. However doing so is fairly inefficient as matrix multiplication over the ring $\mathbb{Z}_{2^{64}}$ (or $\mathbb{Z}_{2^{32}}$) is much faster. This is because (as observed in [85]), native implementation of matrix multiplication over `long` (or `int`) automatically implements the modulo operation over $\mathbb{Z}_{2^{64}}$ (or $\mathbb{Z}_{2^{32}}$) and many libraries heavily optimize matrix multiplication over these rings, which give significant efficiency improvements compared to operations over any other ring. Hence, we provide a protocol that converts values ($\neq L-1$) that are secret shared over $\mathbb{Z}_L$ into shares over $\mathbb{Z}_{L-1}$. This protocol also uses the private compare protocol and may be of independent interest.

Finally, this design (and our protocol) enables us to run our comparison protocol (the protocol that realizes $\mathcal{F}_{\mathsf{PC}}$ above) over a *small* field $\mathbb{Z}_p$ (we choose $p = 67$ concretely) and this reduces the communication complexity significantly. Using these protocols, we obtain our protocol for computing $\mathsf{ReLU}'(x)$ (the derivative of ReLU) beginning with shares over $\mathbb{Z}_{2^{64}}$.

Figure 3.1 shows the different secret sharing schemes used in protocols in SecureNN. Specifically, it shows how various secret sharing schemes are used in ReLU and Private Compare protocols.

Other non-linear functions. In this work, we describe protocols for ReLU, Maxpool, their derivatives, and normalization or division. Maxpool, ReLU, and division are implemented using ReLU' and multiplication. Similar ideas can be used to obtain efficient protocols for other non-linear activation functions such as Leaky ReLU, piecewise linear functions (used to approximate sigmoid), and their derivatives. We also construct an efficient protocol for the derivative of Maxpool exploiting specific number-theoretic properties.

Matrix multiplication and Convolutions. An information-theoretic matrix multiplication protocol over shares when 3 parties are involved is straightforward using matrix-based Beaver multiplication triplets [11] and is omitted from the dis-

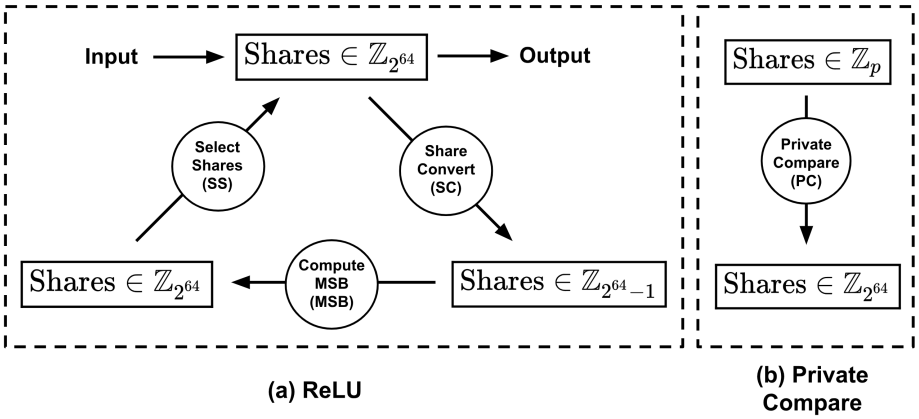


Figure 3.1: Flow of different types of secret sharing schemes used in (a) ReLU, (b) Private Compare.

cussion here. For our implementation we use Beaver triplets generated using PRFs. Convolutions are implemented in a very similar manner to matrix multiplication.

Privacy against malicious adversaries. Since our protocols are fundamentally information-theoretic, it is easy to show that all messages exchanged in the protocol are individually uniformly random. As noted by Araki *et al.* [6], this property then suffices to show that any two executions of the protocol with different inputs are indistinguishable to the malicious adversary and subsequently that the same protocols provide privacy against malicious adversaries. This guarantees that a malicious server cannot learn anything about clients' inputs even if it deviates arbitrarily from the protocol.

3.2 Protocol Constructions

In this section, we describe various building blocks to our main protocols. Some of these protocols deviate from the invariant described before, i.e., P_0 and P_1 do not necessarily begin/end protocols with shares of input/output over $\mathbb{Z}_L$. Further, P_2 receives output in these protocols. A formal description of the functionalities realized by these protocols is given in [112]. We provide the proofs of security for our protocols in Appendix C.1 and A.3. We start with the simplest protocols (such as those for matrix multiplication) and gradually build other protocols that are used in the computation of non-linear functions.

3.2.1 Matrix Multiplication

Algorithm 1 describes our protocol for secure multiplication (functionality $\mathcal{F}_{\mathsf{MATMUL}}$) where parties P_0 and P_1 hold shares of $X \in \mathbb{Z}_L^{m \times n}$ and $Y \in \mathbb{Z}_L^{n \times v}$ and the functionality outputs fresh shares of $Z = X \cdot Y$ to P_0, P_1.

Algorithm 1 Mat. Mul. $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2)$:

Input: P_0 & P_1 hold $(\langle X\rangle_0^L, \langle Y\rangle_0^L)$ & $(\langle X\rangle_1^L, \langle Y\rangle_1^L)$ resp.
Output: P_0 gets $\langle X \cdot Y\rangle_0^L$ and P_1 gets $\langle X \cdot Y\rangle_1^L$.
Common Randomness: P_0 and P_1 hold shares of zero matrices over $\mathbb{Z}_L^{m\times v}$ resp.; i.e., P_0 holds $\langle 0^{m\times v}\rangle_0^L = U_0$ & P_1 holds $\langle 0^{m\times v}\rangle_1^L = U_1$

1: P_2 picks random matrices $A \xleftarrow{\$} \mathbb{Z}_L^{m\times n}$ and $B \xleftarrow{\$} \mathbb{Z}_L^{n\times v}$ and generates for $j \in \{0,1\}$, $\langle A\rangle_j^L, \langle B\rangle_j^L, \langle C\rangle_j^L$ and sends to P_j, where $C = A \cdot B$.
2: For $j \in \{0,1\}$, P_j computes $\langle E\rangle_j^L = \langle X\rangle_j^L - \langle A\rangle_j^L$ and $\langle F\rangle_j^L = \langle Y\rangle_j^L - \langle B\rangle_j^L$.
3: P_0 & P_1 reconstruct E & F by exchanging shares.
4: For $j \in \{0,1\}$, P_j outputs $-jE \cdot F + \langle X\rangle_j^L \cdot F + E \cdot \langle Y\rangle_j^L + \langle C\rangle_j^L + U_j$.

Intuition. Our protocol relies on standard cryptographic technique for multiplication of using Beaver triplets [11] generalized to the matrix setting. P_2 generates these triplet shares and sends to parties P_0, P_1.

3.2.2 Select Share

Algorithm 2 describes our 3-party protocol realizing the select share functionality $\mathcal{F}_{\mathsf{SS}}$, which is as follows: Parties P_0, P_1 hold shares of x, y over $\mathbb{Z}_L$. They also hold shares of a selection bit $\alpha \in \{0,1\}$ over $\mathbb{Z}_L$ ($L = 2^{64}$). Parties P_0, P_1 get fresh shares of x if $\alpha = 0$ and fresh shares of y if $\alpha = 1$ from the functionality.

Algorithm 2 SelectShare $\Pi_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $(\langle\alpha\rangle_0^L, \langle x\rangle_0^L, \langle y\rangle_0^L)$ and $(\langle\alpha\rangle_1^L, \langle x\rangle_1^L, \langle y\rangle_1^L)$, resp.
Output: P_0, P_1 get $\langle z\rangle_0^L$ and $\langle z\rangle_1^L$, resp., where $z = (1-\alpha)x + \alpha y$.
Common Randomness: P_0 and P_1 hold shares of 0 over $\mathbb{Z}_L$ denoted by u_0 and u_1.

1: For $j \in \{0,1\}$, P_j compute $\langle w\rangle_j^L = \langle y\rangle_j^L - \langle x\rangle_j^L$
2: P_0, P_1, P_2 invoke $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle\alpha\rangle_j^L, \langle w\rangle_j^L)$ and P_0, P_1 learn $\langle c\rangle_0^L$ and $\langle c\rangle_1^L$, resp.
3: For $j \in \{0,1\}$, P_j outputs $\langle z\rangle_j^L = \langle x\rangle_j^L + \langle c\rangle_j^L + u_j$.

Intuition. We note that selecting between x and y can be arithmetically expressed as $(1-\alpha)\cdot x + \alpha \cdot y = x + \alpha \cdot (y - x)$. We compute the latter expression in our protocol using one call to Π_{MatMul} for multiplying α and $(y - x)$.

3.2.3 Private Compare

Algorithm 3 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\mathsf{PC}}$ for comparison that is as follows: The parties P_0 and P_1 holds shares of bits of ℓ-bit integer x in $\mathbb{Z}_p$ ($p = 67$ and hence $\mathbb{Z}_p$ is a Field), i.e., $\{\langle x[i]\rangle_0^p\}_{i\in[\ell]}$ and $\{\langle x[i]\rangle_1^p\}_{i\in[\ell]}$, respectively. P_0, P_1 also hold an ℓ-bit integer r and a bit β. At the end of the protocol, P_2 learns

a bit $\beta' = \beta \oplus (x > r)$, where $(x > r)$ denotes the bit which is 1 when $x > r$ over the integers and 0 otherwise.

Algorithm 3 PrivateCompare $\Pi_{\mathsf{PC}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\{\langle x[i]\rangle_0^p\}_{i\in[\ell]}$ and $\{\langle x[i]\rangle_1^p\}_{i\in[\ell]}$, respectively, a common input r (an ℓ bit integer) and a common random bit β.

Output: P_2 gets a bit $\beta \oplus (x > r)$.

Common Randomness: P_0, P_1 hold ℓ common random values $s_i \in \mathbb{Z}_p^*$ for all $i \in [\ell]$ and a random permutation π for ℓ elements. P_0 and P_1 additionally hold ℓ common random values $u_i \in \mathbb{Z}_p^*$.

1: Let $t = r + 1 \mod 2^\ell$.
2: For each $j \in \{0, 1\}$, P_j executes Steps 3–14:
3: **for** $i = \{\ell, \ell - 1, \ldots, 1\}$ **do**
4: **if** $\beta = 0$ **then**
5: $\langle w_i\rangle_j^p = \langle x[i]\rangle_j^p + jr[i] - 2r[i]\langle x[i]\rangle_j^p$
6: $\langle c_i\rangle_j^p = jr[i] - \langle x[i]\rangle_j^p + j + \sum_{k=i+1}^{\ell} \langle w_k\rangle_j^p$
7: **else if** $\beta = 1$ AND $r \neq 2^\ell - 1$ **then**
8: $\langle w_i\rangle_j^p = \langle x[i]\rangle_j^p + jt[i] - 2t[i]\langle x[i]\rangle_j^p$
9: $\langle c_i\rangle_j^p = -jt[i] + \langle x[i]\rangle_j^p + j + \sum_{k=i+1}^{\ell} \langle w_k\rangle_j^p$
10: **else**
11: If $i \neq 1$, $\langle c_i\rangle_j^p = (1 - j)(u_i + 1) - ju_i$, else $\langle c_i\rangle_j^p = (-1)^j \cdot u_i$.
12: **end if**
13: **end for**
14: Send $\{\langle d_i\rangle_j^p\}_i = \pi\left(\left\{s_i\langle c_i\rangle_j^p\right\}_i\right)$ to P_2
15: For all $i \in [\ell]$, P_2 computes $d_i = \mathsf{Reconst}^p(\langle d_i\rangle_0^p, \langle d_i\rangle_1^p)$ and sets $\beta' = 1$ iff $\exists i \in [\ell]$ such that $d_i = 0$.
16: P_2 outputs β'.

Intuition. Our starting point is the 2-party comparison present in [37, 87]. We build on this to give a much more efficient 3-party protocol. We want to compute $\beta' = \beta \oplus (x > r)$. That is, for $\beta = 0$, we compute $x > r$ and for $\beta = 1$, we compute $1 \oplus (x > r) \equiv (x \leq r) \equiv (x < (r + 1))$ over integers. We discuss the corner case of $r = 2^\ell - 1$ below. In this case, $x \leq r$ is always true.

Consider the case of $\beta = 0$. In this case, $\beta' = 1$ iff $(x > r)$ or at the leftmost bit where $x[i] \neq r[i]$, $x[i] = 1$. We compute $w_i = x[i] \oplus r[i] = x[i] + r[i] - 2x[i]r[i]$ and $c[i] = r[i] - x[i] + 1 + \sum_{k=i+1}^{\ell} w_k$. Since r is known to both P_0, P_1, shares of both w_i and c_i can be computed locally. Now, we can prove that $\exists\ i.c_i = 0$ iff $x > r$. Hence, both P_0, P_1 send shares of c_i to P_2 that reconstructs c_i and looks for a 0. To ensure security against a corrupt P_2, we hide exact values of non-zero c_i's and position of (a possible) 0 by multiplying with random s_i and permuting these values by a common permutation π. These s_i and π are common to both P_0 and P_1.

In the case when $\beta = 1$ and $r \neq 2^\ell - 1$, we compute $(r+1) > x$ using similar logic as above. In the corner case of $r = 2^\ell - 1$, both parties P_0, P_1 know that result of $x \leq r \equiv (r+1) > x$ over integers should be true. Hence, they together pick shares of c_i such that there is exactly one 0. This is done by P_0, P_1 having common values u_i that they use to create a valid share of a 0 and $\ell - 1$ shares of 1 (see Step 11). Note that for the re-randomization using s_i's to work, it is crucial that we work over a field such as $\mathbb{Z}_p$.

3.2.4 Share Convert

Algorithm 4 describes our three-party protocol for converting shares over $\mathbb{Z}_L$ to $\mathbb{Z}_{L-1}$ realizing the functionality $\mathcal{F}_{\mathsf{SC}}$ (again, $L = 2^{64}$). Here, parties P_0, P_1 hold shares of $\langle a \rangle^L$ such that $a \neq L-1$. At the end of the protocol, P_0, P_1 hold fresh shares of same value over $L-1$, i.e., $\langle a \rangle^{L-1}$. In this algorithm, $\kappa := \mathsf{wrap}(x, y, L)$ is 1 if $x+y \geq L$ over integers and 0 otherwise. That is, κ denotes the wrap-around bit for the computation $x + y \mod L$.

Intuition: Let $\theta = \mathsf{wrap}(\langle a \rangle_0^L, \langle a \rangle_1^L, L)$. Now, we note that if $\theta = 1$, i.e., if the original shares wrapped around L, then we need to subtract 1, else original shares are also valid shares of same value of $L-1$. Hence, in the protocol we compute shares of bit θ over $L-1$ and subtract from original shares locally. This protocol makes use of novel modular arithmetic to securely compute these shares of θ, an idea which is potentially of independent interest. We explain these relations in the correctness proof below.

Lemma 3.1. *Protocol $\Pi_{\mathsf{SC}}(\{P_0, P_1\}, P_2)$ in Algorithm 4 securely realizes $\mathcal{F}_{\mathsf{SC}}$.*

Proof. For correctness we need to prove that $\mathsf{Reconst}^{L-1}(\langle y \rangle_0^{L-1}, \langle y \rangle_1^{L-1}) = \mathsf{Reconst}^{L}(\langle a \rangle_0^{L}, \langle a \rangle_1^{L}) = a$. Looking at Step 11 of the protocol and the intuition above, it suffices to prove that $\mathsf{Reconst}^{L-1}(\langle \theta \rangle_0^{L-1}, \langle \theta \rangle_1^{L-1}) = \theta = \mathsf{wrap}(\langle a \rangle_0^L, \langle a \rangle_1^L, L)$. First, by correctness of protocol Π_{PC}, $\eta' = \eta'' \oplus (x > r-1)$. Next, let $\eta = \mathsf{Reconst}^{L-1}(\langle \eta \rangle_0^{L-1}, \langle \eta \rangle_1^{L-1}) = \eta' \oplus \eta'' = (x > r-1)$. Next, note that $x \equiv a + r \mod L$. Hence, $\mathsf{wrap}(a, r, L) = 0$ iff $x > r-1$. By the correctness of wrap, following relations hold over the integers:

(A) $r = \langle r \rangle_0^L + \langle r \rangle_1^L - \alpha L$.
(B) $\langle \tilde{a} \rangle_j^L = \langle a \rangle_j^L + \langle r \rangle_j^L - \beta_j L$.
(C) $x = \langle \tilde{a} \rangle_0^L + \langle \tilde{a} \rangle_1^L - \delta L$.
(D) $x = a + r - (1-\eta)L$.
(E) Let θ be such that $a = \langle a \rangle_0^L + \langle a \rangle_1^L - \theta L$.

Computing, (1) − (2) − (3) + (4) + (5) gives us $\theta = \beta_0 + \beta_1 - \alpha + \delta + \eta - 1$. This is exactly what the parties P_0 and P_1 calculate in Step 10. We prove security in Appendix C.1. □

Algorithm 4 ShareConvert $\Pi_{\mathsf{SC}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\langle a\rangle_0^L$ and $\langle a\rangle_1^L$, respectively such that $\mathsf{Reconst}^L(\langle a\rangle_0^L, \langle a\rangle_1^L) \neq L-1$.
Output: P_0, P_1 get $\langle a\rangle_0^{L-1}$ and $\langle a\rangle_1^{L-1}$.

Common Randomness: P_0, P_1 hold a random bit η'', a random $r \in \mathbb{Z}_L$, shares $\langle r\rangle_0^L, \langle r\rangle_1^L, \alpha = \mathsf{wrap}_(\langle r\rangle_0^L, \langle r\rangle_1^L, L)$ and shares of 0 over $\mathbb{Z}_{L-1}$ denoted by u_0 and u_1.

1: For each $j \in \{0,1\}$, P_j executes Steps 2–3
2: $\langle \tilde{a}\rangle_j^L = \langle a\rangle_j^L + \langle r\rangle_j^L$ and $\beta_j = \mathsf{wrap}_(\langle a\rangle_j^L, \langle r\rangle_j^L, L)$.
3: Send $\langle \tilde{a}\rangle_j^L$ to P_2.
4: P_2 computes $x = \mathsf{Reconst}^L(\langle \tilde{a}\rangle_0^L, \langle \tilde{a}\rangle_1^L)$ and $\delta = \mathsf{wrap}_(\langle \tilde{a}\rangle_0^L, \langle \tilde{a}\rangle_1^L, L)$.
5: P_2 generates shares $\{\langle x[i]\rangle_j^p\}_{i\in[\ell]}$ and $\langle \delta\rangle_j^{L-1}$ for $j \in \{0,1\}$ and sends to P_j.
6: P_0, P_1, P_2 invoke[3] $\Pi_{\mathsf{PC}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $\left(\{\langle x[i]\rangle_j^p\}_{i\in[\ell]}, r-1, \eta''\right)$ and P_2 learns η'.
7: For $j \in \{0,1\}$, P_2 generates $\langle \eta'\rangle_j^{L-1}$ and sends to P_j.
8: For each $j \in \{0,1\}$, P_j executes Steps 9–11
9: $\langle \eta\rangle_j^{L-1} = \langle \eta'\rangle_j^{L-1} + (1-j)\eta'' - 2\eta''\langle \eta'\rangle_j^{L-1}$
10: $\langle \theta\rangle_j^{L-1} = \beta_j + (1-j)\cdot(-\alpha - 1) + \langle \delta\rangle_j^{L-1} + \langle \eta\rangle_j^{L-1}$
11: Output $\langle y\rangle_j^{L-1} = \langle a\rangle_j^L - \langle \theta\rangle_j^{L-1} + u_j$ (over $L-1$)

3.2.5 Compute MSB

Algorithm 5 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\mathsf{MSB}}$ that computes the most significant bit[4] (MSB) of a value $a \in \mathbb{Z}_{L-1}$. P_0, P_1 hold shares of a over odd ring $\mathbb{Z}_{L-1}$ and end with shares of $\mathsf{MSB}(a)$ over $\mathbb{Z}_L$.

Intuition: Note that when the shares of the private input (say a) are over an odd ring (such as after using Π_{SC}), the MSB computation can be converted into an LSB computation. More precisely, over an odd ring, $\mathsf{MSB}(a) = \mathsf{LSB}(y)$, where $y = 2a$. Now, P_2 assists in computation of shares of $\mathsf{LSB}(y)$ as follows: P_2 picks a random integer $x \in \mathbb{Z}_{L-1}$ and sends shares of x over $\mathbb{Z}_{L-1}$ and shares of $x[0]$ over $\mathbb{Z}_L$ to P_0, P_1. Next, P_0, P_1 compute shares of $r = y + x$ and reconstruct r by exchanging shares. We note that $\mathsf{LSB}(y) = y[0] = r[0] \oplus x[0] \oplus \mathsf{wrap}(y, x, L-1)$ over an odd ring. Also, $\mathsf{wrap}(y, x, L-1) = (x > r)$, which can be computed using comparison protocol Π_{PC}. To enable this, P_2 also secret shares $\{x[i]\}_{i\in[\ell]}$ over $\mathbb{Z}_p$. Steps 6-10 compute the equation $\mathsf{LSB}(y) = y[0] = r[0] \oplus x[0] \oplus (x > r)$ by using the arithmetic equation for xor computation (Note that $x \oplus r = x + r - 2xr$; when one of x or y is public and known to both P_0 and P_1, then this computation can be done over the shares locally. When both are private and kept as shares, this computation is done using one call to the multiplication functionality, i.e., Step 9 in Algorithm 5).

[3] In the corner case when $r = 0$, both P_0 and P_1 set the output of Π_{PC} to be 1 and execute it. This is similar to the other corner case discussed in Section 3.2.3.

[4] Most significant bit of a number is defined as the value of the leftmost bit in the bit representation.

Algorithm 5 ComputeMSB $\Pi_{\mathsf{MSB}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\langle a \rangle_0^{L-1}$ and $\langle a \rangle_1^{L-1}$, respectively.
Output: P_0, P_1 get $\langle \mathsf{MSB}(a) \rangle_0^L$ and $\langle \mathsf{MSB}(a) \rangle_1^L$.
Common Randomness: P_0, P_1 hold a random bit β and random shares of 0 over L, denoted by u_0 and u_1 resp.

1: P_2 picks $x \xleftarrow{\$} \mathbb{Z}_{L-1}$. Next, P_2 generates $\langle x \rangle_j^{L-1}$, $\{\langle x[i] \rangle_j^p\}_i$, $\langle x[0] \rangle_j^L$ for $j \in \{0,1\}$ and sends to P_j.
2: For $j \in \{0,1\}$, P_j computes $\langle y \rangle_j^{L-1} = 2\langle a \rangle_j^{L-1}$ and $\langle r \rangle_j^{L-1} = \langle y \rangle_j^{L-1} + \langle x \rangle_j^{L-1}$.
3: P_0, P_1 reconstruct r by exchanging shares.
4: P_0, P_1, P_2 call $\Pi_{\mathsf{PC}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $\left(\{\langle x[i] \rangle_j^p\}_{i \in [\ell]}, r, \beta\right)$ and P_2 learns β'.
5: P_2 generates $\langle \beta' \rangle_j^L$ and sends to P_j for $j \in \{0,1\}$.
6: For $j \in \{0,1\}$, P_j executes Steps 7–8
7: $\langle \gamma \rangle_j^L = \langle \beta' \rangle_j^L + j\beta - 2\beta \langle \beta' \rangle_j^L$
8: $\langle \delta \rangle_j^L = \langle x[0] \rangle_j^L + jr[0] - 2r[0] \langle x[0] \rangle_j^L$
9: P_0, P_1, P_2 call $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \gamma \rangle_j^L, \langle \delta \rangle_j^L)$ and P_j learns $\langle \theta \rangle_j^L$.
10: For $j \in \{0,1\}$, P_j outputs $\langle \alpha \rangle_j^L = \langle \gamma \rangle_j^L + \langle \delta \rangle_j^L - 2\langle \theta \rangle_j^L + u_j$.

Next, we describe all our main protocols for functionalities such as linear layer, ReLU, the derivative of ReLU (ReLU'), division needed for normalization during training, Maxpool, and its derivative. We maintain the invariant that parties P_0 and P_1 begin with "fresh" shares of input value (over $\mathbb{Z}_L, L = 2^{64}$) and output a "fresh" share of the output value (again over $\mathbb{Z}_L$) at the end of the protocol. Party P_2 takes the role of "assistant" in all protocols and has no input or output.

3.2.6 Linear and Convolutional Layer

We note that a linear (or fully connected) layer in an NN is exactly a matrix multiplication. Similarly, a convolutional layer can also be expressed as a (larger) matrix multiplication. As an example the 2-dimensional convolution of a 3×3 input matrix X with a kernel K of size 2×2 can be represented by the matrix multiplication shown below.

$$\mathsf{Conv2d}\left(\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}, \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix}\right) = \begin{bmatrix} x_1 & x_2 & x_4 & x_5 \\ x_2 & x_3 & x_5 & x_6 \\ x_4 & x_5 & x_7 & x_8 \\ x_5 & x_6 & x_8 & x_9 \end{bmatrix} \times \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix}$$

For a generalization, see e.g. [104] for an exposition on convolutional layers. Hence both these layers can be directly implemented using Algorithm 1 from Section 3.2.1.

3.2.7 Derivative of ReLU

Algorithm 6 describes our 3-party protocol for realizing the functionality $\mathcal{F}_{\mathsf{DReLU}}$ that computes the derivative of ReLU, denoted by ReLU'. Parties P_0, P_1 hold secret shares of a over ring $\mathbb{Z}_L$ and end up with secret shares of $\mathsf{ReLU}'(a)$ over $\mathbb{Z}_L$. Note that $\mathsf{ReLU}'(a) = 1$ if $\mathsf{MSB}(a) = 0$, else $\mathsf{ReLU}'(a) = 0$.

Algorithm 6 ReLU', $\Pi_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\langle a \rangle_0^L$ and $\langle a \rangle_1^L$, respectively.
Output: P_0, P_1 get $\langle \mathsf{ReLU}'(a) \rangle_0^L$ and $\langle \mathsf{ReLU}'(a) \rangle_1^L$.
Common Randomness: P_0, P_1 hold random shares of 0 over $\mathbb{Z}_L$, denoted by u_0 and u_1 resp.

1: For $j \in \{0, 1\}$, parties P_j computes $\langle c \rangle_j^L = 2\langle a \rangle_j^L$.
2: P_0, P_1, P_2 run $\Pi_{\mathsf{SC}}(\{P_0, P_1\}, P_2)$ with P_0, P_1 having inputs $\langle c \rangle_j^L$ & $\langle c \rangle_1^L$ & P_0, P_1 learn $\langle y \rangle_0^{L-1}$ & $\langle y \rangle_1^{L-1}$, resp.
3: P_0, P_1, P_2 run $\Pi_{\mathsf{MSB}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0, 1\}$ having input $\langle y \rangle_j^{L-1}$ & P_0, P_1 learn $\langle \alpha \rangle_0^L$ & $\langle \alpha \rangle_1^L$, resp.
4: For $j \in \{0, 1\}$, P_j outputs $\langle \gamma \rangle_j^L = j - \langle \alpha \rangle_j^L + u_j$.

Intuition: As is clear from the function ReLU' itself, the protocol computes the shares of $\mathsf{MSB}(a)$ and flips it to compute $\mathsf{ReLU}'(a)$. Recall that the protocol Π_{MSB} expects shares of a over $\mathbb{Z}_{L-1}$. Hence, we need to convert shares over $\mathbb{Z}_L$ to fresh shares over $\mathbb{Z}_{L-1}$ of the same value. Recall that for correctness of the share convert protocol, we require that value is not equal to $L - 1$. This is ensured by first computing shares of $c = 2a$ and then calling Π_{SC}. We ensure[5] that $\mathsf{ReLU}'(a) = \mathsf{ReLU}'(c)$ by requiring that $a \in [0, 2^k) \cup (2^\ell - 2^k, 2^\ell - 1]$, where $k < \ell - 1$.

3.2.8 ReLU

Algorithm 7 describes our 3-party protocol for realizing the functionality $\mathcal{F}_{\mathsf{ReLU}}$ that computes $\mathsf{ReLU}(a)$. Parties P_0, P_1 hold secret shares of a over ring $\mathbb{Z}_L$ and end up with secret shares of $\mathsf{ReLU}(a)$ over $\mathbb{Z}_L$. Note that $\mathsf{ReLU}(a) = a$ if $\mathsf{MSB}(a) = 0$, else 0. That is, $\mathsf{ReLU}(a) = \mathsf{ReLU}'(a) \cdot a$.
Intuition: Our protocol implements the above relation by using one call each to Π_{DReLU} and Π_{MatMul}. Note that Π_{MatMul} is invoked for multiplying two matrices of dimension 1×1 (or just one integer multiplication).

3.2.9 Division

We discuss our 3-party protocol Algorithm 8 realizing the functionality $\mathcal{F}_{\mathsf{DIV}}$. Parties P_0, P_1 hold shares of x and y over $\mathbb{Z}_L$. At the end of the protocol, parties P_0, P_1 hold shares of $\lfloor x/y \rfloor$ over $\mathbb{Z}_L$ when $y \neq 0$.

[5] This essentially means that the absolute value of a is not very large, and in particular not larger than 2^k. This is not a limitation in any of the ML applications that we work with.

Algorithm 7 ReLU, $\Pi_{\mathsf{ReLU}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\langle a \rangle_0^L$ and $\langle a \rangle_1^L$, respectively.
Output: P_0, P_1 get $\langle \mathsf{ReLU}(a) \rangle_0^L$ and $\langle \mathsf{ReLU}(a) \rangle_1^L$.

Common Randomness: P_0, P_1 hold random shares of 0 over $\mathbb{Z}_L$, denoted by u_0 and u_1 resp.

1: P_0, P_1, P_2 run $\Pi_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0, 1\}$ having input $\langle a \rangle_j^L$ and P_0, P_1 learn $\langle \alpha \rangle_0^L$ and $\langle \alpha \rangle_1^L$, resp.
2: P_0, P_1, P_2 call $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0, 1\}$ having input $(\langle \alpha \rangle_j^L, \langle a \rangle_j^L)$ and P_0, P_1 learn $\langle c \rangle_0^L$ and $\langle c \rangle_1^L$, resp.
3: For $j \in \{0, 1\}$, P_j outputs $\langle c \rangle_j^L + u_j$.

Algorithm 8 Division: $\Pi_{\mathsf{DIV}}(\{P_0, P_1\}, P_2)$

Input: P_0, P_1 hold $(\langle x \rangle_0^L, \langle y \rangle_0^L)$ and $(\langle x \rangle_1^L, \langle y \rangle_1^L)$, resp.
Output: P_0, P_1 get $\langle x/y \rangle_0^L$ and $\langle x/y \rangle_1^L$.

Common Randomness: $P_j, j \in \{0, 1\}$ hold ℓ shares 0 over $\mathbb{Z}_L$ denoted by $w_{i,0}$ and $w_{i,1}$ for all $i \in [\ell]$ resp. They additionally also hold another share of 0 over $\mathbb{Z}_L$, denoted by s_0 and s_1.

1: Set $u_\ell = 0$ and for $j \in \{0, 1\}$, P_j holds $\langle u_\ell \rangle_j^L$ (through the common randomness).
2: **for** $i = \{\ell - 1, \ldots, 0\}$ **do**
3: $P_j, j \in \{0, 1\}$ compute $\langle z_i \rangle_j^L = \langle x \rangle_j^L - \langle u_{i+1} \rangle_j^L - 2^i \langle y \rangle_j^L + w_{i,j}$.
4: P_0, P_1, P_2 run $\Pi_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0, 1\}$ having input $\langle z_i \rangle_j^L$ and P_0, P_1 learn $\langle \beta_i \rangle_0^L$ and $\langle \beta_i \rangle_1^L$, resp.
5: P_0, P_1, P_2 call $\mathcal{F}_{\mathsf{MATMUL}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0, 1\}$ having input $(\langle \beta_i \rangle_j^L, \langle 2^i y \rangle_j^L)$ and P_0, P_1 learn $\langle v_i \rangle_0^L$ and $\langle v_i \rangle_1^L$, resp.
6: $P_j, j \in \{0, 1\}$ compute $\langle k_i \rangle_j^L = 2^i \cdot \langle \beta_i \rangle_j^L$.
7: For $j \in \{0, 1\}$, P_j computes $\langle u_i \rangle_j^L = \langle u_{i+1} \rangle_j^L + \langle v_i \rangle_j^L$.
8: **end for**
9: For $j \in \{0, 1\}$, P_j outputs $\langle q \rangle_j^L = \sum_{i=0}^{\ell-1} \langle k_i \rangle_j^L + s_j$.

Intuition: Our protocol implements long division where the quotient is computed bit-by-bit sequentially starting from the most significant bit. In each iteration, we compute the current dividend by subtracting the correct multiple of the divisor. Then we compare the current dividend with a multiple of the divisor ($2^i y$ in round i). Depending on the output of the comparison, i^{th} bit of the quotient is 0 or 1. This comparison can be written as a comparison with 0 and hence can be computed using a single call to Π_{DReLU}. We use this selection bit to select between 0 and 2^i for quotient and 0 and $2^i y$ for what to subtract from dividend. This selection can be implemented using Π_{MatMul} (similar to ReLU computation). Hence, division protocol proceeds in iterations and each iteration makes one call to Π_{DReLU} and one call[6] to Π_{MatMul}.

3.2.10 Maxpool

Algorithm 9 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\mathsf{MAXPOOL}}$ to compute the maximum of n values. Parties P_0, P_1 hold shares of $\{x_i\}_{i\in[n]}$ over $\mathbb{Z}_L$ and end up with fresh shares of $\mathsf{max}(\{x_i\}_{i\in[n]})$.
Intuition: The protocol implements the max algorithm that runs in $(n-1)$ sequential steps. We start with $\mathsf{max}_1 = x_1$. In step i, we compute the shares of $\mathsf{max}_i = \mathsf{max}(x_1, \ldots, x_i)$ as follows: We compute shares of $w_i = x_i - \mathsf{max}_{i-1}$. Then, we compute shares of $\beta_i = \mathsf{ReLU}'(w_i)$ that is 1 if $x_i \geq \mathsf{max}_{i-1}$ and 0 otherwise. Next, we use Π_{SS} to select between max_{i-1} and x_i using β_i to compute max_i. Note, that in a similar manner, we can also calculate the index of maximum value, i.e., k such that $x_k = \mathsf{max}(\{x_i\}_{i\in[n]})$. This is done in steps 6&7. Computing the index of max value is required while doing prediction as well as to compute the derivative of Maxpool activation function needed for back-propagation during training.

3.2.11 Derivative of Maxpool

The derivative of the Maxpool function (functionality $\mathcal{F}_{\mathsf{DMAXPOOL}}$) is defined as the unit vector with a 1 only in the position with the maximum value. Here, we describe the more efficient Algorithm 10 that works for the special (and often-used) case of 2×2 Maxpool, where $n = 4$. In general, this algorithm works when n divides L. For the more general case, we provide an algorithm in Appendix C.1.
Intuition: The key observation behind this protocol is that when n divides L (i.e., $n \mid L$), we have that $a \bmod n = (a \bmod L) \bmod n$. The first step that P_0 and P_1 run is Π_{MP} that gives them shares of the index $\mathsf{ind} \in [n]$ with the maximum value. These shares are over L and must be converted into shares of the unit vector E_{ind} which is a length n vector with 1 in position ind and 0 everywhere else. P_0 and P_1 share a random $r \in \mathbb{Z}_n$ and have P_2 reconstruct $k = (\mathsf{ind} + r) \bmod n$. P_2 then creates shares of E_k and sends the shares back to P_0 and P_1 who "left-shift" these shares by r to obtain shares of E_{ind}. This works because $a \bmod n = (a \bmod L) \bmod n$ is true when $n \mid L$.

[6]Note that multiplication with 2^i can be done locally.

Algorithm 9 Maxpool $\Pi_{\mathsf{MP}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\{\langle x_i \rangle_0^L\}_{i\in[n]}$ and $\{\langle x_i \rangle_1^L\}_{i\in[n]}$, resp.
Output: P_0, P_1 get $\langle z \rangle_0^L$ and $\langle z \rangle_1^L$, resp., where $z = \mathsf{Max}(\{x_i\}_{i\in[n]})$.
Common Randomness: P_0 and P_1 hold two shares of 0 over $\mathbb{Z}_L$ denoted by u_0 and u_1 and v_0 and v_1.

1: For $j \in \{0,1\}$, P_j sets $\langle \mathsf{max}_1 \rangle_j^L = \langle x_1 \rangle_j^L$ and $\langle \mathsf{ind}_1 \rangle_j^L = j$.
2: **for** $i = \{2, \ldots, n\}$ **do**
3: For $j \in \{0,1\}$, P_j computes $\langle w_i \rangle_j^L = \langle x_i \rangle_j^L - \langle \mathsf{max}_{i-1} \rangle_j^L$
4: P_0, P_1, P_2 call $\Pi_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $\langle w_i \rangle_j^L$ and P_0, P_1 learn $\langle \beta_i \rangle_0^L$ and $\langle \beta_i \rangle_1^L$, resp.
5: P_0, P_1, P_2 call $\Pi_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \beta_i \rangle_j^L, \langle \mathsf{max}_{i-1} \rangle_j^L, \langle x_i \rangle_j^L)$ and P_0, P_1 learn $\langle \mathsf{max}_i \rangle_0^L$ and $\langle \mathsf{max}_i \rangle_1^L$, resp.
6: For $j \in \{0,1\}$, P_j sets $\langle k_i \rangle_j^L = j \cdot i$.
7: P_0, P_1, P_2 call $\Pi_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \beta_i \rangle_j^L, \langle \mathsf{ind}_{i-1} \rangle_j^L, \langle k_i \rangle_j^L)$ and P_0, P_1 learn $\langle \mathsf{ind}_i \rangle_0^L$ and $\langle \mathsf{ind}_i \rangle_1^L$, resp.
8: **end for**
9: For $j \in \{0,1\}$, P_j outputs $(\langle \mathsf{max}_n \rangle_j^L + u_j, \langle \mathsf{ind}_n \rangle_j^L + v_j)$.

Algorithm 10 Efficient Derivative of $n_1 \times n_2$ Maxpool $\Pi_{\mathsf{n_1xn_2DMP}}(\{P_0, P_1\}, P_2)$ with $n \mid L$ and $n = n_1 n_2$:

Input: P_0, P_1 hold $\{\langle x_i \rangle_0^L\}_{i\in[n]}$ and $\{\langle x_i \rangle_1^L\}_{i\in[n]}$, resp.
Output: P_0, P_1 get $\{\langle z_i \rangle_0^L\}_{i\in[n]}$ and $\{\langle z_i \rangle_1^L\}_{i\in[n]}$, resp., where $z_i = 1$, when $x_i = \mathsf{Max}(\{x_i\}_{i\in[n]})$ and 0 otherwise.
Common Randomness: P_0 and P_1 hold shares of 0 over $\mathbb{Z}_L^n$ denoted by U_0 and U_1 and a random $r \in \mathbb{Z}_L$.

1: P_0, P_1, P_2 call $\mathcal{F}_{\mathsf{MAXPOOL}}$ with $P_j, j \in \{0,1\}$ having input $\{\langle x_i \rangle_j^L\}_{i\in[n]}$, to obtain $\langle \mathsf{ind}_n \rangle_j^L$ resp. (from the second part of $\mathcal{F}_{\mathsf{MAXPOOL}}$'s output).
2: P_0 sends $\langle k \rangle_0^L = \langle \mathsf{ind}_n \rangle_0^L + r$ to P_2, while P_1 sends $\langle k \rangle_1^L = \langle \mathsf{ind}_n \rangle_1^L$ to P_2.
3: P_2 computes $t = \mathsf{Reconst}^L(\langle k \rangle_0^L, \langle k \rangle_1^L)$, computes $k = t \bmod n$ and creates shares of E_k, denoted by $\langle E \rangle_0^L$ and $\langle E \rangle_1^L$, and sends the shares to P_0 and P_1 resp.
4: P_0 and P_1 locally "cyclic-shift" their shares by $g = r \bmod n$. That is, let $\langle E \rangle_j^L = (\langle E_0 \rangle_j^L, \langle E_1 \rangle_j^L, \cdots, \langle E_{n-1} \rangle_j^L)$. P_j computes $\langle D \rangle_j^L$ as $(\langle E_{(-g \bmod n)} \rangle_j^L, \langle E_{(1-g \bmod n)} \rangle_j^L, \cdots, \langle E_{(n-1-g \bmod n)} \rangle_j^L)$.
5: $P_j, j \in \{0,1\}$ outputs $\langle D \rangle_j^L + U_j$.

3.2.12 End-to-end Protocols

Our main protocols can be easily put together to execute training on a wide class of NNs. For example, consider Network-A, 3-layer NN from SecureML that consists of a fully-connected layer, followed by a ReLU, followed by another fully-connected layer, followed by another ReLU, followed by the function $\mathsf{ASM}(u_i) = \frac{\mathsf{ReLU}(u_i)}{\sum \mathsf{ReLU}(u_i)}$ (for further details on this network, we refer the reader to [85]). To implement this, we first invoke Π_{MatMul}, followed by Π_{ReLU}, then again followed by Π_{MatMul} and Π_{ReLU} and finally we invoke Π_{DIV} to compute $\mathsf{ASM}(\cdot)$[7]. Back-propagation is computed by making calls to Π_{MatMul} as well and Π_{DReLU} with appropriate dimensions[8]. Similarly, we can also do a general CNN with other activations such as Maxpool, AvgPool. We remark that we can put together these protocols easily since our protocols all maintain the invariant that parties begin with arithmetic shares of inputs and complete the protocol with arithmetic shares of the output.

3.3 Theoretical Evaluation

In this section, we describe the theoretical overheads of the building block protocols as well as the main protocols. We defer the proofs of the protocols to Appendix A.

3.3.1 Overheads of Supporting Protocols

The communication and round complexity of our supporting protocols is provided in Table 3.1. $\mathsf{MatMul}_{m,n,v}$ denotes matrix multiplication of an $m \times n$ matrix with an $n \times v$ matrix. The first row states the complexity of $\mathsf{MatMul}_{m,n,v}$ using secure Beaver triplets. In our implementation, we generate the triplets using PRFs as follows: P_0 and P_2 share a PRF key and use it to generate $\langle A\rangle_0^L, \langle B\rangle_0^L, \langle C\rangle_0^L$ locally. Similarly, P_1 and P_2 share a PRF key and use it to generate $\langle A\rangle_1^L, \langle B\rangle_1^L$ locally. Now, P_2 sets $\langle C\rangle_1^L = \mathsf{Reconst}^L(\langle A\rangle_0^L, \langle A\rangle_1^L) \cdot \mathsf{Reconst}^L(\langle B\rangle_0^L, \langle B\rangle_1^L) - \langle C\rangle_0^L$ and send to P_1. This reduces the communication of multiplication by half. All other complexities are for single elements and use this optimized version of multiplication.

3.3.2 Communication and Rounds

The round and communication complexity of our main protocols are presented in Table 3.2. The function $\mathsf{Linear}_{m,n,v}$ denotes a matrix multiplication of dimension $m \times n$ with $n \times v$. $\mathsf{Conv2d}_{m,i,f,o}$ denotes a convolutional layer with input $m \times m$, i input channels, a filter of size $f \times f$, and o output channels. l_D denotes precision of bits. $\mathsf{Maxpool}_n$ and DMP_n denotes Maxpool and its derivative over n elements. For ReLU and DMP_n, the overheads in addition to DReLU and $\mathsf{Maxpool}_n$ respectively are presented as these protocols are always implemented together in an NN. All

[7] $\mathsf{ASM}(\cdot)$ consists of a summation and a division. Summation is a local computation and does not require a protocol to be computed.

[8] We note that Π_{DReLU} is called as part of Π_{ReLU} in forward propagation and its value is stored for back-propagation

Table 3.1: Round & communication complexity of *building blocks*.

Protocol	Rounds	Communication
$\mathsf{MatMul}_{m,n,v}$	2	$2(2mn + 2nv + mv)\ell$
$\mathsf{MatMul}_{m,n,v}$ (with PRF)	2	$(2mn + 2nv + mv)\ell$
SelectShare	2	5ℓ
PrivateCompare	1	$2\ell \log p$
ShareConvert	4	$4\ell \log p + 6\ell$
Compute MSB	5	$4\ell \log p + 13\ell$

Table 3.2: Round & communication complexity of *main protocols*.

Protocol	Rounds	Communication
$\mathsf{Linear}_{m,n,v}$	2	$(2mn + 2nv + mv)\ell$
$\mathsf{Conv2d}_{m,i,f,o}$	2	$(2m^2f^2i + 2f^2oi + m^2o)\ell$
DReLU	8	$8\ell \log p + 19\ell$
ReLU (after DReLU)	2	5ℓ
NORM(l_D) or DIV(l_D)	$10l_D$	$(8\ell \log p + 24\ell)l_D$
$\mathsf{Maxpool}_n$	$9(n-1)$	$(8\ell \log p + 29\ell)(n-1)$
DMP_n (after Maxpool)	2	$2(n+1)\ell$

communication is measured for ℓ-bit inputs and p denotes the field size (which is 67 in our case). All of the complexities are presented using the optimized complexity of multiplication that used PRFs for correlated randomness.

Our gains mainly come from the secure evaluation of non-linear functions such as ReLU and Maxpool and their derivatives. Prior works such as SecureML [85], MiniONN [77], Gazelle [64], etc., took a garbled circuit-based approach to evaluate these functions, i.e., after completion of an arithmetic (linear) computation such as matrix multiplication, they ran a protocol to convert shares of intermediary values into an encoding suitable for garbled circuits. The non-linear function was then evaluated using the garbled circuit after which shares were once again converted back to be suitable for arithmetic computation. This approach leads to a multiplicative factor communication overhead proportional to the security parameter κ, as garbled circuits require communicating encodings proportional to κ, for every bit in the circuit. Overall, this leads to a communication complexity $> 768\ell$ for every ℓ-bit input [41]. As shown in [41], this cost of conversion to garbled circuits is $6\kappa\ell$, and all previous works incur this cost. In our approach, we provide new protocols to compute such non-linear activation functions, while continuing to retain arithmetic shares of the output values. For example, the ReLU protocol that we construct avoids paying κ multiplicative overhead and has communication complexity of $8\ell \log p + 24\ell$, which is approximately 88ℓ (when $p = 67$ as is in our setting). This leads to $> 8\times$ improvement in the communication complexity of the protocols for non-linear functions.

3.4 Experimental Evaluation

SecureNN is implemented in about 7400 lines of `C++` code with the use of standard libraries. We use the Eigen Library [44] for faster matrix multiplications. The ring is set to $\mathbb{Z}_{2^{64}}$ and we use the `uint64_t` native `C++` datatype for all variables. The source code is available at https://www.github.com/snwagh/securenn-public.git and potentially of interest is also the Falcon linked in the following chapter.

3.4.1 System Details

We test our prototype by running experiments over Amazon EC2 c4.8x large instances in two environments, respectively modeling a LAN and WAN setting.

- **LAN setting.** We use 3 Amazon EC2 c4.8xlarge machines running Ubuntu in the same region. The average bandwidth measured was 625MB/s and the average ping time was 0.22ms.
- **WAN setting.** In the WAN setting, we rent machines in different geographical regions with the same machine specifications as in the LAN setting. The average bandwidth measured was 40MB/s and the average ping time was 58ms.

Number encoding. Typical NNs work over floating-point numbers. As observed by all prior works, to make them compatible with efficient cryptographic techniques, they must be encoded into fixed-point form. We use the methodology from [85] to support fixed-point arithmetic in an integer ring (described in Appendix A.1). The fixed-point numbers have 13 bits in their fractional part (cleartext training to get accuracy numbers is also done with these parameters).

3.4.2 Summary of Experiments

We develop a prototype of SecureNN. We test the performance of our protocols by training 3 different NNs over the MNIST dataset [83]. We also evaluate SecureNN on secure inference benchmarks in Section 3.4.5. Finally, in Section 3.4.6, we present microbenchmarks that measure the performance of various sub-protocols implemented in SecureNN such as Linear Layer, Convolutional Layer, ReLU, and Maxpool (and its derivatives) that enables the estimation of the performance cost of other networks using the above functions.

For secure training, we run multiple iterations (10) and take the average - for each iteration, we measured the time for 10 forward-backward passes and used that to extrapolate the numbers for 15 epochs (7000 iterations). Secure inference timings are also averaged over 10 iterations. The learning rate is 2^{-5} in all experiments, except in the SecureML [85] network, where we retain their learning rate of 2^{-7}. In all our experiments, we report overall execution time (and do not split execution time into an offline, data independent phase, and an online, data dependent phase) and treat the same as online time as well. Our experiments show that our total execution times are better than even just the online times of previous works. If we split our work (e.g.,

Table 3.3: Secure training execution times for batch size 128.

	Epochs	Accuracy	LAN (hours)	WAN (hours)
Network-A	15	93.4%	1.03	7.83
Network-B	5	97.94%	5.8	17.99
	10	98.05%	11.6	35.99
	15	98.77%	17.4	53.98
Network-C	5	98.15%	9.98	30.66
	10	98.43%	19.96	61.33
	15	99.15%	29.95	91.99

triplet generation for multiplication) to an offline phase, our online improvements would be even better.

3.4.3 Neural Networks

For benchmarking and comparison, we consider four NN architectures performing training/inference over the MNIST dataset [83] for hand-written digit recognition[9]. Network-A is a 3-layer DNN from [85], Network-B is a 4-layer CNN from [77, 84], Network-C is a 4-layer CNN from [76], and Network-D is a 3-layer CNN from [95, 84]. Note that Network-B, Network-C, Network-D (here) correspond to Network-C, LeNet, and Network-B respectively in Section 2.5.

3.4.4 Secure Training

We evaluate our protocols for secure training in both the LAN and WAN settings over the Networks-A, B, and C listed above. In many cases, the networks we train, achieve more than 99% accuracy for inference (on test dataset). We remark that we are the first work to show the feasibility of secure training on large and complex NNs such as CNNs that achieve high levels of accuracy. We vary the epochs between 5 and 15 for all networks except Network-A which does not achieve good accuracy for smaller epochs and vary the batch size between 4 and 128 for Networks-B and C. Table 3.3 presents a summary of our results in the LAN/WAN setting as a function of the number of epochs for training (batch size fixed to 128), while Table 3.4 presents the results when the batch size is varied and the number of epochs is fixed to 5.

Comparison with prior work. The only prior work to consider NN training was SecureML [85] that considers Network-A only. They give implementations for both 2- and 3-server settings on similar hardware and network settings – we quote experimental numbers from their paper. We provide a comparison of our protocols with their work in Table 3.7. In the LAN setting, our protocol is roughly 6.8× faster than

[9]This dataset has 60,000 training samples of handwritten digits. Each image is a 28-by-28 pixel square, with each pixel represented using 1 byte. The inference set contains 10,000 images.

Table 3.4: Secure training execution times for 5 epochs.

	Batch size	Accuracy	LAN (hours)	WAN (hours)
Network-B	4	99.15%	9.98	112.71
	16	98.99%	8.34	36.46
	128	97.94%	5.8	17.99
Network-C	4	99.01%	18.31	123.96
	16	99.1%	13.43	46.2
	128	98.15%	9.98	30.66

Table 3.5: Single image inference time comparison of various protocols in the LAN setting.

	Framework	Run-time (s)			Communication (MB)		
		Offline	Online	Total	Offline	Online	Total
Network-A	SecureML	4.7	0.18	4.88	-	-	-
	SecureNN	0	0.043	0.043	0	2.1	2.1
Network-B	MiniONN	3.58	5.74	9.32	20.9	636.6	657.5
	Gazelle	0.481	0.33	0.81	47.5	22.5	70.0
	SecureNN	0	0.13	0.13	0	8.86	8.86
Network-C	SecureNN	0	0.23	0.23	0	18.94	18.94
Network-D	DeepSecure	-	-	9.67	-	-	791
	Chameleon 3PC	1.34	1.36	2.7	7.8	5.1	12.9
	Gazelle	0.15	0.05	0.20	5.9	2.1	8.0
	SecureNN	0	0.076	0.076	0	4.05	4.05

their 3-party protocol and 79× faster than their 2-party protocol. In the WAN setting, our improvements are even more dramatic and we get an improvement of 553× over the 2-party protocol[10]. Furthermore, SecureML split their times into a slow (data-independent) offline phase and a faster (data-dependent) online phase. Even comparing only their online time with our overall 3PC time, we obtain an improvement of 1.16× over their 2PC and a 2.7× improvement over their 3PC (their 3PC trades off some offline cost with a larger online cost).

3.4.5 Secure Inference

We also evaluate our protocols for the task of secure inference for Networks-A, B, C, and D. These networks can either be a result of secure training using the 3PC protocol and are secret shared between P_0 and P_1, or a trained model can be secret shared between P_0 and P_1 at the beginning of the protocol.

Comparison with prior work. A sequence of previous works have considered a single secure inference in the LAN setting for various networks. Table 3.5 summarizes our comparison with state-of-the-art secure inference protocols. Networks-A and B were considered in SecureML [85], MiniONN [77], and Gazelle [64] using different

[10]Authors of SecureML do not provide numbers for their 3-party protocol in the WAN setting.

Table 3.6: Prediction timings for batch size 1 vs 128 for SecureNN on Networks A-D over MNIST.

	LAN (s)		WAN (s)		Comm (MB)	
Batch size →	1	128	1	128	1	128
Network-A	0.043	0.38	2.43	2.79	2.1	29
Network-B	0.13	7.18	3.93	21.99	8.86	1066
Network-C	0.23	10.82	4.08	30.45	18.94	1550
Network-D	0.076	2.6	3.06	8.04	4.05	317.7

techniques for secure computation. All these works used similar hardware and network settings as our LAN experiments and we quote experimental numbers from the respective papers.

Each of these works splits its computation into an input-independent offline phase and an input-dependent online phase. In our protocols, we do not do this split and count all cost as online cost – hence, the offline cost is 0. Our protocols in the 3PC setting achieve roughly 3× improvement in small networks that have a small number of non-linear operations (such as Network-D) and between 6×-113× improvements in some larger networks. In fact, in most cases, especially for realistic size networks, our total time is lower than the online time of previous best protocols (ignoring the offline time). We are the first to evaluate on Network-C (which is considerably larger in size) and the table shows our run-time and communication. Finally, for Network-D, we also compare our protocols with the 3PC protocols in Chameleon [95]. This shows SecureNN improves on prior work by about 35×.

In all cases, our performance gains can be attributed to much better communication complexity of our protocols compared to previous works (see comparison in Table 3.5). In particular, as mentioned before, we avoid the use of garbled circuits for the non-linear activation functions such as ReLU. In all previous works, garbled circuits are the major factor in large communication.

Single vs. Batch Prediction. Table 3.6 summarizes our results for secure inference over different networks for 1 prediction and batch of 128 predictions in both the LAN and WAN settings. Due to the use of matrix-based Beaver triplets for secure multiplication protocol in linear and convolutional layers, and batching of communication, the time for multiple predictions grows sub-linearly. SecureML also did predictions for batch size 100 for Network-A and took 14s and 143s in the LAN and the WAN settings, respectively. In contrast, we take only 0.38s and 2.79s for 128 predictions using the 3PC protocol.

Comparison with ABY^3 [84]. ABY^3 considers a similar set-up as SecureNN but develops different techniques for matrix multiplication and non-linear operations. This results in protocols with different communication complexity with performance depending on the network architecture and hardware. For instance, ABY^3 requires 0.5MB of communication for inference on Network-A (SecureNN requires 2.1MB) while it requires 5.2MB of communication over Network-D (SecureNN requires 4.05MB).

Table 3.7: Training time comparison for Network A for batch size 128 and 15 epochs with SecureML [85].

	Framework	LAN (hr)			WAN (hr)		
		Offline	Online	Total	Offline	Online	Total
Network-A	SecureML 2PC	80.5	1.2	81.7	4277	59	4336
	SecureML 3PC	4.15	2.87	7.02	-	-	-
	SecureNN	0	1.03	1.03	0	7.83	7.83

3.4.6 Microbenchmarks

Table 3.8 presents microbenchmark timings for popular functions used in ML algorithms for various sizes. All timings are averaged over 10 iterations. The overheads for DMP and ReLU are additional over the costs of Maxpool and DReLU respectively as these pairs of protocols are always used together in training.

Table 3.8: Microbenchmarks in the LAN & WAN settings.

Protocol	Dimension	LAN (ms)	WAN (ms)	Comm. (MB)
$\mathsf{Conv2d}_{m,f,i,o}$	$8, 5, 16, 50$	3.8	28.4	0.42
	$28, 3, 1, 20$	1.8	26.5	0.2
	$28, 5, 1, 20$	2.8	27.5	0.33
$\mathsf{MatMul}_{m,n,v}$	$1, 100, 1$	0.33	25.2	0.0032
	$1, 500, 100$	4.8	29.4	0.81
	$784, 128, 10$	9.7	34.3	1.69
Maxpool	$8 \times 8 \times 50, 4 \times 4$	59.7	3062.2	2.23
	$24 \times 24 \times 16, 2 \times 2$	61.1	672.6	5.14
	$24 \times 24 \times 20, 2 \times 2$	62.6	685	6.43
DMP	$8 \times 8 \times 50, 4 \times 4$	1.9	51.6	0.18
	$24 \times 24 \times 16, 2 \times 2$	4.8	54.2	0.52
	$24 \times 24 \times 20, 2 \times 2$	4.9	55.2	0.65
DReLu	64×16	11.2	161.9	0.68
	128×128	109.8	288.7	10.88
	576×20	71.5	232.9	7.65
ReLU	64×16	0.42	25.3	0.04
	128×128	2.8	27.1	0.66
	576×20	2.5	26.6	0.46

3.5 Summary

We develop new 3-party secure computation protocols for a variety of NN training and prediction algorithms such that no single party learns any information about the data. Our work makes three fundamental contributions: first, it is the first work to enable secure NN training of large networks such as CNNs that have accuracy of $> 99\%$ over

the MNIST dataset. Second, by designing communication-efficient protocols for non-linear functions, we obtain *several orders of magnitude improvements* over prior works. Finally, our protocols provide both full semi-honest security as well as privacy against malicious adversaries, unlike prior works that only provided semi-honest security.

3.6 Selected References

[85] Payman Mohassel and Yupeng Zhang. "SecureML: A system for scalable privacy-preserving machine learning." In: *IEEE Symposium on Security and Privacy (S&P)*. 2017

[77] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. "Oblivious neural network predictions via MiniONN transformations." In: *ACM Conference on Computer and Communications Security (CCS)*. 2017

[64] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "Gazelle: A low latency framework for secure neural network inference." In: *USENIX Security Symposium (USENIX)*. 2018

[96] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. "DeepSecure: Scalable provably-secure deep learning." In: *Annual Design Automation Conference*. 2018

[95] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. "Chameleon: A hybrid secure computation framework for machine learning applications." In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2018

Chapter 4

Falcon: Scaling-up Privacy-Preserving Machine Learning

This chapter focuses on training and inference of NNs in a manner that protects the privacy of sensitive data[1]. We propose Falcon – an end-to-end 3-party protocol for fast and secure computation of deep learning algorithms on large networks. Falcon presents three main advantages:

(A) Malicious Security: Falcon provides strong security guarantees in an honest-majority adversarial setting. This assumption is similar to prior work where majority of the parties (e.g., 2 out of three) behave honestly [84, 50]. Falcon proposes new protocols that are secure against such corruptions and ensure that either the computation always correctly completes or aborts detecting malicious activity. We achieve this by designing new protocols for the computation of non-linear functions (like ReLU). While MPC protocols are very efficient at computing linear functions, computing non-linear functions like ReLU is much more challenging. We propose solutions both for the malicious security model and provide even more efficient protocols where semi-honest security is sufficient. We formally prove the security of Falcon using the standard simulation paradigm (see Section C.1). We implement both the semi-honest and malicious protocols in our end-to-end framework. In this manner, Falcon provides a choice to the developers to select between either of the security guarantees depending on the trust assumption among the parties and performance requirements (improved performance for semi-honest protocols).

(B) Improved Protocols: Falcon combines techniques from SecureNN [109] and ABY^3 [84] that result in improved protocol efficiency. We improve the theoretical complexity of the central building block – derivative of ReLU – by a factor of 2× through simplified algebra for fixed-point arithmetic. We demonstrate

[1]For additional experiments and details, refer to the full version found at https://snwagh.github.io.

our protocols in a smaller ring size, which is possible using an exact yet expensive truncation algorithm. However, this enables the entire framework to use a smaller datatype, thus reducing their communication complexity by at least 2×. This reduced communication is critical to the communication improvements of Falcon over prior work. Furthermore, as can be seen in Section 4.4, these theoretical improvements lead to even larger practical improvements due to the recursive dependence of the complex functionalities on the improved building blocks. Overall, we demonstrate how to achieve maliciously secure protocols for non-linear operations entirely using arithmetic secret sharing and avoiding the use of inter-conversion protocols (between arithmetic, Boolean, and garbled circuits).

(C) Expressiveness: The focus of this work is to provide simple yet efficient protocols for the fundamental functionalities commonly used in state-of-the-art NNs. Batch normalization, has been previously considered in privacy-preserving inference as linear transformation using HE [24, 61, 32]. However, batch normalization is critical for stable convergence of networks as well as to reduce the parameter tuning required during training of neural networks. Falcon is the first work to demonstrate full support for Batch-Normalization layers, both for forward and backward pass, in private machine learning. This extensive support makes Falcon expressive, thereby supporting evaluation of large networks with hundreds of millions parameters such as VGG16 [102] and AlexNet [69] over datasets such as MNIST [83], CIFAR-10 [68] as well as Tiny ImageNet [114] including in both the LAN and WAN network settings. Designing secure protocols for training is more difficult due to the operations involved in back-propagation which are not required for inference. A number of prior works assume training in a trusted environment and hence provide support for only inference service [94, 77, 95, 25, 64]. However, sensitive data is often inaccessible even during training as described in our motivating application in Section 1.4.

Compared to prior art for private inference, Falcon is about 8× faster than SecureNN (PETS '19) on average and comparable to ABY^3 (CCS '18); Falcon is about 16×-200× more communication-efficient than either of these. For private training, Falcon is about 6× faster than SecureNN, 4.4× faster than ABY^3 and about 2×-60× more communication-efficient. This is also the first work to show via experiments in the WAN setting that for multi-party machine learning computations over large networks and datasets, *compute operations* dominate the overall latency, as opposed to the communication.

4.1 Falcon Overview

In this section, we describe a general application setting for Falcon and provide an executive summary of the technical contributions. We consider the following scenario: There are two types of users, the first own data on which the learning algorithm will

be applied, we call them data holders. The second are users who query the system after the learning period, we call these query users. These two sets of users need not be disjoint. We design a machine learning service. This service is provided by 3 parties which we call computing servers. We assume that government regulations or other social deterrents are sufficient enforcers for non-collusion between these computing servers. The service works in two phases: the training phase where the machine learning model of interest is trained on the data of the data holders and the inference phase where the trained model can be queried by the query users. The data holders share their data in a replicated secret sharing form [6] between the 3 computing servers. These 3 servers utilize the shared data and privately train the network. After this stage, query users can submit queries to the system and receive answers based on the newly constructed model held in shared form by the three servers. This way, the data holders' input has complete privacy from each of the 3 servers. Moreover, the query is also submitted in shared form and thus is kept secret from the 3 servers. Refer to Section 1.4 for concrete use cases.

4.1.1 Threat Model, Assumptions, & Scope

Our threat model assumes an honest majority among the three parties in the setting described above. This is similar to the SecureNN adversarial model – however, this work also provides correctness in the presence of malicious adversaries. This is a common adversarial setting considered in previous secure multi-party computation approaches [84, 85, 6, 50]. Each of the 3 parties has shared point-to-point communication channels and pairwise shared seeds to use AES as a pseudo-random number generator (PRNG) to generate cryptographically secure common randomness. We note that as the users receive the answers to the queries in the clear, Falcon does not guarantee protecting the privacy of the training data from attacks such as model inversion, membership inference, and attribute inference [101, 49, 107]. Defending against these attacks is an orthogonal problem and hence we consider it out-of-scope for this work. We assume that users provide consistent shares and that model poisoning attacks are out of scope.

4.1.2 Technical Contributions

In this section, we summarize some of the main contributions of this work with a focus on techniques used to achieve our results and improvements.

Hybrid Integration for Malicious Security. Falcon consists of a hybrid integration of ideas from SecureNN and ABY^3 along with newer protocol constructions for privacy-preserving deep learning. SecureNN, the closest related prior work, does not provide correctness in the presence of malicious adversaries. Furthermore, the use of semi-honest parties in SecureNN makes it a significant challenge to convert those protocols to provide security against malicious corruptions. To this end, we use replicated secret sharing as our building block and use the redundancy to enforce correct behavior in our protocols [6, 50, 84]. Note that changing from the 2-out-of-2

secret sharing scheme in SecureNN to a 2-out-of-3 replicated secret sharing fundamentally alters some of the building blocks – these protocols are a new contribution of this work. We work in the three-party setting where at most one party can be corrupt. We prove each building block secure in the Universal Composability (UC) framework by proving our protocols are (1) perfectly secure in the stand-alone model, i.e., the distributions are identical and not just statistically close in a model where the protocol is executed only once; and (2) have straight-line black-box simulators, i.e., only assume oracle access to the adversary and do no rewind. Theorem 1.2 from Kushilevitz *et al.* [72] then implies security under general concurrent composition.

Theoretical Improvements to Protocols. FALCON proposes more efficient protocols for common machine learning functionalities while providing stronger security guarantees. We achieve this through a number of theoretical improvements for reducing both the computation as well as the communication. First, in FALCON all parties execute the same protocol in contrast to SecureNN where the protocol is asymmetric. The uniformity of the parties leads to more optimal resource utilization. Second, the protocol for derivative of ReLU[2] in SecureNN [109] first transforms the inputs using a `Share Convert` subroutine (into secret shares modulo an odd ring) and then invokes a `Compute MSB` subroutine to compute the most significant bit (MSB) which is closely related to the DReLU function. Each of these subroutines has roughly the same overhead. In FALCON, we show an easier technique using new mathematical insights to compute DReLU which reduces the overhead by over $2\times$. Note that ReLU and DReLU, non-linear activation functions central to deep learning, are typically the expensive operations in MPC. The first two points above lead to strictly improved protocol for these. Third, FALCON uses a smaller ring size while using an exact yet expensive truncation protocol. This trade-off however allows the entire framework to operate on smaller data-types, thus reducing the communication complexity at least $2\times$. Furthermore, this communication improvement is amplified with the super-linear dependence of the overall communication on the ring size (cf Table 4.2). This reduced communication is critical to the communication improvements of FALCON over prior work. In other words, we notice strictly larger performance improvements (than the theoretical improvements) in our end-to-end deployments of benchmarked networks presented in Section 4.4.

Improved Scope of ML Algorithms. Prior works focus on implementing protocols for linear layers and important non-linear operations. We propose and implement an end-to-end protocol for batch normalization (both forward and backward pass). Batch-normalization is widely used in practice for speedy training of NNs and is critical for machine learning for two reasons. First, it speeds up training by allowing

[2]Note that DReLU, when using fixed-point encoding over a ring $\mathbb{Z}_L$ is defined as follows:

$$\text{DReLU}(x) = \begin{cases} 0 & \text{if } x > L/2 \\ 1 & \text{Otherwise} \end{cases} \tag{4.1}$$

higher learning rates and prevents extreme values of activations [62]. This is an important component of the parameter tuning for NNs as there is limited "seeing and learning" during private training. Second, it reduces over-fitting by providing a slight regularization effect and thus improves the stability of training [62]. In other words, private training of NNs without batch normalization is generally difficult and requires significant pre-training. To truly enable private deep learning, efficient protocols for batch normalization are required. Implementing batch normalization in MPC is hard for two reasons. First computing the inverse of a number is generally difficult in MPC. Second, most approximate approaches require the inputs to be within a certain range, i.e., there is a trade-off between having an approximate function for inverse of a number over a large range and the complexity of implementing it in MPC. Through our implementation, we enable batch normalization that can allow the training of complex network architectures such as AlexNet (about 60 Million parameters).

Comprehensive Evaluation. As shown in Table 4.1, there are a number of factors involved in comparing different MPC protocols and that none of the prior works provide a holistic solution. We also thoroughly benchmark our proposed system – we evaluate our approach over 6 different network architectures and over 3 standard datasets (MNIST, CIFAR-10, and Tiny ImageNet). We also benchmark our system in both the LAN and WAN settings, for training as well as for inference, and in both the semi-honest and actively secure adversarial models. Finally, we provide a thorough performance comparison against prior state-of-the-art works in the space of privacy-preserving machine learning (including 2PC). We believe that such a comparison, across a spectrum of deployment scenarios, is useful for the broader community of MPC practitioners.

Finally, we note that the insights and techniques developed in this work are broadly applicable. For instance, ReLU is essentially a comparison function which can thus enable a number of other applications – private computation of decision trees, privacy-preserving searching and thresholding, and private sorting.

4.2 Protocol Constructions

We begin by describing the notation used in this chapter. We then describe how basic operations are performed over the secret sharing scheme and then move on to describe our protocols in detail.

4.2.1 Notation

Let P_1, P_2, P_3 be the parties. We use P_{i+1}, P_{i-1} to denote the next and previous party for P_i (with periodic boundary conditions). In other words, next party for P_3 is P_1 and previous party for P_1 is P_3. We use $[\![x]\!]^m$ to denote 2-out-of-3 replicated secret sharing (RSS) modulo m for a general modulus m. For any x let $[\![x]\!]^m = (x_1, x_2, x_3)$ denote the RSS of a secret x modulo m, i.e., $x \equiv x_1 + x_2 + x_3 \pmod{m}$, but they are otherwise random. We use the notation $[\![x]\!]^m$ to mean (x_1, x_2) is held by P_1,

Table 4.1: Comparison of various private deep learning frameworks. FALCON proposes efficient protocols for non-linear functionalities such as ReLU and batch normalization (1) purely using modular arithmetic (2) under malicious corruptions (3) supporting both private training and inference. FALCON also provides a comprehensive evaluation (1) over larger networks and datasets (2) extensively compares with related work (3) provides newer insights for future directions of PPML. ● indicates the framework supports a feature, ○ indicates not supported feature, and ◐ refers to fair comparison difficult due to the following reasons: SecureNN provides malicious privacy but not correctness and supports division but not batch norm, XONN supports a simplified batch norm specific to a binary activation layer, ABY^3 does not present WAN results for neural networks, Chameleon evaluates over a network similar to AlexNet but using the simpler mean-pooling operations, and due to the high round complexity and communication, SecureML provides an estimate of their WAN evaluation, Delphi evaluates over network such as ResNet-32, CryptFlow evaluates networks such as DenseNet-121, ResNet-50, uses weaker network parameters in LAN and uses ImageNet dataset. QuantizedNN uses inherent quantization of the underlying NN and performs extensive evaluation over MobileNet architectures and * refers to 3PC version among the 8 protocols. BLAZE uses a Parkinson disease dataset, similar in dimension to MNIST. FLASH and Trident use few other smaller datasets in their evaluation as well as evaluate over increasing number of layers over the network architecture from [85].

	Framework	*Inference*	*Training*	*Semi-honest*	*Malicious*	*Linear*	*Convolution*	*Exact ReLU*	*Maxpool*	*Batch-Norm*	*HE*	*GC*	*SS*	*LAN*	*WAN*	*MNIST*	*CIFAR-10*	*Tiny ImageNet*	*From [85]*	*From [95]*	*From [77]*	*LeNet*	*AlexNet*	*VGG-16*
		Private Capability		Threat Model		Supported Layers					Techniques Used			LAN/ WAN		Evaluation Dataset			Network Architectures					
		Theoretical Metrics												Evaluation Metrics										
2PC	MiniONN [77]	●	○	●	○	●	●	●	●	○	●	●	●	●	○	●	●	○	●	○	●	○	○	○
	Chameleon [95]	●	○	●	○	●	●	●	●	○	○	●	●	●	●	●	●	○	●	●	●	○	◐	○
	EzPC [25]	●	○	●	○	●	●	●	●	○	○	●	●	●	●	●	●	○	●	○	●	○	○	○
	Gazelle [64]	●	○	●	○	●	●	●	●	○	●	●	●	●	○	●	●	○	●	●	●	○	○	○
	SecureML [85]	●	●	●	○	●	●	●	●	○	●	●	●	●	◐	●	○	○	●	○	○	○	○	○
	XONN [94]	●	○	●	●	●	●	●	●	◐	○	●	●	●	○	●	●	○	●	●	●	○	○	◐
	Delphi [82]	●	○	●	○	●	●	●	●	○	●	●	●	●	○	○	●	○	○	○	●	○	○	◐
3PC	ABY^3 [84]	●	●	●	●	●	●	●	●	○	○	●	●	●	◐	●	○	○	●	●	●	○	○	○
	SecureNN [109]	●	●	●	◐	●	●	●	●	◐	○	○	●	●	●	●	○	○	●	●	●	○	○	○
	CryptFlow [71]	●	○	●	●	●	●	●	●	○	○	○	●	◐	○	●	●	●	●	●	●	●	○	◐
	QuantizedNN [35]*	●	○	●	●	●	●	●	●	◐	◐	○	●	●	●	○	○	●	○	○	○	○	○	◐
	ASTRA [27]	●	●	●	◐	●	●	●	●	○	○	●	●	●	●	●	○	○	●	○	○	○	○	○
	BLAZE [88]	●	●	●	●	●	●	●	●	○	○	●	●	●	●	◐	○	○	●	○	○	○	○	○
	FALCON (This Work)	●	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	◐	●	●	●	●	●	●
4PC	FLASH [19]	●	●	●	●	●	●	●	●	○	○	○	●	●	●	●	◐	○	●	◐	○	○	○	○
	Trident [91]	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	○	○	●	○	○	○	○	○

(x_2, x_3) by P_2, and (x_3, x_1) by P_3. We denote by $x[i]$ the i^{th} component of a vector x. In this work, we focus on three different moduli $L = 2^\ell$, a small prime p, and 2. In particular, we use $\ell = 5$, $p = 37$. We use fixed-point encoding with 13 bits of precision. In Π_{Mult} over $\mathbb{Z}_p$, the multiplications are performed using the same procedure with no truncation. ReLU, which compares a value with 0, in this fixed-point representation, corresponds to a comparison with $2^{\ell-1}$.

4.2.2 Basic Operations

To ease the exposition of the protocols, we first describe how basic operations can be performed over the above secret sharing scheme. These operations are extensions of Boolean computations from Araki *et al.* [6] to arithmetic shares, similar to ABY^3 [84].

However, ABY^3 relies on efficient garbled circuits for non-linear function computation which is fundamentally different than the philosophy of this work which relies on simple modular arithmetic. In this manner, we propose a hybrid integration of ideas from SecureNN and ABY^3.

Correlated Randomness: Throughout this work, we will need two basic random number generators. Both of these can be efficiently implemented (using local computation) using PRFs. We describe them below:

(A) **3-out-of-3 randomness:** Random $\alpha_1, \alpha_2, \alpha_3$ such that $\alpha_1 + \alpha_2 + \alpha_3 \equiv 0 \pmod{L}$ and party P_i holds α_i

(B) **2-out-of-3 randomness:** Random $\alpha_1, \alpha_2, \alpha_3$ such that $\alpha_1 + \alpha_2 + \alpha_3 \equiv 0 \pmod{L}$ and party P_i holds (α_i, α_{i+1}).

Given pairwise shared random keys k_i (shared between parties P_i and P_{i+1}), the above two can be computed as $\alpha_i = F_{k_i}(\mathsf{cnt}) - F_{k_{i-1}}(\mathsf{cnt})$ and $(\alpha_i, \alpha_{i-1}) = (F_{k_i}(\mathsf{cnt}), F_{k_{i-1}}(\mathsf{cnt}))$ where cnt is a counter incremented after each invocation. This is more formally described later on in Π_{Prep} in Figure 4.1.

Linear operations: Let a, b, c be public constants and $[\![x]\!]^m$ and $[\![y]\!]^m$ be secret shared. Then $[\![ax+by+c]\!]^m$ can be locally computed as $(ax_1+by_1+c, ax_2+by_2, ax_3+by_3)$ and hence are simply local computations.

Multiplications Π_{Mult}: To multiply two shared values together $[\![x]\!]^m = (x_1, x_2, x_3)$ and $[\![y]\!]^m = (y_1, y_2, y_3)$, parties locally compute $z_1 = x_1y_1 + x_2y_1 + x_1y_2$, $z_2 = x_2y_2 + x_3y_2 + x_2y_3$, and $z_3 = x_3y_3 + x_1y_3 + x_3y_1$. At the end of this, z_1, z_2, and z_3 form a 3-out-of-3 secret sharing of $[\![z = x \cdot y]\!]^m$. Parties then perform *resharing* where 3-out-of-3 randomness is used to generate 2-out-of-3 sharing by sending $\alpha_i + z_i$ to party $i-1$.

Convolutions and Matrix Multiplications: We rely on prior work to perform convolutions and matrix multiplications over secret shares. To perform matrix multiplications, we note that Π_{Mult} described above extends to incorporate matrix multiplications. To perform convolutions, we simply expand the convolutions into matrix multiplications of larger dimensions (cf Section 5.1 of [109]) and invoke the protocol for matrix multiplications. Note that with fixed-point arithmetic, each multiplication protocol has to be followed by the truncation protocol (cf Figure 4.1) to ensure correct fixed-point precision. For more details on fixed-point multiplication, semi-honest, and malicious variants of this refer to [84, 6].

Reconstruction of $[\![x]\!]^m$: In the semi-honest setting, each party sends one ring element to the next party, i.e., P_i sends share x_i to P_{i+1}. In the malicious setting, each party sends x_i to P_{i+1} and x_{i+1} to P_{i-1} and aborts if the two received values do not agree. Note that in either case, a single round of communication is required.

Select Shares Π_{SS}: We define a sub-routine Π_{SS}, which will be used a number of times in the descriptions of other functionalities. It takes as input shares of two random values $[\![x]\!]^L$, $[\![y]\!]^L$, and shares of a random bit $[\![b]\!]^2$. The output $[\![z]\!]^L$ is either $[\![x]\!]^L$ or $[\![y]\!]^L$ depending on whether $b = 0$ or $b = 1$. To do this, we assume access to shares of a random bit $[\![c]\!]^2$ and $[\![c]\!]^L$ (pre-computation). Then we open the bit $(b \oplus c) = e$. If $e = 1$, we set $[\![d]\!]^L = [\![1 - c]\!]^L$ otherwise set $[\![d]\!]^L = [\![c]\!]^L$. Finally, we

compute $[\![z]\!]^L = [\![(y-x)\cdot d]\!]^L + [\![x]\!]^L$ where $[\![(y-x)\cdot d]\!]^L$ can be computed using Π_{Mult} $(y-x, d)$.

XOR with public bit b: Given shares of a bit $[\![x]\!]^m$ and a public bit b, we can locally compute shares of bit $[\![y]\!]^m = [\![x \oplus b]\!]^m$ by noting that $y = x + b - 2b \cdot x$. Since b is public, this is a linear operation and can be computed in both the semi-honest and malicious adversary models.

Evaluating $[\![(-1)^\beta \cdot x]\!]^m$ from $[\![x]\!]^m$ and $[\![\beta]\!]^m$: We assume that $\beta \in \{0, 1\}$. We first compute $1 - 2\beta$ and then perform the multiplication protocol described above to obtain $[\![(1-2\beta)x]\!]^m = [\![(-1)^\beta \cdot x]\!]^m$. We split our computations into data-dependent online computations and data-independent offline computations. Protocols for offline computations are in Figure 4.1.

4.2.3 Private Compare

This function evaluates the bit $x \geq r$ where r is public and the parties hold shares of bits of x in $\mathbb{Z}_p$. Algorithm 11 describes this protocol. Note that β is necessary for privacy as β' reveals information about the output $(x \geq r)$ if not blinded by a random bit β. Each of the bits are independent so a single blinding bit β is sufficient to hide computation of $(x \geq r)$ or $(r > x)$.

(A) Step 2: $u[i]$ can be computed by first evaluating shares of $2\beta - 1$ and then computing the product of $(2\beta - 1)$ and $x[i] - r[i]$. This can be done in a single round using one invocation of Π_{Mult}.

(B) Steps 3,4: These are simply local computations. For instance, $[\![w[i]]\!] = (w[i]_1, w[i]_2, w[i]_3)$ can be computed as $w[i]_j = x[i]_j + \delta_{j1} r[i] - 2r[i]x[i]_j$ where $j \in \{1, 2, 3\}$ and δ_{ij} is the Kronecker delta function and is defined as

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

(C) Step 6 can be computed in $\log_2 \ell + 1$[3] rounds using sequential invocations of the Π_{Mult} with smaller strings.

(D) Steps 7,8: These are once again local computations.

This protocol is an example of the challenges of integrating approaches based on simple modular arithmetic with malicious security. Both SecureNN and Falcon aim to find if there exists an index i such that $c[i] = 0$. However, the existence of a semi-honest third party makes checking this much easier in SecureNN. The two primary parties simply blind and mask their inputs and send them to the third party. This is not possible in Falcon due to the stronger adversarial model and requires newer protocol constructions. In particular, we need to multiply all the $c[i]$'s together along with a mask in $\mathbb{Z}_p^*$ and reveal this final product to compute the answer.

[3]One additional round because of multiplication by random blinding factor.

Π_{Prep}

Usage: Used to generate pre-processing material required for the online protocols.

Setup: This step will have to be executed only once.

(A) Each party P_i chooses a random seed k_i
(B) Send this random seed to party P_{i+1}

Common randomness: Let F be any seeded PRNG. Then 3-out-of-3 and 2-out-of-3 common randomness described in Section 4.2.2 can be generated as follows:

(A) $\alpha_i = F_{k_i}(\mathsf{cnt}) - F_{k_{i-1}}(\mathsf{cnt})$ and $\mathsf{cnt}{+}{+}$
(B) $(\alpha_i, \alpha_{i-1}) = (F_{k_i}(\mathsf{cnt}), F_{k_{i-1}}(\mathsf{cnt}))$ and $\mathsf{cnt}{+}{+}$

Truncation Pair: Generate truncation pair $[\![r]\!], [\![r']\!] = [\![r/2^d]\!]$.

(A) Run protocol Π_{trunc2} from [84] (Figure 3 in [84])

Correlated randomness for Private Compare: Correlated randomness for Π_{PC}

(A) Sample random bit $[\![b]\!]^2$
(B) Use bit injection from [84] $[\![b]\!]^2 \to [\![b]\!]^p$
(C) Sample random values $m_1, \ldots m_k \in \mathbb{Z}_p$.
(D) Compute and open $m_1^{p-1}, \ldots, m_k^{p-1}$.
(E) Remove openings that equal 0 and queue openings that equal 1. Note that this computation takes $\lceil \log_2 p \rceil$ rounds and can be amortized for efficiency (by setting a large value of k).

Correlated randomness for $\mathbf{Wrap_3}$: Correlated randomness required for Π_{WA}

(A) Sample random bits $[\![r_i]\!]^2$ for $i \in [\ell]$
(B) Perform bit composition from [84] to get $[\![r_i]\!]^L$
(C) Use bit injection from [84] $[\![r_i]\!]^2 \to [\![r_i]\!]^p$
(D) Use the optimized full adder FA to compute the final carry bit. Note that this bit is precisely $\mathsf{wrap}_3(\cdot)$

Correlated randomness for ReLU: Correlated randomness required for Π_{ReLU}

(A) Sample random bit $[\![b]\!]^2$
(B) Use bit injection from [84] $[\![b]\!]^2 \to [\![b]\!]^L$

Correlated randomness for Maxpool and Division: No additional correlated randomness necessary other than that used in their subroutines.

Figure 4.1: Protocols for generating various pre-processing material

Algorithm 11 Private Compare $\Pi_{\mathsf{PC}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold replicated secret sharing of bits of x in $\mathbb{Z}_p$.
Output: All parties get shares of the bit $(x \geq r) \in \mathbb{Z}_2$.
Common Randomness: P_1, P_2, P_3 hold a public ℓ bit integer r, shares of a random bit in two rings $[\![\beta]\!]^2$ and $[\![\beta]\!]^p$ and shares of a random, secret integer $m \in \mathbb{Z}_p^*$.

1: **for** $i = \{\ell - 1, \ell - 2, \ldots, 0\}$ **do**
2: Compute shares of $u[i] = (-1)^\beta (x[i] - r[i])$
3: Compute shares of $w[i] = x[i] \oplus r[i]$
4: Compute shares of $c[i] = u[i] + 1 + \sum_{k=i+1}^{\ell} w[k]$
5: **end for**
6: Compute and reveal d given by $d := [\![m]\!]^p \cdot \prod_{i=0}^{\ell-1} c[i] \pmod p$
7: Set $\beta' = 1$ if $(d \neq 0)$ and 0 otherwise.
8: **return** Shares of $\beta' \oplus \beta \in \mathbb{Z}_2$

4.2.4 Wrap Function

Central to the computation of operations such as ReLU and DReLU is a comparison function. The wrap functions, wrap_2 and wrap_3 are defined below as a function of the secret shares of the parties and effectively compute the "carry bit" when the shares are added together as integers. Eq. 4.11 shows that DReLU can be easily computed using the wrap_3 function. So all we require is a secure protocol for wrap_3. Note that we define two similar functions called "wrap" (denoted by wrap_2 and wrap_3). One function takes three inputs and the other one takes four inputs and are formally defined as follows:

$$\mathsf{wrap}_2(a_1, a_2, L) = \begin{cases} 0 & \text{if } \ a_1 + a_2 < L \\ 1 & \text{Otherwise} \end{cases} \tag{4.2}$$

$$\mathsf{wrap}_{3e}(a_1, a_2, a_3, L) = \begin{cases} 0 & \text{if } a_1 + a_2 + a_3 < L \\ 1 & \text{if } L \leq a_1 + a_2 + a_3 < 2L \\ 2 & \text{if } 2L \leq a_1 + a_2 + a_3 < 3L \end{cases} \tag{4.3}$$

In the rest of the chapter, we use the $\pmod 2$ reduction of the wrap function in Eq. 4.4. We call Eq. 4.3 the *exact wrap* function and Eq. 4.4 as simply the wrap function.

$$\mathsf{wrap}_3(a_1, a_2, a_3, L) = \mathsf{wrap}_{3e}(a_1, a_2, a_3, L) \pmod 2 \tag{4.4}$$

Next we briefly describe the connection between wrap_3 computed on shares a_1, a_2, a_3 and the most significant bit (MSB) of the underlying secret a. Note that $a = a_1 + a_2 + a_3 \pmod L$ as a_i's are shares of a modulo L. Considering this sum as a logic circuit (for instance as a ripple carry adder), we can see that $\mathrm{MSB}(a) = \mathrm{MSB}(a_1) + \mathrm{MSB}(a_2) + \mathrm{MSB}(a_3) + c \pmod 2$ where c is the carry bit from the previous index. The key insight here is that the carry c from the previous index is simply the wrap_3 function computed on a_i's (ignoring their MSB's) modulo $L/2$ (this

Algorithm 12 wrap$_3$ $\Pi_{\mathsf{WA}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of a in $\mathbb{Z}_L$.
Output: P_1, P_2, P_3 get shares of a bit $\theta = \mathsf{wrap}_3(a_1, a_2, a_3, L)$
Common Randomness: P_1, P_2, P_3 hold shares $[\![x]\!]^L$ (of a random number x), $[\![x[i]]\!]^p$ (shares of bits of x) and $[\![\alpha]\!]^2$ where $\alpha = \mathsf{wrap}_3(x_1, x_2, x_3, L)$.

1: Compute $r_j \equiv a_j + x_j \pmod{L}$ and $\beta_j = \mathsf{wrap}_2(a_j, x_j, L)$
2: Reconstruct $r \equiv \sum r_j \pmod{L}$
3: Compute $\delta = \mathsf{wrap}_3(r_1, r_2, r_3, L)$ ▷ In the clear
4: Run Π_{PC} on $x, r+1$ to get $\eta = (x \geq r+1)$ ▷ i.e., $\eta = (x > r)$
5: **return** $\theta = \beta_1 + \beta_2 + \beta_3 + \delta - \eta - \alpha$

is evident from Eq. 4.3). And this last operation is synonymous with computing the wrap_3 function on $2a_i$'s modulo L. We will further describe the consequences of this connection in Section 4.2.5 where we describe a protocol to compute the ReLU and DReLU functions. Algorithm 12 gives the protocol for securely computing the wrap_3 function. Note that wrap_2 function is always computed locally and hence a secure algorithm is not needed for the same. Furthermore, note that the wrap_2 function allows us to write exact integer equations as follows: if $a \equiv a_1 + a_2 \pmod{L}$ then $a = a_1 + a_2 - \mathsf{wrap}_2(a_1, a_2, L) \cdot L$ where the former relation is a congruence relation but the latter is an integer relation (and has exact equality). Finally, to see the correctness of the wrap_3 protocol, in reference to Algorithm 12, we can write the following set of equations

$$r = a + x - \eta \cdot L \tag{4.5}$$
$$r = r_1 + r_2 + r_3 - \delta_e \cdot L \tag{4.6}$$
$$r_i = a_i + x_i - \beta_i \cdot L \qquad \forall i \in \{1, 2, 3\} \tag{4.7}$$
$$x = x_1 + x_2 + x_3 - \alpha_e \cdot L \tag{4.8}$$

where δ_e, α_e denote the exact wrap functions, Eq. 4.6,4.8 follow from the definition of the exact wrap function, while Eq 4.7 follows from the definition of wrap_2 function. To see Eq. 4.5, note that $r, a, x \in [0, L-1]$ and that $r \equiv a + x \pmod{L}$. Hence $a + x \geq L$ iff $r < x$ (or $x \geq r+1$). Finally, assuming θ_e is the exact wrap function on a_1, a_2, a_3, i.e.,

$$a = a_1 + a_2 + a_3 - \theta_e \cdot L \tag{4.9}$$

Eqs. 4.5-4.9 together give a constraint among the Greek symbols (in other words, (4.5) - (4.6) - (4.7) + (4.8) + (4.9) gives Eq. 4.10 below)

$$\theta_e = \beta_1 + \beta_2 + \beta_3 + \delta_e - \eta - \alpha_e \tag{4.10}$$

Reducing Eq. 4.10 modulo 2 gives us $\theta = \beta_1 + \beta_2 + \beta_3 + \delta - \eta - \alpha$ which is used to compute wrap_3 as in Algorithm 12.

Algorithm 13 ReLU, $\Pi_{\mathsf{ReLU}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of a in $\mathbb{Z}_L$.
Output: P_1, P_2, P_3 get shares of $\mathsf{ReLU}(a)$.
Common Randomness: $[\![c]\!]^2$ and $[\![c]\!]^L$ (shares of a random bit in two rings)

1: Run Π_{WA} to get $\mathsf{wrap}_3(2a_1, 2a_2, 2a_3, L)$
2: Compute $[\![b]\!]^2$ where $b = \mathsf{DReLU}(a)$ ▷ Local comp. (Eq. 4.11)
3: **return** Output of Π_{SS} run on $\{a, 0\}$ with b as selection.

4.2.5 ReLU and Derivative of ReLU

We now describe how to construct a protocol for securely computing ReLU(a) and DReLU(a) for a given secret a. Recall that we use fixed-point arithmetic over $\mathbb{Z}_{2^\ell}$ for efficiency reasons. Using the natural encoding of native `C++` data-types, we know that positive numbers are the first $2^{\ell-1}$ and have their most significant bit equal to 0. Negative numbers, on the other hand are the last $2^{\ell-1}$ numbers in the ℓ-bit range and have their most significant bit equal to 1. Thus, the DReLU function defined by Eq. 4.1, has a simple connection with the MSB of the fixed-point representation viz., $\mathrm{DReLU}(a) = 1 - \mathrm{MSB}(a)$. Furthermore, in Section 4.2.4, we have seen the connection between MSB(a) and wrap_3. Together, these insights can be distilled into the following equation:

$$\begin{aligned}\mathsf{DReLU(a)} = \mathsf{MSB}(a_1) \oplus \mathsf{MSB}(a_2) \oplus \mathsf{MSB}(a_3) \\ \oplus\ \mathsf{wrap}_3(2a_1, 2a_2, 2a_3, L) \oplus 1\end{aligned} \tag{4.11}$$

In particular, the derivative of ReLU can be computed by combining the output of the wrap function with local computations. Finally, for computing ReLU from DReLU, we simply call Π_{SS} (which effectively performs Π_{Mult} on shares of a and shares of $\mathsf{DReLU}(a)$). With these observations, we can implement the ReLU and Derivative of ReLU protocols (see Algorithm 13). Note that the approach here is crucially different from the approach SecureNN uses due to use of fundamentally different building blocks as well as deeper mathematical insights such as Eq. 4.11. To achieve the DReLU functionality, SecureNN first uses a subroutine to transform the shares of the secret into an odd modulus ring and then uses another subroutine to compute the MSB (cf Section 4.1.2). Both these subroutines have similar complexities. Falcon on the other hand uses the insight presented in Eq. 4.11 to completely eliminate the need for these subroutines, improving the efficiency by about 2× and simplifying the overall protocol. This also drastically improves the end-to-end performance (by over 6.4×) as the ReLU and DReLU functionalities are the building blocks of every comparison in the network.

4.2.6 Maxpool and Derivative of Maxpool

The functionality of maxpool simply takes as input a vector of secret shared values and outputs the maximum value. For derivative of maxpool, we need a one-hot vector of

the same size as the input where the 1 is at the location of the index of the maximum value. Maxpool can be implemented using a binary sort on the vector of inputs and small amounts of bookkeeping, where the comparisons can be performed using ReLUs. Derivative of maxpool can be efficiently implemented along with maxpool. Algorithm 14 describes these in detail.

Algorithm 14 Maxpool, $\Pi_{\mathsf{Maxpool}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of $a_1, a_2, \ldots, a_n$ in $\mathbb{Z}_L$.
Output: P_1, P_2, P_3 get shares of a_k and $\boldsymbol{e_k}$ where $k = \mathsf{argmax}\{a_1, a_2, \ldots, a_n\}$ and where $\boldsymbol{e_k} = \{e_1, e_2, \ldots, e_n\}$ with $e_i = 0\ \forall i \neq k$ and $e_k = 1$.
Common Randomness: No additional common randomness required.

1: Set $\mathsf{max} \leftarrow a_1$ and $\mathbf{ind} \leftarrow \boldsymbol{e_1} = \{1, 0, \ldots, 0\}$
2: **for** $i = \{2, 3, \ldots, n\}$ **do**
3: Set $\mathsf{d_{max}} \leftarrow (\mathsf{max} - a_i)$ and $\mathbf{d_{ind}} \leftarrow (\mathbf{ind} - \boldsymbol{e_i})$
4: $b \leftarrow \Pi_{\mathsf{DReLU}}(\mathsf{d_{max}})$ ▷ b → Derivative of ReLU
5: Set max as Π_{SS} output on inputs $\{a_i, \mathsf{max}\}$ using selection b.
6: Set $\mathbf{ind}$ as Π_{SS} output on inputs $\{\boldsymbol{e_i}, \mathbf{ind}\}$ using selection b.
7: **end for**
8: **return** $\mathsf{max}, \mathbf{ind}$

4.2.7 Division and Batch Normalization

Truncation allows parties to securely eliminate lower bits of a secret shared value (i.e., truncation by k bits of a secret $a \to a/2^k$). However, the problem of dividing by a secret shared number is considerably harder and efficient algorithms rely on either (1) sequential comparison or (2) numerical methods. In this work, we use the numerical methods approach for its efficiency. We use the specific choices of initializations given in [22, 1] to efficiently compute division over secret shares. A crucial component of numerical methods is the need to estimate the value of the secret within a range. We achieve this using Algorithm 15. Note that Algorithm 15 outputs the bounding power of 2, which is also what is guaranteed by the functionality. In this way, we only reveal the bounding power of 2 and nothing else.

Algorithm 16 is used to compute the value of a/b where a, b are secret shared. The first step for the algorithm is to transform $b \to x$ where $x \in [0.5, 1)$. Note that even though b is a fixed-point precision of f_p, for the computations in Algorithm 16, x has to be interpreted as a value with fixed-point precision $\alpha + 1$ where $2^\alpha \leq b < 2^{\alpha+1}$. Thus we first need to extract α (the appropriate range) using Algorithm 15. Let $w_0 = 2.9142 - 2x$, $\epsilon_0 = 1 - x \cdot w_0$ (cf. [22, 1] for choice of constants). Then an initial approximation for $1/x$ is $w_0 \cdot (1 + \epsilon_0)$. For higher-order approximations, set $\epsilon_i = \epsilon_{i-1}^2$ and multiply the previous approximate result by $(1 + \epsilon_i)$ to get a better approximate result. Each successive iteration increases the round complexity by 2. For our value of fixed-point precision, we use the following approximation which works with high

Algorithm 15 Bounding Power, $\Pi_{\mathsf{Pow}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of b in $\mathbb{Z}_L$.
Output: P_1, P_2, P_3 get α in the clear, where $2^\alpha \leq b < 2^{\alpha+1}$.
Common Randomness: No additional common randomness required.

1: Initialize $\alpha \leftarrow 0$
2: **for** $i = \{\ell - 1, \ldots, 1, 0\}$ **do**
3: $\quad c \leftarrow \Pi_{\mathsf{DReLU}}(\mathsf{x} - 2^{2^i+\alpha})$ and reconstruct c
4: $\quad$ Set $\alpha \leftarrow \alpha + 2^i$ if $c = 1$
5: **end for**
6: **return** α

accuracy (refer to Section 4.4 for details):

$$\mathsf{AppDiv}(x) = w_0 \cdot (1 + \epsilon_0)(1 + \epsilon_1) \approx \frac{1}{x} \tag{4.12}$$

Batch-norm is another important component of neural network architectures. They

Algorithm 16 Division, $\Pi_{\mathsf{Div}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of a, b in $\mathbb{Z}_L$.
Output: P_1, P_2, P_3 get shares of a/b in $\mathbb{Z}_L$ computed as integer division with a given fixed precision f_p.
Common Randomness: No additional common randomness required.

1: Run Π_{Pow} on b to get α such that $2^\alpha \leq b < 2^{\alpha+1}$
2: Compute $w_0 \leftarrow 2.9142 - 2b$
3: Compute $\epsilon_0 \leftarrow 1 - b \cdot w_0$ and $\epsilon_1 \leftarrow \epsilon_0^2$
4: **return** $aw_0(1 + \epsilon_0)(1 + \epsilon_1)$

improve the convergence as well as help automate the training process. Algorithm 17 describes the protocol to compute batch-norm. For step 3, required by Eq. B.4c in Appendix, we use Newton's method. We use $2^{-\lfloor \alpha/2 \rfloor}$ as an initial approximation of $1/\sqrt{\sigma^2 + \epsilon}$, where $2^\alpha \leq \sigma^2 + \epsilon < 2^{\alpha+1}$ and use the successive iterative formula:

$$x_{n+1} = \frac{x_n}{2}\left(3 - a x_n^2\right) \tag{4.13}$$

Given the strategic choice of initial guess, the number of rounds required for close approximation for our choice of fixed-point precision is 4. However, batch normalization during training is computed by sequentially computing $\sqrt{\sigma^2 + \epsilon}$ and then computing the inverse. This approach is used to optimize the computation required during back-propagation which requires the values of $\sqrt{\sigma^2 + \epsilon}$. For computing the square root of a value a, we use Newton's method given by Eq. 4.14. This can then be used in conjunction with the inverse computation given by Eq. 4.12 to complete the batch-norm

Algorithm 17 Batch Norm, $\Pi_{\mathsf{BN}}(P_1, P_2, P_3)$:

Input: P_1, P_2, P_3 hold shares of $a_1, a_2 \dots a_m$ in $\mathbb{Z}_L$ where m is the size of each batch and shares of two learnable parameters γ, β.

Output: P_1, P_2, P_3 get shares of $\gamma z_i + \beta$ for $i \in [m]$ and $z_i = (a_i - \mu)/(\sqrt{\sigma^2 + \epsilon})$ where $\mu = 1/m \sum a_i$, $\sigma^2 = 1/m \sum (a_i - \mu)^2$, and ϵ is a set constant.

Common Randomness: No additional common randomness required.

1: Set $\mu \leftarrow 1/m \cdot \sum a_i$
2: Compute $\sigma^2 \leftarrow 1/m \cdot \sum (a_i - \mu)^2$ and let $b = \sigma^2 + \epsilon$
3: Run Π_{Pow} on b to find α such that $2^\alpha \leq b < 2^{\alpha+1}$
4: Set $x_0 \leftarrow 2^{-\lfloor \alpha/2 \rfloor}$
5: **for** $i \in 0, \dots, 3$ **do**
6: Set $x_{i+1} \leftarrow \frac{x_i}{2}(3 - b x_i^2)$
7: **end for**
8: **return** $\gamma \cdot x_{\mathsf{rnds}} \cdot (a_i - \mu) + \beta$ for $i \in [m]$

computations.

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{4.14}$$

4.3 Theoretical Analysis

We provide a detailed theoretical analysis of our framework and protocols. In particular, we provide proofs of security and analyze the theoretical complexity.

4.3.1 Security Proofs

We model and prove the security of our construction in the real-world/ideal-world simulation paradigm [54, 21, 20]. In the real interaction, the parties execute the protocol in the presence of an adversary and the environment. On the other hand, in the ideal interaction, the parties send their inputs to a trusted party that computes the functionality truthfully. Finally, to prove the security of our protocols, for every adversary in the real interaction, there exists a simulator in the ideal interaction such that the environment cannot distinguish between the two scenarios. In other words, whatever information the adversary extracts in the real interaction, the simulator can extract it in the ideal world as well.

We show that our protocols are perfectly secure (i.e., the joint distributions of the inputs, outputs, and the communication transcripts are exactly the same and not statistically close) in the stand-alone model (i.e., protocol is executed only once), and that they have a straight-line black-box simulators (i.e., only assume oracle access to the adversary and hence do no rewind)[4]. We then rely on the result of Kushilevitz *et*

[4]For more details on these, refer to [72]

$\mathcal{F}_{\mathsf{PC}}$

Input: The functionality receives inputs $\{[\![x_i]\!]^p\}_{i=1}^{\ell}, r$

Output: Compute the following

(A) Reconstruct bits x_i and $x = \sum x_i \cdot 2^i$
(B) Compute $b = (x \geq r)$
(C) Generate random shares of b and send back to the parties

Figure 4.2: Ideal functionality for Π_{PC}

$\mathcal{F}_{\mathsf{WA}}$

Input: The functionality receives inputs $[\![a]\!]^L$.

Output: Compute the following

(A) Compute $b = \mathsf{wrap}_3(a_1, a_2, a_3, L)$
(B) Generate random shares of b and send back to the parties

Figure 4.3: Ideal functionality for Π_{WA}

al. [72] to prove that our protocols are secure under concurrent general composition (Theorem 1.2 in [72]).

We formally describe the functionalities in Appendix B. We describe simulators for Π_{PC} (Figure 4.2), Π_{WA} (Figure 4.3), Π_{ReLU} (Figure 4.4), Π_{Maxpool} (Figure 4.5), Π_{Pow} (Figure 4.6), Π_{Div} (Figure 4.7), and Π_{BN} (Figure 4.8) that achieve indistinguishability. $\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Trunc}}, \mathcal{F}_{\mathsf{Reconst}}$ are identical to prior works [84, 50]. We prove security using the standard indistinguishability argument. To prove the security of a particular functionality, we set up hybrid interactions where the sub-protocols used in that protocol are replaced by their corresponding ideal functionalities and then prove that the interactions can be simulated. This hybrid argument in effect sets up a series of interactions $I_0, I_1, \ldots, I_k$ for some k where I_0 corresponds to the real interaction and I_k corresponds to the ideal interaction. Each neighboring interaction, i.e., I_i, I_{i+1} for $i \in \{0, \ldots, k-1\}$ is then shown indistinguishable from each other, in effect showing that the real and ideal interactions are indistinguishable. Without loss of generality, we assume that party P_2 is corrupt. In the *real world*, the adversary Adv interacts with the honest parties P_0 and P_1. In the *ideal world*, the simulator interacts with the adversary and simulates exact transcripts for interactions between the adversary Adv and P_0, P_1. On the other hand, the simulator should be able to extract the adversaries inputs. These inputs are fed to the functionality to generate correct output distributions. Theorems 4.1-4.6 pertain to the indistinguishability of these two interactions.

Theorem 4.1. Π_{PC} *securely realizes* $\mathcal{F}_{\mathsf{PC}}$ *with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Reconst}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

$\mathcal{F}_{\mathsf{ReLU}}$

Input: The functionality receives inputs $[\![a]\!]^L$.

Output: Compute the following

(A) Compute $b = \mathsf{ReLU}(a_1 + a_2 + a_3 \pmod L)$ ▷ Refer to Eq. 4.1
(B) Generate random shares of b and send back to the parties

Figure 4.4: Ideal functionality for Π_{ReLU}

$\mathcal{F}_{\mathsf{Maxpool}}$

Input: The functionality receives inputs $[\![a_1]\!]^L, \ldots, [\![a_n]\!]^L$.

Output: Compute the following

(A) Reconstruct $a_1, \ldots, a_n$ and compute $k = \mathsf{argmax}\{a_1, \ldots, a_n\}$.
(B) Set $\boldsymbol{e_k} = \{e_1, e_2, \ldots, e_n\}$ with $e_i = 0\ \forall i \neq k$ and $e_k = 1$.
(C) Generate random shares of a_k and $\boldsymbol{e_k}$ and send back to the parties.

Figure 4.5: Ideal functionality for Π_{Maxpool}

$\mathcal{F}_{\mathsf{Pow}}$

Input: The functionality receives inputs $[\![b]\!]^L$ and an index $k \in \{0, 1, \ldots, \ell - 1\}$.

Output: Compute each bit of α sequentially as follows:

(A) Reconstruct b.
(B) Compute α such that $2^\alpha \leq b < 2^{\alpha+1}$.
(C) If $k = 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, 0\}$ to all parties.
(D) If $k \neq 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, k\}$ to all parties and then Abort.

Figure 4.6: Ideal functionality for Π_{Pow}

$\mathcal{F}_{\mathsf{Div}}$

Input: The functionality receives inputs $[\![a]\!]^L, [\![b]\!]^L$ and an index $k \in \{0, 1, \ldots, \ell - 1\}$.

Output: Compute the following

(A) Reconstruct a, b.
(B) Compute α such that $2^\alpha \leq b < 2^{\alpha+1}$.
(C) If $k = 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, 0\}$ to all parties.
(D) If $k \neq 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, k\}$ to all parties and then Abort.
(E) Generate random shares of $a \cdot \mathsf{AppDiv}(b)$ and send back to the parties

Figure 4.7: Ideal functionality for Π_{Div}

$\mathcal{F}_{\mathsf{BN}}$

Input: The functionality receives inputs $[\![a_1]\!]^L, \ldots, [\![a_n]\!]^L$ and $[\![\gamma]\!]^L, [\![\beta]\!]^L$ and an index $k \in \{0, 1, \ldots, \ell - 1\}$.

Output: Compute the following

(A) Reconstruct $a_1, \ldots, a_n$ and compute μ and σ^2 as given in Step 1,2 of Algorithm 17.
(B) Set $b = \sigma^2 + \epsilon$ and compute α such that $2^{\alpha-1} \leq b < 2^{\alpha+1}$.
(C) If $k = 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, 0\}$ to all parties.
(D) If $k \neq 0$ send $\alpha[i]$ for $i \in \{\ell - 1, \ldots, k\}$ to all parties and then Abort.
(E) Complete steps 4-8 of Algorithm 17 and return random shares of the output.

Figure 4.8: Ideal functionality for Π_{BN}

Proof. We first set up some detail on the proof strategy that is essential for other proofs as well. For the ease of exposition, we describe it in the context of Π_{PC}. The goal of designing a simulator is to be able to demonstrate the ability to produce transcripts that are indistinguishable from the transcripts in the real world. The joint distribution of the inputs and outputs is a part of these transcripts and hence has to be indistinguishable in the two interactions. However, since the honest parties simply forward their inputs to the functionality, the simulator must be able to extract the inputs of the malicious parties to be able to generate the correct shares for the honest parties.

The usual technique to achieve this is to have the simulator run a simulated version of the protocol internally, i.e., emulating the roles of the honest parties and interacting with the adversary. This is what we call an *internal run*. This internal run can then be used to extract the inputs of the adversarial party (which can then be forwarded to the functionality in the ideal interaction). Note that in the hybrid argument, since the subroutines used in the protocol can be replaced by their corresponding ideal interactions, the simulator can emulate the roles of these trusted functionalities in its internal run.

In the specific context of Π_{PC}, the simulator $\mathcal{S}$ for adversary Adv works by playing the role of the trusted party for $\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Reconst}}$, and $\mathcal{F}_{\mathsf{Prep}}$. To be able to simulate, we need to show that:

(A) All the transcripts from the real interactions can be simulated.
(B) The honest parties receive their outputs correctly.

Simulation follows easily from the protocol and the hybrid argument. The simulator for Π_{Mult} (along with the simulator for Π_{Reconst}) can be used to simulate the transcripts from Steps 2, 6 (from Algorithm 11). Note that the distributions of these transcripts are all uniformly random values (β is required to make the transcript for β' uniformly random, the various bits $u[i], w[i]$, and $c[i]$ are random because x is random) and hence achieve perfect security. Steps 3, 4, 7, and 8 on the other hand are all local and do not need simulation.

To extract the inputs of the malicious party, the simulator uses the fact that it has access to r and β (though $\mathcal{F}_{\mathsf{Prep}}$) and all the internal values for the honest parties

(in the internal run) and hence can extract the shares of $x[i]$ from the corrupt party P_2. Finally, if the protocol aborts at any time in the internal run, then the simulator sends Abort to $\mathcal{F}_{\mathsf{PC}}$ otherwise, it inputs the extracted shares of $x[i]$ to $\mathcal{F}_{\mathsf{PC}}$ and the honest parties receive their outputs. □

Theorem 4.2. Π_{WA} *securely realizes* $\mathcal{F}_{\mathsf{WA}}$ *with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{PC}}, \mathcal{F}_{\mathsf{Reconst}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. We use a similar set-up as the proof of Theorem 4.1. Step 1 is local computation and does not need simulation. Steps 2, 4 can be simulated using the simulators for $\mathcal{F}_{\mathsf{Reconst}}, \mathcal{F}_{\mathsf{PC}}$ respectively. Input extraction follows from having access to r_i (through $\mathcal{F}_{\mathsf{Prep}}$) and output x if the protocol does not abort. Finally, if the protocol does abort at any time in the internal run, then the simulator sends Abort to $\mathcal{F}_{\mathsf{WA}}$. Otherwise, it simply passes on the extracted shares of $a[i]$ to $\mathcal{F}_{\mathsf{WA}}$ and the honest parties receive their outputs. Note that Π_{DReLU} is not formally defined. However, this is simply local computation over Π_{WA} and the proofs can be extended analogously. □

Theorem 4.3. Π_{ReLU} *securely realizes* $\mathcal{F}_{\mathsf{ReLU}}$ *with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{WA}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. Simulation is done as before using the hybrid argument. The protocol simply composes $\mathcal{F}_{\mathsf{WA}}$ and $\mathcal{F}_{\mathsf{Mult}}$ and hence is simulated using the corresponding simulators. □

Theorem 4.4. Π_{Maxpool} *securely realizes* $\mathcal{F}_{\mathsf{Maxpool}}$ *with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{ReLU}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. Similar to the proof of Theorem 4.3, simulation works by sequentially composing the simulators for $\mathcal{F}_{\mathsf{ReLU}}$ and $\mathcal{F}_{\mathsf{Mult}}$. □

Theorem 4.5. Π_{Pow} *securely realizes* $\mathcal{F}_{\mathsf{Pow}}$ *with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{ReLU}}, \mathcal{F}_{\mathsf{Reconst}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. The simulator for Adv works by playing the role of the trusted party for $\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{ReLU}}$, and $\mathcal{F}_{\mathsf{Reconst}}$. The protocol sequentially reveals bits of the scale α. It is important to note the functionality that it emulates (see in Figure 4.6). The simulator runs the first iteration of the loop and in the process extracts the adversaries inputs. Then it proceeds to complete all the iterations of the loop. If the protocol proceeds without aborting till the end, then the simulator sends the extracted shares of b along with $k = 0$ to the functionality $\mathcal{F}_{\mathsf{Pow}}$. If the protocol aborts at iteration k, then the simulator sends the extracted shares of b along with k to $\mathcal{F}_{\mathsf{Pow}}$. □

Theorem 4.6. Π_{Div}, Π_{BN} *securely realize* $\mathcal{F}_{\mathsf{Div}}$, $\mathcal{F}_{\mathsf{BN}}$ *respectively, with abort, in the presence of one malicious party in the* $(\mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Pow}}, \mathcal{F}_{\mathsf{Prep}})$*-hybrid model.*

Proof. Π_{Div}, Π_{BN} are sequential combinations of local computations and invocations of $\mathcal{F}_{\mathsf{Mult}}$. Simulation follows directly from composing the simulators and input extraction follows from the simulator of Π_{Pow}. □

Table 4.2: Theoretical overheads of basic and compound protocols. Communication is in Bytes where ℓ is the logarithm of the ring size and k is its Byte size. We use n to denote the size of the vector in vectorized implementations.

	Protocol	Dependence	Semi-Honest		Malicious	
			Rounds	Comm	Rounds	Comm
Basic Protocols	MatMul	$(x \times y)(y \times z)$	1	kxz	1	$2kxz$
	Private Compare	n	$2 + \log_2 \ell$	$2kn$	$2 + \log_2 \ell$	$4kn$
	wrap_3	n	$3 + \log_2 \ell$	$3kn$	$3 + \log_2 \ell$	$6kn$
Compound Protocols	ReLU and Derivative of ReLU	n	$5 + \log_2 \ell$	$4kn$	$5 + \log_2 \ell$	$8kn$
	MaxPool and Derivative of Maxpool	$n, \{w, h\}$	$(wh-1)(7 + \log_2 \ell)$	$5k + wh$	$(wh-1)(7 + \log_2 \ell)$	$10k + 2wh$
	Pow	n	$5\ell + \ell \cdot \log_2 \ell$	$4kn\ell$	$5\ell + \ell \cdot \log_2 \ell$	$8kn\ell$
	Division	n	$7 + 5\ell + \ell \cdot \log_2 \ell$	$4kn\ell + 7kn$	$7 + 5\ell + \ell \cdot \log_2 \ell$	$8kn\ell + 14kn$
	Batch Norm	r, n	$15 + 5\ell + \ell \cdot \log_2 \ell$	$kr + 4kr\ell + 14krn$	$15 + 5\ell + \ell \cdot \log_2 \ell$	$2kr + 8kr\ell + 28krn$

Protocol Overheads. We theoretically estimate the overheads of our protocols in Table 4.2. The dominant round complexity for Private Compare comes from the string multiplication in Step 6. wrap_3 requires one additional round and one additional ring element (two in malicious security) over Private Compare. Computing derivative of ReLU is a local computation over the wrap_3 function. Computing ReLU requires two additional rounds and one ring element (two for malicious)[5]. Maxpool and the derivative of Maxpool require rounds proportional to the area of the filter. Finally, Pow, division, and batch-norm require a quadratic number of rounds in ℓ.

4.4 Experimental Evaluation

We evaluate the performance of training and inference with Falcon on 6 networks of varying parameter sizes trained using MNIST, CIFAR-10, and Tiny ImageNet datasets (cf. Section 2.5). A number of prior works such as SecureML [85], MiniONN [77], Gazelle [64], SecureNN [109], ABY^3 [84], and Chameleon [95] evaluate some of these networks and we mimic their evaluation set-up for comparison.

4.4.1 Experimental Setup

We implement the Falcon framework in about 14.6k LoC in `C++` using the communication backend of SecureNN[6]. We run our experiments on Amazon EC2 machines over Ubuntu 18.04 LTS with Intel-Core i7 processor and 64GB of RAM. Our evaluation set-up uses similar as compared to prior work [109, 84, 85, 95, 64]. We perform extensive evaluation of our framework in both the LAN and WAN settings. For the LAN setting, our bandwidth is about 625 MBps and ping time is about 0.2ms. For WAN experiments, we run servers in different geographic regions with 70ms ping time and 40 MBps bandwidth. The source code is available online at https://github.com/snwagh/falcon-public.

[5] And one (two) additional bits.

[6] `https://github.com/snwagh/securenn-public`

Optimizations: All data-independent computation, i.e., pre-computation, is parallelized using 16 cores to reduce the run-time. When a ReLU layer is followed by a Maxpool layer, we swap the order of these two layers for optimized run-times. We use the Eigen library for faster matrix multiplication and parallelize the Private Compare computation. We optimize across the forward and backward pass for Maxpool, ReLU, and Batch-Normalization layers, i.e., we compute the relevant derivatives while computing the functions. We use 32-bit integer range with 16 bits of floating-point precision. As the entire codebase is parallelizable, in the future, significant improvement is possible by implementing Falcon using TensorFlow or PyTorch which support easy parallelization as well as computations over GPUs.

Table 4.3: Comparison of inference time of various frameworks for different networks using MNIST dataset. All runtimes are reported in seconds and communication in MB. ABY^3 and XONN do no implement their maliciously secure versions. 2-party (2PC) and 4-party (4PC) protocols are presented here solely for comprehensive evaluation of the literature.

	Framework	Threat Model	LAN/ WAN	Network-A		Network-B		Network-C	
				Time	Comm.	Time	Comm.	Time	Comm.
2PC	SecureML [85]	Semi-honest	LAN	4.88	-	-	-	-	-
	DeepSecure [96]	Semi-honest	LAN	-	-	9.67	791	-	-
	EzPC [25]	Semi-honest	LAN	0.7	76	0.6	70	5.1	501
	Gazelle [64]	Semi-honest	LAN	0.09	0.5	0.29	0.8	1.16	70
	MiniONN [77]	Semi-honest	LAN	1.04	15.8	1.28	47.6	9.32	657.5
	XONN [94]	Semi-honest	LAN	0.13	4.29	0.16	38.3	0.15	32.1
3PC	Chameleon [95]	Semi-honest	LAN	-	-	2.7	12.9	-	-
	ABY^3 [84]	Semi-honest	LAN	0.008	0.5	0.01	5.2	-	-
	SecureNN [109]	Semi-honest	LAN	0.043	2.1	0.076	4.05	0.13	8.86
	Falcon	Semi-honest	LAN	0.011	0.012	0.009	0.049	0.042	0.51
		Malicious	LAN	0.021	0.31	0.022	0.52	0.089	3.37
	SecureNN [109]	Semi-honest	WAN	2.43	2.1	3.06	4.05	3.93	8.86
	Falcon	Semi-honest	WAN	0.99	0.012	0.76	0.049	3.0	0.5
		Malicious	WAN	2.33	0.31	1.7	0.52	7.8	3.37
4PC	FLASH [19]	Malicious	LAN	0.029	-	-	-	-	-
	FLASH [19]	Malicious	WAN	12.6	-	-	-	-	-

Table 4.4: Comparison of inference time of various frameworks over popular benchmarking network architectures from the machine learning domain. All run-times are reported in seconds and communication in MB.

Framework	Threat Model	LAN/WAN	LeNet (MNIST)		AlexNet (CIFAR-10)		VGG16 (CIFAR-10)		AlexNet (ImageNet)		VGG16 (ImageNet)	
			Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.
Falcon	Semi-honest	LAN	0.047	0.74	0.043	1.35	0.79	13.51	1.81	19.21	3.15	52.56
	Malicious	LAN	0.12	5.69	0.14	8.85	2.89	90.1	6.7	130.0	12.04*	395.7*
	Semi-honest	WAN	3.06	0.74	0.13	1.35	1.27	13.51	2.43	19.21	4.67	52.56
	Malicious	WAN	7.87	5.69	0.41	8.85	4.7	90.1	8.68	130.0	37.6*	395.7*

4.4.2 Results for Private Inference

Tables 4.3, 4.4 report the end-to-end latency time (in seconds) and number of bytes (in MB) communicated for performing a single inference query with Falcon. We

compare these values with the numbers reported from prior work wherever applicable. All the numbers are reported for both semi-honest and malicious adversarial setting. Further, we execute the queries in both LAN and WAN settings using Falcon.

Comparison to Prior Work. We compare the inference time of a single query and the communication bytes of Falcon with prior work on Networks-A, B, and C. None of the prior works evaluate the remaining networks and hence we do not compare the performance of Falcon for the networks in Table 4.4[7]. Depending on the network architecture, our results are between 3×-120× faster than existing work. In particular, Falcon is upto 18× faster than XONN [94] (11× on average) and 32× faster than Gazelle (23× on average), 8× faster than SecureNN (3× on average), and comparable to ABY^3 on small networks. Falcon is also 40× more communication-efficient than ABY^3 [84], 200× more communication-efficient than SecureNN [109], and over 760× more communication-efficient compared to XONN [94].

Inference time and communication with Falcon. For both the adversarial settings, the inference latency for Falcon over LAN is within 25ms for smaller networks (A and B) and around 100ms for Network-C and LeNet. For AlexNet and VGG16, the inference time ranges from 0.5 to 12s depending on the model and the input dataset. The inference time increases with the size of the input image. Hence, queries over Tiny ImageNet are slower than CIFAR-10 for the same model architecture. The inference time over the WAN setting ranges from 1 to 3s for the Networks-A, B, and C and from 3 to 37s for the remaining larger networks. However, we emphasize that the inference time with semi-honest adversarial setting is around 3× faster than that for the malicious adversary. Hence, a faster deployment protocol is possible depending on the trust assumptions of the application.

In addition to efficient response times, our results show that Falcon is optimized for communication rounds as well. The parties exchange less than 4MB of data for smaller networks (Table 4.3) and 5MB to 400MB for larger networks (Table 4.4). The amount of data exchanged is the same for both the LAN and WAN settings. However, similar to the inference time, more communication bytes are required for the malicious setting as compared to the semi-honest adversary.

4.4.3 Results for Private Training

Tables 4.5, 4.6 report the execution time and the communication required for training all the 6 network architectures.

Comparison to Prior Work. For private training, Falcon is upto 6× faster than SecureNN [109] (4× on average), 4.4× faster than ABY^3, and 70× faster than SecureML [85]. We highlight that Falcon achieves these speedups due to improved protocols (both round complexity and communication as described in Section 4.1.2).

[7]XONN [94] is evaluated on binarized model parameters of VGG16 and hence we do not compare with it.

Table 4.5: Comparison of training time of various frameworks over popular benchmarking network architectures from the security domain. All run-times are reported in hours and communication in TB. * correspond to 2PC numbers. ABY^3 does not implement their maliciously secure protocols.

Framework	Threat Model	LAN/ WAN	Network-A		Network-B		Network-C	
			Time	Comm.	Time	Comm.	Time	Comm.
SecureML [85]*	Semi-honest	LAN	81.7	-	-	-	-	-
SecureML [85]	Semi-honest	LAN	7.02	-	-	-	-	-
ABY^3 [84]	Semi-honest	LAN	0.75	0.031	-	-	-	-
SecureNN [109]	Semi-honest	LAN	1.03	0.11	-	-	17.4	30.6
Falcon	Semi-honest	LAN	0.17	0.016	0.42	0.056	3.71	0.54
	Malicious	LAN	0.56	0.088	1.17	0.32	11.9	3.29
SecureML [85]*	Semi-honest	WAN	4336	-	-	-	-	-
SecureNN [109]	Semi-honest	WAN	7.83	0.11	-	-	53.98	30.6
Falcon	Semi-honest	WAN	3.76	0.016	3.4	56.14	14.8	0.54
	Malicious	WAN	8.01	0.088	7.5	0.32	39.32	3.29
Batch Size, Epochs			128, 15		128, 15		128, 15	

Table 4.6: Comparison of training time of various frameworks over popular benchmarking network architectures from the machine learning domain. All run-times are reported in hours and communication in TB.

Framework	Threat Model	LAN/ WAN	LeNet		AlexNet (CIFAR-10)		VGG16 (CIFAR-10)		AlexNet (ImageNet)		VGG16 (ImageNet)	
			Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.
Falcon	Semi-honest	LAN	6.05×10^0	0.81	7.89×10^1	7.24	8.43×10^2	45.9	1.23×10^4	222.9	5.19×10^3	156.0
	Malicious	LAN	1.22×10^1	4.82	2.82×10^2	43.4	3.05×10^3	185.3	4.63×10^4	1598	1.95×10^4	1012
	Semi-honest	WAN	1.85×10^1	0.81	2.33×10^2	7.24	2.09×10^3	45.9	1.54×10^4	222.9	6.89×10^3	156.0
	Malicious	WAN	5.20×10^1	4.82	7.24×10^2	43.4	5.26×10^3	185.3	5.71×10^4	1598	2.47×10^4	1012
Batch Size, Epochs			128, 15		128, 90		128, 25		128, 90		128, 25	

As seen from Table 4.5, the communication overhead is 10× to 100× better for Falcon as compared to other solutions.

Execution time for Falcon. The time to privately train Networks-A, B, and C with Falcon is around 3 to 40 hrs. For larger networks, we extrapolate time from a single iteration of a forward and a backward pass. The training time ranges from a few weeks to hundreds of weeks. Although these values seem to be quite large, high-capacity machine learning models are known to take from a few days to weeks to achieve high accuracy when trained (both on CPU as well as GPU). Such networks can also benefit from transfer leaning techniques, where a public pre-trained model is fine-tuned with a private dataset. This fine-tuning requires fewer epochs and hence can speed up the overall run-time considerably.

4.4.4 Compute vs. Communication Cost

Figure 4.9 shows the computation time as compared to the communication time for the inference of a single input over different network sizes. We observe that the computation cost increases with the network size and becomes the dominant reason for the performance overhead in private deep learning with Falcon. The reason for

this is because the complexity of matrix multiplication is "super-quadratic" i.e., to multiply two $n \times n$ matrices, the computation overhead is strictly larger than $O(n^2)$. Note that the communication of the matrix multiplication protocol in this work is only linear in the size of the matrices and has a round complexity of a single round. On the other hand, the non-linear operations, though more communication expensive in MPC, are applied on vectors of size equal to the output of the matrix product and thus are "quadratic." In other words, the non-linear operations such as ReLU are applied on the output of the matrix multiplication (FC/Conv layers) and are applied on vectors of size $O(n^2)$ assuming they are applied on the output of the multiplication of two $n \times n$ matrices. Hence, for large network architectures, the time required for the matrix-multiplication dominates the overall cost.

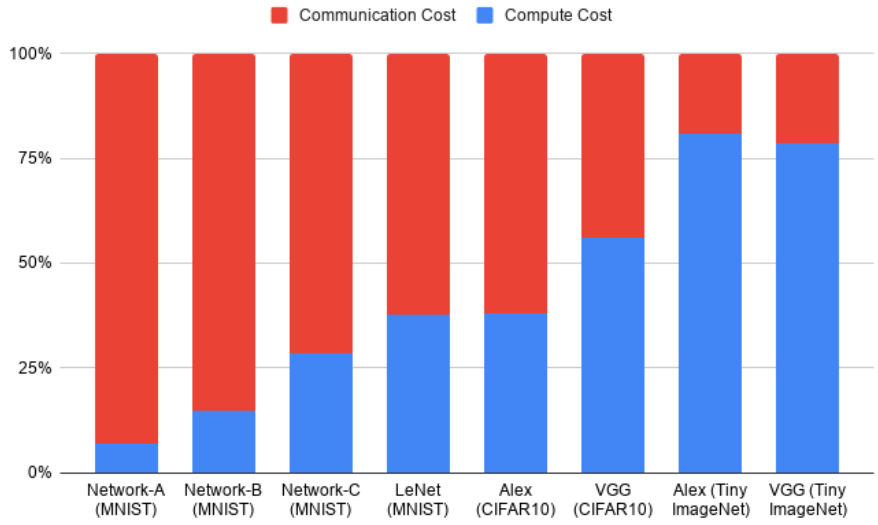


Figure 4.9: Computation vs. communication cost for private inference using FALCON in a WAN deployment for the malicious adversary setting. It is interesting to note that as the network size increases, computation becomes a dominant factor in the overall end-to-end run-time.

This observation is against the conventional wisdom that MPC protocols are communication bound and not computation bound. When running larger networks such as AlexNet and VGG16, and especially for Tiny ImageNet, the computation time starts becoming a significant fraction of the total time. Hence, we claim that FALCON is optimized for communication rounds, specifically when operating over large networks. With our results, we motivate the community to focus on designing faster compute solutions using accelerators such as GPUs, parallelization, efficient matrix multiplications and caching, along with the conventional goals of reducing communication and round complexity.

4.4.5 Comparison vs. Plaintext Computation

Given the surprising insights from Figure 4.9, we also compare the execution of privacy-preserving computations with plaintext computations. These results are sum-

marized in Table 4.7. We use standard PyTorch libraries for the plaintext code, similar hardware as that of privacy-preserving benchmarks for CPU-CPU comparison, and use a single Nvidia P100 GPU for the GPU-CPU comparison. Our findings indicate that private deep learning (over CPU) is within a factor of 40×-1200× of plaintext execution of the same network over CPU and within 50×-four orders of magnitude that of plaintext execution over GPU (using PyTorch) when performed over LAN. The overhead further increases by 1.2×-2.4× when comparing against WAN evaluations. This indicates the importance of supporting GPUs and optimizers for private deep learning and showcases the need for further reducing the overhead of MPC protocols. We believe that it is beneficial for the broader research community to have an estimate of the gap between plaintext and privacy-preserving techniques for realistic size networks and datasets.

Table 4.7: Comparison of private computation (for semi-honest protocols, cf Section 4.4.1 for network parameters) with plaintext over the same hardware using PyTorch and a single NVIDIA P100 GPU. Numbers are for a 128 size batch in milliseconds.

	Run-type		CIFAR-10				Tiny ImageNet			
			Training		Inference		Training		Inference	
			AlexNet	VGG-16	AlexNet	VGG-16	AlexNet	VGG-16	AlexNet	VGG-16
Plaintext	CPU-only	localhost	1.6×10^2	7.3×10^2	7.2×10^1	3.4×10^2	5.0×10^2	3.1×10^3	2.5×10^2	1.3×10^3
	GPU-assisted	localhost	2.8×10^1	6.4×10^1	3.8×10^1	5.8×10^1	3.6×10^1	1.2×10^2	3.8×10^1	5.7×10^1
Private	CPU-only	LAN	6.4×10^3	2.5×10^5	5.6×10^3	1.0×10^5	6.3×10^5	9.5×10^5	2.3×10^5	4.0×10^5
	CPU-only	WAN	2.4×10^4	6.2×10^5	1.7×10^4	1.6×10^5	7.8×10^5	1.2×10^6	3.1×10^5	5.9×10^5
Private	Bandwidth		6.4×10^3	2.5×10^5	5.6×10^3	1.0×10^5	6.3×10^5	9.5×10^5	2.3×10^5	4.0×10^5

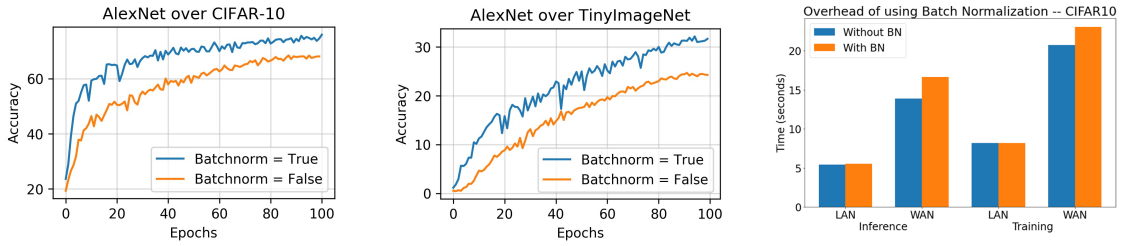
4.4.6 Batch Normalization and Accuracy

We study the benefits of batch normalization for privacy-preserving training of neural networks. We compute the accuracy of partially trained models after each epoch with and without the batch normalization layers. As seen in Figs. 4.10a, 4.10b, batch normalization layers not only help train the network faster but also train better networks. Fig. 4.10c demonstrates the overhead of MPC protocols with and without batch normalization layers. Given the high round complexity of batch normalization, the gap is significant only in the WAN setting.

We also study the effect of our approximations and smaller datatype on the accuracy of the computation. We compare the evaluation of the networks with 64-bit `float` datatypes over PyTorch against a 32-bit datatype `uint32_t` using fixed-point arithmetic for Falcon. The final layer outputs differ by small amounts (less than 1%) in comparison with the high precision 64-bit computation. As a consequence, as seen in Table 4.8, most networks show no/low loss in the overall neural network accuracy when the computation is performed as fixed-point integers over 32-bit datatype. This is because the final prediction is robust to small relative error in individual values at the output. This also makes the final prediction vector inherently noisy and may provide some defense against model inversion attacks.

Table 4.8: Summary of experiments involving accuracy of NNs using secure computation. The first two columns refer to the plaintext accuracies and relative error refers to the average relative error of one forward pass computation using FALCON.

Network	Training Accuracy	Inference Accuracy	FALCON Inference Accuracy	Relative Error
Network-A	98.18%	97.42%	97.42%	0.471%
Network-B	98.93%	97.81%	97.81%	0.635%
Network-C	99.16%	98.64%	98.64%	0.415%
LeNet	99.76%	99.15%	96.85%	0.965%



(a) (b) (c)

Figure 4.10: In Figs. 4.10a, 4.10b, we study the model accuracy with and without batch norm layers as a function of epochs for AlexNet network. As can be seen, batch normalization not only helps train the network faster but also train better networks. In Fig. 4.10c, we study the performance overhead of running the network (using FALCON) with and without batch norm layers.

4.5 Summary

In this work, we develop new protocols for private training and inference in a honest-majority 3-party setting. Theoretically, we propose novel protocols that improve the round and communication complexity and provide security against maliciously corrupt adversaries with an honest majority. FALCON thus provides malicious security and provides several orders of magnitude performance improvements over prior work. Experimentally, FALCON is the first secure deep learning framework to examine performance over large-scale networks such as AlexNet and VGG16 and over large-scale datasets such as Tiny ImageNet. We also are the first work to demonstrate efficient protocols for batch-normalization which is a critical component of present day machine learning.

4.6 Selected References

[109] Sameer Wagh, Divya Gupta, and Nishanth Chandran. "SecureNN: 3-Party secure computation for neural network training." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2019

[84] Payman Mohassel and Peter Rindal. "ABY^3: A mixed protocol framework for machine learning." In: *ACM Conference on Computer and Communications Security (CCS)*. 2018

[94] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. "XONN: XNOR-based oblivious deep neural network inference." In: *USENIX Security Symposium (USENIX)*. 2019

[85] Payman Mohassel and Yupeng Zhang. "SecureML: A system for scalable privacy-preserving machine learning." In: *IEEE Symposium on Security and Privacy (S&P)*. 2017

[77] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. "Oblivious neural network predictions via MiniONN transformations." In: *ACM Conference on Computer and Communications Security (CCS)*. 2017

[64] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "Gazelle: A low latency framework for secure neural network inference." In: *USENIX Security Symposium (USENIX)*. 2018

[25] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. "EzPC: Programmable, efficient, and scalable secure two-party computation for machine learning." In: 2017

[96] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. "DeepSecure: Scalable provably-secure deep learning." In: *Annual Design Automation Conference*. 2018

[95] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. "Chameleon: A hybrid secure computation framework for machine learning applications." In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2018

Chapter 5

Ponytail: Homomorphic Encryption for Faster Multi-Party Computation

Secure Multiparty Computation (MPC) allows a set of parties to compute over their inputs while keeping them private. Over the span of few decades, this field has turned theoretical ideas into practical implementations that allow one to compute over a billion private multiplications per second [5]. The growth of computing on encrypted data has sparked interest in combining MPC with ML, which allows distrusting parties to perform ML tasks such as evaluating private decision trees and support vector machines [95] or evaluating and training NNs, on their joint data [85, 84, 109, 71, 9].

One important building block in all these works is *secure matrix multiplication*, which is often achieved by computing many dot products $\vec{a} \cdot \vec{b}$. In the case of an honest majority corruption model, this problem has a straightforward solution: parties multiply locally each entry $a_i \cdot b_i$ and then re-randomize the sum $\sum_i a_i \cdot b_i$ to the other parties. Hence, the cost of a dot product is a single opening which is independent of the vector sizes. However, in the case of a dishonest majority corruption mode, the dot product protocol must use some correlated randomness, e.g. *Beaver triples*, for each multiplication since the secret sharing scheme is no longer multiplicative. Such triples require expensive public key operations and a lot of research is focused on computing these triples more efficiently via somewhat homomorphic encryption (SHE) or oblivious transfer [13, 39, 65, 66].

The SPDZ framework [39, 38, 66, 10] is a state-of-the-art protocol for dishonest-majority MPC under one of the strongest adversarial settings – it assumes all-but-one corruption and malicious security, meaning that all parties except one can be controlled by the adversary, and can arbitrarily deviate from the protocol description. Moreover, SPDZ is proven secure under the Universal Composability (UC) framework of Cannetti [21], which means in particular that it is still secure when composed arbitrarily with other MPC protocols. Under the SPDZ framework, even if a fast matrix multiplication algorithm such as Strassen's algorithm is used, securely multiplying two $n \times n$ matrices in SPDZ uses at least $O(n^{2.8})$ authenticated Beaver triples. This is prohibitively expensive when targeting applications with a large number and sizes

of matrix multiplications. For instance, the deep convolutional neural network (CNN) ResNet-50 [59] requires more than 4 billion multiplications of plaintext values[1]. Currently, the best known two-party triple generation algorithm over a 128-bit prime field produces $30,000$ triples per second on modest hardware and requires a communication of 15 kbits per party [66]. Using such an approach, the preprocessing phase for evaluating convolution layers of ResNet-50 will require each party to send 5 TB of data. Our work reduces the communication by a factor of about $121\times$, while keeping the same adversarial setting. The contributions of this work can be summarized as below:

(A) We adopted an idea from SecureML [85] of classical Beaver triples to multiply matrices, integrating this idea within the SPDZ framework. This enables computing any bilinear operation efficiently in a *dishonest majority MPC setting*. We focus on two types of bilinear operations, matrix multiplications and two-dimensional convolutions. We call the correlated randomness 'matrix triple' and 'convolution triple' respectively. We then use the state-of-the-art algorithm for HE matrix multiplication [63] to efficiently generate authenticated matrix triples with low communication complexity. Such algorithms allow us to have a communication cost linear in the size of the input and output, and independent of the complexity of the operation itself, in both offline and online phases. For example, in terms of matrix multiplication of n-by-n matrices, our method reduced the communication from $O(n^3)$ to $O(n^2)$ required by SPDZ, with similar computational overhead.

(B) We introduced some further optimizations to the offline phase of SPDZ:

- We avoid the "sacrifice" procedure in SPDZ via switching to slightly larger HE parameters which supports evaluation of circuits of one additional depth. By doing so, we further saved a factor of (almost) two in overall communication and computation.
- We optimized the zero-knowledge proof of plaintext knowledge (ZKPoPK) in the offline phase of SPDZ, reducing the amortized communication overhead for proving each ciphertext from 2.5 to roughly 1.5.

(C) We demonstrated the concrete efficiency of our protocols for (1) private matrix multiplications and (2) private neural network inference in the two-party case. In the former case, we benchmarked the private matrix multiplications over various matrix sizes while in the latter, we benchmarked evaluation of all convolution layers of ResNet-50, a massive, state-of-the-art NN for image classification with 52 layers. The preprocessing phase improves by a factor of at least 121 compared to SPDZ. We integrated the convolution triples in MP-SPDZ [40] to evaluate the online phase ResNet-50 convolutions. Our approach reduces the online communication overhead from 86.9 GB to only 0.54 GB (for a plaintext modulus $p \approx 2^{128}$), which amounts to a factor of at least $150\times$ improvement over the existing matrix multiplication in SPDZ using Strassen's algorithm.

[1]This is considering the scenario that both the model (i.e., ResNet-50 weights) and inference inputs are secret shared.

5.1 Ponytail Overview

In this section, we introduce the notation and other important concepts central to the contributions of this work. In Section 5.2, we introduce our changes to the SPDZ framework to better support bilinear operations, including an algorithm to generate authenticated matrix triples, an optimization which removes the sacrifice procedure, and optimizations on the ZKPoPK. We go on to present the experimental results for private matrix multiplication, private nearest neighbor search, and pirvate evaluation of ResNet-50 in Section 5.4.

Notation. We use $\vec{x}$ to denote vectors, i.e., $\vec{x} = (x_1, \ldots, x_k)$ for some k specified in the context. We also use the notation $[k]$ to denote the set $\{1, 2, \ldots, k\}$. For a positive integer q, we identify $\mathbb{Z}_q = \mathbb{Z} \cap (-q/2, q/2]$. For a finite set S, $U(S)$ denotes a uniform distribution over S.

Adversarial setting. Our protocols in this work follow the same adversarial setting as SPDZ [39, 38], meaning that they are secure under all-but-one corruption and malicious security (we will refer to this setting as *dishonest majority* for short). Also, our protocol is proven secure under the UC framework, a property inherited from SPDZ.

5.1.1 Authenticated Shares in SPDZ

Let n be the number of parties involved in the MPC. In the SPDZ framework, all computations are performed over the finite field $\mathbb{Z}_p$ with prime p. We use $[\![x]\!]_\alpha$ to denote "authenticated shares", i.e., the i-th party holds (x_i, m_i) such that $x \equiv x_0 + \ldots + x_{n-1} \pmod p$ and $\alpha \cdot x \equiv m_0 + \ldots + m_{n-1} \pmod p$, where α is the global MAC key and is shared between the parties as α_i such that $\alpha \equiv \alpha_0 + \ldots + \alpha_{n-1} \pmod p$. In other words,

$$[\![x]\!]_\alpha := \{(x_i, m_i, \alpha_i)\}_{i=1}^n \text{ such that}$$
$$\sum_i m_i \equiv \left(\sum_i \alpha_i\right) \cdot \left(\sum_i x_i\right) \pmod p \tag{5.1}$$

5.1.2 Bilinear Triples

Beaver's multiplication triple technique is widely used in secure computation in both the semi-honest and malicious adversarial settings [13, 38, 85, 109]. Let $\mathbb{F}$ be a finite field. Recall that a multiplication triple is a tuple $([a], [b], [c])$ where $a, b \in \mathbb{F}$ are random elements such that $c = a \cdot b$. Here $[x]$ represents an additive sharing of x where each party has a share x_i such that $\sum_{i=1}^n x_i = x$. These multiplication triples can be utilized to perform private multiplication: in order to multiply secret-shared values x and y. The parties reveal $x - a$ and $y - b$, and compute $[x \cdot y] =$

$(x-a)\cdot(y-b)+[a]\cdot(y-b)+(x-a)\cdot[b]+[c]$. In the dishonest majority malicious adversarial setting, SPDZ enhances the above to authenticated triples $(\llbracket a\rrbracket,\llbracket b\rrbracket,\llbracket c\rrbracket)$.

Mohassel and Zhang [85] generalized the above notion to "matrix triples" and applied it to secure training of ML models in the semi-honest setting. We take this idea further and consider triples for any bilinear operation in the stronger dishonest majority adversarial setting, integrating our protocol with the SPDZ preprocessing framework.

Bilinear triples. Let l, m, k be positive integers and let $\circledast : \mathbb{F}^l \times \mathbb{F}^m \to \mathbb{F}^k$ be a bilinear function[2]. We define a $\circledast$-*triple* as a tuple of secret sharings $[a], [b], [a \circledast b]$ where a, b are uniformly random. Given such a triple, it is simple to securely compute a secret sharing of $x \circledast y$ given secret sharings of x and y following Beaver's method verbatim. Note that when $\circledast$ is scalar multiplication, we get back Beaver's multiplication triple; when $\circledast$ is matrix multiplication, we get the matrix triple in [85]. Another example is convolution, described in more detail below.

Using $\circledast$-triples instead of Beaver triples for securely computing bilinear operations has an advantage of lower communication cost in the triple consumption phase. For example, multiplying two n-by-n matrices with Beaver triples would cost $O(n^3)$ field elements being communicated, or $O(n^{\log 7+o(1)})$ using Strassen's algorithm, whereas using matrix triple only amounts to $O(n^2)$ communication cost. Importantly, we will see that using $\circledast$-triples can also reduce the communication cost in the *triple generation phase* using HE.

Convolutions. *Convolution* is a bilinear operation between tensors widely used by DNNs [74, 70]. Here we will define and discuss *two-dimensional* convolutions, since they are used by a ResNet-50 network [59] we use for benchmarking, but our approach can be easily generalized to all dimensions.

Let A_{ijk} be an input tensor, where $1 \le i \le h$ and $1 \le j \le w$ are spatial coordinates, and $1 \le k \le s$ is the channel. Suppose we would like to compute an $(2l+1)\times(2l+1)$-convolution for some $l \ge 0$, given by a tensor $B_{\Delta i,\Delta j,k,k'}$, where $-l \le \Delta i, \Delta j \le l$ are shifts of the spatial coordinates, and $1 \le k \le s$ and $1 \le k' \le s'$ are the channels. The resulting tensor $C_{ijk'} = \mathsf{conv}(A, B)$ has $h \times w$ spatial coordinates and s' channels and is defined via the formula:

$$C_{ijk'} = \sum_{\Delta i,\Delta j,k} A_{i+\Delta i,j+\Delta j,k} \cdot B_{\Delta i,\Delta j,k,k'},$$

where in the right-hand side, we set the entries of A to be zero if $i+\Delta i$ or $j+\Delta j$ are outside of the ranges $[1;h]$ and $[1;w]$ respectively. Since convolution is bilinear, we can consider *convolution triples*, that is secret shares of uniformly random tensors A, B and secret shares of $\mathsf{conv}(A, B)$.

We can reduce convolution to matrix multiplication as follows: we create an $wh \times (2l+1)^2 \cdot s$ matrix $\mathcal{A}$ with $\mathcal{A}_{(i,j)(\Delta i,\Delta j,k)} = A_{i+\Delta i,j+\Delta j,k}$, as well as an $(2l+1)^2 \cdot s \times s'$

[2] A function $\circledast$ is called *bilinear* if it satisfies the relations $(\alpha x_1 + x_2) \circledast y = \alpha(x_1 \circledast y) + x_2 \circledast y$ and $x \circledast (\alpha y_1 + y_2) = \alpha(x \circledast y_1) + x \circledast y_2$ for arbitrary $\alpha \in \mathbb{F}$, $x_1, x_2, x \in \mathbb{F}^l$ and $y_1, y_2, y \in \mathbb{F}^k$.

matrix $\mathcal{B}$ defined as: $\mathcal{B}_{(\Delta i,\Delta j,k)k'} = B_{\Delta i,\Delta j,k,k'}$. Then one can extract C from the product $\mathcal{C} = \mathcal{A}\mathcal{B}$ (which is of size $wh \times s'$) as follows: $C_{ijk'} = \mathcal{C}_{(i,j)k'}$. Note that 1×1 convolution ($l = 0$) is exactly matrix multiplication. When $l > 0$, one of the matrices $\mathcal{A}$ is obtained from $(2l+1)^2$ stacked permuted instances of the flattening of A. Overall, using this reduction, we can compute the convolution in $O((2l+1)^2 \cdot whss')$ operations[3]. Thus, evaluating the convolution using the authenticated Beaver triples in SPDZ requires $O((2l+1)^2 \cdot whss')$ communication. In contrast, using our convolution triples yields a communication cost of merely $O((wh+s') \cdot s \cdot (2l+1)^2)$. Sometimes, one is willing to *stride* the convolution. This simply corresponds to the regular sampling of the i, j coordinates of the answer. In terms of matrix multiplications, this corresponds to sampling a subset of rows of $\mathcal{A}$.

5.1.3 Matrix Multiplication Using HE

We recall the protocol from [63] which transforms square matrix multiplications into HE-friendly operations. For a $d \times d$ square matrix $A = (a_{i,j})_{0 \le i,j < d}$, we first define useful permutations σ, τ, ϕ, and ψ on the set $\mathbb{Z}_p^{d \times d}$. For simplicity, we assume that $N/2 = d^2$. All the indices will be considered as integers modulo d. Let $\sigma(A)_{i,j} = a_{i,i+j}$, $\tau(A)_{i,j} = a_{i+j,j}$, $\phi(A)_{i,j} = a_{i,j+1}$, and $\psi(A)_{i,j} = a_{i+1,j}$. Then for two square matrices A, B of order d, we can express the matrix product $A \times B$ as follows:

$$A \times B = \sum_{k=0}^{d-1} \left(\phi^k \circ \sigma(A)\right) \odot \left(\psi^k \circ \tau(B)\right), \tag{5.2}$$

where $\odot$ denotes the component-wise multiplication between matrices (see Section 3.1 of [63] for more detail).

We can identify a matrix of order $d \times d$ with a vector of length d^2 via the encoding map $\mathbb{Z}_p^{d^2} \to \mathbb{Z}_p^{d \times d}$, $\vec{a} = (a_0, \ldots, a_{d^2-1}) \mapsto A = (a_{d \cdot i+j})_{0 \le i,j < d}$. A ciphertext will be called an encryption of A if it is an encryption of the plaintext vector $\vec{a}$. Suppose that we are given two ciphertexts $\mathfrak{c}_A$ and $\mathfrak{c}_B$ that encrypt $\sigma(A)$ and $\tau(B)$, respectively. Then we define the homomorphic matrix product by

$$\mathfrak{c}_A \circledast \mathfrak{c}_B = \sum_{k=0}^{d-1} \left(\phi^k(\mathfrak{c}_A) \boxtimes \psi^k(\mathfrak{c}_B)\right), \tag{5.3}$$

where $\mathfrak{c} \boxtimes \mathfrak{c}'$ denotes the homomorphic multiplication between two ciphertexts $\mathfrak{c}$ and $\mathfrak{c}'$. The permutations ϕ^k and ψ^k are fixed linear transformations over $\mathbb{Z}_p^{d^2}$, which can be evaluated as described above. The evaluation of a permutation includes only two homomorphic rotations since the matrix representation of ϕ^k or ψ^k has two nonzero diagonals. It follows from Eq. (5.2) that $\mathfrak{c}_A \circledast \mathfrak{c}_B$ is an encryption of $A \times B$.

The authors of [63] implemented the matrix multiplication algorithm over the CKKS scheme [30], while we apply the same algorithm to the BFV scheme encrypt-

[3] In principle, one can speed it up using Fourier or Winograd transforms [73], but we leave the study of these algorithms in the secure setting for the future work.

ing two vectors of dimension $(N/2)$ with entries in $\mathbb{Z}_p$. We will encrypt two square matrices A and B of size $d = \sqrt{N/2}$ in a single ciphertext. As noted in Section 2.3.1, the BFV scheme supports parallel arithmetic operations and permutations on two vectors. Hence, we can perform two homomorphic matrix multiplications simultaneously by fully utilizing the slots.

5.2 Protocol Constructions

We describe our major contributions in this section. First, we propose our algorithm for generating authenticated matrix triples. Then, we introduce two other optimizations – the first one improves the triple generation phase, by carefully choosing the HE parameters to avoid the sacrifice stage while the second one improves the ZKPoPK in SPDZ.

5.2.1 Generation of Bilinear Triples

We present an improvement to the SPDZ framework to support efficient bilinear operations, in particular matrix multiplications and convolutions. Recall that the offline phase of the SPDZ framework generates Beaver triples, which means that to multiply two square matrices of size d we need to consume $M(d)$ triples, where $M(d)$ is the complexity of the matrix multiplication algorithm of choice. In order to minimize the communication overhead, we designed a new preprocessing protocol for generating matrix and convolution triples and use HE algorithms to generate these triples. In the online phase, they are consumed in essentially the same way as Beaver triples. Such triples allow us to have communication linear in the size of the input and output, and independent of the number of multiplications, in both offline and online phases.

On a high level, our protocol for generating authenticated matrix triples works as follows. First, each party P_i selects uniformly random matrices A_i, B_i and sends an encryption of these matrices to the other parties. Then, the parties engage in the n-party ZKP to obtain verified encryptions of $A = \sum A_i$ and $B = \sum B_i$ with bounded noise. Next, parties use the homomorphic matrix multiplication algorithm (from Section 5.1.3) to compute an encryption of $C = AB$. Finally, parties use homomorphic multiplication to compute encryptions of $\alpha A, \alpha B, \alpha C$, and perform distributed decryption on the resulting ciphertexts. In this way, the parties end up with a valid authenticated triples $(\llbracket A \rrbracket_\alpha, \llbracket B \rrbracket_\alpha, \llbracket C \rrbracket_\alpha)$. We provide the formal description of our pre-processing protocol in Figure 5.1, with the distributed decryption protocol in Figure 5.2. We divide the presentation of this section into the presentation of the protocols for the offline phase Π_{Prep} and the distributed decryption subroutine Π_{DDec}.

Theorem 5.1. *In the $(\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}_{\mathsf{Commit}})$-hybrid model, the protocol Π_{Online} (Figure C.8) implements $\mathcal{F}_{\mathsf{Online}}$ with statistical security against any static, active adversary corrupting up to $n-1$ parties.*

Π_{Prep}

Usage: We execute Π_{PoPK} by batching u ciphertexts together. At the same time, we use the SIMD properties of HE to optimally compute on N plaintext elements at the same time (cf. Sec 5.4.1). Calls to Π_{PoPK} are amortized in batches of u, a detail omitted for simplicity. Also, randomness used in the encryption is implicit and is the randomness used for a fresh ciphertex (cf. Sec 5.1)

Initialize: All parties first invoke $\mathcal{F}_{\mathsf{KeyGenDec}}$ to obtain the public key pk. Then:

(A) Each party generates $\alpha^i \leftarrow \mathbb{Z}_p$. Let $\alpha := \sum_i \alpha^i \pmod p$.
(B) Each party computes and broadcasts a fresh encryption $\mathfrak{c}_\alpha^i \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\alpha^i)$ (Note that this ciphertext has α^i in all the N slots. Refer Sec. 5.1).
(C) The parties invoke protocol Π_{PoPK} on ciphertexts $\mathfrak{c}_{\alpha_i}$ for $i \in [n]$.
(D) All parties compute $\mathfrak{c}_\alpha \leftarrow \sum_i \mathfrak{c}_\alpha^i$.

Authenticated Singles: Parties run this protocol to generate $u \cdot N$ random authenticated shares in $\mathbb{Z}_p$ in one invocation. Let $i \in [n]$ and $k \in [u]$.

(A) All parties sample random $r_k^i \leftarrow U(R_p)$. Each party computes and broadcasts $\mathfrak{c}_{r_k}^i = \mathsf{Enc}_{\mathsf{pk}}(r_k^i)$. Let $\mathfrak{c}_{r_k} \leftarrow \sum_i \mathfrak{c}_{r_k}^i$.
(B) The parties invoke protocol Π_{PoPK} on the u ciphertexts $\mathfrak{c}_{r_k}^i$.
(C) Parties run Π_{AddMacs} to generate $\left(\gamma(r_k)^1, \ldots, \gamma(r_k)^n\right) \leftarrow \mathsf{AddMacs}(\mathfrak{c}_{r_k})$.
(D) Parties output $[\![r_k]\!]_\alpha = \left(\left(r_k^1, \gamma(r_k)^1\right), \ldots, \left(r_k^n, \gamma(r_k)^n\right)\right)$.

Matrix Triples: For the ease of exposition, we encode one matrix in one ciphertext. Refer to Section 5.4.1 for more details on how to optimally use all the ciphertext slots. Let $\circledast$ refer to the HE ciphertext-ciphertext matrix multiplication relation defined in Section 5.1.3. Let $j \in [d_1], k \in [d_2]$, and $l \in [d_3]$. Steps (A)-(J) are done for all j, k, l in their respective ranges. Set $v = (\mathsf{sec_s} + 2)/\log_2(2N+1)$

(A) Each party generates random $A_{jk}^i \leftarrow U(R_p)$ and $B_{kl}^i \leftarrow U(R_p)$.
(B) Compute and broadcast $\mathfrak{c}_{A_{jk}}^i \leftarrow \mathsf{Enc}(\sigma(A_{jk}^i))$ and $\mathfrak{c}_{B_{kl}}^i \leftarrow \mathsf{Enc}(\tau(B_{kl}^i))$.
(C) All parties invoke Π_{PoPK} for $\mathfrak{c}_{A_{jk}}^i$ and $\mathfrak{c}_{B_{kl}}^i$ for each $i \in [n]$.
(D) All parties set $\mathfrak{c}_{A_{jk}} \leftarrow 2 \cdot \sum_i \mathfrak{c}_{A_{jk}}^i$ and $\mathfrak{c}_{B_{kl}} \leftarrow 2 \cdot \sum_i \mathfrak{c}_{B_{kl}}^i$.
(E) All parties compute $\mathfrak{c}_{C_{jl}} \leftarrow \sum_k \mathfrak{c}_{A_{jk}} \circledast \mathfrak{c}_{B_{kl}}$.
(F) Parties run Π_{AddMacs} to generate $(\gamma(A_{jk})^1, \ldots \gamma(A_{jk})^n) \leftarrow \mathsf{AddMacs}(\mathfrak{c}_{A_{jk}})$ and $(\gamma(B_{kl})^1, \ldots \gamma(B_{kl})^n) \leftarrow \mathsf{AddMacs}(\mathfrak{c}_{B_{kl}})$.
(G) Parties run Π_{DDec} to generate $\left(C_{jl}^1, \ldots C_{jl}^n\right) \leftarrow \mathsf{DDec}(\mathfrak{c}_{C_{jl}})$.
(H) Parties run Π_{AddMacs} to generate $\left(\gamma(C_{jl})^1, \ldots \gamma(C_{jl})^n\right) \leftarrow \mathsf{AddMacs}(\mathfrak{c}_{C_{jl}})$.
(I) Set $A_{jk}^i \leftarrow 2 \cdot A_{jk}^i$ and $B_{kl}^i \leftarrow 2 \cdot B_{kl}^i$.
(J) Generate matrix A by using A_{jk}^i as sub-matrix blocks – k blocks per row and j blocks per column. This forms a matrix of dimensions $(d_m \cdot \mathsf{block\ size})$ where $m \in \{1, 2\}$ Similarly, rearrange the $\gamma(A_{jk})^i$ and set this group of 2 matrices as $[\![A]\!]$. Similarly, set $[\![B]\!]$ and $[\![C]\!]$ (no scaling by 2 for C).

Figure 5.1: Protocol for generating various preprocessing material

Π_{DDec}

Distributed Decryption: Parties run the following protocol:

(A) Parties generate $r^i \leftarrow U(R_p)$. Let $\mathfrak{c}_m := (\mathfrak{c}_0, \mathfrak{c}_1)$.

(B) Compute v^i as follows:

$$v^i = \begin{cases} \mathfrak{c}_0 + \mathfrak{c}_1 \cdot s^i & \text{if } i = 1 \\ \mathfrak{c}_1 \cdot s^i & \text{if } i \neq 1 \end{cases}$$

(C) Broadcast $t^i \leftarrow \Delta \cdot r^i + v^i + e^i \pmod q$ where $e^i \leftarrow U(R_{B \cdot 2^{\mathsf{sec}_{\mathsf{dd}}}})$.

(D) Party $i = 1$ outputs $m^1 = \lfloor \Delta^{-1} \cdot (\sum_i t^i) \rceil - r^1 \pmod p$ while all other parties $(i \neq 1)$ output $m^i = -r^i \pmod p$.

(E) Finally, $\mathsf{Decode}(m^i)$ to obtain of vector of plaintexts encoded in each m^i.

Figure 5.2: Protocol for distributed decryption.

Theorem 5.2. *If the underlying cryptosystem is somewhat homomorphic and IND-CPA secure, then* Π_{Prep} *(Figure 5.1) implements* $\mathcal{F}_{\mathsf{Prep}}$ *with computational security against any static, active adversary corrupting up to* $n - 1$ *parties, in the* $(\mathcal{F}_{\mathsf{KeyGen}}, \mathcal{F}_{\mathsf{Rand}})$*-hybrid model.*

Theorem 5.3. *The protocol* Π_{DDec} *securely implements* $\mathcal{F}_{\mathsf{KeyGenDec}}$ *in the* $\mathcal{F}_{\mathsf{KeyGen}}$*-hybrid model with statistical security against any static adversary corrupting upto* $n - 1$ *parties if* B' *is an upper bound on the noise of the input ciphertext, and* $B' \cdot 2n \cdot 2^{\mathsf{sec}_{\mathsf{dd}}} < \Delta$.

Proof of Theorems 5.1, 5.2, and 5.3 are presented in Appendix C.

5.2.2 Authenticating Triples Without Sacrifice

To introduce this optimization, we first recall the technique of authenticated multiplication triples as proposed by the SPDZ line of work [39, 38]. In the framework, there is a global MAC key $\alpha \in \mathbb{F}_p$ and parties have access to a ciphertext $\mathfrak{c}_\alpha$ encrypting α, here the ciphertext is generated via an HE scheme, whose public key is known to all parties and the secret key is secret-shared among the parties[4]. During the triple generation phase, parties obtain ciphertexts $\mathfrak{c}_x, \mathfrak{c}_y, \mathfrak{c}_z$ where supposedly the relation $z = xy$ holds. In order to authenticate the secret values x, y and z, the parties engage in an $\mathsf{AddMacs}$ subroutine (this is a common procedure to prevent malicious behavior for dishonest majority protocols, cf [39, 38]), in which parties compute and then jointly decrypt $\mathfrak{c}_\alpha \boxtimes \mathfrak{c}_t$ to obtain secret shares of $\alpha \cdot t$ for $t \in \{x, y, z\}$. However, a malicious adversary can inject an error term ϵ into z such that $z = xy + \epsilon$, and the $\mathsf{AddMacs}$ subroutine could authenticate such an incorrect triple, which corrupts the final computation result. In order to resolve this issue, a step called *sacrifice*

[4]The initialize phase in Π_{Prep} will require Diag flag similar to [39, 38] to ensure that the ciphertext encodes the same MAC key in the same slots.

was introduced, where one triple is consumed to check the correctness of the other. Sacrificing brings a two times overhead to the complexity of the triple generation phase.

We begin by noting that SPDZ only uses a depth-1 HE, i.e., the underlying HE scheme could support one multiplication. Recall that in the SPDZ triple generation, after computing a ciphertext $\mathfrak{c}_z = \mathfrak{c}_x \boxtimes \mathfrak{c}_y$, the Reshare procedure is called which outputs secret shares of z' and a new ciphertext $\mathfrak{c}_{z'}$ with smaller noise than $\mathfrak{c}_z$. Then, the AddMacs procedure is called, which produces authenticated share $[\![z']\!]_\alpha$. In particular, to generate shares of the MAC on z, prior work requires that the distributed decryption subroutine to be called on z to get a level-1 ciphertext (z') that enables adding the MAC on it. This way, an additive error introduced in z can be "authenticated" using the AddMacs procedure by the adversary. To prevent against such an attack, prior work required a sacrifice of one triple with other which was proved to ensure that the triples do not have an error. The MacCheck ensures that any such additive error introduced is caught with high probability.

In our work, we modify the HE parameters to support larger depth, in particular depth-2 computation. The homomorphic encryption product ($z = xy$) is done over public ciphertexts and hence z is guaranteed to equal xy. However, to add MACs to the product z, we do not need to run a distributed decryption protocol (we only need it for generating the shares of z but *not* for the MAC generation). In our work, we directly call the AddMacs routine on the public ciphertext for z, i.e., $\mathfrak{c}_{\alpha z} = \mathfrak{c}_z \boxtimes \mathfrak{c}_\alpha$, and perform distributed decryption on $\mathfrak{c}_{\alpha z}$ to obtain the MAC shares. This ensure that the additive error introduced by the adversary when running DDec on $\mathfrak{c}_z$ to get shares of z is *independent* of α from the additive error introduced in the DDec of $\mathfrak{c}_{\alpha z}$. This way, we eliminate the need for a sacrifice and simply rely on the MacCheck subroutine to catch malicious behavior.

Thus, we save the computation and communication by a factor of two, with a less-than-two additional overhead due to the need to increase underlying HE parameters to support larger depth computations. This optimization is particularly useful in our bilinear triple generation protocol, since in this case we already need to increase the HE parameters in order to run the homomorphic matrix multiplication algorithm, and the overhead of supporting just one more depth is small.

5.2.3 Improved ZKPoPK Based on BFV Scheme

In the SPDZ offline phase, parties need to use a HE scheme (the BGV scheme of Brakerski, Gentry, and Vaikuntanathan [17]) to encrypt of random values, and broadcast these encryptions. Then, they run homomorphic evaluation and distributed decryption to generate the multiplication triples. Since parties could be malicious, each party needs to prove that it is providing a valid ciphertext. In the context of BGV, this means the coefficients of the message and randomness used in the encryption method must be bounded in size. The ZKPoPK to validate these ciphertext follows a 3-move Schnorr protocol pattern. The goal is to prove knowledge of message x and encryption randomness r with bounded size, such that $\mathfrak{c}_{x,r} = \mathfrak{b}$. The prover chooses some random mask values y_x, y_r and sends $\mathfrak{c}_{y_x,y_r}$ to the verifier. After the verifier

selects a challenge e the prover sends back the masked values $z_x = y_x + e \cdot x$ and $z_r = y_r + e \cdot r$. Finally, the verifier checks whether $\mathfrak{c}_{z_x,z_r} = \mathfrak{c}_{y_x,y_r} + e \cdot \mathfrak{b}$ and whether the noise and plaintext bounds are correct on producing $\mathfrak{c}_x$ by checking the norm of z_x and z_r. The state-of-the-art ZKPoPK in [10] enhances the above approach by designing an n-prover protocol which adds the ability to prove the validity of sum of n ciphertexts instead of proving each individual ones.

Our modification. We note that the BFV HE scheme of Brakerski/Fan-Vercauteren [16, 47] provides the same functionalities as the BGV scheme, while the two schemes have some subtle differences, which we will exploit for our improved ZKP. In particular, BFV allows selecting the plaintext modulus p to divide the ciphertext modulus q, which is not allowed in BGV[5]. We will use this fact to simplify and reduce the complexity of the ZKPoPK component in SPDZ. Recall that the BGV encryption of a message m with public key pk and randomness (u, e_0, e_1) is

$$\mathfrak{c} = u \cdot \mathsf{pk} + (m + pe_0, pe_1) \pmod{q}. \tag{5.4}$$

Although an honest party would encrypt a message $m \in R_p$ with $\|m\|_\infty \leq p/2$, a malicious party can use any $m \in R$, and the excess part $m - [m]_p$ goes into the noise of the ciphertext. Hence the prover needs to prove that $\|m\|_\infty$ is not too large. This is done by having the prover send encryptions of random messages y with $\log \|y\|_\infty \approx \mathsf{sec_{zk}} + \log p$ and later reveal a linear combination of y and m. On the other hand, in the BFV scheme, an encryption of m is the form of

$$\mathfrak{c} = u \cdot \mathsf{pk} + (\Delta \cdot m + e_0, e_1) \pmod{q}, \text{ where } \Delta = \lfloor q/p \rceil. \tag{5.5}$$

Suppose p divides q, then $\Delta = q/p$ exactly, and using a message $m \in R$ in the encryption algorithm is equivalent to using $[m]_p$ due to the automatic reduction modulo q on the ciphertexts. Therefore, the prover in our ZKPoPK only needs to prove upper bounds on the encryption randomness, and it suffices to sample the "masking elements" y as random elements in R_p. This reduces the size of the proof, since we reduce the coefficients of the masked plaintexts sent by the prover (the terms z_i in [10, Figure 1]) from $\log p + \log \mathsf{sec_{zk}}$ bits down to $\log p$ bits.

ZKPoPK. The zero-knowledge proof of plaintext knowledge we describe next (Figure 5.3) is a n-party ZKP used in the preprocessing phase. The n players all simultaneously act as the provers and the verifiers. Sampling is an algorithm that describes the behavior of honest parties to generate their ciphertexts and broadcast them to the other parties. This algorithm satisfies the relation given in Eq. 5.6. However, Π_{PoPK} provides weaker guarantees as given in Eq. 5.7 which will be sufficient for the preprocessing phase[6]. In particular, the protocol introduces a *soundness slack* in the

[5] $\gcd(p, q) = 1$ is required for security of BGV

[6] This is the worst case gaurantee when all provers are dishonest while at least one verifier is honest, which in the case when provers and verifiers are the same entities is the dishonest majority model.

bounds that can be proven on the witness. The protocol works in the standard 3-move Schnorr protocol pattern as described below:

(A) Each party P_i independently runs the "commitment" algorithm on (x_i, w_i) to get $(\mathsf{comm}_i, \mathsf{state}_i) \leftarrow \mathsf{Commit}(x_i, w_i)$ and broadcasts comm_i to all the other parties.

(B) The n parties jointly generate a challenge w (produced via a call to an ideal functionality $\mathcal{F}_{\mathsf{Rand}}$)

(C) Each party P_i independently runs the "response" algorithm to get $\mathsf{resp}_i \leftarrow \mathsf{Response}(\mathsf{state}_i, w)$

(D) Each party P_i independently runs the "verification" algorithm and accept if the output is true: $\mathsf{Verify}(\{\mathsf{comm}_i, \mathsf{resp}_i\}_{i\in[n]}, w) == \mathsf{True}$.

$$\mathcal{R}^{u,\mathsf{Honest}}_{\mathsf{PoPK}} = \Big\{ \big((x^1,\ldots,x^n),(w^1,\ldots,w^n)\big), \; x^i = \big(\mathfrak{c}^i_1,\ldots,\mathfrak{c}^i_u\big), w^i = \big((a^i_1, r^i_{a_1}),\ldots(a^i_u, r^i_{a_u})\big) : \mathfrak{c}_{a_k} = \mathsf{Enc}_{\mathsf{pk}}(a_k, r_{a_k}) \text{ and } \|r_{a_k}\| \le n \text{ where } \mathfrak{c}_{a_k} = \sum_i \mathfrak{c}^i_{a_k} \text{ and } r_{a_k} = \sum_i r^i_{a_k} \Big\} \tag{5.6}$$

$$\mathcal{R}^{u,2}_{\mathsf{PoPK}} = \Big\{ \big((x^1,\ldots,x^n),(w^1,\ldots,w^n)\big), \; x^i = \big(\mathfrak{c}^i_1,\ldots,\mathfrak{c}^i_u\big), w^i = \big((a^i_1, r^i_{a_1}),\ldots(a^i_u, r^i_{a_u})\big) : 2\cdot\mathfrak{c}_{a_k} = \mathsf{Enc}_{\mathsf{pk}}(2\cdot a_k, 2\cdot r_{a_k}) \text{ and } \|2r_{a_k}\| \le Nnu\cdot 2^{\mathsf{sec_{zk}}+1} \text{ where } \mathfrak{c}_{a_k} = \sum_i \mathfrak{c}^i_{a_k} \text{ and } r_{a_k} = \sum_i r^i_{a_k} \Big\} \tag{5.7}$$

Before we describe the protocol, we reiterate some key notation. The *normalized norm* of randomness r_m by $\|r_m\| = \max\{\|u\|_\infty, \rho^{-1}\cdot\|e_0\|_\infty, \rho^{-1}\cdot\|e_1\|_\infty\}$. For $B > 0$, we call $\mathfrak{c}$ a B-*ciphertext* if there exists $m \in R_p$ and $r_m = (u, e_0, e_1) \in R^3$ such that $\|r_m\| \le B$ and $\mathfrak{c} = \mathsf{Enc}_{\mathsf{pk}}(m, r_m)$. We also use U_B to denote a uniform distribution over the set of triples $r = (u, e_0, e_1) \in R^3$ such that $\|r\| \le B$. We set $\rho = 20$ following [10] to ensure the randomness r from an honest party satisfies $\|r\| \le 1$ with overwhelming probability. Furthermore, we also use the following distributions (specifically the third) in the description of the protocol:

(A) $\mathcal{ZO}(0.5, k)$: This distribution generates a vector of size k with elements $\{x_i\}_{i=1}^k$ chosen from $\{-1, 0, +1\}$ such that the $\Pr(x_i = -1) = 0.25, \Pr(x_i = +1) = 0.25$, and $\Pr(x_i = 0) = 0.5$ for all $i \in [k]$.

(B) $\mathcal{DN}(\sigma^2, k)$: This distribution generates a vector of size k with elements drawn according to an approximation to the discrete Gaussian distribution with variance σ^2.

(C) $\mathcal{RG}(0.5, \sigma^2, k)$: This distribution generates a triple of elements (u, e_0, e_1) where $u \leftarrow \mathcal{ZO}(0.5, k)$ and $e_0, e_1 \leftarrow \mathcal{DN}(\sigma^2, k)$.

Improvements compared to prior work. In our protocol, the hiding on the message (z_l^i) is information-theoretic (as opposed to statistical hiding in TopGear [10]) and hence does not need any check during the verification phase. This is due choosing $p \mid q$ in underlying BFV scheme. In addition, the ZKPoPK in [10] sends the polynomials z_l^i and $r_{z_l}^i$ as elements in R_q, which is more than necessary since q is typically large but these polynomials are supposed to have bounded norm. We can reduce this cost by sending z_l^i and $r_{z_l}^i$ in bounded size (since $z_l^i \in U(R_p)$ and all the coefficients of $r_{z_l}^i$ should be bounded by $u \cdot 2^{\mathsf{sec_{zk}}}$ or $\rho \cdot u \cdot 2^{\mathsf{sec_{zk}}}$). In this way, we can also omit the check on size of r_{z_l} in Step 3 of Verify phase.

Note that the "slack" in the ZKP provides looser bounds on the norms of values as well as multiplied the values themselves by a factor of 2. This is a consequence of the ZKP. Figure 5.1 shows how to account for this by modifying the preprocessing protocol to takes these slacks into consideration. The above describes the ZKP protocol. We define the security of the ZKPoPK similar to prior work [10] and present it below for completeness.

Theorem 5.4. *The n-party ZKPoPK-protocol defined by Π_{PoPK} satisfies the following three properties:*

(A) ***Correctness:*** *If all parties P_i, with inputs sampled using the* Sampling *algorithm (in Π_{PoPK}, Figure 5.3), follow the protocol honestly, then an honest verifier will accept with probability one.*

(B) ***Soundness:*** *Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$ be a tuple of PPT algorithms and let $\epsilon \in [0, 1)$. Consider the following game:*

- *(1a)* *$\mathcal{A}_1$ takes no input and outputs $I \subset [n], \{x_i\}_{i \in I}$ and state_{A_1}.*
- *(1b)* *Choose $(x_j, w_j) \leftarrow \mathsf{Sampling}(j)$ for each $P_j, j \notin I$.*
- *(1c)* *Compute $(\mathsf{comm}_j, \mathsf{state}_j) \leftarrow \mathsf{Commit}(x_j, w_j)$ for $j \notin I$.*
- *(2a)* *$\mathcal{A}_2$ on input $\mathsf{state}_{A_1}, \{x_j, \mathsf{comm}_j\}_{j \notin I}$ output $\mathsf{state}_{A_2}, \{\mathsf{comm}_i\}_{i \in I}$.*
- *(3a)* *Choose a uniformly random w and compute $\mathsf{resp}_j \leftarrow \mathsf{Response}(\mathsf{state}_j, w)$ for $j \notin I$.*
- *(4a)* *$\mathcal{A}_3$ on input $\mathsf{state}_{A_2}, w, \{\mathsf{resp}_j\}_{j \notin I}$ outputs $\{\mathsf{resp}_i\}_{i \in I}$.*
- *(4b)* *$\mathcal{A}$ wins the game if $\mathsf{Verify}(\{\mathsf{comm}_i, \mathsf{resp}_i\}_{i \in [n]}, w) = \mathsf{True}$.*

Suppose $\mathcal{A}$ wins the game with probability $\delta > \epsilon$. Then there exists a PPT algorithm Extract *which for any fixed output of $\mathcal{A}_1$, honestly generated inputs given by $\{x_j, w_j, \mathsf{comm}_j, \mathsf{state}_j\}_{j \notin I}$, and black-box access to $\mathcal{A}_2, \mathcal{A}_3$ outputs $\{w_i\}_{i \in I}$ such that $\mathcal{R}^{u,2}_{\mathsf{PoPK}}$ (Eq. 5.7) holds in at most $f(\mathsf{sec_s})/(\delta - \epsilon)$ steps, where $f(\cdot)$ is a positive polynomial and $\epsilon = 2^{-\mathsf{sec_s}}$ ($\mathsf{sec_s}$ is the soundness security parameter).*

(C) ***Honest-verifier zero knowledge:*** *There exists a PPT algorithm $\mathcal{S}_I$ indexed by a set $I \subset [n]$, which takes as input an element in the language given by relation*

Π_{PoPK}

Proof of Plaintext Knowledge (PoPK): This protocol is run between n parties – each acting as a prover and verifier simultaneously. The protocol flow is a standard three-move structure (commitment, challenge, and response) called Σ-protocol with a single challenge produced using an ideal functionality $\mathcal{F}_{\mathsf{Rand}}$. Let u, v be two proof parameters, $\mathsf{Flag} \in \{\mathsf{Diag}, \perp\}$. We use i to denote party index and k, l for variables iterating across ciphertexts ($k \in [u], l \in [v]$). Let n denote the number of parties and N denote the degree of the cyclotimic polynomial used for HE. Ensure that $v \geq (\mathsf{sec_s}+2)/\log_2(2N+1)$.

Sampling **(Sampling phase)**

(A) On input $i \in [n]$, if $\mathsf{Flag} = \perp$ sample $a_k^i \leftarrow U(R_p)$ for each $k \in [u]$. If $\mathsf{Flag} = \mathsf{Diag}$, sample a_k^i as a random diagonal element in $U(R_p)$ for each $k \in [u]$.

(B) Generate $r_{a_k}^i \leftarrow \mathcal{RG}(0.5, \sigma^2, N)$.

(C) Compute ciphertexts $\mathfrak{c}_{a_k}^i = \mathsf{Enc_{pk}}(a_k^i, r_{a_k}^i)$.

(D) Define vectors $\vec{\mathfrak{c}}_a = (\mathfrak{c}_{a_1}^i, \ldots, \mathfrak{c}_{a_u}^i)$, $\vec{a}^i = (a_1^i, \ldots a_u^i)$ and $\vec{r}_a^i = (r_{a_1}^i, \ldots r_{a_u}^i)$. Output $(x^i, w^i) = (\vec{\mathfrak{c}}_a^i, (\vec{a}^i, \vec{r}_a^i))$.

Commit **(Commitment phase)**

(A) Party P_i generates v ciphertexts $\mathfrak{c}_{y_l}^i = \mathsf{Enc_{pk}}(y_l^i, r_{y_l}^i)$ where $l \in [v], y_l^i \leftarrow U(R_p)$, and $r_{y_l}^i \leftarrow U_{u \cdot 2^{\mathsf{sec_{zk}}}}$.

(B) Party P_i broadcasts a commitment $\mathsf{comm}_i \leftarrow \{\mathfrak{c}_{y_l}^i\}_{\forall l}$.

Challenge **(Challenge phase)**

(A) Parties call $\mathcal{F}_{\mathsf{Rand}}$ to obtain a $v \times u$ challenge matrix w with random entries. If $\mathsf{Flag} = \perp$, entries of w come from $\{\pm X^j\}_{0 \leq j < N} \cup \{0\}$. If $\mathsf{Flag} = \mathsf{Diag}$, entries of w come from $\{0, 1\}$.

Response **(Response phase)**

(A) Party P_i computes $z_l^i = y_l^i + (w \cdot \vec{a}^i)_l$ and $r_{z_l}^i = r_{y_l}^i + (w \cdot \vec{r}_a^i)_l$.

(B) Party P_i sets $\mathsf{resp}_i \leftarrow \{z_l^i, r_{z_l}^i\}_{\forall l}$ and broadcasts resp_i.

Verify **(Verification phase)**
Each party then performs the following computations and verifications:

(A) Compute $\mathfrak{c}_{z_l}^i = \mathsf{Enc_{pk}}(z_l^i, r_{z_l}^i)$.

(B) Compute $\vec{\mathfrak{c}}_a \leftarrow \sum_i \vec{\mathfrak{c}}_a^i$, $\mathfrak{c}_{y_l} \leftarrow \sum_i \mathfrak{c}_{y_l}^i$, $\mathfrak{c}_{z_l} \leftarrow \sum_i \mathfrak{c}_{z_l}^i$, $z_l \leftarrow \sum_i z_l^i$, and $r_{z_l} \leftarrow \sum_i r_{z_l}^i$.

(C) Verify $\mathfrak{c}_{z_l} = \mathfrak{c}_{y_l} + (w \cdot \vec{\mathfrak{c}}_a)_l$ and $\|r_{z_l}\| \leq n \cdot u \cdot 2^{\mathsf{sec_{zk}}}$.

(D) If $\mathsf{Flag} = \mathsf{Diag}$ then additionally verify that z_l is a diagonal plaintext element.

(E) If all checks pass, parties accept otherwise they reject.

Figure 5.3: Protocol for zero-knowledge proof of plaintext knowledge.

$\mathcal{R}_{\mathsf{PoPK}}^{u,\mathsf{Honest}}$ *(Eq. 5.6) and a challenge* w*, and outputs tuples* $\{\mathsf{comm}_i, \mathsf{resp}_i\}_{i\in I}$ *such that this output is statistically indistinguishable from a valid execution of the protocol (the statistical indistinguishability parameter is denoted by* $\mathsf{sec}_{\mathsf{zk}}$*).*

Next, we present the proof of Theorem 5.4.

5.3 Theoretical Analysis

In this section, we prove the security of the ZKPoPK. We split the proof into the 3 components – completeness, soundness, and the zero-knowledge property.

5.3.1 ZKPoPK: Security Proof

Completeness

For completeness, a true statement must be verified correctly when both the prover and verifier are honest. In this case, completeness follows directly from the construction as the relation $\mathfrak{c}_{z_l} = \mathfrak{c}_{y_l} + (w \cdot \vec{\mathfrak{c}}_a)_l$ is linear in its arguments and works component-wise as well as from the fact that the BFV encryption procedure is linear in the message and the randomness. The noise bound (in Verify 3 of Figure 5.3) is obtained by:

$$\begin{aligned} \|r_{z_l}\| = \left\| \sum_i r^i_{z_l} \right\| &\leq \sum_i \left(\left\| r^i_{y_l} \right\| + \left\| (w \cdot \vec{r}^{\,i}_a)_l \right\| \right) \\ &\leq nu \cdot 2^{\mathsf{sec}_{\mathsf{zk}}} \end{aligned} \tag{5.8}$$

where the last equality holds with an overwhelming probability since $\|(w \cdot \vec{r}^{\,i}_a)_l\| \leq u$ and $r^i_{y_l}$ is a sample from $U_{u\cdot 2^{\mathsf{sec}_{\mathsf{zk}}}}$.

Zero-Knowledge

To prove zero-knowledge, we need to show that for a true statement, the verifier learns nothing more than the fact that the statement is true. This is done by showing that the verifier (in this case all the parties), given access only to the statement to be proven ($\mathfrak{c}_{a_k} = \mathsf{Enc}_{\mathsf{pk}}(a_k, r_{a_k})$) but no access to prover, can produce a transcript that is statistically indistinguishable from the real transcript, in this case, $\{\mathfrak{c}^i_{a_k}\}, \{\mathfrak{c}^i_{y_l}\}, w, \{z^i_l\}, \{r^i_{z_l}\}$ where $k \in [u], l \in [v]$, and $i \in [n]$.

Assuming a set of corrupt parties $A \subset [n]$, we simulate an accepting transcript for the set of honest parties, i.e., P_i where $i \notin A$ by first choosing the challenge matrix w. Once w is fixed, generate $z^i_l \leftarrow R_p$ and $r^i_{z_l} \leftarrow U_{u\cdot 2^{\mathsf{sec}_{\mathsf{zk}}}}$ for $i \notin A$. Finally, compute $\mathfrak{c}^i_{y_l} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(z^i_l, r^i_{z_l}) - (w \cdot \vec{\mathfrak{c}}^{\,i}_a)_l$. Next, we argue that each of $\{r^i_{z_l}\}, \{z^i_l\}$, and $\{\mathfrak{c}^i_{y_l}\}$ has the same distribution in the real and simulated transcripts (w is straightforward and $\{\mathfrak{c}^i_{a_k}\}$ are in the proof statement). $r^i_{z_l}$ has the same distribution in both the transcripts as it is generated from the same distribution except for an additive factor which is from an exponentially smaller distribution. The distributions of z^i_l are uniformly random

elements from R_p and hence are exactly the same. Finally, the distribution of $\mathfrak{c}^i_{y_l}$ is a uniformly random $u \cdot 2^{\mathsf{sec_{zk}}}$-ciphertext in both the real and simulated transcript as $(w \cdot \vec{\mathfrak{c}}^i_a)_l$ is a u-ciphertext.

Soundness

To prove knowledge soundness, we follow the techniques of [14, 10]. Informally, we show that if there exists a prover $\mathcal{P}$ (as a function of the adversarial corruptions) that can succeed with probability $\epsilon > 2^{-\mathsf{sec_s}}$, then there exists a knowledge extractor running in $\mathsf{poly}(\mathsf{sec_s}) \cdot \epsilon^{-1}$ that can extract the witnesses $\{(a^i_k, r^i_{a_k})\}_{k\in[u]}$. We effectively construct a polynomial time extractor $\mathcal{E}_k$ for each witness $(a^i_k, r^i_{a_k})$ and $k \in [u]$ (intuition provided in Figure 5.4). The extractor $\mathcal{E}_k$, which acts as the verifier, given access to such a prover P, performs the following steps:

(i) Send random challenges w to the prover $\mathcal{P}$ until it outputs an accepting transcript. Let us denote this accepting transcript by $(z^i_l, r^i_{z_l})$. This runs in expected time $1/\epsilon$.

(ii) Select a new random challenge $\tilde{w}$ identical to w except the k-th column. This ensures that $w - \tilde{w}$ is a matrix with all zeros except in the k-th column, where the entries are elements of R of the form $a - b \neq 0$ where $a, b \in \{0\} \cup \{\pm X^j\}_{0\leq j<N}$.

(iii) Send challenge matrices to the prover $\mathcal{P}$ until one of two things happen

  (a) A successful transcript is generated with $\tilde{w}$.
  (b) There are $t = \lceil \mathsf{sec_s} \cdot \epsilon^{-1} \rceil$ unsuccessful challenges.

(iv) The extractors aborts in case (iii)(b). In case (iii)(a), the extractor outputs the two successful transcripts along with the challenges.

If the extractor outputs two transcripts successfully, then we can use the resulting two conversations to compute the witness $(a^i_k, r^i_{a_k})$ efficiently. We describe this argument next. However, it is important to note here that the soundness argument is not complete until we show that (1) the above extractor runs in $\mathsf{poly}(\mathsf{sec_s})/\epsilon$ time and (2) aborts with low probability. We break down the proof into the above three steps.

Runtime. The runtime is easiest to argue and follows directly from the description of the extractor.

Probability of aborting. To bound the failure probability of the extractor, we follow the line of argument from [36]. Let w_k denote the k-th column of the challenge matrix w and w_{-k} the rest of the challenge matrix, i.e., w except the k-th column. We construct a binary matrix H such that each row corresponds to a choice of randomness σ used by the prover $\mathcal{P}$ and a choice of challenge w_{-k} and each column corresponds to a choice of w_k. The entry H_{σ,w_{-k},w_k} is 1 if the verifier accepts the transcripts for this random choice σ and challenge w. When the extractor uses $\mathcal{P}$ as a blackbox and submits a random challenge w, it is equivalent to probing an entry in the matrix H. By rewinding the prover $\mathcal{P}$, we can probe another entry in the matrix H in the

$$\begin{bmatrix} \mathfrak{c}^1_{d_1} & \dots & \mathfrak{c}^n_{d_1} \\ \vdots & \ddots & \vdots \\ \mathfrak{c}^1_{d_v} & \dots & \mathfrak{c}^n_{d_v} \end{bmatrix} = \begin{bmatrix} & e_{1k} & \\ \vec{0} & \vdots & \vec{0} \\ & e_{vk} & \end{bmatrix} \times \begin{bmatrix} \mathfrak{c}^1_{a_1} & \dots & \mathfrak{c}^n_{a_1} \\ \\ \vdots & \ddots & \vdots \\ \\ \mathfrak{c}^1_{a_u} & \dots & \mathfrak{c}^n_{a_u} \end{bmatrix}$$

Figure 5.4: Visual aid to assist the exposition of the witness extraction. Here $\mathfrak{c}^i_{d_l} = \mathfrak{c}^i_{z_l} - \tilde{\mathfrak{c}}^i_{z_l}$ and $e = w - \tilde{w}$ is a matrix with zeros everywhere except the k-th column.

same row (same internal randomness, i.e., $\tilde{w}$) and these two transcripts can be used to extract the witness $(a^i_k, r^i_{a_k})$ efficiently.

Now, we look at the number of ones in each row of H. We note that each row has $(2N+1)^v$ entries (the size of the challenge space w_k). A row is called heavy if it contains at least $(\epsilon/2) \times (2N+1)^v$ ones. A simple application of Markov inequality implies that at least half of the ones are located in the heavy rows since ϵ is the ratio of the number of ones to the size of entire matrix H. Setting $v \geq (\mathsf{sec_s}+2)/\log_2(2N+1)$, we get at least $(\epsilon/2) \cdot (2N+1)^v \geq 2$ ones in each of the heavy rows. Now, from the description, it is clear that the extractor aborts in the following two cases:

(A) The first successful challenge is not in a heavy row.

(B) The first successful challenge is in a heavy row but we do not hit another one in $t = \lceil 4\mathsf{sec_s}/\epsilon \rceil$ tries.

The first probability as we just saw is $\leq 1/2$. For second probability, each successful attempt happens with probability $\geq \epsilon/2 - (2N+1)^{-v} > \epsilon/4$. Hence, the probability of aborting from the second case is at most

$$(1 - \epsilon/4)^t < \exp(-t \cdot \epsilon/4) < 2^{-\mathsf{sec_s}} \tag{5.9}$$

Adding these up, the probability that the extractor aborts is $< 1/2 + 2^{-\mathsf{sec_s}}$.

Witness extraction. The final piece of completing the soundness proof is the witness extraction and associated bounds. Given two accepting transcripts $(w, \{z^i_l, r^i_{z_l}\})$ and $(\tilde{w}, \{\tilde{z}^i_l, \tilde{r}^i_{z_l}\})$, we set $\mathfrak{c}^i_{z_l} = \mathsf{Enc}_{\mathsf{pk}}(z^i_l, r^i_{z_l})$ and $\tilde{\mathfrak{c}}^i_{z_l} = \mathsf{Enc}_{\mathsf{pk}}(\tilde{z}^i_l, \tilde{r}^i_{z_l})$. Let us consider the matrix with entries $\mathfrak{c}_{d_l} = \mathfrak{c}_{z_l} - \tilde{\mathfrak{c}}_{z_l}$ and another matrix $w - \tilde{w}$ with 0's everywhere except the k-th column.

We can see that this set of linear constraints allows us to find the witness, one index at a time. In particular, at least one of the $e_{lk} \neq 0$ and consequently, $z^i_l, r^i_{z_l}, \tilde{z}^i_l$, and $\tilde{r}^i_{z_l}$ along with e_{lk} can be used to extract, respectively, the plaintext and randomness a^i_k and $r^i_{a_k}$ (which encrypts to C^i_k). The exact relations can be written as follows:

$$\begin{aligned} a^i_k &= e^{-1}_{lk} \cdot (z^i_l - \tilde{z}^i_l) \\ r^i_{a_k} &= e^{-1}_{lk} \cdot (r^i_{z_l} - \tilde{r}^i_{z_l}) \end{aligned} \tag{5.10}$$

Finally, to estimate the noise, we use the following result from [14]:

Lemma 5.1. *The quantity* $2/(X^i - X^j)$ *for* $0 \leq i \neq j < N$ *is a polynomial in* R *with coefficients in* $\{0, \pm 1\}$.

As a consequence of the above, $\|2/(X^i - X^j)\|_\infty \leq 1$. We use this to bound the norm of $2 \cdot a_k^i$ and $2 \cdot r_{a_k}^i$ from Eq. 5.10. In particular,

$$\left\|2 \cdot r_{a_k}^i\right\| \leq N \cdot \|2/e_{lk}\|_\infty \cdot \left\|r_{z_l}^i - \tilde{r}_{z_l}^i\right\| \leq 2N \cdot u \cdot 2^{\mathsf{sec}_{\mathsf{zk}}}. \tag{5.11}$$

Therefore, $2 \cdot \mathfrak{c}_{a_k}^i = \mathsf{Enc}(2 \cdot a_k, 2 \cdot r_{a_k}^i)$ and $\|2 \cdot r_{a_k}\| \leq Nnu \cdot 2^{\mathsf{sec}_{\mathsf{zk}}+1}$. This completes the proof. □

5.4 Experimental Evaluation

We present our experimental results for the applications of our protocols to private matrix multiplication and neural network inference. We start with describing some further optimizations. Then, we present noise growth estimates for the homomorphic matrix multiplication algorithms, followed by our concrete parameter instantiation, before proceeding to present our experimental results. The main results are presented over 3 application scenarios (1) private matrix multiplications (2) private nearest neighbor search and (3) private inference of ResNet-50.

5.4.1 Evaluation Set-up and Parameter Estimation

Next, we describe the optimization used for the homomorphic matrix multiplication, the general noise estimation bounds, and lastly, describe a choice of parameters that satisfy all these constraints which we use in the following evaluations.

Further Optimizations. On top of the baseline implementation, we apply the following optimization techniques for the homomorphic matrix multiplication.

- A *lazy key-switching* technique can be applied to the last multiplication step of Eq. (5.3). To be precise, we compute tensor products between $\phi^k(\mathfrak{c}_A)$ and $\psi^k(\mathfrak{c}_B)$ and aggregate all the resulting ciphertexts. In the end, the key-switching operation is performed only once to relinearize the output ciphertext.
- The *hoisting* technique of [57] can be applied to our case to reduce the complexity of rotations in the generation of $\phi^k \circ \sigma(A)$ and $\psi^k \circ \tau(B)$. Since there are many rotations done on the same input ciphertext, one can compute the common part of computation that only depend on the input, and therefore it can be significantly faster than applying each rotation separately.
- As described in [63], homomorphic matrix multiplication can be extended to matrices of an arbitrary size. Given the packing structure of BFV (presented in Section 2.3.1, 5.1), the two rows of BFV encoding operate identically and without interference, so it is easy to pack two matrices in a single ciphertext.

Additionally, we can use the interlacing technique of [63] to encrypt multiple matrices in each plaintext row and carry out matrix operations in parallel, thereby amortizing it over many operations. On the other hand, when an input matrix is too large to be encrypted in a single ciphertext, we split it into *block-size* matrices and encrypt them separately in different ciphertexts. A large matrix operation can be expressed as a composition of several block-size matrix operations. Instead of computing block-wise multiplications separately, we precompute and store the permutations of block matrices not to repeat the same computation in individual products.

Noise Estimation of Homomorphic Matrix Multiplication. In order to optimally choose the parameters of the HE scheme, we perform a noise analysis of our algorithms. The noise bounds of ciphertexts are updated during the computation with respect to the following analysis.

- Encryption: Suppose that $\mathfrak{c} = \mathsf{Enc}_{\mathsf{pk}}(m, r_m)$ for a message m and randomness $r_m = (u, e_0, e_1)$ such that $\|r_m\| \leq B$. Then, we have

$$\mathfrak{c}[0] + \mathfrak{c}[1] \cdot s = \Delta \cdot m + (u \cdot e + e_0 + e_1 \cdot s) \pmod{q}$$

  and the encryption noise $e_{enc} = u \cdot e + e_0 + e_1 \cdot s$ is bounded by $\|e_{enc}\|_\infty \leq B\rho(1 + 2N)$. If a ciphertext is honestly generated, then we derive the bound $B_{\mathsf{clean}} = \rho(1 + 2N)$ since $\|r_m\| \leq 1$. However, our ZKPoPK only guarantees that $2\mathfrak{c}_m = Enc_{\mathsf{pk}}(2m, 2r_m)$ for some $\|2r_m\| \leq Nnu \cdot 2^{\mathsf{sec}_{\mathsf{zk}}+1}$ and so the noise of $2\mathfrak{c}_m$ is bounded by $B^{\mathsf{dishonest}}_{\mathsf{clean}} = Nnu \cdot 2^{\mathsf{sec}_{\mathsf{zk}}+1} \cdot \rho(1 + 2N)$.

- Plaintext-ciphertext product: The noise of resulting ciphertext is the product of an initial noise $e \in R$ and a plaintext $\mathfrak{p}$ such that $\|\mathfrak{p}\|_\infty \leq p$. Hence a new noise bound is $\|\mathfrak{p} \cdot e\|_\infty \leq N \cdot \|\mathfrak{p}\|_\infty \|e\|_\infty \leq Np \cdot \|e\|_\infty$.

- Rotation: In our protocols, all ciphertexts are generated with PoPKs which provide an upper bound $Nnu \cdot 2^{\mathsf{sec}_{\mathsf{zk}}}$ of the size of encryption randomness $r = (u, e_0, e_1)$. Hence the noise of a ciphertext $u \cdot (\mathsf{pk}[0] + \mathsf{pk}[1] \cdot s) + (e_0 + e_1 \cdot s)$ also has an exponential bound in $\mathsf{sec}_{\mathsf{zk}}$. Since we introduce a special modulus to use the modulus-raising technique in our key-switching algorithm, the noise from homomorphic rotation is $\tilde{O}(N)$ which is negligible compared to the noise parameter of ciphertexts. Hence the homomorphic rotation does not change the upper bound of noise.

- Multiplication: Given two ciphertexts $\mathfrak{c}_1, \mathfrak{c}_2$, we have $\mathfrak{c}_i[0] + \mathfrak{c}_i[1] \cdot s = qI_i + \Delta \cdot m_i + e_i$ over R for some $I_i \in R$, plaintext $m_i \in R_p$ and noise $e_i \in R$. Their product scaled by Δ is $\Delta \cdot m_1 m_2 + e'$ modulo q for some noise $e' \approx p(I_1 e_2 + I_2 e_1)$ (other terms are exponentially small compared to this dominating one). We note that $\|I_i\|_\infty \leq N$ and so $\|e'\|_\infty \leq 2N^2 p \cdot \max\{\|e_1\|_\infty, \|e_2\|_\infty\}$. In certain cases, multiplication is followed by a key-switching procedure, which introduces a negligible noise, similar to the case of rotation.

- Matrix product: The permutation $\psi^k(\cdot)$ is not simply a rotation but the composition of two *maskings* and rotations, where a masking refers a specific scalar multiplication which zeros out some values in plaintext slots. It increases the noise bound of input ciphertext by a factor of Np. To sum up, for input ciphertexts $\mathfrak{c}_A, \mathfrak{c}_B$ of noise e_A and e_B, respectively, the noise of each term $\sigma^k(\mathfrak{c}_A) \boxtimes \tau^k(\mathfrak{c}_B)$ is bounded by $2N^2p \cdot 2Np \cdot \max\{\|e_A\|_\infty, \|e_B\|_\infty\}$ and their sum $\mathfrak{c}_A \circledast \mathfrak{c}_B$ has a noise with the upper bound $4dN^3p^2 \cdot \max\{\|e_A\|_\infty, \|e_B\|_\infty\}$.

Concrete Parameter Choices. In our experiments, we set $\mathsf{sec_{zk}} = 128$, $\mathsf{sec_{dd}} = 80$, and $\log p = 128$. For the BFV scheme, we chose $N = 2^{15}$, $\log q = 720$ and the standard deviation $\sigma = 8/\sqrt{2\pi}$, same as in [10] and [66]. This parameter set enjoys computational security of more than 128 bits [26]. In the ZKPoPK protocol (Figure 5.3), we use $u = 2v$ and similar to TopGear [10] set $v = 16$. For notational convenience, we let $|R_m|$ denote the set of polynomials of degree N with non-negative integer coefficients bounded above by m, and let $|R_m|$ denote the number of bits needed to represent an element of R_m. Hence $|R_m| = N \log m$.

5.4.2 Application I: Private Matrix Multiplication

The first application of our protocol we consider is that of private matrix multiplication. We split the study of our techniques for this application into multiple components (1) communication overhead (2) comparison with prior art (3) concrete efficiency and (4) theoretical complexity.

Communcation overhead. We calculate the communication cost of our private matrix multiplication protocol for 128×128 matrices, noting that the communication cost scales linearly with the number of entries in the matrix[7]. In the online phase, the parties open two matrices (say of size $d \times d$), so the communication is $2d^2 \log p$ bits per matrix multiplication. The dominating cost occurs in the offline phase, which we break down further into three parts: the ciphertexts, the ZKPoPK procedure, and the distributed decryption (i.e. DDec) procedure. Each ciphertext takes $2|R_q|$ bits; the ZKPoPK can be used to prove u ciphertexts while it sends $v = u/2$ additional ciphertexts together with v "openings." Here, as seen in Figure 5.3, each opening consists of one element in in R_p, one element in $R_{u \cdot 2^{\mathsf{sec_{zk}}}}$ and two elements in $R_{\rho \cdot u \cdot 2^{\mathsf{sec_{zk}}}}$; finally, the protocol requires 4 invocations to DDec, which requires each party to send $4|R_q|$ bits.

Note that one invocation of the protocol generates two matrix triples, due to the fact that we optimally use the $2^{15} = 128^2 \cdot 2$ slots in our HE scheme. Hence, the

[7]Note that we did not include the cost of one-time set-up, which consists of generating all the required keys keys for the HE scheme and generating and proving the encryptions of shares of the MAC key.

amortized communication cost *sent by each party* in the offline phase is

$$\begin{aligned}&\frac{1}{2}\left(6|R_q| + \frac{1}{u}v(2|R_q| + u \cdot \log_2 N + (1 + 2\log_2 \rho)|R_{u\cdot 2^{\mathsf{sec}_{\mathsf{zk}}}}| + |R_p|)\right)\\ &\approx \frac{1}{2}\left(6|R_q| + \frac{1}{u}v(2|R_q| + u \cdot \log_2 N + 9.64|R_{u\cdot 2^{\mathsf{sec}_{\mathsf{zk}}}}| + |R_p|)\right)\end{aligned} \tag{5.12}$$

With our parameter settings, this amounts to around 12.46MB of data sent by each party.

Comparison with LowGear [66]. We compare our communication cost with the preprocessing required by the SPDZ protocol to multiply 128×128 matrices: the LowGear protocol takes 15 kbits per triple, and we assume that we need $d^{2.8}$ triples. Setting $d = 128$, this amounts to a 1.54GB communication cost of sent by each party. So we reduced the communication by roughly two orders of magnitude for 128-dimensional matrix multiplication.

Concrete efficiency. We now present the performance of our secure matrix multiplication protocol over various matrix sizes. Our source code was developed in C++ with Microsoft SEAL version 3.3 [81]. All the experiments were done on a machine with an Intel Xeon Platinum 8168 at 2.7 GHz featuring 16 cores. The compiler was GNU version 7.4.0 (-O3), and we used GMP version 6.1.2 and NTL version 11.3.3.

Table 5.1 shows results for microbenchmarks on homomorphic matrix computation for a two party scenario and various components of the matrix triple generation process. We split the input matrices into 128×128 matrix blocks. We found that key generation takes about 83 seconds and it takes about 191 milliseconds to encrypt two input square matrices of size 128 as a single ciphertext, yielding an amortized rate of 96 milliseconds per matrix. The second column gives the amortized encryption timing per matrix. We note that a one time set-up cost is to prepare appropriate masking plaintext polynomials that will be used for performing permutation $\psi^k(\cdot)$, which takes around 14.5 seconds. In the third and fourth columns labeled "Permutation", we give timings per matrix for generating the encrypted permutations of blocks of A and B, respectively. The fifth column labeled "Block comp." gives the amortized time taken for additions and multiplications on block matrices.

Theoretical complexity. Suppose the input matrix of size n is partitioned into k^2 blocks of size d (we have $d = 128$ in our experiments). Then the encryption cost is $O(k^2)$. On the other hand, the computational costs of generating permutations of block matrices and performing block computation are $O(k^2)$ and $O(k^3)$, respectively. These trends can be seen in Table 5.1.

In Table 5.2 we document the experimental latency associated with the communication cost of our protocol. In the LAN setting, two parties are deployed in the same geographic network (N. Virginia on Amazon EC2, bandwidth about 5Gbps, ping time 20 ms). In the WAN setting, they were deployed in different geographic settings (N.

Table 5.1: Microbenchmarks: All timings measured in seconds; 16 threads were used for columns labeled "Permutation" and "Block comp", and a single thread was used for other operations; the ZkPoPK time is amortized over $u = 32$ ciphertexts.

Matrix size	Encrypt time	Permutation		Block comp.	ZkPoPK		AddMacs time	DDec time
		of A	of B		Prover	Verifier		
128×128	0.10	1.8	0.9	1.4	0.047	0.09	0.6	1
256×256	0.38	5.6	2.3	10.1	0.188	0.35	2.4	4
384×384	0.86	12.8	4.9	34.0	0.79	0.81	5.4	9
512×512	1.52	21.8	8.0	79.6	1.41	1.44	9.6	16
1024×1024	6.08	79.6	32.9	648	3	5.63	38.4	64

Virginia and N. California on Amazon EC2, bandwidth about 320 Mbps, ping time 70 ms). SPDZ uses a 25 Gbps link for LAN and 50 Mbps for WAN (WAN numbers are extrapolated from Overdrive [**EC:KelPasRot18**]).

Finally, Table 5.3 provides total time estimates on matrix multiplications in the LAN and WAN settings respectively. Total-16, SPDZ-16 refer to timings using 16 threads and Total-1, SPDZ-1 refer to single-threaded implementations. As can be seen from the table, our approach is between 16×-40× faster than prior art and improves with larger matrix sizes.

5.4.3 Application II: Private Nearest Neighbors

The next application we explore is that of private nearest neighbor search (NNS). In the batched version of the private NNS problem, one party holds a dataset X of n vectors in d-dimensional Euclidean space, and the other party holds several d-dimensional query vectors $q_1, q_2, \ldots, q_b$. The task is to compute securely for each query k nearest data vectors with respect to the Euclidean distance. There is a large body of work on this topic (see [28] for an overview). However, we are not aware of any previous work that solves the problem in the dishonest majority malicious adversarial model. Most of the secure NNS algorithms first (securely) compute secret shares of distances between every query vector and every dataset vector and then perform top-k selection. Distance computation can easily be reduced to matrix multiplication for matrices of size $n \times d$ and $d \times b$ and thus in the dishonest majority security model, we can use our protocol to perform distance computation.

As an example, we will consider the largest NNS instance that was solved securely to date [28]: the subset of the Deep1B dataset [8] with $n = 10^7$, $d = 96$. If we would like to compute distances between $b = 128$ queries and the whole dataset, we would need to multiply 78125 pairs of square matrices of size 128. Since each matrix multiplication requires 12.46 MB of communication per party in the offline phase, the overall distance computation requires 7.6 GB per party per query. On 16 threads, our protocols roughly require 30 minutes per query. LowGear equipped with the Strassen algorithm, on the other hand, requires at least 500 million Beavers triples per query. Running on 16 threads, this amounts to at least 80 minutes, and takes more than 1

Table 5.2: Communication overhead accounting for the round complexity and amount of data sent between parties.

Matrix Sizes	Communication Time	
	LAN	WAN
128×128	0.010 sec	2.05 sec
256×256	0.039 sec	8.19 sec
384×384	0.091 sec	18.44 sec
512×512	0.161 sec	32.78 sec
1024×1024	0.647 sec	131.15 sec

Table 5.3: Benchmarks for private matrix multiplication over various sizes. Note that the timings for SPDZ are obtained by measuring the throughput of triple generation.

	Matrix sizes	Total-16 time	Total-1 time	SPDZ-16	SPDZ-1
LAN	128×128	5.9 sec	36.1 sec	8.41 sec	128 sec
	256×256	25.5 sec	214.5 sec	58.9 sec	900 sec
	384×384	68.3 sec	653.6 sec	3 min	46.8 min
	512×512	2.3 min	24.5 min	6.87 min	105 min
	1024×1024	14.5 min	173 min	52.02 min	735 min
WAN	128×128	7.95 sec	38.15 sec	1.61 min	24.6 min
	256×256	33.5 sec	222.6 sec	11.32 min	2.88 hours
	384×384	68.34 sec	672.0 sec	34.6 min	9 hours
	512×512	2.35 min	25.0 min	1.32 hours	20.2 hours
	1024×1024	16.51 min	175.1 min	10 hours	5.88 days

TB of communication. Note that these performances numbers are obtained from our microbenchmarks rather than from running actual experiments.

5.4.4 Application III: Private Inference of ResNet-50

We can use our protocol to perform convolutions of a neural network securely. Here we discuss it in the context of the ResNet-50 network [59]. Note that for this discussion we ignore ReLUs, batch normalization, and pooling layers and focus on convolutions only.

All the convolutions in the ResNet-50 network require 3298 multiplications of pairs of 128×128 matrices. We will now follow the benchmarks from Table 5.3 to estimate the preprocessing cost of computing these products securely. Since each multiplication requires 12.46 MB of communication per party, the total communication would be 41 GB per party. Estimating the running time for preprocessing phase on 16 threads, we obtain 7.4 hours per query. On the other hand, doing multiplications using Strassen with LowGear would require at least 2.7 billion Beavers triples, so when run with 16

Table 5.4: Two-party costs for ResNet-50 without the batch norm layer over $\mathbb{Z}_p$.

Protocol	Communication (GB)			
	Preprocessing		Online	
Conv [66]	5,092	170×	86.91	160×
Conv (ours)	30		0.54	
Conv + RELUs [66]	9,225	2.2×	105.2	5.6×
Conv + RELUs (ours)	4,133		18.83	

triple generation threads, this amounts to at least 7.6 hours of running time and 5 TB of communication.

Adding RELUs into the costs. ResNet-50 architecture requires a total of 9,608,704 ReLUs. To compute a RELU in MPC, one needs to have access to a protocol for random shared bit generation $[\![b]\!]$. Using existing techniques, the cost of such a RELU protocol is two-fold: in terms of preprocessing, it requires 122 triples and 105 random bits[8] whereas the online cost of RELU is 8 rounds of communication and 1 extra openings. A more careful analysis of SCALE/MP-SPDZ implementation of RELU reveals that there are exactly 119 field elements sent per party in the online phase.

On top of the RELUs, each multiplication involving a Beaver triple requires two field elements opened per party hence some extra 256 bits. In Table 5.4 we summarize the estimated costs using LowGear and SPDZ-online versus our implementation of the online phase which uses convolution triples. Note that our current implementation does not support RELUs so we estimate that part. In Table 5.4 the "Conv" keyword denotes the evaluation of the convolution layers only. As can be seen from the table, our approach brings down the online cost of the convolution layers by at least two orders of magnitude compared with classic SPDZ Beaver triples.

5.5 Summary

In this work, we reduced the overhead of computing linear operations in the SPDZ framework for dishonest-majority MPC. First, we demonstrate a novel way of generating pre-processing data for bilinear operations such as matrix multiplication and convolutions in the SPDZ framework, where the communication cost does not depend on the number of multiplications but only depends on the input and output size. We achieved this by leveraging state-of-the-art homomorphic encryption algorithms for linear operations into SPDZ. We generalized the notion of authenticated Beaver triples to arbitrary bilinear operations and adapted the state-of-the-art homomorphic matrix multiplication algorithm to generate authenticated "matrix triples"

[8]This is assuming $p \approx 2^{128}$ and a comparison with statistical security $\mathsf{sec_s} = 40$ - see SCALE-MAMBA documentation for more details [3].

and "convolution triples." We also removed the sacrifice stage of SPDZ via increasing the parameters of the HE scheme to allow one more multiplication, and optimized the SPDZ zero-knowledge proof via the usage of BFV homomorphic encryption scheme, which further improved performance. Our protocol requires $O(n^2)$ total communication to multiply two $n \times n$ matrices, compared to $O(n^{2.8})$ from SPDZ. In terms of concrete efficiency, to securely multiply two 128×128 matrices, our protocol is at least *one order of magnitude* faster in terms of latency and as much as *two orders of magnitude* more communication efficient compared to prior art. Furthermore, this improvement only increases as the dimensions of the matrices increase. We believe our protocols improves the state-of-the-art in dishonest-majority secure computation, particularly in tasks that require a large number of linear operations such as private machine learning inference and training.

5.6 Selected References

[66] Marcel Keller, Valerio Pastro, and Dragos Rotaru. "Overdrive: making SPDZ great again." In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2018, pp. 158–189

[10] Carsten Baum, Daniele Cozzo, and Nigel P Smart. "Using TopGear in Overdrive: A more efficient ZKPoK for SPDZ." in: *International Conference on Selected Areas in Cryptography*. 2019

[39] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. "Multiparty computation from somewhat homomorphic encryption." In: *Annual Cryptology Conference*. Springer. 2012, pp. 643–662

[38] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. "Practical covertly secure MPC for dishonest majority–or: breaking the SPDZ limits." In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 1–18

[47] Junfeng Fan and Frederik Vercauteren. *Somewhat practical fully homomorphic encryption*. https://eprint.iacr.org/2012/144.pdf. 2012

[85] Payman Mohassel and Yupeng Zhang. "SecureML: A system for scalable privacy-preserving machine learning." In: *IEEE Symposium on Security and Privacy (S&P)*. 2017

Chapter 6

Conclusion

Motivated by the trend of reducing the overhead of privacy-preserving systems, specifically those targeted towards machine learning, we develop techniques that improve the performance of critical building blocks of such applications. A key insight is a hybrid approach towards designing such systems, particularly cross-layer design and combination of MPC and HE.

Using a novel cross-layer design approach, we propose a set of highly efficient protocols for commonly used functions in NNs. This unique approach enables joint optimization over the secret sharing layer as well as the function computation layer. As a consequence, we can jointly optimize and design optimally efficient protocols in this integrated architecture. We propose a framework called SecureNN based on these insights. Furthermore, since these protocols rely on simple modular arithmetic, this approach also provides concrete efficiency when implemented.

Though highly efficient in practice, SecureNN operates in a weaker semi-honest security model. In Falcon, we have seen how to improve upon the adversarial model and design protocols that enjoy active security using a hybrid protocol design. We build upon the redundancy of a replicated secret sharing and propose a number of theoretical improvements that reduce both the communication and computation complexity of protocols commonly used in NNs. Furthermore, the improved support for various types of layers in this work enhances the applicability of privacy-preserving techniques to machine learning.

Finally, we further strengthen the adversarial model from honest majority to a dishonest majority and design protocols against such an adversary. Matrix multiplication, a critical component of ML is performance-intensive in such an adversarial setting. In Ponytail, we have shown how to reduce the communication complexity of matrix multiplication from $O(n^3)$ to $O(n^2)$. Furthermore, implementations demonstrate that even for modest matrix sizes of 100×100, our approach of using a combination of FHE and MPC outperforms all prior approaches.

Collectively, these techniques provide a new foundation for the design of privacy-preserving algorithms while improving both the asymptotic and concrete efficiency of such systems. Overall, these protocols provide orders of magnitude performance improvements for commonly used functions in machine learning, thereby significantly reducing the gap between privacy-preserving and plaintext computation.

6.1 Future Work

Moving forward, the vision of this dissertation can be more broadly stated as:

> *Design and development of efficient protocols for adoption of privacy-preserving technologies.*

I envision a future in which the digital infrastructure enables privacy-conscious collection and handling of user data, thus enabling exciting new applications while simultaneously complying with changing digital and social norms. There is still work to be done and this broad view leads to a number of other research directions, a few of which I describe below:

Dishonest majority adversarial model. While SecureNN and Falcon demonstrated purely arithmetic secret sharing-based protocols for non-linear function computation, it is an open line of research to improve these function computations in the dishonest majority adversarial model (one similar to Ponytail). The insights from Chapters 3 and 4 can be distilled and combined with advances such as edaBits [46] to provide efficient comparison operations.

Expanding ML base. While CNNs form an important class of NN algorithms, they are by no means the only class of NN algorithms. Extending privacy-preserving ML techniques to other network architectures such as RNNs [59], LSTMs [60], GANs [55], Reinforcement Learning [105], etc., poses important new questions. Furthermore, to truly enable privacy-preserving ML, a number of open problems around ML must be solved. These include hyper-parameter setting, support for optimizers, support for GPUs, and a sufficiently wide range of supported functionalities.

Improving usability. Finally, privacy-preserving techniques are currently restricted to a few expert programmers and there is work to be done to increase adoption of these technologies by ML experts. This will require easy-to-use libraries that abstract the complexities of the underlying cryptography from the system user. Crypten [34] is one such effort with limited support for protocols and functionalities.

Appendix A

SecureNN: Supplementary Material

A.1 Arithmetic Operations on Shared Decimal Numbers

In order for NN algorithms to be compatible with cryptographic applications, they must typically be encoded into integer form (most NN algorithms work over floating-point numbers). Now, decimal arithmetic must be performed over these values in an integer ring which requires careful detail. We follow [85] and describe details below. We use fixed-point arithmetic to perform all computations. In other words, all numbers are represented as integers in the native `C++` datatype `uint64_t`. We use a precision of $l_D = 13$ bits for representing numbers. For instance, an integer 2^{15} in this encoding corresponds to the float 4 and an integer $2^{64} - 2^{13}$ corresponds to a float -1. Since we use unsigned integers for encoding, $\mathsf{ReLU}(\cdot)$ compares its argument with 2^{63}. Such an encoding is gaining popularity in the systems community with the introduction of fixed-point data types [48].

To perform decimal arithmetic in an integer ring, we use the same solution as is used in [85]. Addition of two fixed-point decimal numbers is straightforward. To perform multiplication, we multiply the two decimal numbers and *truncate* the last l_D bits of the product. Theorem 1 in [85] shows that this truncation technique also works over shared secrets (2-out-of-2 shares), i.e., the two parties can simply truncate their shares locally preserving correctness with an error of at most 1 bit with high probability. Denoting an arithmetic shift by $\Pi_{\mathsf{AS}}(a, \alpha)$, truncation of shares, i.e., dividing shares by a power of 2 is described in Algorithm 18. We refer the reader to [85] for further details.

A.2 Security Proofs

Here, we provide proofs of semi-honest simulation based security of our protocols. If a protocol invokes another sub-protocol for a functionality $\mathcal{F}$, we prove the security

Algorithm 18 Truncate $\Pi_{\mathsf{Truncate}}(\{P_0, P_1\})$:

Input: P_0 & P_1 hold an positive integer α and $\langle X \rangle_0^L$ & $\langle X \rangle_1^L$ resp.
Output: P_0 gets $\langle X/2^\alpha \rangle_0^L$ and P_1 gets $\langle X/2^\alpha \rangle_1^L$.

1: P_0 computes $\Pi_{\mathsf{AS}}(\langle X \rangle_0^L, \alpha)$.
2: P_1 computes $-\Pi_{\mathsf{AS}}(-\langle X \rangle_1^L, \alpha)$.

by replacing the sub-protocol invocation with the corresponding functionality call. This refers to the $\mathcal{F}$-hybrid model.

Private Compare

Lemma A.1. *Protocol* $\Pi_{\mathsf{PC}}(\{P_0, P_1\}, P_2)$ *in Algorithm 3 securely realizes* $\mathcal{F}_{\mathsf{PC}}$ *when* $p > \ell + 2$.

Proof. We first prove correctness of our protocol, i.e., $\beta' = \beta \oplus (x > r)$. Define $x[i]$ as $x[i] := \mathsf{Reconst}^p(\langle x[i] \rangle_0^p, \langle x[i] \rangle_1^p) \in \{0, 1\}$ for all $i \in [\ell]$. We treat x and r as ℓ bit integers and $x > r$ tells if x is greater[1] than r. Below, we do a case analysis on the value of β.
Case $\beta = 0$. For correctness, we require $\beta' = (x > r)$. For each $i \in [\ell]$, define $w_i = \mathsf{Reconst}^p(\langle w_i \rangle_0^p, \langle w_i \rangle_1^p)$. Note that $w[i] = x[i] + r[i] - 2r[i]x[i] = x[i] \oplus r[i]$. For each $i \in [\ell]$, define $c_i = \mathsf{Reconst}^p(\langle c_i \rangle_0^p, \langle c_i \rangle_1^p)$. Note that $c[i] = r[i] - x[i] + 1 + \sum_{k=i+1}^{\ell} w_k$. Let i^* be such that for all $i > i^*$, $x[i] = r[i]$ and $x[i^*] \neq r[i^*]$. We claim that the following holds:

- For all $i > i^*$, $c[i] = 1$. This is because both $r[i] - x[i]$ and $\sum_{k=i+1}^{\ell} w_k$ are 0.
- For $i = i^*$, if $x[i] = 1$, $c[i] = 0$, else $c[i] = 2$.
- For $i < i^*$, $c[i] > 1$. This is because $r[i] - x[i]$ is either 1 or -1 and $\sum_{k=i+1}^{\ell} w_k > 1$. For this step, we require that there is no wrap-around modulo p, which is guaranteed by $p > \ell + 2$.

This proves that $x > r$ iff there exists a $i \in [\ell]$ such that $c[i] = 0$. Finally, the last step of multiplying with random non-zero s_i and permuting all the $s_i c_i$ preserves this characteristic. This condition is exactly what P_2 checks.
Case $\beta = 1$. For correctness, we require $\beta' = 1 \oplus (x > r) = (x \leq r)$. The last expression is equivalent to $x < (r + 1)$ when $r \neq 2^\ell - 1$ and otherwise $x \leq r$ is always true. Note that $t = r + 1$. Now, similar to the above logic, we compute $t > x$ when $r \neq 2^\ell - 1$. This condition is easy to check since r is known to both P_0 and P_1.
When $r = 2^\ell - 1$, we know that $\beta' = 1$. Also, $\beta' = 1$ iff there exists a unique i such that d_i is 0. Hence, the parties create a vector starting with 1 followed by $\ell - 1$ zeroes. Scaling by s_i and permutation creates a uniform vector with exactly one 0.

Now we prove security of our protocol. First note that P_0 and P_1 receive no messages in the protocol and hence, our protocol is trivially secure against corruption

[1] $x > r$ iff the leftmost bit where $x[i] \neq r[i]$, $x[i] = 1$.

of P_0 or P_1. Now, we have to simulate the messages seen by P_2 given P_2's output, namely β'. To do this, if $\beta' = 0$, pick $d_i \xleftarrow{\$} \mathbb{Z}_p^*$, for all $i \in [\ell]$. If $\beta' = 1$, then pick an $i^* \xleftarrow{\$} [\ell]$, set $d_{i^*} = 0$ with all other $d_i \xleftarrow{\$} \mathbb{Z}_p^*$. Now, compute $(\langle d_i \rangle_0^p, \langle d_i \rangle_1^p) \leftarrow \mathsf{Share}^p(d_i)$ and send $\langle d_i \rangle_j^p$ for all $i \in [\ell], j \in \{0,1\}$ as the message from P_j to P_2. This completes the simulation. To see that the simulation is perfect, observe that whether or not $\exists i^*$, with $d_{i^*} = 0$ depends only on β'. Additionally, when $\beta' = 1$, the index i^* where $d_{i^*} = 0$ is uniformly random in $[\ell]$ due to the random permutation π. Finally, the non-zero d_i values are uniform over $\mathbb{Z}_p^*$ since the s_i values are random in $\mathbb{Z}_p^*$. □

Compute MSB

Lemma A.2. *Protocol* $\Pi_{\mathsf{MSB}}(\{P_0, P_1\}, P_2)$ *in Algorithm 5 securely realizes* $\mathcal{F}_{\mathsf{MSB}}$ *in the* $(\mathcal{F}_{\mathsf{PC}}, \mathcal{F}_{\mathsf{MATMUL}})$*-hybrid model.*

Proof. First, we prove correctness of our protocol, i.e., $\alpha := \mathsf{Reconst}^L(\langle \alpha \rangle_0^L, \langle \alpha \rangle_1^L) = \mathsf{MSB}(a)$. As already mentioned, over an odd ring, the MSB computation can be reduced to LSB computation. More precisely, over an odd ring, $\mathsf{MSB}(a) = \mathsf{LSB}(y)$, where $y = 2a$. Hence, it suffices to compute $\mathsf{LSB}(2a)$.

In the protocol, $r = y + x(\mathsf{mod} L - 1)$. Hence, $\mathsf{LSB}(y) = y[0] = r[0] \oplus x[0] \oplus \mathsf{wrap}_(y, x, L-1)$. Next, we note that $\mathsf{wrap}_(y, x, L-1) = (x > r)$. First, P_0, P_1, P_2 compute $x > r$ as follows. They invoke Π_{PC} and its correctness ensures that P_2 learns $\beta' = \beta \oplus (x > r)$. Next, P_2 secret shares β' to P_0, P_1. Note that $\gamma = \beta' + \beta - 2\beta\beta' = \beta \oplus \beta' = (x > r) = \mathsf{wrap}_(y, x, L-1)$. Next, similarly, $\delta = r[0] \oplus x[0]$. Then, $\theta = \gamma\delta$ and $\alpha = \gamma + \delta - 2\theta = \gamma \oplus \delta = \mathsf{LSB}(y) = \mathsf{MSB}(a)$.

Next, we prove security of our protocol. Parties P_0 and P_1 learn the following information: $2a + x$ (from Step 3), $\langle r \rangle_j^{L-1}$, $\{\langle x[i] \rangle_j^p\}_i$, $\langle x[0] \rangle_j^B$ (Step 1) and $\langle \beta' \rangle_j^B$ (Step 5). However, these are all fresh shares of these values and hence can be perfectly simulated by sending random fresh share of 0. Finally, P_j outputs a fresh share of $\mathsf{MSB}(a)$ as the share is randomized with u_j. The only information that P_2 learns is bit β'. However, $\beta' = \beta \oplus (r > c)$, where β is a random bit unknown to P_2. Hence, the distribution of β' is uniformly random from P_2's view and hence the information learned by P_2 can be perfectly simulated. □

Derivative of ReLU

Lemma A.3. *Protocol* $\Pi_{\mathsf{DRELU}}(\{P_0, P_1\}, P_2)$ *in Algorithm 6 securely realizes* $\mathcal{F}_{\mathsf{DReLU}}$ *in the* $(\mathcal{F}_{\mathsf{SC}}, \mathcal{F}_{\mathsf{MSB}})$*-hybrid model for all* $a \in [0, 2^k] \cup [2^\ell - 2^k, 2^\ell - 1]$*, where* $k < \ell - 1$.

Proof. First, we prove the correctness of our protocol when $a \in [0, 2^k) \cup (2^\ell - 2^k, 2^\ell - 1]$, where $k < \ell - 1$, i.e., $\gamma := \mathsf{Reconst}^L(\langle \gamma \rangle_0^L, \langle \gamma \rangle_1^L) = \mathsf{ReLU}'(a) = 1 \oplus \mathsf{MSB}(a)$, where a is the value underlying the input shares. Note that when a belongs to the range $[0, 2^k] \cup [2^\ell - 2^k, 2^\ell - 1]$, where $k < \ell - 1$, $\mathsf{MSB}(a) = \mathsf{MSB}(2a) = \mathsf{MSB}(c)$. Also, it holds that $2a \neq L - 1$, and precondition of $\mathcal{F}_{\mathsf{SC}}$ is satisfied. From correctness of $\mathcal{F}_{\mathsf{SC}}$, $y := \mathsf{Reconst}^{L-1}(\langle y \rangle_0^{L-1}, \langle y \rangle_1^{L-1}) = 2a$. Next, from correctness of $\mathcal{F}_{\mathsf{MSB}}$, $\alpha := \mathsf{Reconst}^L(\langle \alpha \rangle_0^L, \langle \alpha \rangle_1^L) = \mathsf{MSB}(y) = \mathsf{MSB}(2a)$. Finally, $\gamma = 1 - \alpha = 1 - \mathsf{MSB}(a)$ as

required. Also, note that $\langle\gamma\rangle_j^L$ are fresh shares of γ since both parties locally add shares of 0 to randomize the shares.

For security, first see that P_2 learns no information from the protocol (as both $\mathcal{F}_{\mathsf{SC}}(\{P_0, P_1\}, P_2)$ and $\mathcal{F}_{\mathsf{MSB}}(\{P_0, P_1\}, P_2)$ provide outputs only to P_0 and P_1). Now, $P_j, j \in \{0, 1\}$ only learns a fresh share of $2a$ (over $\mathbb{Z}_{L-1}$) in Step 2 and a fresh share of $\alpha = \mathsf{MSB}(2a)$ in Step 3 and hence any information learned by either party can be perfectly simulated through appropriate shares of 0. Finally, P_j outputs a fresh share of $\mathsf{ReLU}'(a)$ as the respective shares are randomized by u_j. □

ReLU

Lemma A.4. *Protocol* $\Pi_{\mathsf{ReLU}}(\{P_0, P_1\}, P_2)$ *in Algorithm 7 securely realizes* $\mathcal{F}_{\mathsf{ReLU}}$ *in the* $(\mathcal{F}_{\mathsf{DReLU}}, \mathcal{F}_{\mathsf{MATMUL}})$*-hybrid model.*

Proof. First, we prove the correctness, i.e., $c := \mathsf{Reconst}^L(\langle c\rangle_0^L, \langle c\rangle_1^L) = \mathsf{ReLU}(a) = \mathsf{ReLU}'(a) \cdot a$, where a is the value underlying the input shares. It follows from correctness[2] of $\mathcal{F}_{\mathsf{DReLU}}$ that $\alpha := \mathsf{Reconst}^L(\langle\alpha\rangle_0^L, \langle\alpha\rangle_1^L) = \mathsf{ReLU}'(a)$. Now from the correctness of $\mathcal{F}_{\mathsf{MATMUL}}$ it follows that $c = \alpha \cdot a$.

For security, see that P_2 learns no information from the protocol (as both $\mathcal{F}_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ and $\mathcal{F}_{\mathsf{MATMUL}}(\{P_0, P_1\}, P_2)$ provide outputs only to P_0 and P_1). Now, $P_j, j \in \{0, 1\}$ only learns a fresh share of $\alpha = \mathsf{ReLU}'(a)$ in Step 1 and a fresh share of αa (over $\mathbb{Z}_L$) in Step 2 and hence any information learned by either party can be perfectly simulated through appropriate shares of 0. Finally, P_j outputs a fresh share of $\mathsf{ReLU}(a)$ as the respective shares are randomized by u_j. □

Matrix Multiplication

Lemma A.5. *Protocol* $\Pi_{\mathsf{MatMul}}(\{P_0, P_1\}, P_2)$ *in Algorithm 1 securely realizes* $\mathcal{F}_{\mathsf{MATMUL}}$.

Proof. Let Z_j be the output of party P_j. For correctness we need to prove that $\mathsf{Reconst}^L(Z_0, Z_1) = X \cdot Y$. We calculate $Z_0 + Z_1 = \big(\langle X\rangle_0^L \cdot F + E \cdot \langle Y\rangle_0^L + \langle C\rangle_0^L + U_0\big) + \big(-E \cdot F + \langle X\rangle_1^L \cdot F + E \cdot \langle Y\rangle_1^L + \langle C\rangle_1^L + U_1\big) = -E \cdot F + X \cdot F + E \cdot Y + C = -(X - A) \cdot (Y - B) + X \cdot (Y - B) + (X - A) \cdot Y + A \cdot B = X \cdot Y$.

Security against corrupt P_2 is easy to see since it gets no message and only generates a fresh matrix Beaver triplet of correct dimensions. Now, we prove security against corruption of either P_0 or P_1. Party P_0 receives $\langle A\rangle_0^L, \langle B\rangle_0^L, \langle C\rangle_0^L$ and $\langle E\rangle_1^L, \langle F\rangle_1^L$. We note that all of these uniform random matrices because A, B are uniformly chosen and fresh shares are generated of A, B, C. Also, the final output of $P_j, j \in \{0, 1\}$, is a fresh random share of $X \cdot Y$ (as they have each been randomized by random matrix U_j) and contain no information about X and Y. □

[2]When we instantiate the functionality $\mathcal{F}_{\mathsf{DReLU}}$ using protocol Π_{DReLU}, we would ensure that the conditions on the range of input to Π_{DReLU} are met.

Select Share

Lemma A.6. *Protocol* $\Pi_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ *in Algorithm 2 securely realizes* $\mathcal{F}_{\mathsf{SS}}$ *in the* $\mathcal{F}_{\mathsf{MATMUL}}$*-hybrid model.*

Proof. We first prove the correctness of our protocol, i.e., $z := \mathsf{Reconst}^L(\langle z\rangle_0^L, \langle z\rangle_1^L)$ is x when $\alpha = 0$ and y when α is 1. Note that $w = y - x$ and from correctness of $\mathcal{F}_{\mathsf{MATMUL}}$, $c = \mathsf{Reconst}^L(\langle c\rangle_0^L, \langle c\rangle_1^L) = \alpha \cdot w = \alpha \cdot (y - x)$. And finally, $z = x + c = (1 - \alpha) \cdot x + \alpha \cdot y$. Hence, correctness holds.

To argue security, first observe that P_2 learns no information from the protocol (as $\mathcal{F}_{\mathsf{MATMUL}}(\{P_0, P_1\}, P_2)$ provides outputs only to P_0 and P_1). On the other hand, $P_j, j \in \{0, 1\}$, only learn fresh shares of the outputs in Step 2 and hence any information learned by either party can be perfectly simulated through appropriate shares of 0 (over $\mathbb{Z}_L$). Finally, P_j outputs a fresh share of the output in Step 3 as they are randomized by u_j. □

Share Convert

Proof of Lemma 3.1: We have already seen correctness. To see the security, first observe that the only information that P_2 sees is $x = a + r$ (over $\mathbb{Z}_L$) and η'. Since $r \xleftarrow{\$} \mathbb{Z}_L$ and is not observed by P_2, we have that x is uniform over $\mathbb{Z}_L$ and so information sent to P_2 can be simulated by sampling $x \xleftarrow{\$} \mathbb{Z}_L$ and sending shares of x from P_j to P_2 for $j \in \{0, 1\}$. Next, η'' is a random bit not observed by P_2 and thus, η' is a uniform random bit to P_2. Hence, η' can be perfectly simulated.

Finally, the only information that P_0 and P_1 observe are fresh shares of the following values: $\forall i \in [\ell], x[i]$, δ, and η' that can be perfectly simulated by sharing 0. The outputs of P_0 and P_1 are fresh shares of a over $\mathbb{Z}_{L-1}$ as they are randomized using u_0 and u_1 respectively.

Division

Lemma A.7. *Protocol* $\Pi_{\mathsf{DIV}}(\{P_0, P_1\}, P_2)$ *in Algorithm 8 securely realizes* $\mathcal{F}_{\mathsf{DIV}}$ *in the* $(\mathcal{F}_{\mathsf{DReLU}}, \mathcal{F}_{\mathsf{MATMUL}})$*-hybrid model when* $y \neq 0$.

Proof. We first prove the correctness of our protocol, i.e., $q := \mathsf{Reconst}^L(\langle q\rangle_0^L, \langle q\rangle_1^L) = \lfloor x/y \rfloor$. Our protocol mimics the standard long division algorithm and proceeds in ℓ iterations. In the i^{th} iteration we compute the $q[i]$, the i^{th} bit of q starting from the MSB.

We will prove by induction and maintain the invariant: $\beta_i = q[i]$, $k_i = 2^i \beta_i$, $u_i = y \cdot \sum_{j=i}^{\ell-1} k_j$. Assume that invariant holds for $i > m$, then we will prove that it holds for $i = m$. Note that $z_m = (x - u_{m+1} - 2^m y)$. We note that β_m or $q[m]$ is 1 iff $x - u_{m+1} > 2^m y$, that is, when $\mathsf{ReLU}'(z_m) = 1$. By correctness[3] of $\mathcal{F}_{\mathsf{DReLU}}$, $\beta_m =$

[3]When we instantiate the functionality $\mathcal{F}_{\mathsf{DReLU}}$ using protocol Π_{DReLU}, we would ensure that the conditions of Lemma A.3 are met.

$\mathsf{Reconst}^L(\langle\beta_m\rangle_0^L, \langle\beta_m\rangle_1^L) = \mathsf{ReLU}'(z_m)$. Next by correctness of $\mathcal{F}_{\mathsf{MATMUL}}$, $k_m = \beta_m 2^m$ and $v_m = \beta_m \cdot 2^m y = k_m y$. Hence, $u_m = u_{m+1} + v_m = y \cdot \sum_{j=m}^{\ell-1} k_j$.

To argue security, first observe that P_2 learns no information from the protocol (as both $\mathcal{F}_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ and $\mathcal{F}_{\mathsf{MATMUL}}(\{P_0, P_1\}, P_2)$ provide outputs only to P_0 and P_1). Now, $P_j, j \in \{0, 1\}$, only learn fresh shares of the outputs in Step 4, 5, and 6 and hence any information learned by either party can be perfectly simulated through appropriate shares of 0 (over $\mathbb{Z}_L$). Finally, P_j outputs a fresh share of the final output in Step 9 as they are randomized by s_j. □

Maxpool

Lemma A.8. *Protocol* $\Pi_{\mathsf{MP}}(\{P_0, P_1\}, P_2)$ *in Algorithm 9 securely realizes* $\mathcal{F}_{\mathsf{MAXPOOL}}$ *in the* $(\mathcal{F}_{\mathsf{DReLU}}, \mathcal{F}_{\mathsf{SS}})$*-hybrid model.*

Proof. We first prove $\mathsf{max}_n := \mathsf{Reconst}^L(\langle\mathsf{max}_n\rangle_0^L, \langle\mathsf{max}_n\rangle_1^L)$ stores the maximum value of the elements $\{x_i\}_{i\in[n]}$ and $\mathsf{ind}_n := \mathsf{Reconst}^L(\langle\mathsf{ind}_n\rangle_0^L, \langle\mathsf{ind}_n\rangle_1^L)$ stores the index of maximum value. This establishes the correctness of the protocol.

We will prove this by induction and will maintain the invariant that max_i holds the value of $\mathsf{max}(x_1, \ldots, x_i)$ and ind_i holds a value of k s.t. $\mathsf{max}_i = x_k$. It is easy to see that this holds for $i = 1$. Suppose this holds for $i = m - 1$. Then we will prove that it holds for $i = m$. In Step 3, we calculate $w_m = x_m - \mathsf{max}_{m-1}$. By correctness of $\mathcal{F}_{\mathsf{DReLU}}$, $\beta_m = \mathsf{ReLU}'(w_m)$. That is, $\beta_m = 1$ iff $x_m > \mathsf{max}_{m-1}$. Next, by correctness of $\mathcal{F}_{\mathsf{SS}}$, max_m is max_{m-1} if $\beta_m = 0$ and x_m otherwise. In Step 6, we compute shares of $k_m = m$. In Step 7, by correctness of $\mathcal{F}_{\mathsf{SS}}$, $\mathsf{ind}_m = \mathsf{ind}_{m-1}$ if $\beta_m = 0$ and m otherwise. This proves correctness.

To argue security, first observe that P_2 learns no information from the protocol (as $\mathcal{F}_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ and $\mathcal{F}_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ provide outputs only to P_0 and P_1). Now, $P_j, j \in \{0, 1\}$, only learn fresh shares of the values $\beta_i, \mathsf{max}_i, \mathsf{ind}_i$ and hence any information learned by either party can be perfectly simulated through appropriate shares of 0 (over $\mathbb{Z}_L$). Finally, P_j outputs a fresh share of the final output in Step 9 as the respective shares are randomized by u_j and v_j. □

Derivative of Maxpool

We provide a proof of correctness and security of Algorithm 10 followed by the general case algorithm.

Lemma A.9. $\Pi_{\mathsf{n_1 \times n_2 DMP}}(\{P_0, P_1\}, P_2)$ *in Algorithm 10 securely realizes* $\mathcal{F}_{\mathsf{DMAXPOOL}}$ *in the* $\mathcal{F}_{\mathsf{MAXPOOL}}$*-hybrid model.*

Proof. Let k^* be the index of the maximum value and E_r denote the unit vector with 1 in the r^{th} position and 0 everywhere else. For correctness, we show that $\mathsf{Reconst}^L(\langle D\rangle_0^L + U_0, \langle D\rangle_1^L + U_1) = E_{k^*}$ in Algorithm 10.

From the correctness of $\mathcal{F}_{\mathsf{MAXPOOL}}$, we have that P_0 and P_1 hold shares of ind_n (which is the index of the maximum value). P_2 receives $\langle\mathsf{ind}_n\rangle_0^L + r$ and $\langle\mathsf{ind}_n\rangle_1^L$ from P_0 and P_1 respectively and reconstructs $t = \mathsf{ind}_n + r \bmod L$ and then computes $k =$

t mod n. P_2 provides P_0 and P_1 with shares $\langle E\rangle_0^L$ and $\langle E\rangle_1^L$ that reconstruct to E_k. Now, observe that $k = ((\mathsf{ind}_n + r) \bmod L) \bmod n$. Let $g = r \bmod n$. Since $n \mid L$, we have that $k = (\mathsf{ind}_n + g) \bmod n$. Now, let shares $\langle E\rangle_j^L = (\langle E^0\rangle_j^L, \langle E^1\rangle_j^L, \cdots, \langle E^{n-1}\rangle_j^L)$. In this, $\langle E^k\rangle_0^L$ and $\langle E^k\rangle_1^L$ reconstruct to 1, while all other $k' \neq k$ reconstruct to 0. Since $\langle D\rangle_j^L = (\langle E^{(-g \bmod n)}\rangle_j^L, \langle E^{(1-g \bmod n)}\rangle_j^L, \cdots, \langle E^{(n-1-g \bmod n)}\rangle_j^L)$, $\langle D^{(k-g) \bmod n}\rangle_0^L$ and $\langle D^{(k-g) \bmod n}\rangle_1^L$ alone will reconstruct to 1 with all other indices reconstructing to 0. Since $(k-g) \bmod n = \mathsf{ind}_n \bmod n$, we have that $\langle D\rangle_0^L$ and $\langle D\rangle_1^L$ reconstruct to E_{k^*}, hence proving the statement.

To argue security, first observe that P_0 and P_1 obtain shares of ind_n from the call to $\mathcal{F}_{\mathsf{MAXPOOL}}$. Now, since r is uniformly random in $\mathbb{Z}_L$, P_2 learns no information from shares $\langle k\rangle_0^L$ and $\langle k\rangle_1^L$ (which reconstruct to $\mathsf{ind}_n + r$). Finally, $P_j, j \in \{0,1\}$, only learn fresh shares of the values $E_{(\mathsf{ind}_n + r) \bmod n}$ and hence any information learned by either party can be perfectly simulated through appropriate shares of 0 (over $\mathbb{Z}_L$). Finally, P_j outputs a fresh share of the final output in Step 5 as the shares are randomized by U_0 and U_1. □

Derivative of Maxpool in the general case. We first observe that this function can be computed using steps similar to 6 & 7 from Algorithm 9. The idea is for the parties to invoke $\mathcal{F}_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ sequentially with shares of the unit vector representing the current maximum. Let E_k, $k \in [n]$, denote the unit vector of length n with 1 in its k^{th} position and 0 everywhere else. E_0 denotes the all-zero vector. Details are presented in Algorithm 19.

Algorithm 19 Derivative of Maxpool $\Pi_{\mathsf{DMP}}(\{P_0, P_1\}, P_2)$:

Input: P_0, P_1 hold $\{\langle x_i\rangle_0^L\}_{i\in[n]}$ and $\{\langle x_i\rangle_1^L\}_{i\in[n]}$, respectively.

Output: P_0, P_1 get $\{\langle z_i\rangle_0^L\}_{i\in[n]}$ and $\{\langle z_i\rangle_1^L\}_{i\in[n]}$, respectively, where $z_i = 1$, when $x_i = \mathsf{Max}(\{x_i\}_{i\in[n]})$ and 0 otherwise.

Common Randomness: P_0 and P_1 hold shares of 0 over $\mathbb{Z}_L^n$ denoted by U_0 and U_1.

1: For $j \in \{0,1\}$, P_j sets $\langle \mathsf{max}_1\rangle_j^L = \langle x_1\rangle_j^L$ and $\langle \mathsf{DMP}_1\rangle_j^L = E_j$.
2: **for** $i = \{2, \ldots, n\}$ **do**
3: For $j \in \{0,1\}$, P_j computes $\langle w_i\rangle_j^L = \langle x_i\rangle_j^L - \langle \mathsf{max}_{i-1}\rangle_j^L$
4: P_0, P_1, P_2 call $\mathcal{F}_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$, having input $\langle w_i\rangle_j^L$ and P_0, P_1 learn $\langle \beta_i\rangle_0^L$ and $\langle \beta_i\rangle_1^L$, respectively.
5: P_0, P_1, P_2 call $\mathcal{F}_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \beta_i\rangle_j^L, \langle \mathsf{max}_{i-1}\rangle_j^L, \langle x_i\rangle_j^L)$ and P_0, P_1 learn $\langle \mathsf{max}_i\rangle_0^L$ and $\langle \mathsf{max}_i\rangle_1^L$, respectively.
6: For $j \in \{0,1\}$, P_j sets $\langle K_i\rangle_j^L = E_{j \cdot i}$.
7: P_0, P_1, P_2 call $\mathcal{F}_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ with $P_j, j \in \{0,1\}$ having input $(\langle \beta_i\rangle_j^L, \langle \mathsf{DMP}_{i-1}\rangle_j^L, \langle K_i\rangle_j^L)$ and P_0, P_1 learn $\langle \mathsf{DMP}_i\rangle_0^L$ and $\langle \mathsf{DMP}_i\rangle_1^L$, respectively.
8: **end for**
9: For $j \in \{0,1\}$, P_j outputs $\langle \mathsf{DMP}_n\rangle_j^L + U_j$.

Lemma A.10. *Protocol $\Pi_{\mathsf{DMP}}(\{P_0, P_1\}, P_2)$ in Algorithm 19 securely realizes $\mathcal{F}_{\mathsf{DMAXPOOL}}$ in the $(\mathcal{F}_{\mathsf{DReLU}}, \mathcal{F}_{\mathsf{SS}})$-hybrid model.*

Proof. For correctness, we show that $\mathsf{Reconst}^L(\langle \mathsf{DMP}_n \rangle_0^L + U_0, \langle \mathsf{DMP}_n \rangle_1^L + U_1) = E_{k^*}$ in Algorithm 19. This proof is nearly identical to the proof of correctness of Algorithm 9. As before, we prove this by induction and will maintain the invariant that max_i holds the value of $\mathsf{max}(x_1, \ldots, x_i)$ and now show that DMP_i holds the value E_k for k s.t. $\mathsf{max}_i = x_k$. It is easy to see that this holds for $i = 1$. Suppose this holds for $i = m - 1$. Then we will prove that it holds for $i = m$. Now, in Step 3, we calculate $w_m = x_m - \mathsf{max}_{m-1}$. By correctness of $\mathcal{F}_{\mathsf{DReLU}}$, $\beta_m = \mathsf{ReLU}'(w_m)$. That is, $\beta_m = 1$ iff $x_m > \mathsf{max}_{m-1}$. Next, by correctness of $\mathcal{F}_{\mathsf{SS}}$, max_m is max_{m-1} if $\beta_m = 0$ and x_m otherwise. In Step 6, we compute shares of $k_m = E_m$. In Step 7, by correctness of $\mathcal{F}_{\mathsf{SS}}$, $\mathsf{DMP}_m = \mathsf{DMP}_{m-1}$ if $\beta_m = 0$ and E_m otherwise. This proves correctness.

To argue security, first observe that P_2 learns no information from the protocol (as $\mathcal{F}_{\mathsf{DReLU}}(\{P_0, P_1\}, P_2)$ and $\mathcal{F}_{\mathsf{SS}}(\{P_0, P_1\}, P_2)$ provide outputs only to P_0 and P_1). Now, $P_j, j \in \{0, 1\}$, only learn fresh shares of the values $\beta_i, \mathsf{max}_i, \mathsf{DMP}_i$, and hence any information learned by either party can be perfectly simulated through appropriate shares of 0 (over $\mathbb{Z}_L$). Finally, P_j outputs a fresh shares of the final output in Step 9 as the shares are randomized by U_0 and U_1.

□

A.3 Privacy against Malicious Adversary

In this section, we show that all our protocols described in Section 3.2 as well as protocols for general NNs obtained by stitching these together satisfy stronger a security, namely, privacy against a malicious server in the client-server model (formalized by [6]). As was already pointed out by Araki *et al.* [6], this can only be achieved when the servers receive no information about the output of the protocol. Formally, we show that, for any malicious server, for any two inputs of the honest clients (holding the data) the view of the server is indistinguishable.

First, intuitively, we show that views are identical with secure correlated randomness. This holds because in all our protocols, the incoming messages to a server are either a fresh share of a value or can be generated using a uniformly random value (e.g., incoming messages of P_2 in private-compare protocol). Thus, irrespective of what the adversary sends in each round, the view of a malicious server can be simulated using uniform randomness and is completely independent of the inputs being used by the clients. Second, in the case when correlated randomness is generated using shared pseudo random function (PRF) keys, to argue security against malicious P_0, we rely on security of the PRF key shared between P_1, P_2 that is unknown to P_0. Using this, we show that incoming messages of P_0 are computationally close to uniform. It is critical that to argue security against a malicious P_0 we do not rely on security of PRF keys known to P_0, i.e., shared keys between (P_0, P_1) or (P_0, P_2). Hence, we do not need to use a malicious secure coin-tossing protocol to generate secure keys between an adversary and an honest server. We only rely on the security of the PRF key shared between two honest servers. Therefore, the exact same protocol gives privacy against a single malicious server. Similar arguments can be made to argue security against a malicious P_1 or malicious P_2.

Appendix B

Falcon: Supplementary Material

We present a brief summary of various NN layers and their equations. NNs, in particular CNNs, form the state-of-the-art techniques for image classification. The operation of NNs is most widely based on stochastic gradient descent and usually iterates over the following three components: a forward pass, a backward pass, and a parameter update phase.

An NN architecture is defined by the combination of layers that compose the network. Various types of layers such as convolution, fully-connected, pooling layers, and activation functions are used in different combinations to form the network. In the training phase, an NN takes in a batch of inputs and outputs "a guess" (forward pass). The ground truth is then used to compute errors using chain rule (back-propagation) and finally update the network parameters (update phase). In the inference phase, the output of the forward pass is used for prediction purposes. Below we look at the various components required by state-of-the-art NNs.

Our general framework supports the following types of layers: convolutional, fully-connected, pooling layers (max and mean pooling), normalization layers, and the ReLU activation function. Together these enable a vast majority of networks used in the ML literature. In the forward pass, each layer takes in an input from the previous layer and generates the output (input for the following layer) using learnable parameters such as weights, biases, etc. The final layer output is used to then compute the loss using a loss function (such as cross-entropy, mean squared etc.). In the backward pass, the final layer loss is propagated backwards through each layer using the chain rule. Finally, each layer uses the associated loss to update its learnable parameters. Below we look at each layer in detail. We use Einstein tensor notation [45] with ϵ_{ab} to denote the Kronecker Delta (to avoid confusion with the error δ) to describe each layer.

B.1 Convolutional Layer

The input to a convolutional layer is a 4D tensor $\mathbb{R}^{w_{\mathsf{in}},h_{\mathsf{in}},D_{\mathsf{in}},B}$ where $w_{\mathsf{in}}, h_{\mathsf{in}}$ are the width and height of the input, D_{in} is the number of input filters, and B is the batch size. The hyper-parameters are the number of output filters D_{out}, the filter size

F, the stride S and the amount of zero padding P. The output of the layer is another 4D tensor $\mathbb{R}^{w_{\mathsf{out}},h_{\mathsf{out}},D_{\mathsf{out}},B}$ where $w_{\mathsf{out}} = (w_{\mathsf{in}} - F + 2 * P)/S + 1$ and $h_{\mathsf{out}} = (h_{\mathsf{in}} - F + 2 * P)/S + 1$. The weights are 4D tensors in $\mathbb{R}^{F,F,D_{\mathsf{in}},D_{\mathsf{out}}}$ and biases are a vector in $\mathbb{R}^{D_{\mathsf{out}}}$.

The forward pass is simply a convolution between the inputs activation and the weights plus the bias. The backward pass as well as the update equations are also convolutions which can all be implemented as matrix multiplications. We use the following notation: activations are represented by a^l and indexed by the layer number $l \in \{1\ldots, L\}$, δ^l represents $\frac{\partial C}{\partial a^l}$, the error of layer l, and weights and biases are respectively represented by w and b. Dimension variables are: $\alpha \in \{1, \ldots, w_{\mathsf{in}}\}, \beta \in \{1, \ldots, h_{\mathsf{in}}\}, r \in \{1, \ldots, D_{\mathsf{in}}\}, d \in \{1, \ldots, D_{\mathsf{out}}\}, b \in \{1, \ldots, B\}, x \in \{1, \ldots, w_{\mathsf{out}}\}$, and $y \in \{1, \ldots, h_{\mathsf{out}}\}$

$$
\begin{aligned}
a^l_{x,y,d,b} &= w_{p,q,r,d} \cdot a^{l-1}_{(xS-P+p),(yS-P+q),r,b} + b_d && \text{(B.1a)}\\
\delta^{l-1}_{\alpha,\beta,r,b} &= \delta^l_{x,y,d,b} \cdot w_{(\alpha+P-xS),(\beta+P-yS),r,d} && \text{(B.1b)}\\
\frac{\partial C}{\partial w_{p,q,r,d}} &= a^{l-1}_{(xS-P+p),(yS-P+q),r,b} \cdot \delta^l_{x,y,d,b} && \text{(B.1c)}\\
\frac{\partial C}{\partial b_d} &= \delta^l_{x',y',d,b'} \cdot \epsilon_{xx'}\epsilon_{yy'}\epsilon_{bb'} && \text{(B.1d)}
\end{aligned}
$$

Eq. B.1a is used for the forward pass, Eq. B.1b is used for back-propagation, and Eqs. B.1c, B.1d are used for updating layer parameters.

B.2 Fully-Connected Layer

The input to a convolutional layer is a matrix in $\mathbb{R}^{c_{\mathsf{in}},B}$ where B is the batch size. The layer is defined by the number of input and output channels $c_{\mathsf{in}}, c_{\mathsf{out}}$. The output of the layer is a matrix in $\mathbb{R}^{c_{\mathsf{in}},B}$. The weights are a matrix in $\mathbb{R}^{c_{\mathsf{in}},c_{\mathsf{out}}}$ and biases form a vector of size $\mathbb{R}^{c_{\mathsf{out}}}$.

The forward pass is a matrix multiplication of the input matrix with the weights matrix and bias added. The backward pass as well as the update equations require matrix multiplications. Using the notation as in the convolutional layer, the equations defining the fully-connected layer are described as follows:

$$
\begin{aligned}
a^l_{y,b} &= w_{p,y} \cdot a^{l-1}_{p,b} + b_y && \text{(B.2a)}\\
\delta^{l-1}_{x,b} &= \delta^l_{y,b} \cdot w_{x,y} && \text{(B.2b)}\\
\frac{\partial C}{\partial w_{p,q}} &= a^{l-1}_{p,b} \cdot \delta^l_{q,b} && \text{(B.2c)}\\
\frac{\partial C}{\partial b_y} &= \delta^l_{y,b'} \cdot \epsilon_{bb'} && \text{(B.2d)}
\end{aligned}
$$

Eq. B.2a is used for the forward pass, Eq. B.2b is used for back-propagation, and Eqs. B.2c, B.2d are used for updating layer parameters.

B.3 Pooling Layer

The input to a pooling layer (specifically Maxpool) is a 4D tensor $\mathbb{R}^{w_{\mathsf{in}},h_{\mathsf{in}},D_{\mathsf{in}},B}$ where $w_{\mathsf{in}}, h_{\mathsf{in}}$ are the width and height of the input, D_{in} the number of input filters, and B the batch size. The hyper-parameters are the filter size F and the stride S. The output of the layer is another 4D tensor $\mathbb{R}^{w_{\mathsf{out}},h_{\mathsf{out}},D_{\mathsf{in}},B}$ where $w_{\mathsf{out}} = (w_{\mathsf{in}} - F)/S + 1$ and $h_{\mathsf{out}} = (h_{\mathsf{in}} - F)/S + 1$. There are no learnable parameters as the output is a fixed function of the input.

The forward pass is max operation over the filter and can be implemented using sequential comparisons. The backward pass requires a matrix multiplication with the derivative of Maxpool (which is a unit vector with 0's everywhere except at the location of the argmax). For optimization, we compute this while computing the Maxpool in the forward pass. Since pooling layers do not introduce any parameters, there is no parameter update required for this layer.

$$a^l_{x,y,d,b} = \left(\max_{p,q} a^{l-1}_{xS+p,yS+q,d',b'}\right) \cdot \epsilon_{dd'}\epsilon_{bb'} \tag{B.3a}$$

$$\delta^{l-1}_{\alpha,\beta,r,b} = \left(\delta^l_{x,y,r,b} \otimes f_{xS+p,yS+q,r',b'}\right) \cdot \tag{B.3b}$$

$$\epsilon_{rr'}\epsilon_{bb'}\epsilon_{\alpha(xS+p)}\epsilon_{\beta(yS+q)} \tag{B.3c}$$

Here, f denotes the derivative of the Maxpool function. Eq. B.3a governs the forward pass and Eq. B.3c governs the back-propagation.

B.4 Normalization Layer

Normalization is typically applied to the output of the first few layers for improved performance on two fronts – stability and efficiency of training. Activations are normalized across a batch by subtracting the mean and dividing by the standard deviation. Finally, these normalized inputs are then scaled using two learnable parameters γ, β.

$$\mu_b = \sum_{\alpha,\beta,r} a^{l-1}_{\alpha,\beta,r,b} \tag{B.4a}$$

$$\sigma_b^2 = \frac{1}{m}\sum_{\alpha,\beta,r}(a^{l-1}_{\alpha,\beta,r,b} - \mu_b)^2 \tag{B.4b}$$

$$z^{l-1}_{\alpha,\beta,r,b} = \frac{(a^{l-1}_{\alpha,\beta,r,b} - \mu_b)}{\sqrt{\sigma_b^2 + \epsilon}} \tag{B.4c}$$

$$a^{l}_{\alpha,\beta,r,b} = \gamma z^{l-1}_{\alpha,\beta,r,b} + \beta \tag{B.4d}$$

where m is the size of each batch. We set $\epsilon = 2^{-10}$. Eqs. B.4a-B.4d form the forward pass of the batch norm layer. The back-prop and update parameters are simply matrix multiplications and can be read off from the source code available at https://github.com/snwagh/falcon-public.

B.5 ReLU Activation

Rectified Linear Unit (ReLU) defined as $(x) = \mathsf{max}(0, x)$ is one of the most popular activation functions used in deep learning. It is applied to the output of most layers and simply applies the ReLU function to each input. Hence, the input and output both are matrices in $\mathbb{R}^{s_{\text{in}},B}$. Since the output is a fixed function of the inputs, there are no learnable parameters in this layer. The forward pass involves computing the ReLU function on each input whereas the backward pass involves a matrix multiplication with the derivative of ReLU function (which is 0 if the input is negative and 1 otherwise). There is no parameter update as there are no learnable parameters.

We use Stochastic Gradient Descent (SGD) to iteratively train the network to learn the right set of parameter values. We use the cross entropy loss function for training given by:

$$C = -\frac{1}{n}\sum_{b}\sum_{j}\left(y_j \ln a^L_{j,b} + (1-y_j)\ln(1-a^L_{j,b})\right) \tag{B.5}$$

where n is the batch size. These above 5 layers can be used to implement a large fraction of the NNs used in deep learning and specifically in computer vision.

Appendix C

Ponytail: Supplementary Material

C.1 Proofs for the Preprocessing Phase

The proof for the online phase Π_{Online}, i.e., of Theorem 5.1 is identical to that of [39] and we omit its extended discussion here. We present proof of the distributed decryption protocols Π_{DDec}, i.e., of Theorem 5.3, later in this section. Finally, we begin by presenting the proof for the preprocessing phase (Theorem 5.2).

C.1.1 Proof of Theorem 5.2

Figure C.1 presents the proof structure for proving malicious security of Π_{Prep}. The ideal functionality for Π_{Prep} is presented in Figure C.2 and the simulator in Figure C.3. The simulator runs a virtual copy of the protocol to simulate interactions with the

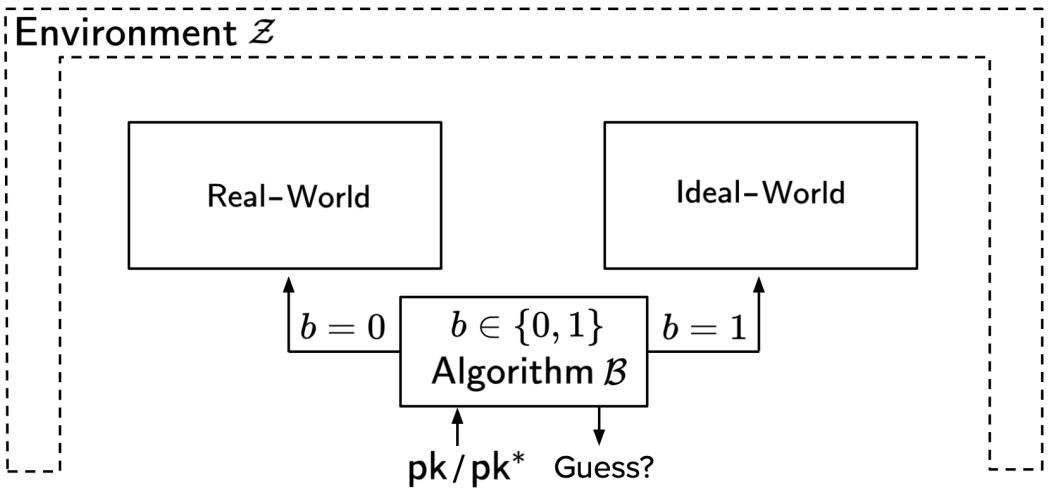


Figure C.1: Ponytail security proof works using a contrapositive argument: assuming there exists an environment $\mathcal{Z}$ that can distinguish between the real-world (Figure 2.1a) and ideal-world (Figure 2.1b), there exists an algorithm $\mathcal{B}$ that can use the distinguishing ability of such an environment to distinguish between a $\mathcal{F}_{\mathsf{KeyGen}}$ generated key and a meaningless one.

adversary. The ability to run $\mathcal{F}_{\mathsf{KeyGen}}$ and $\mathcal{F}_{\mathsf{Rand}}$ allows the simulator to decrypt the inputs of the corrupt parties (since it knows their secret keys). Finally, the simulator then uses these extracted inputs to query the ideal functionality and obtain outputs for all parties. To show that there exists no environment $\mathcal{Z}$ that can distinguish between the real-world and the ideal-world, we use a contrapositive argument. In other words, if there exists such an environment that can distinguish between these two interactions with a non-negligible advantage (let ϵ denote this advantage), then we show the existence of an algorithm $\mathcal{B}$ that can use the distinguishing ability of such an environment to distinguish between a $\mathcal{F}_{\mathsf{KeyGen}}$ generated key and a meaningless one (with advantage $\approx \epsilon/2$).

Such an algorithm $\mathcal{B}$ is given either a normal public key pk or a meaningless one pk^*. It randomly decides to simulate either a real or an ideal world. To notationally distinguish these from the real-world and ideal-world interactions presented to the environment $\mathcal{Z}$, we call the former $\text{real}_{\mathcal{B}}$, $\text{ideal}_{\mathcal{B}}$ and the latter $\text{real}_{\mathcal{Z}}$, $\text{ideal}_{\mathcal{Z}}$. We next prove the following two statements:

(a) (Claim 1) If the public key is meaningless, we show that the environment's view does not depend on whether $\mathcal{B}$ chooses $\text{real}_{\mathcal{B}}$ or $\text{ideal}_{\mathcal{B}}$.

(b) (Claim 2) However, on the other hand, if the public key is normal, we show that $\text{real}_{\mathcal{B}}$ simulated by $\mathcal{B}$ is statistically indistinguishable from $\text{real}_{\mathcal{Z}}$ and $\text{ideal}_{\mathcal{B}}$ is statistically indistinguishable from $\text{ideal}_{\mathcal{Z}}$. In this case, $\mathcal{Z}$ can correctly guess the random choice used by $\mathcal{B}$, i.e., $\text{real}_{\mathcal{B}}$ or $\text{ideal}_{\mathcal{B}}$.

Combining these two observations, $\mathcal{B}$ can correctly guess whether it was given a meaningless key or meaningful key by estimating how well the environment $\mathcal{Z}$ can guess its own random choice $\text{real}_{\mathcal{B}}$ or $\text{ideal}_{\mathcal{B}}$. Since, $\mathcal{B}$ is given meaningful keys (with probability 1/2), the environment guesses the choice of $\mathcal{B}$ with advantage ϵ within this probability space. If that happens, $\mathcal{B}$ outputs a meaningful key otherwise it outputs a meaningless key. It is easy to see that $\mathcal{B}$ succeeds with advantage $\approx \epsilon/2$. Finally, note that Algorithm $\mathcal{B}$ works akin to Simulator $\mathcal{S}$. However, such an Algorithm $\mathcal{B}$ has two additional challenges compared to Simulator $\mathcal{S}$, we present each followed by a statistically indistinguishable workaround.

(a) (Challenge 1) $\mathcal{B}$ does not have access to secret keys (since it is given a public key instead of being generated using $\mathcal{F}_{\mathsf{KeyGen}}$).

- Extractions of the adversarial inputs is done using the knowledge extractors of zero-knowledge proofs. Note that Simulator $\mathcal{B}$ internally runs copies of $\mathcal{F}_{\mathsf{Rand}}$ and the distinguishing environment $\mathcal{Z}$. This allows it to rewind the adversary and issue challenges of its choice. This enables us to run the extractors presented in the soundness argument in Sec. 5.2.

(b) (Challenge 2) $\mathcal{B}$ does not have access to the inputs of the honest parties.

- $\mathcal{B}$ simulates the proof using the honest verifier simulator presented in the zero-knowledge argument in Sec. 5.2. The challenge matrix w of the accepting transcript can be output from $\mathcal{F}_{\mathsf{Rand}}$ (as $\mathcal{B}$ controls the copy of $\mathcal{F}_{\mathsf{Rand}}$).

$\mathcal{F}_{\mathsf{Prep}}$

Let $\mathcal{A}$ denote the set of indices corresponding to the corrupt parties. GenMAC is a macro called multiple times by the functionality.

GenMAC(a, Δ, α): This subroutine will be called multiple times by the functionality.
(A) Receive MAC shares $\{\gamma^i\}_{i\in\mathcal{A}}$ from the adversary.
(B) Set $\gamma(a) \leftarrow \alpha \cdot a$ and $\gamma \leftarrow \gamma(a) + \Delta$.
(C) Sample random values for $\gamma(a)^i \xleftarrow{r} \mathbb{Z}_p$ for $i \notin \mathcal{A}$ subject to $\gamma = \sum_{i=1}^{n} \gamma(a)^i$.
(D) Return $\gamma(a)^i$ to party P_i for $i \notin \mathcal{A}$.

Initialize: On input (init,p) from all the parties, do the following:

(A) Receive share α^i from the adversary for $i \in \mathcal{A}$ and sample $\alpha^i \xleftarrow{r} \mathbb{Z}_p$ for each $i \notin \mathcal{A}$. Set $\alpha \leftarrow \alpha^1 + \ldots + \alpha^n \pmod p$.
(B) Wait for Ok or Abort from the adversary. If the adversary sends Abort, send Abort to all parties and abort otherwise send α^i to party P_i.

Authenticated Singles: On receiving input (Authenticated Singles) from all parties, do the following:

(A) Wait for Ok or Abort from the adversary. If the adversary sends Abort, send Abort to all parties and abort. Otherwise choose random values $r_k^i \in \mathbb{Z}_p$ for $i \notin \mathcal{A}$ and send them to P_i for all $i \notin \mathcal{A}$.
(B) For each corrupt party $P_i, i \in \mathcal{A}$, the adversary specifies a share r^i.
(C) The environment specifies MAC errors Δ_k. Let $r_k = \sum_i r_k^i$.
(D) Run the Macro GenMAC(r_k, Δ_k, α).

Matrix Triples: On receiving input (Matrix Triples, d_1, d_2, d_3) from all parties, do the following:

(A) Wait for Ok or Abort from the adversary. If the adversary sends Abort, send Abort to all parties and abort. Otherwise choose random matrices $A^i \leftarrow U(R_p)^{d_1\times d_2}$ and $B^i \leftarrow U(R_p)^{d_2\times d_3}$.
(B) For each corrupt party $P_i, i \in A$, the environment specifies shares $A^i \leftarrow U(R_p)^{d_1\times d_2}$, $B^i \leftarrow U(R_p)^{d_2\times d_3}$, and $C^i \leftarrow U(R_p)^{d_1\times d_3}$.
(C) The environment specifies the MAC errors Δ_A, Δ_B, and Δ_C.
(D) Set $A = \sum_i A^i, B = \sum_i B^i$, and $C = A \times B + \delta_{AB}$.
(E) For each honest party $P_i, i \notin \mathcal{A}$, randomly choose C^i subject to $C = \sum_i C^i$.
(F) Run Macros GenMAC(A, Δ_A, α), GenMAC(B, Δ_B, α), and GenMAC$((A \times B), \Delta_C, \alpha)$.

Figure C.2: Ideal functionality for Π_{Prep}

$\mathcal{S}_{\mathsf{Prep}}$

$\mathcal{S}_{\mathsf{ext}}(\mathfrak{c}_m)$: This subroutine will be called multiple times by the simulator to extract the error Δ introduced by Π_{DDec}. The simulator uses $\mathcal{S}_{\mathsf{DDec}}$ as a sub-routine to extract Δ. In particular, the simulator has access to m as well as the final output m' and sets $\Delta = m' - m$ (as it runs its own $\mathcal{F}_{\mathsf{KeyGenDec}}$)

Initialize:

(A) The simulator performs initialization steps of Π_{Prep}. A call to simulated $\mathcal{F}_{\mathsf{KeyGen}}$ is made and pk, sk are locally stored.
(B) Simulator decrypts all encrypted ciphertexts and obtains $\alpha^1, \ldots, \alpha^n$.

Authenticated Singles:

(A) Simulator performs step 1,2 as per protocol and decrypts every broadcast ciphertext to obtain r_k^i.
(B) Step 3 is performed as per the protocol setting $\Delta_k \leftarrow \mathcal{S}_{\mathsf{ext}}(\mathfrak{c}_{r_k \cdot \alpha})$.
(C) Call Authenticated Singles on $\mathcal{F}_{\mathsf{Prep}}$ using r_k^i at step 2 and Δ_k at step 3.

Matrix Triples:

(A) Simulator performs step 1,2 as per protocol and decrypts every broadcast ciphertext to obtain A_{jk}^i and B_{kl}^i (using σ^{-1} and τ^{-1}).
(B) Steps 3, 4, 5 are performed as per protocol and generate $\Delta_{A_{jk}} \leftarrow \mathcal{S}_{\mathsf{ext}}(\mathfrak{c}_{A_{jk} \cdot \alpha})$ and $\Delta_{B_{jk}} \leftarrow \mathcal{S}_{\mathsf{ext}}(\mathfrak{c}_{B_{jk} \cdot \alpha})$.
(C) Compute steps 6, 7 as per the protocol and compute $\delta_{C_{jl}} \leftarrow \mathcal{S}_{\mathsf{ext}}(\mathfrak{c}_{C_{jl}})$.
(D) Compute steps 8 as per the protocol and compute $\Delta_{C_{jl}} \leftarrow \mathcal{S}_{\mathsf{ext}}(\mathfrak{c}_{C_{jl} \cdot \alpha})$.
(E) Execute steps 9, 10 as per protocol.
(F) Call Matrix Triples in $\mathcal{F}_{\mathsf{Prep}}$ with inputs A^i, B^i where A^i is a matrix formed by padding blocks A_{jk}^i appropriately and B^i is a matrix formed by padding B_{kl}^i appropriately in step 2, inputs Δ_A, Δ_B, and Δ_C in step 4, 7 and input δ_C in step 5, where $\Delta_A, \Delta_B, \Delta_C$, and δ_C are formed by appropriately padding $\Delta_{A_{jk}}, \Delta_{B_{kl}}, \Delta_{C_{jl}}$, and $\delta_{C_{jl}}$ respectively.

Figure C.3: Simulator for $\mathcal{F}_{\mathsf{Prep}}$

Other than these differences, $\mathcal{B}$ works exactly as Simulator $\mathcal{S}$ described in Figure C.3. Finally, to complete the proof, we prove Claim 1 and Claim 2 above.

Proving Claim 1. If the key is meaningless, the encryptions contain statistically no information about the encrypted values. Since the zero-knowledge proofs are simulated and the outputs of Π_{Reshare} are secret shared, the honest parties inputs are not revealed to the environment.

Proving Claim 2. If the key is meaningful, $\mathcal{B}$ chooses either $\text{real}_{\mathcal{B}}$ or $\text{ideal}_{\mathcal{B}}$. In $\text{ideal}_{\mathcal{B}}$, the only differences between Simulator $\mathcal{S}$ and $\mathcal{B}$ are those presented in Challenge 1 and 2 above, the workarounds for which work with statistical indistinguishability from those of Simulator $\mathcal{S}$. Similarly, in $\text{real}_{\mathcal{B}}$, $\mathcal{B}$ generates a statistically indistinguishable view from the real protocol execution.

C.1.2 Proof of Theorem 5.3

Simulator $\mathcal{S}_{\mathsf{DDec}}$ for Π_{DDec} is described in Figure C.5 and the ideal functionality is described in Figure C.4. To prove the security result, we need to show the following:

(a) (Part 1) The transcripts t^i generated by the simulator (in the "internal run") and sent to the adversary are indistinguishable from the real-world transcripts.

(b) (Part 2) Extract δ and the outputs of the adversary to be able to send them to the functionality $\mathcal{F}_{\mathsf{KeyGenDec}}$.

(c) (Part 3) Finally, generate the correct distribution of the outputs.

Proving Part 1. To see this, note that the simulator behaves honestly in generating t^i for all parties except for one honest party P_j. Hence, we only need to show that the distribution of t^j when generated by the simulator is indistinguishable from an honest generation of t^j. Note that

$$\begin{aligned} t^j - \tilde{t}^j &= (\Delta \cdot r^j + v^j + e^j) - (\Delta \cdot r^j + e^j - \sum_{i \neq j} v^i) \pmod q \\ &= \sum_i v^i \pmod q \\ &= \Delta \cdot m + e \end{aligned} \tag{C.1}$$

We use a hybrid approach to show that these distributions are indistinguishable. Let $t^j_{\mathsf{h}} = \tilde{t}^j + \Delta \cdot m$. Since r^j is uniformly random in R_p and $\Delta = q/p$, the distributions of t^j_{h} and $\tilde{t}^j$ are statistically indistinguishable. Finally, since $e \leq e^j \cdot 2^{\mathsf{sec_{dd}}}$, the distributions of $t^j = t^j_{\mathsf{h}} + e$ and t^j_{h} are statistically indistinguishable.

Proving Part 2. This is easy to see, the extraction is provided in Steps (F),(G) of $\mathcal{S}_{\mathsf{DDec}}$.

$\mathcal{F}_{\mathsf{KeyGenDec}}$

Upon receiving start, get $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$. Send pk to adversary and store sk.

Distributed Key Generation:

(A) Receive the shares of the secret key s_j from the adversary.
(B) Construct a complete set of shares $\{s_i\}_{i=1}^n$ consistent with the adversary's input and the secret key sk. Send pk to all the players and s_i to each honest party P_i.

Distributed Decryption:

(A) "Adversary" sends δ and random shares $\tilde{m}^i$ for $i \in \mathcal{A}$ to the functionality.
(B) Upon receiving $\mathsf{decrypt}(\mathfrak{c}_m, B)$ decrypt $m \leftarrow \mathfrak{c}_m$ and store it.
(C) Generate $\tilde{m}^i$ for $i \notin \mathcal{A}$ such that $m + \delta = \sum_i \tilde{m}^i$. Send $\tilde{m}^i$ for $i \notin \mathcal{A}$ to the honest parties.

Figure C.4: Functionality for distributed key generation and decryption

Proving Part 3. In the ideal simulation, the outputs of all parties are random shares with the constraint that $\sum \tilde{m}^i = m + \delta$ where δ is as given in Step (G). On the other hand, in the real simulation, the output is as given by Π_{DDec}, i.e., random shares of $\lfloor \Delta^{-1} \cdot (\sum_i t^i) \rceil - \sum_i r^i \pmod p$. From the real simulation, we know that $(\sum_i t^i) = \Delta \cdot (m + \sum_i r^i) + (e + \sum_i e^i)$ and from the ideal simulation, we know that

$$(\delta - 1/2) \cdot \Delta \leq \sum_i e^i \leq (\delta + 1/2) \cdot \Delta \tag{C.2}$$

Hence, $\lfloor \Delta^{-1} \cdot (\sum_i t^i) \rceil - \sum_i r^i \pmod p = m + \delta$ when $(e + \sum_i e^i) \leq (\delta + 1/2) \cdot \Delta$. But since $e << \sum_i e^i$, this holds with overwhelming probability and that completes the proof.

C.2 Additional Protocols and Functionalities

This section presents the following protocols and functionalities for the sake of completeness.

(A) Protocol for adding MACs Π_{AddMacs} (Figure C.6).
(B) Protocol for MAC check Π_{MACCheck} (Figure C.7).
(C) Protocol for online phase Π_{Online} (Figure C.8).
(D) Ideal functionality $\mathcal{F}_{\mathsf{Online}}$ (Figure C.9).
(E) Ideal functionality $\mathcal{F}_{\mathsf{Rand}}$ (Figure C.10).

$\mathcal{S}_{\mathsf{DDec}}$

Key Generation: Key distribution stage.

(A) Simulator obtains pk and $\{\mathsf{sk}^i\}_{i\in\mathcal{A}}$ and internally sets random $\{\mathsf{sk}^i\}_{i\notin\mathcal{A}}$ such that sk is a full vector of 0's.

(B) Send pk to the adversary.

Distributed Decryption: Simulates distributed decryption

(A) Upon decrypt $(\mathfrak{c}_m, B)$, compute the value v^i for all players except one honest player P^j.

(B) For each $i \in A$, on receiving the message t^i from malicious party P^i, it computes unique $e^i = t^i - v^i \pmod{\Delta}$ and $r^i = \lfloor \Delta^{-1} \cdot (t^i - v^i) \rceil$ so that $\Delta r^i + e^i = t^i - v^i$.

(C) It samples $r^j \leftarrow U(R_p)$ and $e^j \leftarrow U(R_{B\cdot 2^{\mathsf{sec}_{\mathsf{dd}}}})$ and computes

$$\tilde{t}^j = \Delta \cdot r^j + e^j - \sum_{i\neq j} v^i \pmod{q}.$$

(D) For each honest player, it computes t^i honestly.

(E) The simulator sends these t^i for all $i \notin A$, $i \neq j$, and $\tilde{t}^j$ to Adversary.

(F) For all $i \in A$, the simulator sets $\tilde{m}^i = -r^i$.

(G) The simulator sets

$$\delta := \left\lfloor \frac{\sum_i e^i}{\Delta} \right\rceil$$

(H) The simulator sends δ, and $\tilde{m}^i$ to the functionality $\mathcal{F}_{\mathsf{KeyGenDec}}$.

Figure C.5: Simulator for distributed decryption.

Π_{AddMacs}

Usage: On public input $\mathfrak{c}_a$ (and $\mathfrak{c}_\alpha$ generated during initialization phase), the protocol generates shares $\gamma(a)^i$.

AddMacs: All parties

(A) All parties set $\mathfrak{c}_{\alpha\cdot a} \leftarrow \mathfrak{c}_\alpha \boxtimes \mathfrak{c}_a$.

(B) Parties run Π_{DDec} to generate $\gamma(a)^1, \ldots \gamma(a)^n \leftarrow \mathsf{DDec}(\mathfrak{c}_{\alpha\cdot a})$.

(C) Output $\big(\gamma(a)^1, \ldots \gamma(a)^n\big)$.

Figure C.6: Sub-protocol for adding MACs.

Π_{MACCheck}

Each party has inputs α_i and $(\gamma(a)_j)_i$ for $j \in \{1, 2, \ldots, t\}$. All players have a public set opened values $\{a_1, a_2, \ldots, a_t\}$. The protocol either succeeds or aborts (if an inconsistent MAC value is found)

(A) Parties call $\mathcal{F}_{\mathsf{Rand}}$ to generate a random seed s.

(B) Players sample a random set of values $\{r_j\}_{j=1}^{t}$ using s as a seed.

(C) Each player computes the following values

  (a) Public value $a = \sum_{j=1}^{t} r_j \cdot a_j$.

  (b) Share $\gamma_i = \sum_{j=1}^{t} r_j \cdot (\gamma(a)_j)_i$.

  (c) Share $\sigma_i = \gamma_i - \alpha_i \cdot a$.

(D) Parties call $\mathcal{F}_{\mathsf{Commit}}$ with $(\mathsf{Commit}, \sigma_i)$ for $i \in \{1, 2, \ldots, n\}$.

(E) Parties then call $\mathcal{F}_{\mathsf{Commit}}$ with $(\mathsf{Open}, \sigma_i)$ for $i \in \{1, 2, \ldots, n\}$ and all players obtain $\{\sigma_i\}_{i=1}^{n}$.

(F) If $\sigma_1 + \ldots + \sigma_n \neq 0$, players abort the protocol.

Figure C.7: The MAC check protocol verifies the consistency of a list of opened values

Π_{Online}

Initialize: The parties first invoke the preprocessing Π_{Prep} to get the shared secret key $[\![\alpha]\!]$ and verified $\mathfrak{c}_\alpha = \mathsf{Enc}(\alpha)$, a sufficient number of the following correlated random objects of appropriate dimensions:

(A) Matrix multiplication triples $([\![a]\!], [\![b]\!], [\![c]\!])$.
(B) Convolution triples $([\![x]\!], [\![y]\!], [\![z]\!])$.
(C) Multiplication triples $([\![a]\!], [\![b]\!], [\![c]\!])$.

Then the steps below are performed according to the computation circuit.

Input: To share party P_i's input, all parties pick a fresh $\mathsf{Authenticated\ Single}[\![a]\!]$ from the set of available ones and then the following is performed:

(A) $[\![a]\!]$ is opened to P_i.
(B) P_i broadcasts $\delta = x_i - a$.
(C) The parties compute $[\![x]\!] = [\![a]\!] + \delta$ using local computation.

Add: To add two shares $[\![x]\!], [\![y]\!]$, parties locally compute $[\![x + y]\!]$.

Matrix Multiply: To multiply two matrices $[\![X]\!]$ and $[\![Y]\!]$, with each dimension a multiple of ysl, the parties do the following:

(A) Take a fresh matrix multiplication triple $([\![A]\!], [\![B]\!], [\![C]\!])$ from the available set of appropriate dimensions.
(B) $\epsilon = [\![X - A]\!]$ and $\delta = [\![Y - B]\!]$ are opened.
(C) Compute $[\![Z]\!] = [\![C]\!] + \epsilon \times [\![B]\!] + [\![A]\!] \times \delta + \epsilon \times \delta$.

Output: Parties perform $\mathsf{MACCheck}$ on all openings so far. If no party aborts, the players open the output value $[\![\mathsf{Out}]\!]$ and if $\mathsf{MACCheck}$ goes through, each receives the output Out.

Figure C.8: The online phase for MPC

$\mathcal{F}_{\mathsf{Online}}$

Initialize: On receiving (init, p) from all parties, the functionality stores the modulus p and initializes an empty array $\mathsf{Value}[]$. If the adversary sends abort, send abort to all parties and abort.

Input: On input $(\mathsf{input}, \mathsf{id}, x)$ from P_i and $(\mathsf{input}, \mathsf{id}, \perp)$ from all other parties, with fresh identifier id, set $\mathsf{Value}[\mathsf{id}] = x$.

Add: On input $(\mathsf{Add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ with $\mathsf{id}_1, \mathsf{id}_2$ already defined, set $\mathsf{Value}[\mathsf{id}_3] = \mathsf{Value}[\mathsf{id}_1] + \mathsf{Value}[\mathsf{id}_2]$.

Matrix Multiply: On input $(\mathsf{Matrix\ Multiply}, \{\mathsf{id}^{\mathsf{A}}_{\mathsf{ij}}\}, \{\mathsf{id}^{\mathsf{B}}_{\mathsf{jk}}\}, \{\mathsf{id}^{\mathsf{C}}_{\mathsf{ik}}\})$ with appropriate dimensions and OK from adversary, set $\mathsf{Value}[\mathsf{id}^{\mathsf{C}}_{\mathsf{ik}}] = \sum_{\mathsf{j}} \mathsf{Value}[\mathsf{id}^{\mathsf{A}}_{\mathsf{ij}}] \cdot \mathsf{Value}[\mathsf{id}^{\mathsf{B}}_{\mathsf{jk}}]$. If adversary sends Abort, send the message abort to all parties and abort.

Output: On input $(\mathsf{Output}, \mathsf{id}, i)$ from all the parties and $\mathsf{Value}[\mathsf{id}]$ is defined, then

(A) Public output: If $i = 0$, send $\mathsf{Value}[\mathsf{id}]$ to the adversary and wait for OK or Abort. If adversary sends Abort, send Abort to all parties and abort. If adversary sends OK, send $\mathsf{Value}[\mathsf{id}]$ to all the parties.

(B) Private output (corrupt): If $i \neq 0$ and P_i is corrupt, then send $\mathsf{Value}[\mathsf{id}]$ to the adversary and wait for OK or Abort. If the adversary sends Abort, send Abort to all parties and abort.

(C) Private output (honest): If $i \neq 0$ and P_i is honest, wait for the adversary to send OK or Abort. If the adversary sends Abort, send Abort to all parties and abort otherwise send $\mathsf{Value}[\mathsf{id}]$ to party P_i.

Figure C.9: Ideal functionality for Π_{Online}

$\mathcal{F}_{\mathsf{Rand}}$

Init: On input $(\mathsf{Init}, \mathsf{sid}, \mathbb{F})$ from all parties set the sampling field $\mathbb{F}$ and wait for incoming messages.

Random: On input $(\mathsf{Random}, \mathsf{sid})$ from all parties sample $r \in \mathbb{F}$ and send r to the Simulator $\mathcal{S}$. If $\mathcal{S}$ replies with Abort then send Abort to all parties. If the reply is OK then output r to all parties.

Figure C.10: Functionality for jointly sampling a random public element in MPC

Appendix D

Network Architectures

Layer	Input Size	Description	Output
Fully-Connected Layer	28×28	Fully-Connected layer	128
ReLU Activation	128	$\mathsf{ReLU}(\cdot)$ on each input	128
Fully-Connected Layer	128	Fully-Connected layer	128
ReLU Activation	128	$\mathsf{ReLU}(\cdot)$ on each input	128
Fully-Connected Layer	128	Fully-Connected layer	10
ReLU Activation	10	$\mathsf{ReLU}(\cdot)$ on each input	10

Figure D.1: NN architecture from SecureML [85] for training over the MNIST dataset.

Layer	Input Size	Description	Output
Convolution	$28 \times 28 \times 1$	Window size 2×2, Stride $(2, 2)$, Padding $(0, 0)$, output channels 5	$14 \times 14 \times 5$
ReLU Activation	$14 \times 14 \times 5$	$\mathsf{ReLU}(\cdot)$ on each input	$14 \times 14 \times 5$
Fully-Connected Layer	980	Fully-Connected layer	100
ReLU Activation	100	$\mathsf{ReLU}(\cdot)$ on each input	100
Fully-Connected Layer	100	Fully-Connected layer	10
ReLU Activation	10	$\mathsf{ReLU}(\cdot)$ on each input	10

Figure D.2: NN architecture used in Chameleon [95] for training over the MNIST dataset.

Layer	Input Size	Description	Output
Convolution	$28 \times 28 \times 1$	Window size 5×5, Stride $(1, 1)$, Padding $(0, 0)$, output channels 16	$24 \times 24 \times 16$
ReLU Activation	$24 \times 24 \times 16$	$\mathsf{ReLU}(\cdot)$ on each input	$24 \times 24 \times 16$
Max Pooling	$24 \times 24 \times 16$	Window size 2×2, Stride $(2, 2)$	$12 \times 12 \times 16$
Convolution	$12 \times 12 \times 16$	Window size 5×5, Stride $(1, 1)$, Padding $(0, 0)$, output channels 16	$8 \times 8 \times 16$
ReLU Activation	$8 \times 8 \times 16$	$\mathsf{ReLU}(\cdot)$ on each input	$8 \times 8 \times 16$
Max Pooling	$8 \times 8 \times 16$	Window size 2×2, Stride $(2, 2)$	$4 \times 4 \times 16$
Fully-Connected Layer	256	Fully-Connected layer	100
ReLU Activation	100	$\mathsf{ReLU}(\cdot)$ on each input	100
Fully-Connected Layer	100	Fully-Connected layer	10
ReLU Activation	10	$\mathsf{ReLU}(\cdot)$ on each input	10

Figure D.3: NN architecture used in MiniONN [77] for training over the MNIST dataset.

Layer	Input Size	Description	Output
Convolution	$28 \times 28 \times 1$	Window size 5×5, Stride $(1, 1)$, Padding $(0, 0)$, output channels 20	$24 \times 24 \times 20$
ReLU Activation	$24 \times 24 \times 20$	$\mathsf{ReLU}(\cdot)$ on each input	$24 \times 24 \times 20$
Max Pooling	$24 \times 24 \times 20$	Window size 2×2, Stride $(2, 2)$	$12 \times 12 \times 20$
Convolution	$12 \times 12 \times 20$	Window size 5×5, Stride $(1, 1)$, Padding $(0, 0)$, output channels 50	$8 \times 8 \times 50$
ReLU Activation	$8 \times 8 \times 50$	$\mathsf{ReLU}(\cdot)$ on each input	$8 \times 8 \times 50$
Max Pooling	$8 \times 8 \times 50$	Window size 2×2, Stride $(2, 2)$	$4 \times 4 \times 50$
Fully-Connected Layer	800	Fully-Connected layer	500
ReLU Activation	500	$\mathsf{ReLU}(\cdot)$ on each input	500
Fully-Connected Layer	500	Fully-Connected layer	10
ReLU Activation	10	$\mathsf{ReLU}(\cdot)$ on each input	10

Figure D.4: The LeNet network architecture [76] for training over the MNIST dataset.

Layer	Input Size	Description	Output
Convolution	$32 \times 32 \times 3$	Window size 11×11, Stride $(4, 4)$, Padding $(9, 9)$, output channels 96	$11 \times 11 \times 96$
ReLU Activation	$11 \times 11 \times 96$	$\mathsf{ReLU}(\cdot)$ on each input	$11 \times 11 \times 96$
Max Pooling	$11 \times 11 \times 96$	Window size 3×3, Stride $(2, 2)$	$5 \times 5 \times 96$
Batch Normalization	$5 \times 5 \times 96$	Normalization and linear scaling using learnable parameters γ, β	$5 \times 5 \times 96$
Convolution	$5 \times 5 \times 96$	Window size 5×5, Stride $(1, 1)$, Padding $(1, 1)$, output channels 256	$3 \times 3 \times 256$
ReLU Activation	$3 \times 3 \times 256$	$\mathsf{ReLU}(\cdot)$ on each input	$3 \times 3 \times 256$
Max Pooling	$3 \times 3 \times 256$	Window size 3×3, Stride $(2, 2)$	$1 \times 1 \times 256$
Batch Normalization	$1 \times 1 \times 256$	Normalization and linear scaling using learnable parameters γ, β	$1 \times 1 \times 256$
Convolution	$1 \times 1 \times 256$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 384	$1 \times 1 \times 384$
ReLU Activation	$1 \times 1 \times 384$	$\mathsf{ReLU}(\cdot)$ on each input	$1 \times 1 \times 384$
Convolution	$1 \times 1 \times 384$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 384	$1 \times 1 \times 384$
ReLU Activation	$1 \times 1 \times 384$	$\mathsf{ReLU}(\cdot)$ on each input	$1 \times 1 \times 384$
Convolution	$1 \times 1 \times 384$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 256	$1 \times 1 \times 256$
ReLU Activation	$1 \times 1 \times 256$	$\mathsf{ReLU}(\cdot)$ on each input	$1 \times 1 \times 256$
Fully-Connected Layer	256	Fully-Connected layer	256
ReLU Activation	256	$\mathsf{ReLU}(\cdot)$ on each input	256
Fully-Connected Layer	256	Fully-Connected layer	256
ReLU Activation	256	$\mathsf{ReLU}(\cdot)$ on each input	256
Fully-Connected Layer	256	Fully-Connected layer	10
ReLU Activation	10	$\mathsf{ReLU}(\cdot)$ on each input	10

Figure D.5: The AlexNet network architecture [69] for training over the CIFAR-10 dataset.

Layer	Input Size	Description	Output
Convolution	$32 \times 32 \times 3$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 64	$32 \times 32 \times 64$
ReLU Activation	$32 \times 32 \times 64$	$\mathsf{ReLU}(\cdot)$ on each input	$32 \times 32 \times 64$
Convolution	$32 \times 32 \times 64$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 64	$32 \times 32 \times 64$
ReLU Activation	$32 \times 32 \times 64$	$\mathsf{ReLU}(\cdot)$ on each input	$32 \times 32 \times 64$
Max Pooling	$32 \times 32 \times 64$	Window size 2×2, Stride $(2, 2)$	$16 \times 16 \times 64$
Convolution	$16 \times 16 \times 64$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 128	$16 \times 16 \times 128$
ReLU Activation	$16 \times 16 \times 128$	$\mathsf{ReLU}(\cdot)$ on each input	$16 \times 16 \times 128$
Convolution	$16 \times 16 \times 128$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 128	$16 \times 16 \times 128$
ReLU Activation	$16 \times 16 \times 128$	$\mathsf{ReLU}(\cdot)$ on each input	$16 \times 16 \times 128$
Max Pooling	$16 \times 16 \times 128$	Window size 2×2, Stride $(2, 2)$	$8 \times 8 \times 128$
Convolution	$8 \times 8 \times 128$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 256	$8 \times 8 \times 256$
ReLU Activation	$8 \times 8 \times 256$	$\mathsf{ReLU}(\cdot)$ on each input	$8 \times 8 \times 256$
Convolution	$8 \times 8 \times 256$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 256	$8 \times 8 \times 256$
ReLU Activation	$8 \times 8 \times 256$	$\mathsf{ReLU}(\cdot)$ on each input	$8 \times 8 \times 256$
Convolution	$8 \times 8 \times 256$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 256	$8 \times 8 \times 256$
ReLU Activation	$8 \times 8 \times 256$	$\mathsf{ReLU}(\cdot)$ on each input	$8 \times 8 \times 256$
Max Pooling	$8 \times 8 \times 256$	Window size 2×2, Stride $(2, 2)$	$4 \times 4 \times 256$
Convolution	$4 \times 4 \times 256$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 512	$4 \times 4 \times 512$
ReLU Activation	$4 \times 4 \times 512$	$\mathsf{ReLU}(\cdot)$ on each input	$4 \times 4 \times 512$
Convolution	$4 \times 4 \times 512$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 512	$4 \times 4 \times 512$
ReLU Activation	$4 \times 4 \times 512$	$\mathsf{ReLU}(\cdot)$ on each input	$4 \times 4 \times 512$
Convolution	$4 \times 4 \times 512$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 512	$4 \times 4 \times 512$
ReLU Activation	$4 \times 4 \times 512$	$\mathsf{ReLU}(\cdot)$ on each input	$4 \times 4 \times 512$
Max Pooling	$4 \times 4 \times 512$	Window size 2×2, Stride $(2, 2)$	$2 \times 2 \times 512$
Convolution	$2 \times 2 \times 512$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 512	$2 \times 2 \times 512$
ReLU Activation	$2 \times 2 \times 512$	$\mathsf{ReLU}(\cdot)$ on each input	$2 \times 2 \times 512$
Convolution	$2 \times 2 \times 512$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 512	$2 \times 2 \times 512$
ReLU Activation	$2 \times 2 \times 512$	$\mathsf{ReLU}(\cdot)$ on each input	$2 \times 2 \times 512$
Convolution	$2 \times 2 \times 512$	Window size 3×3, Stride $(1, 1)$, Padding $(1, 1)$, output channels 512	$2 \times 2 \times 512$
ReLU Activation	$2 \times 2 \times 512$	$\mathsf{ReLU}(\cdot)$ on each input	$2 \times 2 \times 512$
Max Pooling	$2 \times 2 \times 512$	Window size 2×2, Stride $(2, 2)$	$1 \times 1 \times 512$
Fully-Connected Layer	512	Fully-Connected layer	4096
ReLU Activation	4096	$\mathsf{ReLU}(\cdot)$ on each input	4096
Fully-Connected Layer	4096	Fully-Connected layer	4096
ReLU Activation	4096	$\mathsf{ReLU}(\cdot)$ on each input	4096
Fully-Connected Layer	4096	Fully-Connected layer	1000
ReLU Activation	1000	$\mathsf{ReLU}(\cdot)$ on each input	1000

Figure D.6: The VGG16 network architecture [102] for training over the CIFAR-10 dataset.

Bibliography

All of the URLs listed here are valid as of April 2020.

[1] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. "Secure computation on floating point numbers." In: *Symposium on Network and Distributed System Security (NDSS)*. 2013 (Referenced on page 57).

[2] Joshua Allen, Bolin Ding, Janardhan Kulkarni, Harsha Nori, Olga Ohrimenko, and Sergey Yekhanin. "An algorithmic framework for differentially private data analysis on trusted processors." In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019 (Referenced on page 3).

[3] Abdelrahaman Aly, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. *SCALE-MAMBA v1.2: Documentation*. 2018. URL: https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf (Referenced on page 94).

[4] *Announcing SecureNN in tf-encrypted*. https://mc.ai/announcing-securenn-in-tf-encrypted/. 2018 (Referenced on page 8).

[5] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. "Optimized honest-majority MPC for malicious adversaries – breaking the 1 billion-gate per second barrier." In: *IEEE Symposium on Security and Privacy (S&P)*. 2017 (Referenced on page 72).

[6] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. "High-throughput semi-honest secure three-party computation with an honest majority." In: *ACM Conference on Computer and Communications Security (CCS)*. 2016 (Referenced on pages 15, 27, 47, 50, 51, 105).

[7] Charles Arthur. *Twitter to Introduce PhotoDNA System to Block Child Abuse Images*. https://www.theguardian.com/technology/2013/jul/22/twitter-photodna-child-abuse. 2013 (Referenced on page 6).

[8] Artem Babenko and Victor Lempitsky. "Efficient indexing of billion-scale datasets of deep descriptors." In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2016 (Referenced on page 92).

[9] Assi Barak, Daniel Escudero, Anders Dalskov, and Marcel Keller. *Secure evaluation of quantized neural networks*. https://eprint.iacr.org/2019/131. 2019 (Referenced on page 72).

[10] Carsten Baum, Daniele Cozzo, and Nigel P Smart. "Using TopGear in Overdrive: A more efficient ZKPoK for SPDZ." In: *International Conference on Selected Areas in Cryptography*. 2019 (Referenced on pages 72, 81–83, 86, 90, 95).

[11] Donald Beaver. "Efficient multiparty protocols using circuit randomization." In: *Annual International Cryptology Conference*. Springer. 1991, pp. 420–432 (Referenced on pages 26, 28).

[12] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)." In: *ACM Symposium on Theory of Computing (STOC)*. 1988 (Referenced on pages 3, 15).

[13] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. "Semi-homomorphic encryption and multiparty computation." In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2011, pp. 169–188 (Referenced on pages 72, 74).

[14] Fabrice Benhamouda, Jan Camenisch, Stephan Krenn, Vadim Lyubashevsky, and Gregory Neven. "Better zero-knowledge proofs for lattice encryption and their application to group signatures." In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2014, pp. 551–572 (Referenced on pages 86, 87).

[15] George Robert Blakley. "Safeguarding cryptographic keys." In: *International Workshop on Managing Requirements Knowledge (MARK)*. 1979 (Referenced on page 15).

[16] Zvika Brakerski. "Fully homomorphic encryption without modulus switching from classical GapSVP." In: *Advances in Cryptology—CRYPTO*. Springer. 2012, pp. 868–886 (Referenced on pages 19, 81).

[17] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping." In: *ACM Transactions on Computation Theory (TOCT)*. 2014 (Referenced on pages 3, 18, 80).

[18] Elie Bursztein, Einat Clarke, Michelle DeLaune, David M. Elifff, Nick Hsu, Lindsey Olson, John Shehan, Madhukar Thakur, Kurt Thomas, and Travis Bright. "Rethinking the detection of child sexual abuse imagery on the Internet." In: *The World Wide Web Conference*. ACM. 2019, pp. 2601–2607 (Referenced on pages 5, 6).

[19] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. "FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2020 (Referenced on pages 50, 65).

[20] R. Canetti. "Universally composable security: A new paradigm for cryptographic protocols." In: *IEEE Symposium on Foundations of Computer Science (FOCS)*. 2001, pp. 136– (Referenced on pages 16, 59).

[21] Ran Canetti. "Security and composition of multiparty cryptographic protocols." In: *Journal of CRYPTOLOGY*. Vol. 13. 1. 2000, pp. 143–202 (Referenced on pages 16, 59, 72).

[22] Octavian Catrina and Amitabh Saxena. "Secure computation with fixed-point numbers." In: *International Conference on Financial Cryptography and Data Security*. 2010, pp. 35–50 (Referenced on page 57).

[23] Centers for Medicare & Medicaid Services. *The Health Insurance Portability and Accountability Act of 1996 (HIPAA)*. http://www.cms.hhs.gov/hipaa/. 1996 (Referenced on page 4).

[24] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. "Privacy-Preserving Classification on Deep Neural Network." In: *IACR Cryptol. ePrint Arch.* 2017 (2017), p. 35 (Referenced on page 46).

[25] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. "EzPC: Programmable, efficient, and scalable secure two-party computation for machine learning." In: 2017 (Referenced on pages 46, 50, 65, 71).

[26] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. *Security of Homomorphic Encryption*. Tech. rep. Redmond WA, USA: HomomorphicEncryption.org, 2017 (Referenced on page 90).

[27] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. "Astra: High throughput 3pc over rings with application to secure prediction." In: *ACM SIGSAC Conference on Cloud Computing Security Workshop*. 2019 (Referenced on page 50).

[28] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya Razenshteyn, and M. Sadegh Riazi. *SANNS: Scaling up secure approximate k-nearest neighbors search*. https://arxiv.org/pdf/1904.02033.pdf. 2019 (Referenced on page 92).

[29] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragoş Rotaru, Yongsoo Song, and Sameer Wagh. "Maliciously secure matrix multiplication with applications to private deep learning." In: *Advances in Cryptology—ASIACRYPT*. *Author order alphabetical. 2020 (Referenced on page 8).

[30] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. "Homomorphic encryption for arithmetic of approximate numbers." In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, pp. 409–437 (Referenced on pages 18, 76).

[31] *Child Abusers Run Rampant as Tech Companies Look the Other Way*. https://www.nytimes.com/interactive/2019/11/09/us/internet-child-sex-abuse.html. 2019 (Referenced on page 5).

[32] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. "Faster cryptonets: Leveraging sparsity for real-world encrypted inference." In: *arXiv preprint arXiv:1811.09953* (2018) (Referenced on page 46).

[33] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. "The Pyramid scheme: Oblivious RAM for trusted processors." In: *Tech Report*. https://arxiv.org/abs/1712.07882. 2017 (Referenced on page 9).

[34] *CrypTen: Privacy-Preserving Machine Learning built on PyTorch*. 2019. URL: https://github.com/facebookresearch/CrypTen (Referenced on page 97).

[35] Anders Dalskov, Daniel Escudero, and Marcel Keller. *Secure evaluation of quantized neural networks*. https://eprint.iacr.org/2019/131. 2019 (Referenced on page 50).

[36] Ivan Damgård. "On Σ-protocols." In: *Lecture Notes, University of Aarhus, Department for Computer Science*. 2002 (Referenced on page 86).

[37] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. "Homomorphic encryption and secure comparison." In: *International Journal of Applied Cryptography*. 2008 (Referenced on pages 26, 29).

[38] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. "Practical covertly secure MPC for dishonest majority–or: breaking the SPDZ limits." In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 1–18 (Referenced on pages 72, 74, 79, 95).

[39] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. "Multiparty computation from somewhat homomorphic encryption." In: *Annual Cryptology Conference*. Springer. 2012, pp. 643–662 (Referenced on pages 72, 74, 79, 95, 110).

[40] Data61. *MP-SPDZ: Versatile framework for multi-party computation*. https://github.com/data61/MP-SPDZ. 2019 (Referenced on page 73).

[41] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY – A framework for efficient mixed-protocol secure two-party computation." In: *Symposium on Network and Distributed System Security (NDSS)*. 2015 (Referenced on page 38).

[42] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. "Calibrating noise to sensitivity in private data analysis." In: *Theory of Cryptography Conference (TCC)*. Springer. 2006, pp. 265–284 (Referenced on page 3).

[43] Cynthia Dwork, Aaron Roth, et al. "The algorithmic foundations of differential privacy." In: *Foundations and Trends in Theoretical Computer Science*. 2014 (Referenced on page 3).

[44] *Eigen Library*. http://eigen.tuxfamily.org/. Version: 3.3.3 (Referenced on page 39).

[45] Albert Einstein et al. "The foundation of the general theory of relativity." In: *Annalen der Physik* 49.7 (1916), pp. 769–822 (Referenced on page 106).

[46] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. *Improved primitives for MPC over mixed arithmetic-binary circuits*. https://eprint.iacr.org/2020/338.pdf. 2020 (Referenced on page 97).

[47] Junfeng Fan and Frederik Vercauteren. *Somewhat practical fully homomorphic encryption*. https://eprint.iacr.org/2012/144.pdf. 2012 (Referenced on pages 3, 18, 19, 81, 95).

[48] *Fixed-point data type*. http://dec64.com. Last Updted: 2018-01-20 (Referenced on page 98).

[49] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. "Model inversion attacks that exploit confidence information and basic countermeasures." In: *ACM Conference on Computer and Communications Security (CCS)*. ACM. 2015 (Referenced on page 47).

[50] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. "High-throughput secure three-party computation for malicious adversaries and an honest majority." In: *Advances in Cryptology—EUROCRYPT*. 2017 (Referenced on pages 45, 47, 60).

[51] Simson L Garfinkel, John M Abowd, and Christian Martindale. "Understanding database reconstruction attacks on public data." In: *Communications of the ACM*. Vol. 62. 3. 2019, pp. 46–53 (Referenced on page 3).

[52] Craig Gentry. "A fully homomorphic encryption scheme." crypto.stanford.edu/craig. PhD thesis. Stanford University, 2009 (Referenced on pages 3, 18).

[53] Craig Gentry. "Fully homomorphic encryption using ideal lattices." In: *ACM Symposium on Theory of Computing (STOC)*. 2009, pp. 169–178 (Referenced on page 18).

[54] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to play any mental game or a completeness theorem for protocols with honest majority." In: *ACM Symposium on Theory of Computing (STOC)*. 1987 (Referenced on pages 3, 15, 59).

[55] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." In: *Advances in neural information processing systems*. 2014, pp. 2672–2680 (Referenced on page 97).

[56] Shai Halevi and Victor Shoup. "Algorithms in helib." In: *Annual Cryptology Conference*. Springer. 2014, pp. 554–571 (Referenced on page 20).

[57] Shai Halevi and Victor Shoup. "Faster homomorphic linear transformations in HElib." In: *Annual International Cryptology Conference*. Springer. 2018, pp. 93–120 (Referenced on page 88).

[58] Hans Hanley, Yixin Sun, Sameer Wagh, and Prateek Mittal. "DPSelect: A differential privacy based guard relay selection algorithm for Tor." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2019 (Referenced on page 9).

[59] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In: *Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778 (Referenced on pages 23, 73, 75, 93, 97).

[60] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory." In: vol. 9. 8. MIT Press, 1997, pp. 1735–1780 (Referenced on page 97).

[61] Alberto Ibarrondo and Melek Önen. "FHE-compatible batch normalization for privacy preserving deep learning." In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 2018, pp. 389–404 (Referenced on page 46).

[62] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating deep network training by reducing internal covariate shift." In: *International Conference on Machine Learning (ICML)*. 2015, pp. 448–456 (Referenced on page 49).

[63] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. "Secure outsourced matrix computation and application to neural networks." In: *ACM Conference on Computer and Communications Security (CCS)*. 2018, pp. 1209–1222 (Referenced on pages 73, 76, 88, 89).

[64] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "Gazelle: A low latency framework for secure neural network inference." In: *USENIX Security Symposium (USENIX)*. 2018 (Referenced on pages 38, 41, 44, 46, 50, 64, 65, 71).

[65] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: Faster malicious arithmetic secure computation with oblivious transfer." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 830–842 (Referenced on page 72).

[66] Marcel Keller, Valerio Pastro, and Dragos Rotaru. "Overdrive: making SPDZ great again." In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2018, pp. 158–189 (Referenced on pages 8, 72, 73, 90, 91, 94, 95).

[67] Daniel Kifer and Ashwin Machanavajjhala. "Pufferfish: A framework for mathematical privacy definitions." In: *ACM Transactions on Database Systems (TODS)*. Vol. 39. 1. ACM, 2014 (Referenced on page 3).

[68] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. "The CIFAR-10 Dataset." In: 2014 (Referenced on pages 23, 46).

[69] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2012 (Referenced on pages 22, 46, 122).

[70] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2012 (Referenced on page 75).

[71] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. "CrypTFlow: Secure tensorflow inference." In: *IEEE Symposium on Security and Privacy (S&P)*. 2020 (Referenced on pages 50, 72).

[72] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. "Information-theoretically secure protocols and security under composition." In: *SIAM Journal on Computing* 39.5 (2010), pp. 2090–2112 (Referenced on pages 48, 59, 60).

[73] Andrew Lavin and Scott Gray. "Fast algorithms for convolutional neural networks." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4013–4021 (Referenced on page 76).

[74] Steve Lawrence, C. Lee Giles, Ah Chung Tsoi, and Andrew D. Back. "Face recognition: A convolutional neural-network approach." In: *IEEE Transactions on Neural Networks*. Vol. 8. 1. IEEE, 1997, pp. 98–113 (Referenced on page 75).

[75] Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. "Backpropagation applied to handwritten zip code recognition." In: *Neural Computation* 1.4 (1989), pp. 541–551 (Referenced on page 22).

[76] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (Referenced on pages 22, 40, 121).

[77] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. "Oblivious neural network predictions via MiniONN transformations." In: *ACM Conference on Computer and Communications Security (CCS)*. 2017 (Referenced on pages 22, 38, 40, 41, 44, 46, 50, 64, 65, 71, 121).

[78] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings." In: *Journal of the ACM (JACM)* 60.6 (2013), pp. 1–35 (Referenced on page 19).

[79] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. "Fairplay – A secure two-party computation system." In: *USENIX Security Symposium (USENIX)*. 2004 (Referenced on page 4).

[80] *Microsoft PhotoDNA Cloud Service*. 2018. URL: https://www.microsoft.com/en-us/photodna (Referenced on page 6).

[81] *Microsoft SEAL (release 3.3)*. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. 2019 (Referenced on page 91).

[82] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. "DELPHI: A Cryptographic Inference Service for Neural Networks." In: *USENIX Security Symposium (USENIX)*. 2020 (Referenced on page 50).

[83] *MNIST database*. http://yann.lecun.com/exdb/mnist/. Accessed: 2017-09-24 (Referenced on pages 23, 39, 40, 46).

[84] Payman Mohassel and Peter Rindal. "ABY^3: A mixed protocol framework for machine learning." In: *ACM Conference on Computer and Communications Security (CCS)*. 2018 (Referenced on pages 7, 40, 42, 45, 47, 50, 51, 53, 60, 64–67, 70, 72).

[85] Payman Mohassel and Yupeng Zhang. "SecureML: A system for scalable privacy-preserving machine learning." In: *IEEE Symposium on Security and Privacy (S&P)*. 2017 (Referenced on pages 22, 26, 37–41, 43, 44, 47, 50, 64–67, 71–75, 95, 98, 120).

[86] *Netscope: ResNet-50 Architecture*. 2020. URL: http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006 (Referenced on page 23).

[87] Takashi Nishide and Kazuo Ohta. "Multiparty computation for interval, equality, and comparison without bit-decomposition protocol." In: *Public Key Cryptography (PKC)*. 2007 (Referenced on pages 26, 29).

[88] Arpita Patra and Ajith Suresh. "BLAZE: Blazing Fast Privacy-Preserving Machine Learning." In: *Symposium on Network and Distributed System Security (NDSS)*. 2020 (Referenced on page 50).

[89] PySyft. *Implement SecureNN within PySyft #1990*. https://github.com/OpenMined/PySyft/issues/1990. 2019 (Referenced on page 8).

[90] Tal Rabin and Michael Ben-Or. "Verifiable secret sharing and multiparty protocols with honest majority." In: *ACM Symposium on Theory of Computing (STOC)*. 1989, pp. 73–85 (Referenced on page 15).

[91] Rahul Rachuri and Ajith Suresh. "Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning." In: *Symposium on Network and Distributed System Security (NDSS)*. 2019 (Referenced on page 50).

[92] "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/EC (GDPR)." In: *Official Journal of the European Union* L119 (May 2016) (Referenced on page 4).

[93] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. *HEAX: An architecture for computing on encrypted data*. https://eprint.iacr.org/2019/1066. 2019 (Referenced on page 2).

[94] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. "XONN: XNOR-based oblivious deep neural network inference." In: *USENIX Security Symposium (USENIX)*. 2019 (Referenced on pages 46, 50, 65, 66, 71).

[95] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. "Chameleon: A hybrid secure computation framework for machine learning applications." In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2018 (Referenced on pages 22, 40, 42, 44, 46, 50, 64, 65, 71, 72, 120).

[96] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. "DeepSecure: Scalable provably-secure deep learning." In: *Annual Design Automation Conference*. 2018 (Referenced on pages 44, 65, 71).

[97] *SecureNN gives efficient secure ReLU's and Maxpooling neural network layers*. https://news.ycombinator.com/item?id=18674414. 2018 (Referenced on page 8).

[98] Adi Shamir. "How to share a secret." In: *Communications of the ACM* 22.11 (1979), pp. 612–613 (Referenced on page 15).

[99] Claude E Shannon. "Communication theory of secrecy systems." In: *Bell system technical journal*. Vol. 28. 4. Wiley Online Library, 1949, pp. 656–715 (Referenced on page 12).

[100] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. "Panoply: Low-TCB linux applications with SGX enclaves." In: *Symposium on Network and Distributed System Security (NDSS)*. 2017 (Referenced on page 3).

[101] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. "Membership inference attacks against machine learning models." In: *IEEE Symposium on Security and Privacy (S&P)*. 2017 (Referenced on page 47).

[102] Karen Simonyan and Andrew Zisserman. *Very deep convolutional networks for large-scale image recognition*. https://arxiv.org/abs/1409.1556. 2014 (Referenced on pages 23, 46, 123).

[103] David Marco Sommer, Liwei Song, Sameer Wagh, and Prateek Mittal. *Towards probabilistic verification of machine unlearning*. https://arxiv.org/pdf/2003.04247.pdf. 2020 (Referenced on page 9).

[104] *Stanford CS231n: Convolutional Neural Networks for Visual Recognition*. http://cs231n.github.io/convolutional-networks/ (Referenced on page 32).

[105] Richard S Sutton, Andrew G Barto, et al. *Introduction to Reinforcement Learning*. Vol. 135. MIT press Cambridge, 1998 (Referenced on page 97).

[106] *This Club had its Own Chairman and Treasurer. Its Business was Child Abuse*. https://www.theguardian.com/uk/2001/feb/11/tracymcveigh.martinbright. 2001 (Referenced on page 6).

[107] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. "Stealing machine learning models via prediction APIs." In: *USENIX Security Symposium (USENIX)*. 2016 (Referenced on page 47).

[108] Sameer Wagh, Paul Cuff, and Prateek Mittal. "Differentially private oblivious RAM." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2018 (Referenced on page 9).

[109] Sameer Wagh, Divya Gupta, and Nishanth Chandran. "SecureNN: 3-Party secure computation for neural network training." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2019 (Referenced on pages 7, 8, 45, 48, 50, 51, 64–67, 70, 72, 74).

[110] Sameer Wagh, Xi He, Ashwin Machanavajjhala, and Prateek Mittal. "DP-Cryptography:Marrying differential privacy and cryptography in emerging applications." In: *Communications of the ACM*. 2020 (Referenced on page 9).

[111] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. "FALCON: Honest-majority maliciously secure framework for private deep learning." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2021 (Referenced on pages 2, 8).

[112] Wagh, Sameer and Gupta, Divya and Chandran, Nishanth. *SecureNN: 3-Party secure computation for neural network training*. https://eprint.iacr.org/2018/442.pdf. 2019 (Referenced on page 27).

[113] Gerry Wan, Aaron Johnson, Ryan Wails, Sameer Wagh, and Prateek Mittal. "Guard placement attacks on path selection algorithms for Tor." In: *Privacy Enhancing Technologies Symposium (PETS)*. 2019 (Referenced on page 9).

[114] Jiayu Wu, Qixiang Zhang, and Guoxi Xu. *Tiny ImageNet Challenge*. http://cs231n.stanford.edu/reports/2017/pdfs/930.pdf (Referenced on pages 23, 46).

[115] Andrew C Yao. "Protocols for secure computations." In: *IEEE Symposium on Foundations of Computer Science (FOCS)*. 1982 (Referenced on pages 3, 4).

[116] Andrew Chi-Chih Yao. "How to generate and exchange secrets (extended abstract)." In: *IEEE Symposium on Foundations of Computer Science (FOCS)*. 1986 (Referenced on pages 3, 4).

[117] Yanqi Zhou, Sameer Wagh, Prateek Mittal, and David Wentzlaff. "Camouflage: Memory traffic shaping to mitigate timing attacks." In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017 (Referenced on page 9).

[118] Xiangxin Zhu, Carl Vondrick, Charless Fowlkes, and Deva Ramanan. "Do we need more training data?" In: *International Journal of Computer Vision*. 2016 (Referenced on page 4).