TU WIEN Informatics

# Enabling Privacy-Preserving Machine Learning with Secure Multi-Party Computation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering/Internet Computing**

eingereicht von

**BSc. Adam Skuta**
Matrikelnummer 12208176

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.univ.Prof. Dr. Andreas Rauber
Mitwirkung: DI Dr. Thomas Lorünser

Wien, 14. November 2025

Adam Skuta

Andreas Rauber

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

# Enabling Privacy-Preserving Machine Learning with Secure Multi-Party Computation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering and Internet Computing**

by

**BSc. Adam Skuta**
Registration Number 12208176

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.univ.Prof. Dr. Andreas Rauber
Assistance: DI Dr. Thomas Lorünser

Vienna, November 14, 2025

Adam Skuta Andreas Rauber

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

# Erklärung zur Verfassung der Arbeit

BSc. Adam Skuta

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 14. November 2025

Adam Skuta

# Acknowledgements

I would like to thank first and foremost my supervisor Ao.univ.Prof. Dr. Andreas Rauber for his great help and insights. I would also like to express my gratitude towards DI Dr. Thomas Lorünser for working with me on this thesis. Last but not least I would also like to thank my girlfriend Julia for her unconditional support as well as my family and friends.

# Kurzfassung

Die Einführung von Machine Learning in Bereichen wie Gesundheitswesen, Biometrie und industrieller Automatisierung wirft Bedenken hinsichtlich des Datenschutzes auf. Secure Multi-Party Computation (SMPC) bietet einen interessanten Ansatz, der datenschutzwahrende Berechnungen auf sensiblen Daten ermöglicht. Diese Arbeit untersucht die Praxistauglichkeit von SMPC-basierter Machine-Learning-Inferenz, indem SMPC-Frameworks evaluiert, neuronale Netzwerkarchitekturen benchmarkt und zwei Fallstudien implementiert werden. Zunächst werden drei SMPC-Frameworks analysiert und miteinander verglichen. Auf Grundlage dieses Vergleichs wird SecretFlow-SPU aufgrund seiner benutzerfreundlichen Unterstützung für weitere Experimente ausgewählt. Als Nächstes wird ein systematischer Benchmark dreier unterschiedlicher neuronaler Netzwerkarchitekturen durchgeführt, um den Inferenz-Overhead sowie den Zusammenhang zwischen der Anzahl der Parameter und der Schichten, der diesen Overhead beeinflusst, zu untersuchen. Abschließend werden zwei SMPC-Anwendungsfälle vorgestellt: zum einen eine datenschutzwahrende Gesichtsverifikation und zum anderen eine sichere Energieverbrauchsprognose für Industrieroboter. Beide Fallstudien zeigen, dass SMPC einen erheblichen Inferenz-Overhead einführt, insbesondere bei der Gesichtsverifikation, die größere Modelle für gute Ergebnisse erfordert. Gleichzeitig zeigt sich jedoch, dass Modelle, die für ressourcenbeschränkte Geräte optimiert sind, auch im SMPC-Kontext deutlich profitieren. Darüber hinaus wurde der Einfluss von Netzwerkbedingungen wie Latenz und Paketverlust untersucht.

# Abstract

The adoption of machine learning in domains such as healthcare, biometrics and industrial automation raises concerns around data privacy. Secure Multi-Party Computation offers an interesting approach that enables privacy-preserving computation on sensitive data. This thesis investigates the practicality of SMPC-based machine learning inference, by evaluating SMPC frameworks, benchmarking neural network architectures and implementing two case studies. Firstly three SMPC frameworks were reviewed and compared. Based on this comparison Secretflow-SPU is selected for further experimentation due to its user-friendly support. Second, a systematic benchmark of three different neural network architectures is conducted, to show the inference overhead and the relationship between the number of parameters and layers that influences this overhead. Finally, two SMPC use cases are presented. One is a privacy-preserving face verification and the second is a secure energy prediction for industrial robots. Both case studies show that SMPC introduces a significant inference overhead, especially for face verification that requires a more larger models to perform well. But it also shows that using models that are optimized for resource-constrained devices benefits significantly in SMPC as well. In addition the effect of network conditions such as network delay and packet loss was examined as well.

# Contents

**Kurzfassung** **ix**

**Abstract** **xi**

**Contents** **xiii**

**1 Introduction** **1**
1.1 Research Questions . . . 2
1.2 Approaches to privacy . . . 2
1.3 Methodology . . . 4

**2 Preliminaries** **7**
2.1 Secure Multi-Party Computation . . . 7
2.2 Neural networks . . . 9

**3 Framework analysis** **13**
3.1 Secretflow-SPU . . . 13
3.2 Crypten . . . 17
3.3 MP-SPDZ . . . 19
3.4 Conclusion . . . 20

**4 Benchmarks of generic models** **21**
4.1 Network Configurations . . . 21
4.2 Results . . . 22
4.3 Conclusion . . . 26

**5 Use-case Studies** **27**
5.1 Generic use-case model . . . 27
5.2 Privacy-preserving Face Verification . . . 28
5.3 Privacy-Preserving Energy Prediction for Industrial Robots . . . 34
5.4 Conclusion . . . 42

**6 Conclusion** **43**

**Overview of Generative AI Tools Used** **45**

**List of Figures** **47**

**List of Tables** **49**

**Bibliography** **51**

CHAPTER 1

# Introduction

The rapid advancement of machine learning technologies has revolutionized numerous aspects of our daily lives, from healthcare diagnostics to financial services, forecasting services, as well as personal assistants and chatbots like ChatGPT, etc. These systems range from very simple to sophisticated, the more sophisticated ones requiring large amounts of data to properly improve. A big portion of the valuable data comes from a highly sensitive source. Medical records that could advance diagnostic capabilities, biometric data that could enhance security systems, and proprietary companies' information all fall into this category of sensitive yet valuable information.

Traditional machine learning approaches present organizations with a seemingly binary choice: either work with raw data and accept the associated privacy risks, or implement privacy measures that often significantly degrade the utility of the data and the resulting models. The techniques range from data preprocessing in the form of privacy-preserving data publishing, synthetic data, to different secure computation techniques. This dilemma has become particularly acute in fields such as healthcare, where the potential benefits of advanced machine learning models must be carefully balanced against strict privacy requirements. Similarly, in industrial applications, companies often hesitate to leverage machine learning fully due to concerns about protecting proprietary data and trade secrets.

Secure Multi-Party Computation (SMPC) is a promising solution to this problem. Numerous frameworks and optimization strategies have been developed and tested in recent years, showing promising results that can be applicable in various fields. As most of the recent solutions focus mainly on optimization of inference of Large Language Models, these optimization strategies have not been applied to other fields. Particularly, the domains of Robotics and Biometrics present compelling yet unexplored use cases for SMPC implementation. Biometrics represents a classical Machine Learning problem of computer vision, and this field is especially relevant due to its inherent need to process highly sensitive personal identifiers, while robotics applications represent a problem of time series analysis and often involve proprietary motion planning and control algorithms that could potentially require privacy-preserving computation. Concrete use-case studies of SMPC in these fields would serve as practical reference architectures for computer vision and time series analysis, lowering the barrier to entry for domain experts from these fields considering privacy-preserving approaches. This thesis aims to show the maturity of SMPC on these two representative use cases. By demonstrating real-world performance metrics and implementation strategies specific to robotics and biometrics applications, this research aims to bridge the gap between theoretical SMPC capabilities and domain-specific requirements, potentially accelerating the adoption of privacy-preserving techniques in these communities.

## 1.1 Research Questions

The main research question this thesis aims to answer is "To what extent can current SMPC frameworks effectively enable privacy-preserving machine learning inference across different application domains while maintaining acceptable performance and accuracy?". To help answer this question, three distinct specific research questions have been laid out.

1. What are the key architectural differences, security guarantees, and practical implementation considerations across the current SMPC frameworks for machine learning?
2. How do different neural network configurations affect the inference time under SMPC?
3. How effectively can an SMPC framework be implemented in privacy-sensitive applications compared to their non-encrypted implementation through two representative case studies within biometrics and robotics?

By answering these three questions, an informed answer can be made about the main research question.

## 1.2 Approaches to privacy

In addressing privacy concerns in machine learning, two primary approaches have emerged: privacy-preserving data publishing and privacy-preserving computation. Each approach offers distinct strategies for protecting sensitive information.

### 1.2.1 Privacy-Preserving Data Publishing

Privacy-preserving data publishing (PPDP) focuses on transforming raw data into privacy-protected versions before releasing or sharing them. The fundamental principle behind PPDP is to modify the original dataset in ways that obscure individual identities or sensitive attributes while preserving statistical properties necessary for meaningful analysis. Key techniques in this domain include:

- **k-anonymity**: This technique ensures that each record in a dataset is indistinguishable from at least $k-1$ other records with respect to certain quasi-identifier attributes[1]. Through generalization, k-anonymity reduces the granularity of data representation, thereby preventing the unique identification of individuals.
- **Differential Privacy**: Unlike k-anonymity, differential privacy[2] provides formal mathematical guarantees about the privacy risk of participating in a dataset. It functions by adding calibrated noise to query results or data transformations, ensuring that the presence or absence of any single individual has a limited impact on the output. The privacy guarantee is controlled by a parameter $\epsilon$, where smaller values indicate stronger privacy protection.
- **Synthetic Data Generation**: This approach involves creating artificial datasets that maintain the statistical properties and relationships of the original data without containing actual records from real individuals[3]. Advanced methods leverage generative models, such as Generative Adversarial Networks or Variational Autoencoders, to produce synthetic records that still maintain complex patterns while eliminating the direct privacy risks associated with real data.

While PPDP techniques offer advantages in terms of one-time processing and compatibility with existing analysis tools, they often involve irrevocable transformations of data that may reduce utility for certain applications when high precision is required.

### 1.2.2 Privacy-Preserving Computation

Privacy-preserving computation (PPC) takes a different approach by protecting data during the computation process rather than modifying the data itself. These techniques enable computations on sensitive data while ensuring that neither the input data nor intermediate results are exposed to unauthorized parties. Key methodologies in this domain include:

- **Secure Multi-Party Computation (SMPC)**: SMPC protocols allow multiple parties to jointly compute functions over their private inputs without revealing those inputs to other participants [4]. Through cryptographic techniques such as secret sharing, SMPC enables collaborative computation while maintaining the confidentiality of each party's data. SMPC typically introduces significant computational overhead compared to non-secure processing.
- **Homomorphic Encryption (HE)**: HE schemes enable computations to be performed directly on encrypted data without prior decryption [5]. The results of these computations, when decrypted, match the results that would have been obtained by

performing the same operations on the unencrypted data. While fully homomorphic encryption supports arbitrary computations, practical implementations often use partially homomorphic schemes that support specific operations (such as addition or multiplication) to improve efficiency.

Privacy-preserving computation techniques offer the advantage of maintaining data in its original form, potentially preserving more utility than data publishing approaches. However, they typically produce more computational and communicational overhead.

### 1.2.3 Privacy-preserving machine learning

There has been a substantial amount of work done on privacy-preserving machine learning in recent years.

In [6], a privacy-preserving machine learning technique based on SMPC was proposed that trains on genomic data from different stakeholders who consider the data sensitive without revealing anything to the participating parties except the final trained model.

In [7], an attempt at speeding up the transformer inference is proposed. In order to mitigate the exponentiation bottleneck in transformers, a different variant is employed, which replaces the sigmoid function with ReLU activations, which represents a more SMPC-friendly non-linearity. Along with an approximation technique for attention matrices in transformers, the attention layer can be processed in a faster time. The results show that for the translation task of an output sequence of length 64, the computation takes 19 minutes in the LAN environment.

In [8], a framework MPCFormer is proposed for fast Transformer model inference. The main contributions lie in two approximations of GeLU and softmax activation functions and a later Knowledge Distillation process that transforms knowledge from the full model to the approximated one, enabling aggressive approximations in the first step, while preserving the quality of the original models after approximations.

In [9], a framework PUMA is proposed to enable fast and secure Transformer model inference. The core of the design lies in an accurate approximation of GeLU and softmax functions, which normally significantly bottleneck the inference speed of models. Another contribution is a new Embedding and LayerNorm layer SMPC protocols that keep the desired functionality intact. PUMA claims to have a two times faster inference speed than the state-of-the-art and can evaluate LLaMA-7B in around 5 minutes to generate 1 token. PUMA internally uses the Secretflow-SPU framework.

## 1.3 Methodology

The methodology of this thesis is threefold. Firstly, a review of three distinct SMPC frameworks will be made. Their architectures and supported protocols will be examined, as well as their implementation details and documentation. An examination of their maturity as well as usage in other projects will be highlighted as well.

Secondly, an evaluation of three different neural network architectures will be made, with different configurations regarding their width and height. The inference time under SMPC will be evaluated.

Thirdly, an implementation and evaluation of one of the selected frameworks will be performed on two use-case studies, which highlight two different and widely used areas of Machine Learning, namely computer vision and time-series analysis. The ease of integration, benchmarks, and other problems during the development will be highlighted as well.

This thesis is structured as follows. In Chapter 2, a preliminary knowledge needed for understanding the basic concepts discussed in this thesis will be presented. In Chapter 3, a comparison and analysis of three popular SMPC frameworks is made, helping answer the first research question. In Chapter 4, generic network benchmarks will be conducted, helping answer the second research question. In Chapter 5, two use-case studies are presented that help answer the third research question. The thesis concludes in Chapter 6.

# CHAPTER 2

# Preliminaries

In this chapter, the two most important concepts for this thesis will be presented. First, an introduction to Secure Multi-Party Computation will be stated, followed by an introduction to deep neural networks.

## 2.1 Secure Multi-Party Computation

Secure Multi-Party Computation (SMPC) is a collection of cryptographic techniques that enable multiple parties to jointly compute a function over their private data without revealing that data to one another [10]. The goal is to achieve the same result of the function as if all data had been pooled together and processed by a trusted central party, but without ever needing to rely on such a trusted party. Formally, SMPC considers a scenario in which $n$ parties $P_1, P_2, ..., P_n$ each hold a private input $x_i$. The goal is to compute the function 2.1, such that the result $y$ is revealed to the predetermined party or parties and nothing else is exposed.

$$y \leftarrow f(x_1, x_2, ..., x_n) \tag{2.1}$$

### 2.1.1 Security models

The role of SMPC protocols is to ensure the privacy and integrity of the underlying computation. Therefore, the protocols are often described based on the security model they support.

Most often, this security is described based on the adversary types of the corrupted parties. Generally, these are divided into three categories:

- **Semi-honest** adversaries (sometimes called Honest-but-curious) are corrupted nodes that correctly follow the computation. Their only goal is to obtain the information about the computation, like inputs or intermediate results from other parties. These parties can collude with each other,

- **Malicious** adversaries do not follow the protocol and often can act against the protocol to disrupt the privacy and integrity of the computation,
- **Covert** adversaries do not have to follow protocol and can switch between being malicious or semi-honest, based on the potential gains of each respective mode.

Another possible description of the security model is the number of adversaries relative to the overall party size. These are divided into two categories:

- **Honest-majority** setting assumes that more than half of the party members are honest and not corrupted,
- **Dishonest-majority** setting assumes that half or more of the party members are corrupted.

Based on the adversary type, different security requirements are needed. For **semi-honest** adversary type, privacy and correctness are required from the SMPC protocol. The privacy aspect requires that each party learns nothing about the computation of other parties apart from the final output. The correctness aspect requires that the computation results in the correct output if the honest parties are computing the protocol correctly, regardless of the eavesdropping of the semi-honest adversaries. For **malicious** adversary type requires privacy, correctness, and additionally robustness and verifiability. The robustness aspect requires that the computation protocol behaves uninterrupted despite possible efforts from malicious adversaries to change the outcome or disrupt the computation. The verifiability aspect ensures that the malicious activity is detected and that measures are taken against it. For **covert** adversary, a correct balance between efficiency and robustness must be met. A requirement needed for this type is accountability, which ensures that the covert parties can be detected with high probability. Another requirement that can be used is fairness, which ensures that no party can obtain the output before others.

### 2.1.2 Secret-sharing

Secret-sharing is a cryptographic technique that is a very important building block of SMPC applications. It splits a secret value into $n$ shares that are then distributed to $n$ parties, in such a way that only a predetermined number of parties are required to successfully reconstruct the original secret. These shares can then be computed by allowing addition and multiplication, with some operations requiring further communication between the parties. Two of the most important operations that can be used to implement further complex operations are addition and multiplication of two numbers. For the addition of two shares $x_i$ and $y_i$, where $i$ represents the share of the node $i$, each node needs to locally compute $x_i + y_i$. This operation is the simplest one as it does not require any inter-node communication. The multiplication is a more complex operation and its exact algorithm is dependent on the underlying protocol that is used (e.g, Beaver triples [11] are used in Crypten [12]). In general, it requires inter-node communication, making it a more expensive computation than addition, as it is influenced by network complications, like delay, loss, and bandwidth. For machine learning purposes, two different types of shares will be discussed, namely arithmetic and binary secret shares. Arithmetic secret shares [12] represent values as elements of a finite ring, e.g.. $\mathbb{Z}/Q\mathbb{Z}$ for some large integer $Q$. A value $x$ is shared among $n$ parties by splitting it into additive shares $[x]_p$ such that $x = \sum_{p \in P}[x]_p \bmod Q$. This type of sharing is suitable for the computation of arithmetic operations such as multiplication and addition, which translates to usefulness in neural network operations like matrix multiplication, convolutions, and other linear operations. Binary secret sharing is, on the other hand, based on the field $\mathbb{Z}/2\mathbb{Z}$ (a ring with two binary values) and operates on the individual bits of values. Each bit is split into binary shares so that their XOR (Exclusive OR logical operation) reconstructs the original bit. Binary secret shares are inefficient in terms of arithmetic operations, but are suitable for operations like comparisons, which in the machine learning context are used for activation functions like ReLU.

## 2.2 Neural networks

A neural network is a mathematical model that can learn and approximate different complex systems. The main theoretical foundation to learn these complex systems is the universal approximation theorem, which states that, given non-linear activation, a network consisting of a single hidden layer can, with enough neurons, approximate any continuous function in $\mathbb{R}^n$ [13]. In reality, neural networks are split into shallow and deep ones. The shallow neural networks consist of one or two hidden layers, while deep neural networks contain several hidden layers and are often more sophisticated. There are two main components of neural networks, which are layers and activation functions.

- **Layers** are the main building blocks of neural networks. They contain learnable parameters that, during training, are shifted to approximate the output as close as possible. These layers can range from very simple linear layers that compute a dot product between weights and inputs, more complex convolutions that interact with neighboring inputs and produce an output, to long short-term memory (LSTM) layers that take into consideration the order and time of the inputs and produce the output. These layers can be stacked in an arbitrary order to approximate more and more complex functions,

- **Activation functions** are another main building block of neural networks. They introduce non-linearity to the network, making it more complex and therefore able to approximate more complex relations. Some common activation functions are, for example, ReLU, which has the following definition: $ReLU(x) = max(0, x)$ or softmax function: $softmax(x) = \frac{1}{1+e^{-x}}$.

### 2.2.1 Relevant architectures

Several foundational neural network architectures have been developed over the years in various domains. In this section, two architectures are going to be presented that are relevant for this thesis.

#### Convolutional Neural Networks

Convolutional neural networks (CNNs) are most often used in computer vision applications. They use convolutional layers to recognize patterns such as edges and shapes. CNNs typically consist of convolutional, pooling, and fully connected layers.

**Convolutional** layers use a set of learnable kernels on the input data. The output is then computed using the equation described in 2.2.

$$S(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X(i+m, j+n) \cdot K(m,n) + b. \tag{2.2}$$

Here:

- $S(i,j)$ is the output feature map at position $(i,j)$,
- $X$ is the input matrix, e.g. image or previous layer's output,
- $K$ is the kernel,
- $M \times N$ is the size of the kernel,
- $\cdot$ is the convolution operation,
- $b$ is the bias, which is a learnable parameter.

**Pooling** layers downsample the outputs of the convolutions and prevent overfitting of the network. The most common pooling layer is the max-pool layer and is described as follows:

$$P(i,j) = \max_{0 \leq m < M, 0 \leq n < N} X(i \cdot s + m, j \cdot s + n).$$

Where:

- $P(i,j)$ is the output at position $(i,j)$,
- $X$ is the input feature map,
- $M \times N$ is the size of the pooling window,

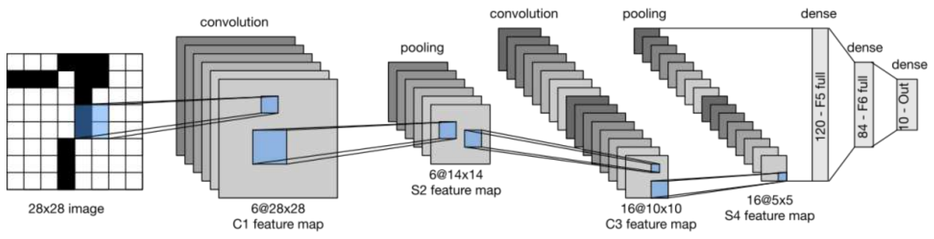


Figure 2.1: LeNet-5 architecture. Taken from `https://github.com/d2l-ai/d2l-en`

- $s$ is the stride, i.e., how far the window moves each step.

Besides convolutional and pooling layers, the CNNs usually utilize linear layers at the end to produce the final classification or regression. Some notable architectures for CNNs are LeNet [14], AlexNet [15] or ResNet [16]. An example CNN architecture can be seen on **Fig**. 2.1. Here, the network is supposed to identify the digit shown in the picture, and it does so with convolutions (C1, C3) as well as pooling layers (S2, S4). The representation of the picture is then processed using linear layers to finally output the probabilities for each digit, where the biggest probability represents the final classification.

### Long Short-Term Memory networks

Long Short-Term Memory networks (LSTMs) [17] fall under the broader category of Recurrent Neural Networks (RNNs). In general, RNNs are designed for processing sequential data like time series or natural language. Unlike classical networks like Linear or CNNs, they have hidden states that contain information from previous time-steps. Traditional RNNs suffer from a problem called vanishing or exploding gradients, which hinder them from realistically learning longer sequences or relationships in sequential data. One of the solutions for this problem is LSTM. These networks introduce gating mechanisms to better manage information flow over time. LSTMs maintain a memory cell that holds a state, which is capable of holding information across longer sequences. The three gates that regulate the state are: the input gate, forget gate, and output gate. Their role in the LSTM can be described as follows:

- Input gate: Determines how much new information should be added to the cell in the current time stamp, from the current input as well as the hidden state from the previous step.
- Forget gate: Determines what information should be removed from the previous cell state.

- Output gate: Determines what information should be put as the output of the current cell based on the current input, as well as the output of the cell in the previous timestamp.

This concludes the summary of all the important building blocks needed for this thesis. In the next chapter, the first research question will be analyzed and answered.

CHAPTER 3

# Framework analysis

There is a wide range of SMPC frameworks available, developed for different purposes and with varying levels of maturity. Examples include TF-Encrypted [1], MPyC [2] and Falcon [18].

To choose frameworks for comparative study, the following inclusion criteria were applied:

- Open-source: The framework must be freely available with open code,
- Active development: The framework should be actively maintained at least within the last two years,
- Connection to Machine Learning: The framework should support machine learning models from the most popular frameworks (Tensorflow, PyTorch, Jax)

Based on these criteria, three different prominent SMPC frameworks will be compared, namely Crypten, MP-SPDZ, and Secretflow-SPU. Each framework will be examined across multiple key dimensions - security models, supported ML operations/architectures, ease of integration, documentation and community, development activity, and real-world adoption. At the end, the findings will be summarized in a comparison matrix. The comparison will draw from each framework's respective literature and documentation [12, 19, 20].

## 3.1 Secretflow-SPU

Secretflow-SPU [20] is a general-purpose Privacy-preserving Machine Learning (ppML) framework designed to make the development of ML models in SMPC easier for ML

[1]https://github.com/tf-encrypted/tf-encrypted
[2]https://github.com/lschoe/mpyc

researchers. It consists of a frontend compiler that provides developers with a Python API and a backend runtime. The frontend accepts an ML program and compiles it into an intermediate representation called Privacy-preserving high-level operations (pphlo). The backend then consumes the pphlo and executes it on a virtual device that consists of multiple interconnected nodes that implement a configurable MPC protocol. The overview of the architecture can be seen in Figure 3.1.

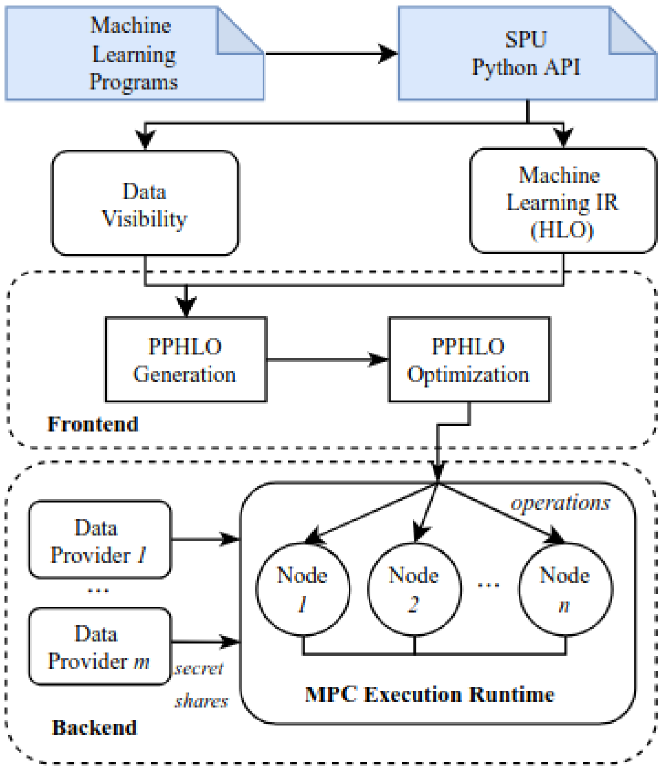


Figure 3.1: SPU architecture. Taken from [20].

The compilation of the ML program into intermediate representation is inspired by other non-MPC ML compilers like Google's XLA [3]. The purpose of them is to simplify the development of ML models from different frameworks on different hardware. XLA uses High-level operation (HLO) as an intermediate representation to represent the computational graph of the ML model. This code is further optimized for the underlying hardware and finally compiled into machine code to be run on CPU, GPU, or TPU. SPU uses this approach as well in its design. It leverages the XLA to produce the HLO representation, applies further compilation to mark the privacy and data type of the computational graph, and produces the pphlo representation. Listing 3.1 shows a Python code that describes an addition function. On lines 4-6, the *add* function is defined, on lines 8-9, two tensors are defined with shape 3 and data type of *jnp.float32*, and finally on line 10, the result of the add operation is stored in tensor *z*.

[3]https://github.com/openxla/xla

Listing 3.1: Addition function in Jax

```
import jax
from jax import numpy as jnp

@jax.jit
def add(x, y):
    return jnp.add(x, y)

x = jnp.array([1, 2, 3], dtype=jnp.float32)
y = jnp.array([4, 5, 6], dtype=jnp.float32)
z = add(x, y)
```

This *add* function with the concrete inputs can be compiled into the HLO representation as can be seen on Listing 3.2.

Listing 3.2: HLO representation of the addition function

```
func.func public @main(%arg0: tensor<3xf32>, %arg1: tensor
    <3xf32>) -> (tensor<3xf32>) {
  %0 = stablehlo.add %arg0, %arg1 : tensor<3xf32>
  return %0 : tensor<3xf32>
}
```

This HLO representation is then further processed by SPU and compiled into PPHLO representation, as can be seen on Listing 3.3.

Listing 3.3: PPHLO representation of the addition function

```
func.func @main(%arg0: tensor<3x!pphlo.secret<f32>>, %arg1:
      tensor<3x!pphlo.secret<f32>>) -> tensor<3x!pphlo.secret
    <f32>> {
  %0 = pphlo.add %arg0, %arg1 : tensor<3x!pphlo.secret<f32>>
  return %0 : tensor<3x!pphlo.secret<f32>>
}
```

Further SMPC-related optimizations are made, and the output is consumed by the backend runtime implementing a specific SMPC protocol. This creates a clear separation between ML development and SMPC development. The ML developer is only required to use a framework that supports XLA compilation to develop its models in SMPC, while, on the other hand, the SMPC developer is only required to develop SMPC primitives for the operations defined in pphlo.

SPU runs user-written programs, secret-sharing a predetermined number of input data and all intermediate results. In an ML context, SPU protects both user inputs as well as model weights, where the user can also specify whether one of them should be used in plaintext form to speed up the computation process. As SPU is a protocol-agnostic framework, the specific threat model and number of parties are dictated by the underlying MPC protocol used during the computation. As of right now, SPU implements the following protocols:

- ABY3 [21], a honest-majority semi-honest three party protocol
- SPDZ2k [22], a semi-honest n-party protocol,
- Cheetah [23], a semi-honest two party protocol.

In theory, the SPU frontend should accept any machine learning program that can be compiled using XLA into HLO representation. Because of its younger nature, currently only programs written in a popular framework JAX [24] by Google are fully supported, with other frameworks like PyTorch[4] and Tensorflow[5] having experimental support. Creating a machine learning program for SPU is straightforward and requires minimal changes from a plaintext JAX version. An example of an SPU program can be seen in Listing 3.4. Python decorator *@ppd.device("SPU")* is used to specify which function should be computed using SMPC. In this case, it is a compare function that returns the element-wise maximum number between two vectors. The decorators *@ppd.device("P1")* and *@ppd.device("P2")* are used to specify that both parties' inputs should be secret-shared and therefore kept private during computation. This example can be extended to ML usage as well, where the SMPC function could be an inference of the model, and the two private inputs would be the input features and model weights.

Listing 3.4: SPU function that calculates maximum between two vectors

```python
import json
from jax import numpy as jnp
from spu.utils import distributed as ppd

with open('config.json', 'r') as f:
    conf = json.load(f)
ppd.init(conf['nodes'], conf['devices'])

@ppd.device("SPU")
def calc_max(x, y):
    return jnp.maximum(x, y)

x = jnp.array([1, 2, 3])
y = jnp.array([4, 5, 6])
x_s = ppd.device("P1")(lambda x: x)(x)
y_s = ppd.device("P2")(lambda x: x)(y)
```

[4]https://github.com/pytorch/pytorch
[5]https://github.com/tensorflow/tensorflow

```
    z_s = calc_max(x_s, y_s)
    z = ppd.get(z_s)
```

SPU is part of a larger Secretflow environment [6], which aims to be used as a production suite for privacy-preserving machine learning and data science. That being said, SPU so far has only been used in scientific papers, mainly aiming to lower and optimize the inference times of transformer models like PUMA [9] or Ditto [25]. The source repository receives periodical commits, and a new version is released around two times a year.

## 3.2 Crypten

Crypten is an SMPC PyTorch-based library, supporting a range of tensor operations, geared towards machine learning. It uses a tensor abstraction via Cryptensor, behaving similarly to a PyTorch tensor. The Cryptensor supports element-wise arithmetic, comparisons, matrix multiplications, convolution, and others in a secret-sharing form. As with PyTorch, Crypten implements automatic differentiation, enabling model training in encrypted form. Automatic differentiation is a set of algorithms that allows the calculation of a programmed numerical function. It is an important feature for any deep learning framework as it effectively enables training of the network via the backpropagation algorithm. There are two ways to implement a neural network in Crypten. One is to implement the network in PyTorch and use the Crypten API to translate it into a secret-shared model. Another one is to directly use Crypten's model neural network package, which contains widely used layers like fully-connected linear layers or convolutional layers, and a wide variety of activation functions like ReLU or Softmax. An example of a CrypTen program can be seen in Listing 3.5 taken from GitHub. On lines 1–3, the main module and other important submodules are imported from Crypten. On line 8, a decorator is used specifying the number of parties used for this SMPC computation. On line 11, a pretrained model is loaded from one of the parties. This party then provides the pretrained parameters of the model for the computation. On line 15, Crypten requires an initialization of the model with some dummy input. This input is only required to have the same shape as the input of the actual data that would be used with this model. On line 16, the model is encrypted, with one party again being the source. On lines 19–21, a similar process is made by the second party, but now with the data that would be used for the inference. Here, the data do not have to be explicitly encrypted compared to the model. On lines 24–25, the model is evaluated in an SMPC setting with the secret-shared data. On line 28, the results are revealed to both parties, and accuracy is computed.

Listing 3.5: Crypten function to evaluate a private model on some private data.

```
1    import crypten
2    import crypten.mpc as mpc
3    import crypten.communicator as comm
```

[6] https://www.secretflow.org.cn/en/

```python

labels = torch.load('/tmp/bob_test_labels.pth').long()
count = 100

@mpc.run_multiprocess(world_size=2)
def encrypt_model_and_data():
    # Load pre-trained model to Alice
    model = crypten.load_from_party('models/
        tutorial4_alice_model.pth', src=ALICE)

    # Encrypt model from Alice
    dummy_input = torch.empty((1, 784))
    private_model = crypten.nn.from_pytorch(model,
        dummy_input)
    private_model.encrypt(src=ALICE)

    # Load data to Bob
    data_enc = crypten.load_from_party('/tmp/bob_test.pth',
        src=BOB)
    data_enc2 = data_enc[:count]
    data_flatten = data_enc2.flatten(start_dim=1)

    # Classify the encrypted data
    private_model.eval()
    output_enc = private_model(data_flatten)

    # Compute the accuracy
    output = output_enc.get_plain_text()
    accuracy = compute_accuracy(output, labels[:count])
    crypten.print("Accuracy:␣{0:.4f}".format(accuracy.item
        ()))

encrypt_model_and_data()
```

Crypten assumes a semi-honest adversary model. All parties are assumed to follow the computation correctly, but may try to learn additional information. An arbitrary number of parties are supported. Unlike SPU, Crypten does not rely on an underlying SMPC protocol but rather implements its own. At the core of CrypTen's secure computation are two secret-sharing schemes: arithmetic secret sharing and binary secret sharing. Arithmetic secret sharing is well-suited for linear operations like matrix multiplications and convolutions, while binary secret sharing is necessary for functions such as comparisons and ReLU activations. Efficient conversions between the two types of sharing (A2B and B2A) are implemented.

Crypten has not been used in any production setting so far, staying only in the academic space for benchmarking and potential improvements of inference time of transformer architectures. One such example is the Centaur [26] paper, which aims to improve the inference time by introducing inference with plaintext weights that are permuted to disallow brute-force model stealing, as well as doing activation functions in plaintext. The development of the network has recently stopped, and the repository was deprecated.

## 3.3 MP-SPDZ

MP-SPDZ [19] is a fork of SPDZ-2 that extends the original framework to include 34 different variants of secure multi-party computation (MPC) protocols. These protocols cover a wide spectrum of security models, including semi-honest, dishonest, covert, honest majority, and dishonest majority settings. The broad protocol support allows MP-SPDZ to serve as a flexible platform for experimenting with and comparing different MPC approaches within a single framework. An important feature of MP-SPDZ is its high-level programming interface, which is based on Python. This interface works uniformly across all supported protocol variants, making it easier to develop MPC applications without needing to adapt their code for each protocol. The system is designed primarily for benchmarking different MPC protocols in a consistent, universal way. While machine learning is supported in MP-SPDZ, the framework is intended for a wider range of MPC use cases. Programs in MP-SPDZ run on a custom virtual machine (VM). The compiler translates the high-level Python code provided by the user into MPC-specific bytecode, which the virtual machine then executes.

The core part of MP-SPDZ is its virtual machine, originally introduced with SPDZ-2. Unlike standard processors, the MP-SPDZ VM is designed around the communication-heavy nature of MPC. The main characteristic of the MP-SPDZ virtual machine is that instructions involving communication take an unrestricted number of arguments, which minimizes the number of communication rounds and introduces parallelism. This is an important property especially for SMPC workflows, because compared to a traditional instruction set, where instructions have a different complexity between each other (i.e., instructions can vary based on how many clock cycles they require), instructions in SMPC also have a qualitative difference, because, e.g.. addition can be done locally by one node compared to multiplication, which requires inter-node communication as stated in Section 2.1.2.

MP-SPDZ, as a framework, belongs to one of the most mature frameworks, when it comes to academic research, being used to study, implement, and benchmark various SMPC protocols [27, 28]. The source repository is supported by a large number of contributors, and its documentation is extensive. But as of right now, MP-SPDZ only seems to be used in academic and research fields to further study SMPC and its protocols, and has not yet been used in a production setting.

## 3.4 Conclusion

In summary, Secretflow-SPU, Crypten, and MP-SPDZ each address secure multi-party computation differently, making them suitable for different scenarios. Secretflow-SPU offers a protocol-agnostic design centered on machine learning, with a compilation-based workflow that relies on intermediate representation. It supports a few well-known protocols and aims to make SMPC development seamless for ML researchers. Crypten focuses on tight integration with PyTorch in a semi-honest setting. MP-SPDZ also offers a protocol-agnostic design via compilation into an intermediate representation and supports a wide range of protocols, covering 34 variants across various adversary models, and is particularly strong as a benchmarking and experimental platform. The summarisation of the results is provided in Table 3.1.

Table 3.1: Qualitative comparison of Secretflow-SPU, Crypten, and MP-SPDZ.

| **Feature / Framework** | **Secretflow-SPU** | **Crypten** | **MP-SPDZ** |
|---|---|---|---|
| **Security Models** | Protocol-dependent | Semi-honest | Protocol-dependent |
| **Primary Focus** | Privacy-preserving ML, primarily for XLA-coded workflows | Privacy-preserving ML with PyTorch integration | Benchmarking multiple MPC protocols |
| **Protocol Support** | ABY3, SPDZ2k, Cheetah | Built-in custom protocol | 34 protocol variants |
| **Ease of Integration** | Python API, XLA-compatible ML frameworks | PyTorch-like API, with support to import already existing PyTorch models | Python API |
| **Development Scope** | ML-centric | ML-centric | Broad MPC use cases including ML |
| **Execution Model** | Compiles to pphlo, executed on virtual devices | Direct encrypted tensor ops | Compiles to MPC bytecode for VM execution |

Although all of these networks could be used for further benchmarking, the SPU framework was selected. This is mainly due to the ease of implementation, as dealing with SPU in terms of machine learning means dealing with the Jax machine learning framework, which is well documented. In the next chapter, a generic benchmark on a few machine learning architectures will be conducted to assess the performance impact based on size and depth of the network.

CHAPTER 4

# Benchmarks of generic models

In this chapter, three different neural network architectures are going to be benchmarked, namely Multi-layer perceptron (MLP), Convolutional Neural Network  (CNN), and Long Short-term Memory network (LSTM), under plaintext and SMPC setting to establish a baseline comparison of their performance and understand the computational overhead introduced by privacy-preserving inference. All the benchmarks will be done using the Secretflow-SPU framework.

## 4.1 Network Configurations

Each architecture type was tested with multiple configurations, varying the depth (number of layers) and width (units per layer) while exploring different model sizes in terms of parameters. For MLP and CNN, four different parameter sizes and for LSTM, three different sizes were tested, and for each parameter size fixed, four different variations of depth and width of the network were tested.

For MLP, the range of sizes is from 1000 parameters to 1000000 parameters. Each MLP configuration consists of ReLU activation functions after each layer. Configuration details can be seen in Table 4.1. The input tensor tested for MLP is of shape $(1, 10)$.

| **Number of parameters** | **Width $\times$ Depth** | | | |
|---|---|---|---|---|
| 1000 | 100 $\times$ 1 | 28 $\times$ 2 | 20 $\times$ 3 | 17 $\times$ 4 |
| 10000 | 1000 $\times$ 1 | 95 $\times$ 2 | 70 $\times$ 3 | 56 $\times$ 4 |
| 500000 | 50000 $\times$ 1 | 702 $\times$ 2 | 498 $\times$ 3 | 407 $\times$ 4 |
| 1000000 | 10000 $\times$ 1 | 995 $\times$ 2 | 705 $\times$ 3 | 575 $\times$ 4 |

Table 4.1: MLP configurations. Each row corresponds to a configuration with approximately the same number of network parameters.

For CNN, the parameter range is from 60000 parameters, which roughly equals the parameter size of the LeNet network, to 25000000 parameters, which is roughly equal to the size of the ResNet50 network. Each CNN consists of convolutional layers followed by a max-pooling layer and a ReLU activation. The addition of a max-pooling layer is due to it being a very common operation used in convolutional networks. The configurations can be seen in Table 4.2. The input tensor tested for CNN is of shape $(1, 112, 112, 3)$.

| **Number of parameters** | **Width × Depth** | | | |
|---|---|---|---|---|
| 60000 | 25 × 12 | 17 × 24 | 14 × 36 | 12 × 50 |
| 1000000 | 100 × 12 | 70 × 24 | 56 × 36 | 48 × 50 |
| 15000000 | 389 × 12 | 270 × 24 | 218 × 36 | 185 × 50 |
| 25000000 | 500 × 12 | 350 × 24 | 280 × 36 | 240 × 50 |

Table 4.2: CNN configurations. Each row corresponds to a configuration with approximately the same number of network parameters.

For LSTM networks, the parameter range is from 500 parameters to 5000. LSTM does not contain ReLU activations as opposed to the other two network types, due to the fact that LSTM already contains activation functions sigmoid and tanh. The configurations can be seen in Table 4.3. The input tensor tested for LSTM is of shape $(1, 1, 14)$.

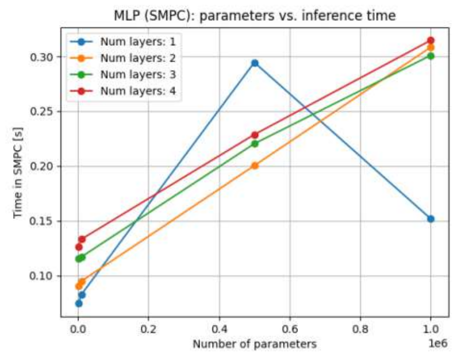
| **Number of parameters** | **Width × Depth** | | | |
|---|---|---|---|---|
| 500 | 6 × 1 | 4 × 3 | 3 × 6 | 2 × 10 |
| 1000 | 10 × 1 | 6 × 3 | 4 × 6 | 3 × 10 |
| 5000 | 29 × 1 | 14 × 3 | 10 × 6 | 8 × 10 |

Table 4.3: LSTM configurations. Each row corresponds to a configuration with approximately the same number of network parameters.

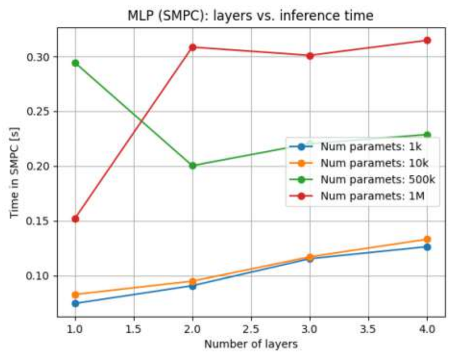
## 4.2 Results

All benchmarks are done on an Intel Xeon W-2245 CPU, 126 GB of RAM, using Ubuntu 24.04 and Python 3.10.14. The SPU framework was used, utilizing ABY3 underlying protocol with three computing nodes and two provider nodes, one for the input features and the other for trained model parameters. All tests are done in the localhost network. Each network was tested on 100 iterations, with random inputs taken from a normal distribution.

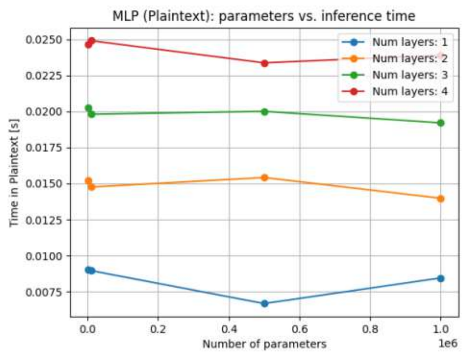
Impact of different network configurations was first tested on the MLP network shown in Figure 4.1. In Figure 4.1a, the relationship between the number of parameters and inference time is shown. Each line represents a different number of layers present in the network. It can be observed that networks with two or more layers behave very similarly. The exception comes with a one-layer MLP network that exhibits a larger increase with the rising number of parameters, but also a sharp decrease for a million-parameter, one-layer MLP. For plaintext inference in Figure 4.1c we can see that the number of



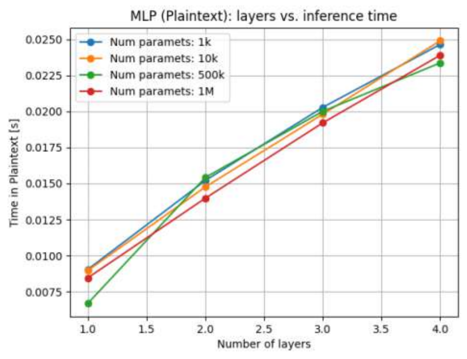
(a) Impact of number of parameters on inference time under SMPC, with fixed number of layers in the network.



(b) Impact of number of layers on inference time under SMPC, with fixed number of parameters in the network.



(c) Impact of number of parameters on inference time in plaintext, with fixed number of layers in the network.

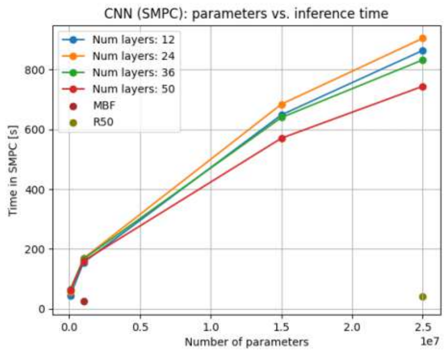


(d) Impact of number of layers on inference time in plaintext, with fixed number of parameters in the network.

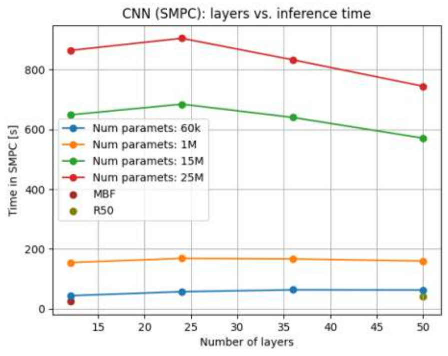
Figure 4.1: Two tested configurations on the MLP network.

parameters does not affect the inference time in this parameter range and is mostly affected by the increasing number of layers as can be seen in Figure 4.1d. In Figure 4.1b, it can be observed that both networks with one thousand parameters and ten thousand parameters behave similarly, even with an increased number of layers. For a million-parameter network, the inference time is large with one layer, but falls and stabilizes with more added layers. An inverted pattern can be observed for a million-parameter network, where with one layer, the inference time is lower but rises and stabilizes after adding more layers. Overall it can be observed that the inference reacts differently to number of parameters or layers depending if the inference is done in plaintext or SMPC.

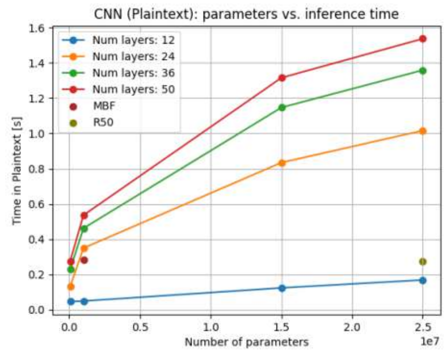
The second tested network was the CNN network shown in Figure 4.2. In Figure 4.2a, an almost linear relationship can be observed between the number of parameters and the inference time for all numbers of layers. This follows a similar pattern shown for plaintext inference of the same setting on Figure 4.2c, except for 12-layer network, which follows



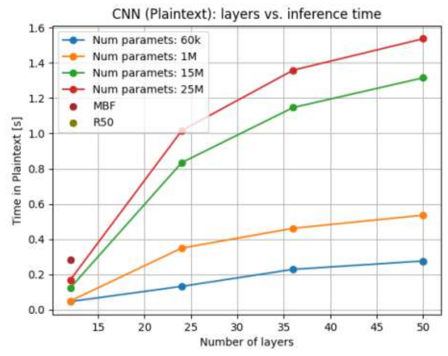
(a) Impact of number of parameters on inference time under SMPC, with fixed number of layers in the network.



(b) Impact of number of layers on inference time under SMPC, with fixed number of parameters in the network.



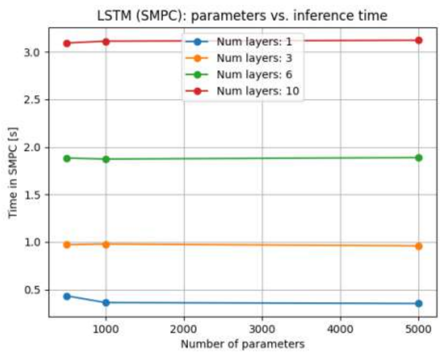
(c) Impact of number of parameters on inference time in plaintext, with fixed number of layers in the network.



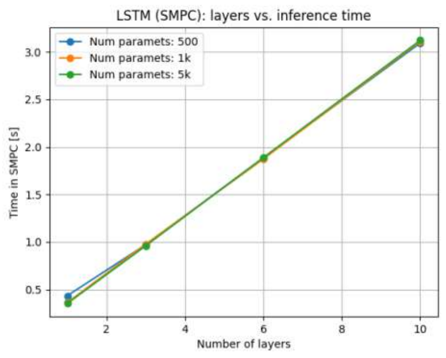
(d) Impact of number of layers on inference time in plaintext, with fixed number of parameters in the network.

Figure 4.2: Two tested configurations on the CNN network.

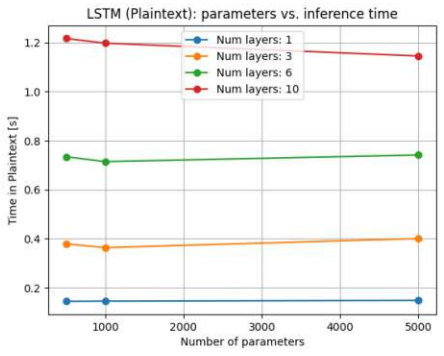
a linear relationship with a much smaller slope than the other networks. Interestingly, networks with fewer layers perform worse with a higher number of parameters in SMPC, in contrast to plaintext inference. This phenomenon can also be observed in Figure 4.2b, where networks with a higher number of parameters have a downward tendency with increased number of layers, compared to smaller networks, which appear to have a constant time when comparing number of layers and inference time. This behaviour is very different when comparing the plaintext inference on Figure 4.2d. Here, the relationship steadily increases the inference time when more layers are added to the network. For comparison, these Figures also contain two optimized and specialized networks, MobileFaceNet (MBF) and ResNet50 (R50). The MBF contains around 1 million parameters, with about twelve layers, while R50 contains around 25 million parameters and 50 layers. This shows the importance of network optimization and how it can affect the performance, compared to a generic Convolutional neural network.



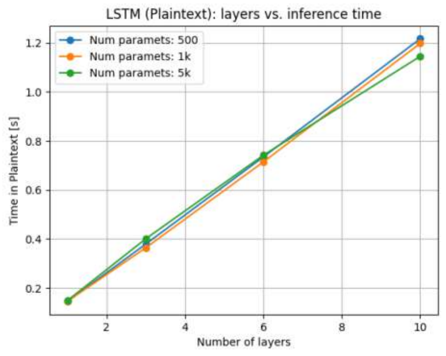
(a) Impact of number of parameters on inference time under SMPC, with fixed number of layers in the network.



(b) Impact of number of layers on inference time under SMPC, with fixed number of parameters in the network.



(c) Impact of number of parameters on inference time in plaintext, with fixed number of layers in the network.



(d) Impact of number of layers on inference time in plaintext, with fixed number of parameters in the network.

Figure 4.3: Two tested configurations on the LSTM network.

The last tested network structure was LSTM, which can be seen in Figure 4.3. In Figure 4.3a, it can be observed that the relationship between the number of parameters and inference time is constant across all the tested layers. This is an interesting observation, when compared to the other two networks, where increasing the number of parameters meant increasing the inference time. Similar pattern can be observed for plaintext inference as well as seen in Figure 4.3c. In Figure 4.3b, a linear relationship can be observed between the number of layers and the inference time, across all numbers of parameters. Once again, this is an interesting observation compared to the other two types of networks, where the increase in the number of layers did not necessarily increase the inference time. Again very similar pattern can be observed for plaintext inference as well, as seen in Figure 4.3d.

## 4.3 Conclusion

In this chapter, multiple configurations were tested on three different neural network architectures, namely MLP, CNN, and LSTM. MLP and CNN networks behave similarly, where the increase in the number of parameters generally meant an increase in the inference time, while the increase in the number of layers did not affect the inference time, with some exceptions. Interestingly, this relationship is seemingly flipped for LSTM networks. Here, the observation can be made that the number of layers linearly affects the inference time, while the parameter count does not affect the inference time itself, at least in the tested range. In the next chapter, two general use case studies will be presented in two specific machine learning scenarios, namely computer vision and time series analysis.

CHAPTER 5

# Use-case Studies

In this chapter, two different traditional machine learning applications will be transformed for an SMPC setting, namely computer vision and time-series analysis. These two applications will be tested based on two use case studies: Face recognition application and Energy Prediction for Industrial Robots. The framework of choice is Secretflow-SPU.

Since Secretflow-SPU fully supports only Jax-based [24] models, both use cases require an additional conversion step to make the models compatible with the framework. This posed a practical challenge as the original implementation of some models was available in other frameworks (PyTorch for Face verification use-case and Matlab for Robotics use-case). To address this, models were either rewritten in the Jax framework or directly developed in it from scratch.

## 5.1 Generic use-case model

In general, for both use-cases considered in this work, the generic inference model can be visualised as shown in Figure 5.1. Machine learning model inference can be described as a function of two inputs: the input features and the trained model parameters. This setup translates into three parties: the input owner (User), the model owner (Weights), and the SMPC cluster, which in this case consists of three nodes.

For the inference of a single feature, the process is as follows. The user and the model owner both secret-share their respective inputs into the function. The shares are distributed according to the protocol to each of the nodes in the SMPC cluster. The distribution is performed by the input owners themselves, so no third party is required to redistribute them. Once the inputs are distributed, the nodes perform the SMPC computation, which corresponds to the forward pass of the model.

After the computation is completed, each node sends its shares of the result back to the user. The user then locally reconstructs the correct output from the shares received.

If the SMPC cluster serves the same model to multiple users, it can retain its shares of the weights. This way, the distribution from the model's owner does not need to be repeated for each new inference request, improving efficiency. In contrast, if the model is open-source, the weights can also be sent in plaintext form. This significantly speeds up the process, at the cost of revealing the model weights to other parties.

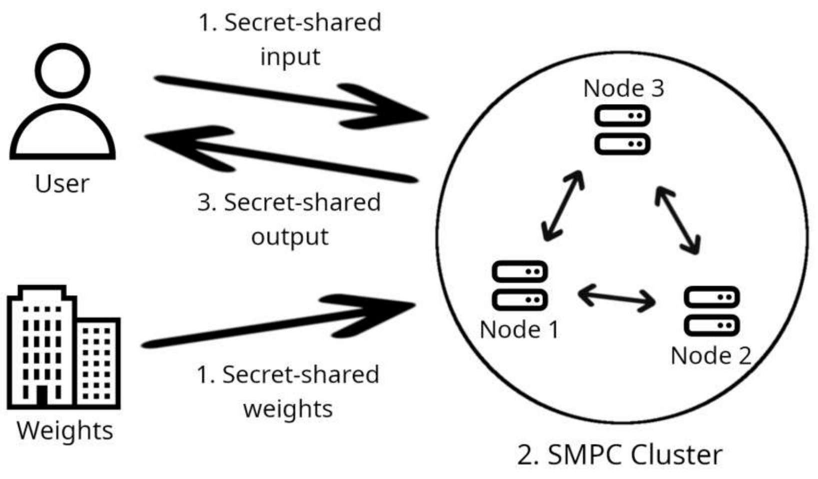


Figure 5.1: Base SMPC use-case for Machine learning inference; 1. User input and Weights input providers secret-share their respective inputs and send them to the SMPC cluster; 2. SMPC cluster does the forward pass of the network; 3. SMPC cluster sends back the secret-shared output to the user, which is then locally reconstructed to get the proper output.

## 5.2 Privacy-preserving Face Verification

Face verification is a biometric technology that verifies individuals by analyzing their facial features and comparing them with an existing database to find a match. This process is typically done by extracting vector embeddings from the face image, which are fixed-size numerical vectors that represent the face. Face verification systems offer convenience and wide applicability in areas like security, surveillance, and user access control, but are inherently very privacy sensitive as they directly process the user's face.

### 5.2.1 Motivation

Facial verification and recognition technologies have witnessed a massive growth and adoption across multiple sectors such as border control, staff management, and personal devices. The most powerful facial verification systems use machine learning models, which result in high accuracy and efficiency. However, despite its benefits, the technology itself raises security and privacy concerns. Unlike traditional authentication methods like passwords, facial biometrics are immutable features and therefore cannot be changed after a security breach or theft. Another concern is that, compared to other features used for biometric verification, like fingerprints, facial data can be taken covertly, without consent. To mitigate these risks, privacy-preserving machine learning can help mitigate these concerns by enabling secure and private biometric verification. This process can further help and deepen the trust of the public in the usage of facial verification technologies.

State-of-the-art models for face verification rely on large CNNs [29][30][31]. While very powerful, the complexity can hinder the inference time of the networks when the inference is performed under an SMPC setting. This is due to the fact that operations like multiplication or activation functions in SMPC incur significant time overhead, because computational nodes require inter-node communication to complete these operations in a secure and private way. To tackle the complexity of these larger CNNs, a range of neural networks have been proposed to work on smaller, less resource-heavy devices like mobile phones or embedded devices. While SMPC is not directly tied to such devices, it is still a very resource-demanding process and can greatly benefit from the usage of such networks. Networks like MobileNetV1 [32], ShuffleNet [33], or MobileNetV2 [34] have been developed for general visual recognition tasks. This family of networks achieves computational efficiency and reduces the number of parameters and operations compared to standard CNNs. However, these architectures are optimized for general object classification and do not perform well on the more specialized task of face verification. Due to these limitations, a new family of mobile networks specifically designed for face verification and recognition has emerged, including architectures such as MobileFaceNet [35] and GhostFaceNet [36].

In this section, two Convolutional neural networks will be presented, one smaller but highly optimized network for face verification and the other a bigger, more general network, to determine whether the benefits gained in plaintext inference speed can be translated to inference speed under SMPC.

### 5.2.2 ResNet

ResNet [16] is a CNN architecture that mitigates the problem arising from training very deep neural networks. It is a widely used structure.

#### Model architecture

The main contribution ResNet brings to the CNN family of neural networks is the introduction of Residual layers. These are shown in Figure 5.2.

The idea behind residual layers is that multiple stacked non-linear layers can approximate some complex function $\mathcal{H}(x)$, where $x$ is the input to the first layer. By the same hypothesis, then it is possible for stacked layers to also approximate some function $\mathcal{F}(x) := \mathcal{H}(x) - x$. The original function to approximate then becomes $\mathcal{F}(x) + x$. Both functions are approximately the same, but the ease of learning one is not. This phenomenon is still not fully understood, but the experiments show the increased capability of such a network to learn [16]. In the case of ResNet, the latter function becomes easier to train when it comes to deep (50-layer) stacked networks. ResNet implements this mapping by the use of shortcut connections. The input $x$ is processed by the residual layer, and after that, it is summed up with the input again, continuing further into the network. Some ResNet variants (like ResNet-50) also make use of bottleneck layers. These layers first reduce the dimension of the features, then perform traditional convolution, and finally restore the dimensions. This reduces the computational complexity to train very deep networks.

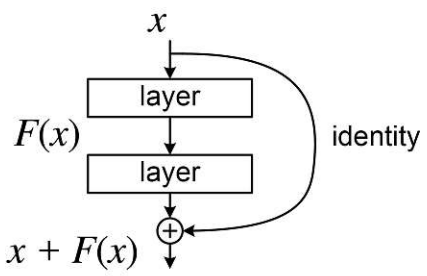


Figure 5.2: Visualisation of the residual layer used in ResNet. Taken from [16]

**Configuration**

The ResNet variant that was used is ResNet50, which consists of 50 layers. The model contains roughly 25.6 million parameters. The model was trained on the MS1MV3 dataset [37]. It contains 5179510 face images and 93431 labels. The images were cropped to the resolution $112 \times 112$ pixels, with three color channels.

### 5.2.3 MobileFaceNet

MobileFaceNet [38] is a CNN architecture that is specifically designed for high-accuracy real-time face verification on resource-constrained devices. It contains fewer than 1 million parameters.

**Model architecture**

The main building block utilised in MobileFaceNet is the residual bottleneck layer introduced in MobileNetV2 [34]. This layer consists of three important techniques used in resource-constrained convolutional neural networks, namely Depthwise Separable Convolutions, Linear Bottlenecks, and Inverted Residuals.

**Depthwise Separable Convolutions** serve as a drop-in replacement for traditional convolutional layers, greatly reducing the amount of computations. Traditional convolutional layers take an input tensor $T_i$ of size $h_i \times w_i \times d_i$ and apply a convolutional kernel $K \in \mathcal{R}^{k \times k \times d_i \times d_j}$ producing an output tensor $T_j$. The computational cost is $h_i \cdot w_i \cdot d_i \cdot d_j \cdot k \cdot k$. In contrast, Depthwise separable convolution is composed of two convolutional layers: a depthwise convolution and a pointwise convolution. The depthwise convolution applies a single kernel $K_i \in \mathcal{R}^{k \cdot k}$ for each input channel of the tensor $T_i$, producing the same number of output channels as the input. The pointwise convolution then takes this output and applies a kernel $K \in \mathcal{R}^{1 \times 1 \times d_i \times d_j}$, resulting in a tensor $T_j$. The output is identical to traditional convolution, but the computation cost is $h_i \cdot w_i \cdot d_i(k^2 + d_j)$. In a case of MobileNetV2 and MobileFaceNet, where kernel size of $k = 3$ is used, this results in a computational cost 8 to 9 times smaller than that of a standard convolution with only a small reduction in accuracy.

**Linear Bottlenecks** are bottleneck layers that use linear activation. Bottleneck layers serve as compression, reducing the size of their input. Usually, after applying the bottleneck, the output is again expanded. This results in a smaller computational cost in contrast to applying an expansive convolution at once. Using linear activation on the results of the bottleneck provided better experimental results according to [34]. Activation functions are still used after the expansion.

**Inverted Residuals** are a modification of residual blocks introduced in ResNet [16]. The difference between traditional residual blocks and inverted residuals is visualised in Fig. 5.3. In a normal residual block, the full input goes through the bottleneck, which is again expanded and summed up with the original input into the block. The inverted block does the opposite. It assumes that all the information needed for the task is already present in the bottleneck, and therefore, it uses a skip connection to sum up the "compressed" input.

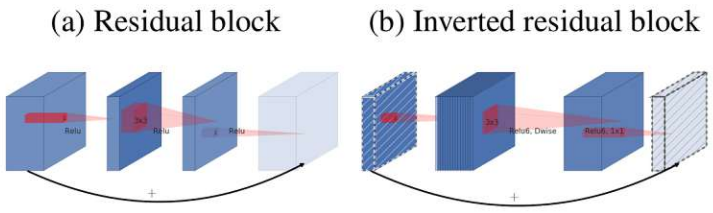


Figure 5.3: Comparison of residual block and inverted residual. Taken from [34]

In addition to the residual bottleneck layers, MobileFaceNet also introduces a Global depthwise convolution layer instead of the Global average pooling layer that MobileNetV2 uses. This layer consists of a depthwise convolution with a kernel size that is equal to the input size with zero padding and a stride equaling 1. The input of the layer is a tensor $F$ with size $w \times h \times d$ and the output is a tensor $J$ with size $1 \times 1 \times d$. This layer from experiments achieves significantly better results on LFW and AgeDB datasets compared to the network that uses a Global average pooling layer. Another change from MobileNetV2 is the usage of PReLU (Parametric Rectified Linear Unit) activation function instead of the original ReLU6 function. From the experiments, this proved to be a better-performing activation for face verification purposes.

### Configuration

MobileFaceNet architecture used in this chapter consists of roughly 1 million parameters and was trained on Casia-WebFace dataset [39]. The dataset consists of 494414 face images of 10575 real identities collected from the web. As with Resnet, the images were cropped to 112 × 112 pixels with three color channels.

## 5.2.4 Evaluation

### Qualitative results

In this section, the qualitative results of the two studied models will be presented. The accuracy of both ResNet50 and Mobilefacenet was assessed on the Labeled Faces in the Wild (LFW) dataset [40]. LFW is a widely used benchmark for face verification and face recognition. It consists of 13,233 images of 5,749 people, where 1,680 of the people have two or more distinct photos in the dataset.

Both ResNet50 and Mobilefacenet are available as open-source pretrained models from the InsightFace [1] repository, which provides implementation and weights for a wide range of state-of-the-art face recognition models. These models were used in plaintext inference to show their qualitative performance. The results presented are also taken from the InsightFace repository. The benchmark is conducted in a way where a model is appended with a face classifier that predicts the label presented in the image. This results in an accuracy metric. The backbone of that model (which is the model without the classifier, that just returns face embeddings) can be used then for verification purposes.

Table 5.1 shows the accuracy results of both models on the LFW dataset. ResNet50 achieves an accuracy of 99.73%. Mobilefacenet, on the other hand, achieves an accuracy of 99.45%, which is lower, but considering the significantly smaller size of Mobilefacenet, this highlights the possibility of a smaller, more mobile architecture to achieve comparable performance as their bigger, slower counterparts.

| Model | Accuracy [%] |
|---|---|
| ResNet50 | 99.73 |
| Mobilefacenet | 99.45 |

Table 5.1: Face verification accuracy of ResNet50 and MobileFaceNet on the LFW dataset.

### Quantitative results

In this section, the two models presented above will be evaluated based on their quantitative performance in plaintext form and SMPC form. The metric under analysis will be the inference time of pre-trained networks. The effect of other network complications, such as network delay or packet loss, will be analysed as well. All benchmarks are done

[1]https://github.com/deepinsight/insightface

on an Intel Xeon W-2245 CPU, 126 GB of RAM, using Ubuntu 24.04 and Python 3.10.14. The SPU framework was used, utilizing ABY3 underlying protocol with three computing nodes and two provider nodes, one for the input features and the other for trained model parameters. ABY3 is the protocol that's recommended for usage by the SPU developers. All tests are done in the localhost network.

Table 5.2 shows the inference times of both ResNet50 and MobileFaceNet when executed on both plaintext and encrypted data. The results show the substantial added overhead to both models that was introduced by encrypted inference. More specifically, the inference time of ResNet50 increases from 0.2741 seconds in the plaintext inference to 41.0637 seconds when doing inference under SMPC. A similar pattern can be observed for MobileFaceNet, where inference time rises from 0.2851 seconds in plaintext to 24.6422 under SMPC. Both architectures receive significant inference penalty under SMPC, but MobileFaceNet achieves lower inference times than ResNet50, both in plaintext and SMPC settings. This is caused by MobileFaceNets' efficiency and lightweight architecture that translates even more drastically in an encrypted domain, suggesting that small networks are much more suitable for privacy-preserving deployment but still fairly distant from real-world usage due to the substantial increase in inference time.

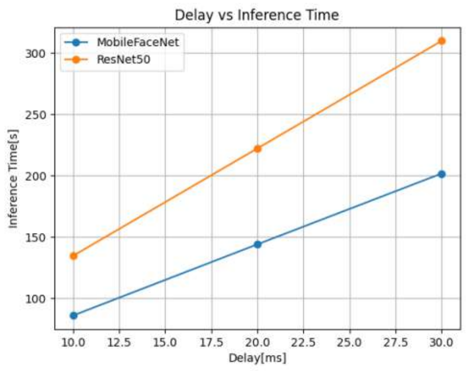
| ResNet50 | | MobileFaceNet | |
|---|---|---|---|
| Time plaintext [s] | Time encrypted [s] | Time plaintext [s] | Time encrypted [s] |
| $0.2741 \pm 0.0349$ | $41.0637 \pm 0.3600$ | $0.2851 \pm 0.0202$ | $24.6422 \pm 0.2144$ |

Table 5.2: Inference time for both ResNet50 and MobileFaceNet in encrypted and plaintext domains. Inference time is for one sample.

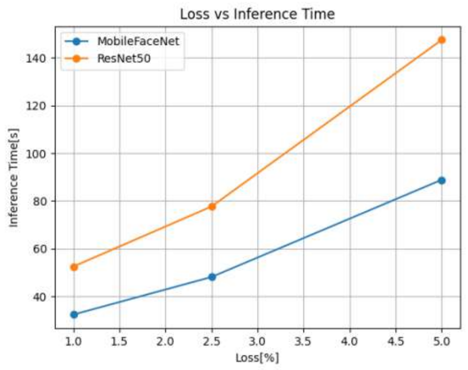
Table 5.3 shows the effect of an optimization technique, where the inputs of the network are kept private but the model weights remain unencrypted. This introduces a trade-off in privacy and performance, where the inference is substantially decreased, while the model weights can be leaked to an adversary. This does not pose a problem for machine learning models, where the pre-trained weights are open-source, but it can be problematic for parties where the trained model falls under their proprietary intellectual property. The decrease with this optimization is more substantial for ResNet50, where encrypted inference is decreased from 41.0637 seconds to 26.2798 seconds, compared to MobileFaceNet's decrease from 24.6422 seconds to 17.0413 seconds. This is likely due to the substantial difference in the architecture of both models.

| Model | Time encrypted [s] |
|---|---|
| ResNet50 | $26.2798 \pm 0.2632$ |
| MobileFaceNet | $17.0413 \pm 0.1844$ |

Table 5.3: Inference time for both ResNet50 and MobileFaceNet in the encrypted domain with model weights kept in plaintext. Inference time is for one sample.



(a) Impact of added delay on inference time



(b) Impact of added packet loss on inference time

Figure 5.4: Impact of network complications on inference time

**Deployment considerations**

The effects of network delay and packet loss on inference have also been analyzed, as these factors are critical to consider, particularly for SMPC inference, which depends heavily on inter-node communication for certain operations. Both network delay and packet loss were simulated on localhost using the Unix *tc* utility. As shown in Figure 5.4, network delay increases inference time linearly across all models. However, ResNet50 exhibits a much steeper slope. A similar trend is observed with packet loss, but with the relationship exhibiting non-linear properties. That said, extreme packet loss scenarios are not the main focus of this work, as they represent severe communication failures rather than typical real-world conditions.

## 5.3 Privacy-Preserving Energy Prediction for Industrial Robots

This section explores the implementation of a privacy-preserving framework for predicting energy consumption in industrial robots based on their trajectory using SMPC [41]. This use case demonstrates the feasibility of using ppML on a time-series problem.

### 5.3.1 Motivation

The rising focus on energy efficiency in manufacturing processes is largely motivated by escalating operational expenses and stricter environmental regulations. Since industrial production systems require substantial energy, energy modeling plays a vital role in improving resource efficiency. Within these systems, industrial robots stand out due to their broad use and considerable energy consumption in automated operations. Developing models for such systems, particularly with deep learning techniques, can demand large volumes of data and significant computational power. These models can offer substantial benefits to the industry, but their deployment often requires sharing sensitive intellectual property, such as neural network parameters of robot trajectory data. This creates challenges in collaborative settings where multiple stakeholders, like manufacturers and cloud service providers, must cooperate without disclosing proprietary information. At the same time, the industrial sector has seen rapid growth in machine learning adoption, with applications spanning predictive maintenance to process optimization. As these models become more advanced, their increasing computational demands have driven a shift towards cloud-based solutions typically managed by external providers. While this transition offers considerable computational advantages, it also raises significant privacy and security concerns for industrial stakeholders. Due to this fact, Privacy-preserving machine learning (ppML) has the potential to serve as a safeguard to address these concerns.

### 5.3.2 Data acquisition and preprocessing

The robot trajectory was generated from a seven-axis collaborative industrial robot (KUKA LBR iiwa R800). This robot is equipped with electrical actuators in each of its joints, which allow it to move. The actuators convert electrical energy into mechanical torque to rotate each axis, and the total energy consumption of the robot is therefore directly influenced by the commanded joint motions.

Two different approaches were used during generation to ensure different levels of complexity were captured:

- Limited Space Random – Random target positions are selected within a restricted part of the robot's reachable workspace.
- All Space Random – Random robot joint configurations are generated across the full range of the robot's configuration space. Only physically valid configurations are used, meaning those that do not cause collisions or violate mechanical limits.

The capture contains robot axis angles $\theta(t)$ as well as consumed active electrical power $p(t)$. All recordings are made with a sampling frequency of $\Delta T = 40ms$. To model the expected electrical power that would be used by the system, three different input features are used, namely the robot axis angles $\theta$ as well as two additional and optional derived features, angle velocity $\dot{\theta}$ and angle acceleration $\ddot{\theta}$. These are obtained using equations 5.1 and 5.2.

$$\dot{\theta}(t) \approx \frac{\theta(t+\Delta T) - \theta(t-\Delta T)}{2\Delta T} \tag{5.1}$$

$$\ddot{\theta}(t) \approx \frac{\theta(t+\Delta T) - 2\theta(t) + \theta(t-\Delta T)}{(\Delta T)^2} \tag{5.2}$$

For the preprocessing of the features, the electrical power $p(t)$ is normalized using the formula 5.3, where $p_{\text{offset}} = 155$ and $p_{\text{scale}} = 166$ are chosen empirically based on the training data.

$$p_{\text{norm}}(t) = (p(t) - p_{\text{offset}})/p_{\text{scale}} \tag{5.3}$$

The input features are scaled by factors $\theta_{\text{scale}} = 3, \dot{\theta}_{\text{scale}} = 1.6, \ddot{\theta}_{\text{scale}} = 13$.

### 5.3.3 Model training and evaluation

Three separate model architectures were designed: dense models, LSTM models, and LSTM models combined with convolutional layers. Both the dense and LSTM models are capable of handling a variable number of input features. The first network uses only the normalized seven-axis angles as input. The second network includes both the normalized axis angles and normalized angular velocities, resulting in a 14-dimensional input. The third network incorporates all 21 constructed features as input.

The dense models are composed of four linear layers with ReLU activation. They take input in the format $(b, f)$, where $b$ represents the batch size and $f$ the number of features at a single timestamp. The LSTM architecture starts with a sequence input layer, followed by an LSTM layer with 10 hidden units, and then a four-layer dense network without activation functions. Its final layer is a one-dimensional output layer. This network expects input in the format $(b, t, f)$, where $b$ is the batch size, $t$ the number of timestamps, and $f$ the number of features. Using this setup, three networks were created that differ only in input dimensionality. Notably, the datasets for both LSTM and dense models are the same; only the input arrays are reshaped to fit the required format for each architecture before training and evaluation.

The LSTM model with a convolutional layer for feature extraction is designed to automatically derive features from the data without any prior knowledge of the system. Notably, the derivative of a signal can be obtained by convolving the signal with a suitable filter kernel. Commonly used filter kernels include the first- and second-order derivatives of a Gaussian function g(t) (5.4 and 5.5) [42].

$$\frac{d}{dt} f(t) \approx f(t) \cdot \left(\frac{d}{dt} g(t)\right) \tag{5.4}$$

$$\frac{d^2}{dt^2} f(t) \approx f(t) \cdot \left(\frac{d^2}{dt^2} g(t)\right) \tag{5.5}$$

When the kernel width is reduced to $L = 3$, convolving these kernels with a signal is equivalent to computing the first and second finite differences as described in 5.1 and 5.2 for $\Delta T = 1$. Therefore, a convolutional layer with multiple filter kernels of width 3 placed before the LSTM layer can, in principle, achieve the same performance as networks using manually extracted features. In this architecture, input features are organized in the format $(b, L, f)$, where $b$ is the batch size, $f$ is the number of individual features, and

$L$ is the kernel width. The datasets remain unchanged but are reshaped so that the CNN always receives $L$ timestamps, with appropriate padding applied at the sequence boundaries.

The LSTM and CNN with LSTM models were originally implemented in MATLAB. As SPU requires JAX models, the implementation of these two architectures had to be manually rewritten.

The number of trainable parameters of all developed models and their variants is shown in Table 5.4. LSTM and dense networks are employed with varying input features. The inputs consist of either only the angular values $\theta$, angular values together with angular velocities $\theta, \dot{\theta}$, or all features including angular accelerations $\theta, \dot{\theta}, \ddot{\theta}$. For the LSTM with a convolutional layer, two networks were trained using three kernels with kernel widths of $L = 3$ and $L = 5$. This configuration uses only the angular values $\theta$ as input.

| **Architecture** | **#Input features** | **#Parameters** |
|---|---|---|
| LSTM | 7 | 903 |
| LSTM | 14 | 1183 |
| LSTM | 21 | 1463 |
| MLP | 7 | 2625 |
| MLP | 14 | 3073 |
| MLP | 21 | 3521 |
| CNN+LSTM (L=3) | 7 | 1475 |
| CNN+LSTM (L=5) | 7 | 1481 |

Table 5.4: Number of trainable parameters for different network configurations.

For training and evaluation of the models, the dataset was split into training, testing, and validation sets with a ratio of approximately 74:14:12 (248 training samples, 47 test samples, and 42 validation samples from a total of 337 samples). The training configuration is displayed on Table 5.5.

| | |
|---|---|
| Max nr. of Epochs | 2000 |
| Batch size | 12 |
| Initial learning rate | 0.02 |

Table 5.5: Training configuration.

The loss function used was the Mean Squared Error loss function 5.6, where $B$ is the batch size, $y_k$ are the logits from the last layer of the network, and $\hat{y}_k$ are the labels. The optimizer used was the Adam optimizer [43] with a piecewise constant schedule learning rate that scales the initial learning rate by 0.9 after the hundredth epoch. Early stopping was also implemented, where the training would stop after the loss would not change by 0.0001 compared to the previous epoch.

$$MSE = \frac{1}{B}\sum_{k=1}^{B}(y_k - \hat{y}_k)^2 \tag{5.6}$$

For the quality assessment of the models, the metrics Mean Absolute Error (MAE) Eq. 5.7, Root Mean Squared Error (RMSE) Eq. 5.8, and Mean Absolute Percentage Error (MAPE) Eq. 5.9 were utilized.

$$MAE = \frac{1}{N}\sum_{k=1}^{N}|y_k - \hat{y_k}| \tag{5.7}$$

$$RMSE = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(y_k - \hat{y_k})^2} \tag{5.8}$$

$$MAPE = \frac{100}{N}\sum_{k=1}^{N}\frac{|y_k - \hat{y_k}|}{|y_k|} \tag{5.9}$$

An attempt was also made to train the networks directly under SMPC. However, even for relatively small architectures, the computational and communication overhead proved to be prohibitive: completing a single training epoch required an impractically long time. For this reason, the scope of SMPC evaluation in this work was limited to the inference stage, while all model training was carried out in plaintext. This reflects a broader limitation of current SMPC frameworks, where training remains largely impractical, but inference can already be realistically applied in constrained settings.

### 5.3.4 Privacy-Preserving Collaborative Energy Prediction

Regarding testing privacy-preserving inference of the developed models, several different strategies were utilized. The first was a pure SMPC strategy in which both the trained model weights and the testing input features were secretly shared with the SMPC cluster, and the results were sent back and reconstructed by the input owner, as visualized in Figure 5.5.

Another strategy involves an optimization presented in [26]. The model weights are kept public, but are permuted before the inference with predetermined permutation matrices. Combined with the permutation of the inputs, this results in a mathematically correct forward pass of the network, while at the same time the permutation disables the honest-but-curious adversary from extracting meaningful data from the exposed weights without knowing the permutation matrices. The only option the attacker has is to brute-force the correct model weights, which is computationally very expensive.

The last strategy used is the evaluation of activation functions in plaintext. Activation functions in SMPC have a much larger impact on the inference time compared to plaintext

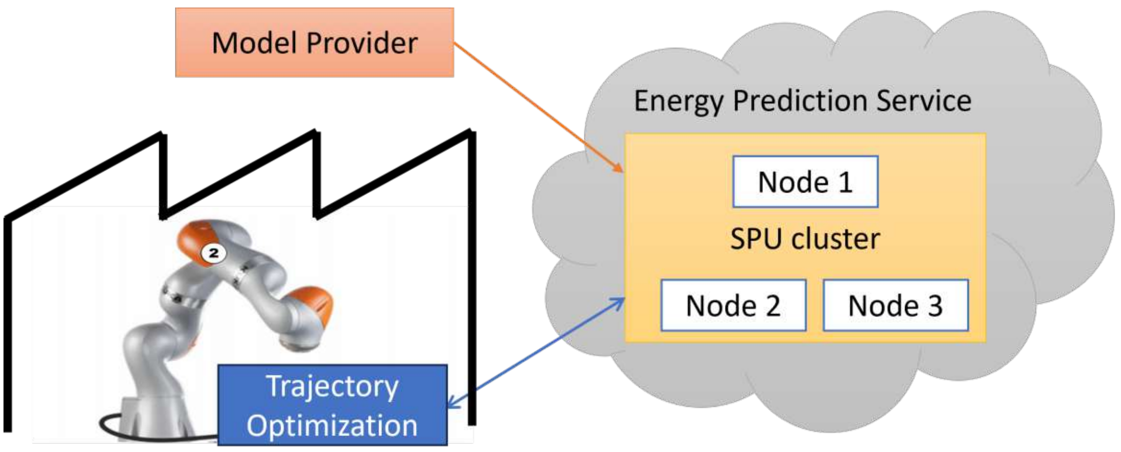


Figure 5.5: High-level overview of the basic SMPC workflow

inference [44]. One of the options to mitigate this effect is to simplify the activation functions, which often results in a decreased utility of the model. This can be mitigated by using techniques such as Knowledge Distillation (KD). In the case of the activation functions used in this use case, they already belong to a class of very simple activations. To further simplify them would mean a very drastic decrease in the model's utility. Another option is to evaluate the activations in plaintext, meaning the intermediate results would be sent back to the input owner, where they will be reconstructed. The input owner will then perform the activation and secret-share the results back to the cluster. This process will be repeated until the last activation is performed.

### 5.3.5 Evaluation

In this section, the evaluation of the developed models is discussed. First, the qualitative performance of the models is presented, followed by the performance impact of using SMPC and various optimization methods. The model naming convention in the tables below indicates the network architecture and input configuration. Three network types were tested: Multi-Layer Perceptron (MLP), Long Short-Term Memory (LSTM), and CNN-LSTM hybrid. The suffix '-Nf' denotes the number of input features used: 7 features (-Nf7), 14 features (-Nf14), or 21 features (-Nf21). For the CNN-LSTM models, 'k3' indicates three parallel kernel filters, each processing one set of the original 7 features, while 'L3' and 'L5' specify kernel sizes of 3 and 5, respectively. Thus, for example, 'mlpNf21' represents an MLP with 21 input features, while 'cnn-k3L5' represents a CNN-LSTM with three kernels of size 5.

#### Models quality evaluation

Metrics calculated on the testing dataset are presented in Table 5.6. The results show that using manually derived features is beneficial in terms of predictive quality of the model. Using all input features $\theta, \dot{\theta}$ and $\ddot{\theta}$ (-Nf21) had the lowest error rates. The most interesting outcome is from the MLP model, which achieves the best overall results. This comes from the fact that they have the largest amount of trainable parameters and therefore are able to capture the largest amount of information from the dataset. The temporal aspect of the dataset from the results does not seem to contain much additional information for the LSTM model to improve its performance over the MLP model. The results also show the viability of the CNN to automatically extract the additional features without additional preprocessing.

| Name | MAE [W] | RMSE [W] | MAPE [%] |
|---|---|---|---|
| mlpNf21 | 3.279 | 5.127 | 1.800 |
| lstmNf21 | 3.558 | 5.548 | 1.943 |
| cnn-k3L5 | 3.595 | 5.719 | 1.958 |
| lstmNf14 | 3.676 | 6.659 | 2.033 |
| cnn-k3L3 | 4.341 | 7.417 | 2.299 |
| mlpNf14 | 5.446 | 9.427 | 2.950 |
| lstmNf7 | 10.041 | 13.850 | 5.540 |
| mlpNf7 | 16.368 | 21.343 | 9.111 |

Table 5.6: Comparison of metrics for tested model structures

### Models evaluation under SMPC

The trained models are evaluated using inference time under SMPC and inference time in plaintext form as the main metric. All benchmarks are done on an Intel Xeon W-2245 CPU, 126 GB of RAM, using Ubuntu 24.04 and Python 3.10.14. The SPU framework was used, utilizing ABY3 underlying protocol with three computing nodes and two provider nodes, one for the input features and the other for trained model parameters. All tests are done in the localhost network. Table 5.7 illustrates the performance measurements. From the results, a few conclusions about the inference in SMPC can be made.

- The number of input features (7, 14, or 21) does not significantly influence the inference speed in the SMPC domain,
- The architecture of the neural network (LSTM, Dense or combination of CNN and LSTM) has the biggest influence on the inference speed,
- The number of trainable parameters does not directly correlate with the inference times.

### Analyzing performance-privacy tradeoff

The impact on inference times of the two proposed strategies in Section 5.3.4 is analyzed as well. The model chosen under analysis was the *mlpNf21* due to its best qualitative

| Model | SMPC [s] | Plaintext [s] |
|---|---|---|
| lstmNf7 | $0.5624 \pm 0.0037$ | $0.1799 \pm 0.0049$ |
| lstmNf14 | $0.5618 \pm 0.0036$ | $0.1746 \pm 0.0048$ |
| lstmNf21 | $0.5624 \pm 0.0037$ | $0.1766 \pm 0.0046$ |
| cnn-k3L5 | $0.5731 \pm 0.0026$ | $0.1827 \pm 0.0047$ |
| cnn-k3L3 | $0.5728 \pm 0.0023$ | $0.1834 \pm 0.0049$ |
| mlpNf7 | $0.1577 \pm 0.0003$ | $0.0248 \pm 0.0001$ |
| mlpNf14 | $0.1536 \pm 0.0003$ | $0.0249 \pm 0.0001$ |
| mlpNf21 | $0.1537 \pm 0.0002$ | $0.0249 \pm 0.0001$ |

Table 5.7: Comparison of inference time for all models in both SMPC and plaintext form. For SMPC inference, both model weights and inputs are secret shared.

results, fastest SMPC inference times, and simplest architecture. The inference times of the two different strategies can be seen in Table 5.8.

| Optimization used | SMPC [s] | Plaintext [s] |
|---|---|---|
| Activation functions in plaintext | 0.156 | 0.024 |
| Permutation of weights | 0.065 | 0.022 |

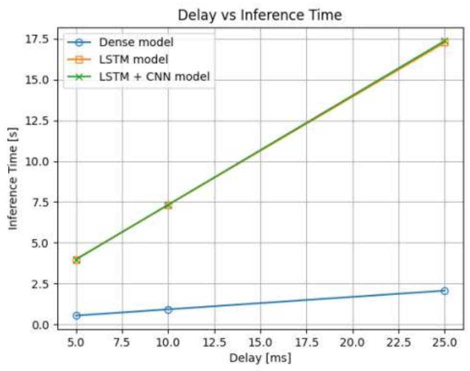
Table 5.8: Impact of different optimization attempts on mlpNf21

The evaluation of activation functions in plaintext form resulted in slightly worse inference times. This stems from the fact that the time required for collecting shares, reconstructing, and then again resharing the outputs multiple times is computationally more expensive than just an evaluation purely in the SMPC domain. This optimization can prove useful for more complex models with less frequent activation functions.

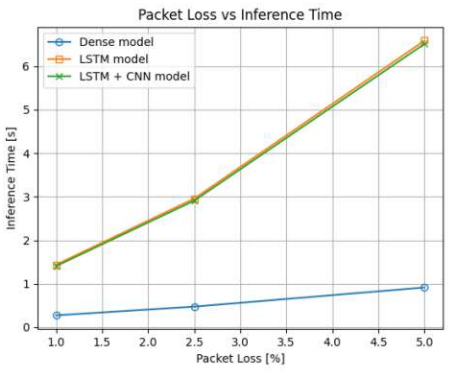
The strategy of permuting the weights and inputs and keeping the weights revealed to all parties shows significant speedup over purely SMPC implementation. This outcome is expected as SMPC computation on a revealed input significantly reduces the communication and computation overhead. The disadvantage of this approach is the limited applicability to some network architectures, e.g., CNN layers cannot be permuted this way.

### Deployment considerations

As with the previous use case, the impact of network delay and packet loss was examined. The results can be seen in Figure 5.6. From the figure, the effect of network delay linearly increases the inference time for all models, but the LSTM and LSTM with a convolutional layer are more affected. It can also be observed that both LSTM and LSTM with a convolutional layer have almost the same increase, which hints at the fact that the LSTM affects this metric the most. Similarly, for packet loss, the relationship is almost linear, with some nonlinearities visible for higher packet loss, more prevalent for models using



(a) Impact of added delay on inference time



(b) Impact of added packet loss on inference time

Figure 5.6: Impact of network complications on inference time

LSTM. However, such high packet loss does not mirror typical real-world conditions and would rather be a sign of severe communication issues on the network.

## 5.4 Conclusion

In this chapter, a use-case study was made on two different traditional Machine learning applications and the effect of SMPC on their inference time. In the first part, a face verification case study was examined, exploring two different models, namely ResNet50 and MobileFaceNet. Both models report high accuracy, but also suffer a slowdown on privacy-preserving inference. MobileFaceNet, due to its lightweight design, is much less affected, making it more suitable for inference usage with SMPC. Although not really competitive with plaintext inference, it provides important insights into studying small, mobile-friendly models and their performance in SMPC. The second part of the chapter, a privacy-preserving energy prediction model for robotics, was studied. Here, several models were developed, including an LSTM model with and without CNN attachment and a purely dense model. Here, the best performing model was the dense one with 21 input features, both in terms of accuracy metrics as well as inference time metrics in SMPC. The SMPC inference times for these dense models can be comparable to the times of plaintext LSTM models.

CHAPTER 6

# Conclusion

This thesis set out to answer the main research question: "To what extent can current SMPC frameworks effectively enable privacy-preserving machine learning inference across different application domains while maintaining acceptable performance and accuracy?" To address this, the work was divided into three parts: a review of existing frameworks, benchmarking of generic deep neural networks, and use-case studies in different application domains.

The first part of the study evaluated three state-of-the-art SMPC frameworks: MP-SPDZ, CrypTen, and SecretFlow-SPU. The analysis demonstrated that each framework has inherent design trade-offs and is best suited for different contexts. MP-SPDZ provides a versatile platform for benchmarking and academic exploration of diverse SMPC protocols. CrypTen, while conceptually useful, is no longer recommended due to its deprecation. SecretFlow-SPU, on the other hand, stands out as a more user-friendly solution, making it the most promising candidate for real-world adoption at present.

The second part of the thesis benchmarked three generic neural networks, with varying sizes and depths. MLP and CNN architectures showed expected parameter-dependent scaling, with inference time increasing roughly linearly with model size. On the other hand, LSTM networks showed a different behavior, where the increase in inference time is mostly driven by the network depth. This shows that translation from plaintext to SMPC inference is not the same across architectures. The results also showed the importance of model optimization, which is even more prevalent in the SMPC setting, shown on examples such as ResNet50 and MobileFaceNet.

The third part of the thesis involved two case studies, one in robotics and another in biometrics, to evaluate the practical feasibility of SMPC implementations in real-world scenarios. The results highlighted a clear distinction between application domains. In the biometric case study (face verification), the complexity of the required models introduced large computational and communication overheads, even when optimizations such as model weight revelation were applied. This indicates that current SMPC technology is not yet suitable for such highly complex domains, particularly under realistic network conditions with latency and delays. By contrast, the robotics use case proved significantly more suited to SMPC deployment. The models in this domain were comparatively lightweight, and as a result, the additional overhead introduced by SMPC was manageable. Furthermore, optimizations such as weight permutation reduced inference times even further, strengthening the case for realistic adoption in such settings.

These findings provide an answer to the main research question. Current SMPC frameworks can, under certain conditions, enable privacy-preserving machine learning without severely compromising performance or accuracy. However, their effectiveness is highly dependent on the application domain and the complexity of the models employed, as well as the concrete structure of the model employed. For domains requiring large-scale, high-complexity models, such as biometrics, the current generation of SMPC frameworks remains impractical for real-world deployment. On the other hand, for applications that rely on less complex models, such as robotic energy consumption prediction, SMPC is already a feasible and promising solution.

In summary, while SMPC is not yet a universal solution for privacy-preserving machine learning, it shows strong potential for adoption in specific domains with manageable model complexity. Continued research into protocol efficiency, model optimization, and pipeline integration will be crucial to extending its applicability to more demanding use cases in the future.

Future work would entail a benchmark of additional neural network models for added insights, as well as the implementation of additional case studies in different areas.

# Overview of Generative AI Tools Used

Generative AI was only used as an assistive tool for this thesis. ChatGPT and Claude helped to proofread the thesis as well as rewrite some parts to be more expressive and academic. No knowledge and/or ideas were generated and the Generative AI only worked with my own text or my own written notes. ChatGPT was also used to translate required parts into German. Generative AI was also used for some parts of the code writing but only as the starting point and was carefully reviewed and further improved by me.

# List of Figures

2.1 LeNet-5 architecture. Taken from `https://github.com/d2l-ai/d2l-en` 11

3.1 SPU architecture. Taken from [20]. . . . . . . . . . . . . . . . . . . . . . . . 14

4.1 Two tested configurations on the MLP network. . . . . . . . . . . . . . . . . 23
4.2 Two tested configurations on the CNN network. . . . . . . . . . . . . . . . . 24
4.3 Two tested configurations on the LSTM network. . . . . . . . . . . . . . . . 25

5.1 Base SMPC use-case for Machine learning inference; 1. User input and Weights input providers secret-share their respective inputs and send them to the SMPC cluster; 2. SMPC cluster does the forward pass of the network; 3. SMPC cluster sends back the secret-shared output to the user, which is then locally reconstructed to get the proper output. . . . . . . . . . . . . . . . 28
5.2 Visualisation of the residual layer used in ResNet. Taken from [16] . . . . 30
5.3 Comparison of residual block and inverted residual. Taken from [34] . . . 31
5.4 Impact of network complications on inference time . . . . . . . . . . . . . 34
5.5 High-level overview of the basic SMPC workflow . . . . . . . . . . . . . . 39
5.6 Impact of network complications on inference time . . . . . . . . . . . . . 42

# List of Tables

3.1 Qualitative comparison of Secretflow-SPU, Crypten, and MP-SPDZ. . . . 20

4.1 MLP configurations. Each row corresponds to a configuration with approximately the same number of network parameters. . . . . . . . . . . . . . . 21
4.2 CNN configurations. Each row corresponds to a configuration with approximately the same number of network parameters. . . . . . . . . . . . . . . 22
4.3 LSTM configurations. Each row corresponds to a configuration with approximately the same number of network parameters. . . . . . . . . . . . . . . 22

5.1 Face verification accuracy of ResNet50 and MobileFaceNet on the LFW dataset. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
5.2 Inference time for both ResNet50 and MobileFaceNet in encrypted and plaintext domains. Inference time is for one sample. . . . . . . . . . . . . . . . 33
5.3 Inference time for both ResNet50 and MobileFaceNet in the encrypted domain with model weights kept in plaintext. Inference time is for one sample. . . 33
5.4 Number of trainable parameters for different network configurations. . . . 37
5.5 Training configuration. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.6 Comparison of metrics for tested model structures . . . . . . . . . . . . . 40
5.7 Comparison of inference time for all models in both SMPC and plaintext form. For SMPC inference, both model weights and inputs are secret shared. . . 41
5.8 Impact of different optimization attempts on mlpNf21 . . . . . . . . . . . 41

# Bibliography

[1] P. Samarati and L. Sweeney, “Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression,” 1998.

[2] C. Dwork, “Differential privacy: A survey of results,” in *Theory and Applications of Models of Computation*, M. Agrawal, D. Du, Z. Duan, and A. Li, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–19.

[3] A. Bauer, S. Trapp, M. Stenger, R. Leppich, S. Kounev, M. Leznik, K. Chard, and I. Foster, “Comprehensive exploration of synthetic data generation: A survey,” 2024. [Online]. Available: https://arxiv.org/abs/2401.02524

[4] I. Zhou, F. Tofigh, M. Piccardi, M. Abolhasan, D. Franklin, and J. Lipman, “Secure multi-party computation for machine learning: A survey,” *IEEE Access*, vol. 12, pp. 53 881–53 899, 2024.

[5] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. P. Fitzek, and N. Aaraj, “Survey on fully homomorphic encryption, theory, and applications,” *Proceedings of the IEEE*, vol. 110, no. 10, pp. 1572–1609, 2022.

[6] C. Hong, Z. Huang, W. jie Lu, H. Qu, L. Ma, M. Dahl, and J. Mancuso, “Privacy-preserving collaborative machine learning on genomic data using tensorflow,” 2020. [Online]. Available: https://arxiv.org/abs/2002.04344

[7] Y. Akimoto, K. Fukuchi, Y. Akimoto, and J. Sakuma, “Privformer: Privacy-preserving transformer with mpc,” in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023, pp. 392–410.

[8] D. Li, R. Shao, H. Wang, H. Guo, E. P. Xing, and H. Zhang, “MPCFormer: fast, performant and private Transformer inference with MPC,” Mar. 2023, arXiv:2211.01452 [cs]. [Online]. Available: http://arxiv.org/abs/2211.01452

[9] Y. Dong, W.-j. Lu, Y. Zheng, H. Wu, D. Zhao, J. Tan, Z. Huang, C. Hong, T. Wei, and W. Chen, “PUMA: Secure Inference of LLaMA-7B in Five Minutes,” Sep. 2023, arXiv:2307.12533 [cs]. [Online]. Available: http://arxiv.org/abs/2307.12533

[10] D. Evans, V. Kolesnikov, M. Rosulek *et al.*, “A pragmatic introduction to secure multi-party computation,” *Foundations and Trends® in Privacy and Security*, vol. 2, no. 2-3, pp. 70–246, 2018.

[11] D. Beaver, “Efficient multiparty protocols using circuit randomization,” in *Advances in Cryptology — CRYPTO ’91*, J. Feigenbaum, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 420–432.

[12] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. v. d. Maaten, “CrypTen: Secure Multi-Party Computation Meets Machine Learning,” Sep. 2022, arXiv:2109.00984 [cs]. [Online]. Available: http://arxiv.org/abs/2109.00984

[13] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0893608089900208

[14] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998, conference Name: Proceedings of the IEEE. [Online]. Available: https://ieeexplore.ieee.org/document/726791

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3065386

[16] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[17] R. C. Staudemeyer and E. R. Morris, “Understanding lstm – a tutorial into long short-term memory recurrent neural networks,” 2019. [Online]. Available: https://arxiv.org/abs/1909.09586

[18] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, “Falcon: Honest-majority maliciously secure framework for private deep learning,” 2020. [Online]. Available: https://arxiv.org/abs/2004.02229

[19] M. Keller, “MP-SPDZ: A Versatile Framework for Multi-Party Computation,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’20. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1575–1590. [Online]. Available: https://dl.acm.org/doi/10.1145/3372297.3417872

[20] J. Ma, Y. Zheng, J. Feng, D. Zhao, H. Wu, W. Fang, J. Tan, C. Yu, B. Zhang, and L. Wang, “{SecretFlow-SPU}: A Performant and {User-Friendly} Framework for {Privacy-Preserving} Machine Learning,” 2023, pp. 17–33. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/ma

[21] P. Mohassel and P. Rindal, "ABY3: A Mixed Protocol Framework for Machine Learning," 2018, publication info: Published elsewhere. Minor revision. 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18). [Online]. Available: https://eprint.iacr.org/2018/403

[22] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "SPDZ2k: Efficient MPC mod 2^k for Dishonest Majority," 2018, publication info: A minor revision of an IACR publication in CRYPTO 2018. [Online]. Available: https://eprint.iacr.org/2018/482

[23] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference," 2022, publication info: Published elsewhere. Minor revision. USENIX Security'22. [Online]. Available: https://eprint.iacr.org/2022/207

[24] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/jax-ml/jax

[25] H. Wu, W. Fang, Y. Zheng, J. Ma, J. Tan, and L. Wang, "Ditto: Quantization-aware secure inference of transformers upon MPC," in *Proceedings of the 41st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, Eds., vol. 235. PMLR, 21–27 Jul 2024, pp. 53 346–53 365. [Online]. Available: https://proceedings.mlr.press/v235/wu24d.html

[26] J. Luo, G. Chen, Y. Zhang, S. Liu, H. Wang, Y. Yu, X. Zhou, Y. Qi, and Z. Xu, "Centaur: Bridging the Impossible Trinity of Privacy, Efficiency, and Performance in Privacy-Preserving Transformer Inference," Dec. 2024, arXiv:2412.10652 [cs]. [Online]. Available: http://arxiv.org/abs/2412.10652

[27] T. Gehlhar, F. Marx, T. Schneider, A. Suresh, T. Wehrle, and H. Yalame, "Safefl: Mpc-friendly framework for private and robust federated learning," in *2023 IEEE Security and Privacy Workshops (SPW)*, 2023, pp. 69–76.

[28] T. Lorünser and F. Wohner, "Performance comparison of two generic mpc-frameworks with symmetric ciphers." *ICETE (2)*, vol. 587594, 2020.

[29] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A Unified Embedding for Face Recognition and Clustering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015.

[30] J. Deng, J. Guo, N. Xue, and S. Zafeiriou, "ArcFace: Additive Angular Margin Loss for Deep Face Recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019.

[31] Y. Wen, K. Zhang, Z. Li, and Y. Qiao, “A Discriminative Feature Learning Approach for Deep Face Recognition,” in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, vol. 9911, pp. 499–515, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-46478-7_31

[32] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” Apr. 2017, arXiv:1704.04861 [cs]. [Online]. Available: http://arxiv.org/abs/1704.04861

[33] X. Zhang, X. Zhou, M. Lin, and J. Sun, “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2018.

[34] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2018.

[35] S. Chen, Y. Liu, X. Gao, and Z. Han, “MobileFaceNets: Efficient CNNs for Accurate Real-Time Face Verification on Mobile Devices,” Jun. 2018, arXiv:1804.07573 [cs]. [Online]. Available: http://arxiv.org/abs/1804.07573

[36] M. Alansari, O. A. Hay, S. Javed, A. Shoufan, Y. Zweiri, and N. Werghi, “Ghostfacenets: Lightweight face recognition model from cheap operations,” *IEEE Access*, vol. 11, pp. 35 429–35 446, 2023.

[37] J. Deng, J. Guo, D. Zhang, Y. Deng, X. Lu, and S. Shi, “Lightweight face recognition challenge,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, Oct 2019.

[38] S. Chen, Y. Liu, X. Gao, and Z. Han, “MobileFaceNets: Efficient CNNs for Accurate Real-Time Face Verification on Mobile Devices,” in *Biometric Recognition*, J. Zhou, Y. Wang, Z. Sun, Z. Jia, J. Feng, S. Shan, K. Ubul, and Z. Guo, Eds. Cham: Springer International Publishing, 2018, vol. 10996, pp. 428–438, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-97909-0_46

[39] D. Yi, Z. Lei, S. Liao, and S. Z. Li, “Learning face representation from scratch,” 2014. [Online]. Available: https://arxiv.org/abs/1411.7923

[40] G. B. Huang, M. Mattar, T. Berg, and E. Learned-Miller, “Labeled Faces in the Wild: A Database forStudying Face Recognition in Unconstrained Environments,” in *Workshop on Faces in ’Real-Life’ Images: Detection, Alignment, and Recognition*. Marseille, France: Erik Learned-Miller and Andras Ferencz and Frédéric Jurie, Oct. 2008. [Online]. Available: https://inria.hal.science/inria-00321923

[41] A. Skuta, P. Steurer, S. Hegenbart, R. Hoch, and T. Loruenser, "Towards privacy-preserving machine learning for energy prediction in industrial robotics: Modeling, evaluation and integration," *Machines*, vol. 13, no. 9, 2025. [Online]. Available: https://www.mdpi.com/2075-1702/13/9/780

[42] S. Birchfield, *Image Processing and Analysis.* Cengage Learning, 2016. [Online]. Available: https://books.google.at/books?id=hOu5DQAAQBAJ

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. [Online]. Available: https://arxiv.org/abs/1412.6980

[44] J. Luo, Y. Zhang, Z. Zhang, J. Zhang, X. Mu, H. Wang, Y. Yu, and Z. Xu, "SecFormer: Fast and Accurate Privacy-Preserving Inference for Transformer Models via SMPC," Dec. 2024, arXiv:2401.00793 [cs]. [Online]. Available: http://arxiv.org/abs/2401.00793