Towards Practical Secure Neural Network Inference: The Journey So Far and the Road Ahead

ZOLTÁN ÁDÁM MANN, University of Amsterdam, The Netherlands
CHRISTIAN WEINERT, Royal Holloway, University of London, United Kingdom
DAPHNEE CHABAL, University of Amsterdam, The Netherlands
JOPPE W. BOS, NXP Semiconductors, Belgium

Neural networks (NNs) have become one of the most important tools for artificial intelligence. Well-designed and trained NNs can perform inference (e.g., make decisions or predictions) on unseen inputs with high accuracy. Using NNs often involves sensitive data: Depending on the specific use case, the input to the NN and/or the internals of the NN (e.g., the weights and biases) may be sensitive. Thus, there is a need for techniques for performing NN inference securely, ensuring that sensitive data remain secret.

In the past few years, several approaches have been proposed for secure neural network inference. These approaches achieve better and better results in terms of efficiency, security, accuracy, and applicability, thus making big progress toward practical secure neural network inference. The proposed approaches make use of many different techniques, such as homomorphic encryption and secure multi-party computation. The aim of this article is to give an overview of the main approaches proposed so far, their different properties, and the techniques used. In addition, remaining challenges toward large-scale deployments are identified.

CCS Concepts: • **Security and privacy** → *Privacy protections*; **Privacy-preserving protocols**; • **Computing methodologies** → **Neural networks;**

Additional Key Words and Phrases: Privacy-preserving machine learning, secure inference, neural networks, deep learning, secure computation, homomorphic encryption, multi-party computation

ACM Reference format:
Zoltán Ádám Mann, Christian Weinert, Daphnee Chabal, and Joppe W. Bos. 2023. Towards Practical Secure Neural Network Inference: The Journey So Far and the Road Ahead. *ACM Comput. Surv.* 56, 5, Article 117 (November 2023), 37 pages.
https://doi.org/10.1145/3628446

1 INTRODUCTION

The field of machine learning (ML) has been subject to enormous uptake in the recent past. One of the main drivers behind this success is the progress in deep neural networks (NNs), also known as deep learning (DL) [99]. Deep NNs are now routinely used in a variety of applications, including image recognition, natural language understanding, and drug discovery [168]. In these and many

Authors' addresses: Z. Á. Mann and D. Chabal, University of Amsterdam, Science Park 900, 1098 XH Amsterdam, The Netherlands; C. Weinert, Royal Holloway, University of London, Egham Hill B2-02 (Bedford Building), Egham TW20 0EX, United Kingdom; J. W. Bos, NXP Semiconductors, Interleuvenlaan 80, 3001 Leuven, Belgium.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0360-0300/2023/11-ART117 $15.00
https://doi.org/10.1145/3628446

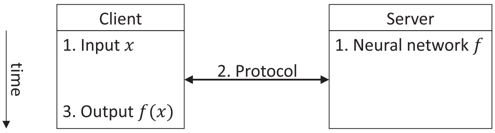


Fig. 1. Typical setup of secure neural network inference. The client obtains $f(x)$ through a protocol that ensures that both x and f remain secret.

other domains, deep NNs have outperformed other ML techniques in terms of accuracy and are often competitive with the performance of humans [67].

Deep learning is fueled by data. In many cases, the involved data may be sensitive. For example, it can be personal data, subject to privacy concerns and data protection laws [156]. Also non-personal data can be sensitive, e.g., if it represents business secrets or other intellectual property. If the involved data are sensitive, then this puts constraints on how it can be used in DL. Thus, there is a need for approaches that enable DL while ensuring that security and privacy requirements are met [21, 168]. Developing such approaches is challenging because of an intrinsic conflict: To achieve good accuracy, DL needs full access to a large amount of precise data, whereas security and privacy concerns entail limiting the access to data. Nevertheless, modern cryptographic methods offer possibilities to achieve a good tradeoff between the conflicting objectives of accuracy and security.

A NN is evaluated on an input to produce an output. Typically, both the input and the output are vectors, i.e., lists of numbers. For example, in the widely used MNIST benchmark [100], the input encodes a 28×28 grayscale picture of a handwritten digit by listing the darkness of the picture's pixels, while the output is a vector of size 10 containing the likelihood that the picture shows the digit $0, 1, \ldots, 9$. DL is often used for classification, i.e., to decide which of a predefined set of classes the input belongs to. In the MNIST example, the picture needs to be classified as one of the 10 possible digits. This is achieved by determining which of the 10 outputs is the highest.

ML typically consists of two phases: training and inference. In the training phase, a large amount of training data is used to determine the best parameter values of a NN. In the inference phase, the already trained NN is applied to a new input. While the operations performed in the two phases are similar, the security and privacy requirements are very different in the two phases. In this article, we *focus on the inference phase*. Key requirements for inference include low latency (i.e., inference should be fast) and high accuracy (i.e., the output should be correct as often as possible). In the inference phase, security and privacy become a concern if the NN and the input are held by different parties. This is the case in the context of Machine Learning as a Service (MLaaS) [145]. In this scenario, a company offers inference with a pre-trained NN as a service to its clients. As Figure 1 shows, the client holds an input x, while the NN realizing function f resides on a remote server. The client wants to obtain $f(x)$, the output of the NN when applied to x. However, the client wants to keep its sensitive input x private, i.e., x must not be uploaded to the server. Downloading the NN to the client is also not an option, because the NN represents the service provider's intellectual property that must not be shared with clients. The secure neural network inference (SNNI)[1] problem entails calculating $f(x)$ for the client while satisfying these security requirements.

Solving this problem is challenging. Nevertheless, in recent years, several approaches have been proposed for SNNI. These approaches are often based on cryptographic techniques, such as homomorphic encryption and secure multi-party computation, although also other methods have been proposed, such as hardware-based trusted execution environments. The proposed

[1] Other names sometimes used in the literature for the same include *privacy-preserving inference* and *oblivious prediction*.

approaches achieve different tradeoffs in terms of efficiency, security, accuracy, and applicability. Significant progress has been achieved in all of these dimensions, making SNNI increasingly practical. However, since many different techniques are used and different tradeoffs are achieved, the fast progress has made it difficult to get an overview of this field.

The aim of this article is to provide a guided tour of this fascinating research area. Previous relevant surveys presented a high-level overview of larger areas, covering security and privacy aspects of different types of ML models, different attack possibilities, and different phases of the ML pipeline [21, 105, 124, 168]. In contrast, we focus on the specific topic of secure neural network inference, which allows us to provide much deeper insights into the properties and the internals of different SNNI approaches.[2]

We limit the scope of the article to solution techniques that offer strong cryptographic security guarantees and do not require special hardware. Thus, we exclude techniques based on model splitting, where the client evaluates some layers of the NN and the server evaluates the other layers [122], because these techniques do not provide security guarantees. Also, solution techniques based on trusted execution environments [157] are out of scope as they require special hardware.

The rest of this article is organized as follows. Section 2 presents basics in NNs and relevant security techniques. Section 3 reviews the characteristics of SNNI approaches, treating the approaches as black box, focusing on what they assume and what they guarantee. In contrast, Section 4 reviews the inner working of SNNI approaches. Section 5 reviews implementation issues and Section 6 discusses how SNNI approaches are evaluated. Finally, Section 7 discusses our insights and proposes future research directions, and Section 8 concludes the article.

2 PRELIMINARIES

In the following, we summarize relevant background information on NNs as well as secure computation techniques that are frequently used to build SNNI solutions. We assume that the reader is familiar with at least the basic notions of cryptography and the related mathematical concepts. Readers aiming at understanding the details of the relevant secure computation techniques will need advanced cryptographic knowledge. We refer readers looking for more background information in cryptography to relevant books such as References [61, 66, 84].

2.1 Neural Networks

A NN is an algorithm that transforms an input vector of numerical data into an output vector representing some insight about the input. The input vector can represent different types of data [134]. For instance, words in a sentence can be represented by numbers such that words with similar meanings are represented with similar values [119]. An image of $N \times M$ pixels can be used as an input vector of length $N \cdot M$ for grey-scale images or of length $3 \cdot N \cdot M$ for RGB-encoded images. Input vectors can also represent records of data, where each number represents an attribute of interest, for instance, different characteristics of a person.

The interpretation of the output vector depends on the task of the NN. For classification tasks, usually each output number corresponds to one possible class and expresses how likely it is that the input belongs to the given class. For example, if an NN is used to classify images of animals into the classes cat, dog, and elephant, then the output is a vector of length 3, and the highest number in the output vector determines the classification result.

Traditionally, NNs are composed of neurons. A *neuron* has a list of weights $w_1, \ldots, w_n \in \mathbb{R}$ and a bias $b \in \mathbb{R}$. As illustrated in Figure 2, the neuron gets as input a list of numbers $x_1, \ldots, x_n \in \mathbb{R}$. The

[2] We limit our survey to papers that were published or accepted for publication in peer-reviewed scientific conferences or journals and only in exceptional cases refer to preprints.

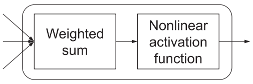


Fig. 2. A neuron.

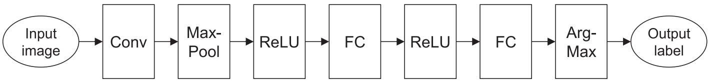


Fig. 3. An example convolutional neural network (based on Reference [144]).

neuron first computes $y = w_1x_1 + \cdots + w_nx_n + b$ and then applies a non-linear function $f : \mathbb{R} \rightarrow \mathbb{R}$ to compute the output $z = f(y)$. In an alternative representation, y is seen simply as the dot product of the weight vector and the input vector, with b being an element of the weight vector and the constant 1 signal being the corresponding element in the input vector. Typical choices for the non-linear function f include the tanh function, the sigmoid function $\sigma(y) = 1/(1 + e^{-y})$, and the ReLU function $\text{ReLU}(y) = \max(0, y)$.

An NN consists of a sequence of layers.[3] The first layer contains the inputs to the NN, and the output of the last layer is the result of the NN. The layers in between, called hidden layers, are traditionally seen as being composed of neurons. The outputs of the neurons of layer i are used as the inputs to the neurons of layer $i + 1$.

In recent years, it has become more common to look at the functionalities of layers instead of the operation of individual neurons. A layer performs one type of computation on its input vector to create its output vector, which is then used as the input vector of the next layer.[4] Many different types of layers are used in modern NNs, such as the following.

- Fully connected (FC) layer: Contains a matrix of weights. Computes the output vector by multiplying the input vector with the weight matrix.
- Convolutional (Conv) layer: Contains a set of weight matrices called feature maps that are smaller than the input. Computes output numbers by sliding a window of the size of the feature map over the input and computing the dot product of the feature map with the part of the input in the sliding window. Conv layers play an important role in convolutional neural networks (CNNs), which are widely used in image processing tasks.
- Activation layer: Applies the same $\mathbb{R} \rightarrow \mathbb{R}$ activation function to each element of the input to compute the elements of the output. Depending on the function, there can be ReLU activation layers, tanh activation layers, and so on.
- Pooling layer: Computes output numbers by sliding a window of size k over the input, and applying an $\mathbb{R}^k \rightarrow \mathbb{R}$ pooling function. The most common pooling function is the max function, resulting in a Max-Pool layer.

FC and Conv layers are called linear layers, because their output is a linear function of their input. In fact, Conv layers can also be represented by matrix multiplications, just as FC layers. Most of the other layer types are non-linear.

A NN is created by first determining its *architecture*. This entails the number of layers, their types, and their order (cf. Figure 3 for an example). In addition, the sizes of the layers are determined (note that FC, Conv, and pooling layers may change the size of the processed vectors, thus creating some degrees of freedom).

After the architecture is determined, the NN has to be trained. During *training*, the parameters in the NN, in particular weights in FC layers and feature maps in Conv layers, are iteratively tuned so that the outputs of the NN match the expectations as much as possible. This is performed by

[3] We describe here feed-forward NNs, the most common type of NNs. There are also other types, such as recurrent neural networks that represent variations of this general scheme.

[4] A traditional layer of neurons performs the task of two consecutive layers in this more modern view. As a consequence, different authors may count the number of layers differently.

applying the NN to inputs for which the expected output is known. The actual output of the NN is compared with the expected output, and the differences are back-propagated to tune the parameters by an appropriate algorithm. After the training is finished, the NN can be used for *inference*, i.e., be applied to new inputs. The *accuracy* of a trained NN is the ratio of inputs for which the NN's result is correct. Accuracy is typically measured using a dedicated validation dataset.

2.2 Homomorphic Encryption

Privacy concerns often result in slow adoption of new data-driven techniques such as processing sensitive data as required in the MLaaS scenario. The privacy of sensitive information can be guaranteed if these data are encrypted by the user before being uploaded to a cloud service. In that way, only the legitimate user can access the data by decrypting it using their private decryption key. But encryption limits the possibility to outsource *computation* on the externally stored information. In services such as MLaaS, if the user provides their input in an encrypted form, then it seems impossible for the service provider to perform the required computations without first decrypting the input.

This seemingly impossible task can be solved with homomorphic encryption (HE), a privacy-enhancing technology introduced in the late 1970s by Rivest, Adleman, and Dertouzos [146]: the ability to compute meaningful operations on encrypted data. That is, if the input is encrypted using an HE scheme, then the service provider can perform operations on the encrypted data and send the result, still in encrypted form, back to the user. The user can decrypt the output and obtain the correct result (cf. Figure 4). Throughout the computation, the service provider only has access to encrypted data; thus, the secrecy of the user's data is guaranteed.

It is not too difficult to create an encryption scheme that supports one type of homomorphic operation (i.e., one type of operation, such as addition or multiplication, can be performed by appropriate transformation of the encrypted data). For example, the RSA cryptosystem [147] allows us to perform multiplications on the encrypted data (multiplicative homomorphic encryption), while the Paillier cryptosystem [123] allows homomorphic additions (additive homomorphic encryption). Such encryption schemes are called partially homomorphic encryption.

It is much more difficult to devise an encryption scheme that supports both homomorphic addition and homomorphic multiplication. Such schemes are called fully homomorphic encryption (FHE). It took until 2009 until a concrete FHE scheme was found by Gentry [59]. Using addition and multiplication as building blocks, arbitrarily complex computations can be realized using *arithmetic circuits*. Thus, an FHE scheme allows a service provider to compute arbitrarily complex functions on encrypted data without learning anything about the content of these data.

In today's FHE schemes, the ciphertext typically contains a small random additive term called *noise*. The noise is essential for the security of these cryptosystems, and it causes no problems as long as it remains within given bounds. However, homomorphic operations lead to an increase of the noise (i.e., the noise in the result of the operation is higher than the noise in the operands). If the noise in a ciphertext exceeds a predefined threshold, then the decryption of the ciphertext may lead to a wrong result. To cope with this problem, FHE schemes must reset the noise in ciphertexts, before it becomes too large, using an approach called *bootstrapping*. The main idea behind bootstrapping is to evaluate the decryption circuit homomorphically. Using bootstrapping, an unlimited number of homomorphic operations can be carried out, and the output can still be correctly decrypted. Unfortunately, bootstrapping has a significant performance cost in practice. The research field of FHE is now in its fourth generation of schemes, and the overall performance is getting more practical (especially for use cases such as NN inference).

If the arithmetic circuit to be applied to the encrypted data is fixed and known in advance, then it is possible to avoid the costly bootstrapping operations. This requires a careful analysis of the

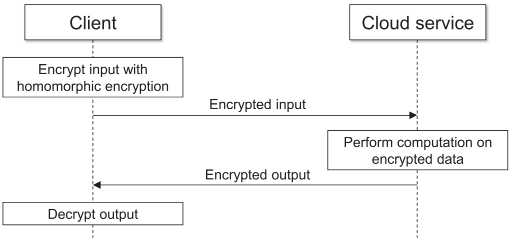

Client
Cloud service
Encrypt input with homomorphic encryption
Encrypted input
Perform computation on encrypted data
Encrypted output
Decrypt output

Fig. 4. Using homomorphic encryption for offloading computation to a cloud service.

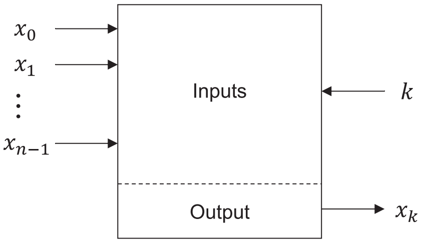


Fig. 5. High-level overview of 1-out-of-n oblivious transfer.

noise growth caused by the arithmetic circuit and setting the parameters of the HE scheme in such a way that the output's noise is guaranteed to remain within the allowed bounds. Note that this way, unlike with FHE, it is only possible to perform a limited number of homomorphic operations. This type of approach is called *leveled* FHE or somewhat homomorphic encryption (SHE).

In FHE schemes, the mathematical objects one computes on are typically polynomials chosen carefully such that they allow efficient arithmetic [133]. More details on HE schemes relevant for SNNI can be found in Section 4.1.2.

2.3 Oblivious Transfer

Oblivious transfer (OT) is a cryptographic two-party protocol. It allows the receiving party to obliviously select one of the sending party's inputs [136]. The protocol's privacy guarantees ensure that the sender does not learn the choice of the receiver and the receiver does not learn the non-selected inputs. More formally, in classic 1-out-of-2 OT, the sender has inputs x_0, x_1 and the receiver has a choice bit b. After invoking the OT protocol, the receiver learns x_b, but no information about x_{1-b}, and the sender learns nothing. More generally, in 1-out-of-n OT, the sender has inputs $x_0, x_1, \dots, x_{n-1}$, and the receiver has a number $k \in \{0, 1, \dots, n-1\}$. As a result of the protocol, the receiver learns x_k but nothing about the sender's other inputs, and the sender does not learn anything (cf. Figure 5). As will be discussed in the following sections, this functionality can be utilized in different ways as a building block for interactive secure computation protocols, e.g., in the garbled circuit protocol (Section 2.4).

Several protocols were proposed to securely realize the described OT functionality [12, 45, 50, 117, 129]. Unfortunately, all known implementations incur non-negligible computation and communication cost, which is problematic if a higher-level protocol uses a large number of OTs. This is not accidental: It was shown in Reference [79] that OT inherently requires some form of cryptographic hardness assumptions and therefore has to utilize computationally expensive asymmetric cryptography. For example, the protocol of Reference [45] is based on the Diffie–Hellman key-exchange protocol, which in turn requires relatively expensive modular exponentiations.

To benefit from significantly faster symmetric-key cryptography primitives (e.g., via hardware-accelerated AES evaluations), *OT extension* protocols were constructed [5, 80, 86, 90, 150]. OT extension allows performing a large number of OTs from a small constant number of base OTs, where only the base OTs require asymmetric cryptography, and cheaper symmetric-key cryptography primitives suffice for the further OTs. This way, if a large number of OTs is needed, then their average (also called amortized) complexity can be significantly reduced. For example, the amortized communication complexity of passively secure 1-out-of-2 OT extension protocols on l bit inputs [5, 80] is $\kappa + 2l$ bits per OT, where κ is the symmetric security parameter.

There are also variants of the OT functionality that can be implemented with a reduced communication overhead and are sufficient for many applications. For example, in *random OT*,

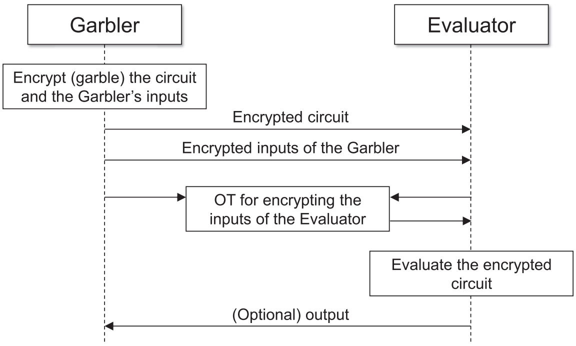

Garbler
Evaluator
Encrypt (garble) the circuit and the Garbler's inputs
Encrypted circuit
Encrypted inputs of the Garbler
OT for encrypting the inputs of the Evaluator
Evaluate the encrypted circuit
(Optional) output

Fig. 6. High-level overview of the garbled circuit protocol.

the inputs x_0, x_1 are randomly chosen by the protocol, which allows reducing the amortized communication complexity to κ bits per OT [5, 80]. It is also possible to pre-compute OTs [10] to ensure fast protocol execution when the inputs are known.

A recent line of work proposed *silent* OT extension [23, 47, 164]. Silent OT introduces a computation–communication performance tradeoff, as it makes it possible to generate a larger number of OT instances from small correlated seeds without further communication. For example, the communication of the protocol in Reference [23] for random OT extension is amortized to 0.1 bit per random OT for 10^7 instances.

2.4 Multi-party Computation and Garbled Circuits

Multi-party computation (MPC) entails computing the value of a function $f(x_1, \dots, x_n)$ by n parties. The function is publicly known, but the inputs are secrets of the individual parties: x_1 is a secret of party P_1, x_2 is a secret of party P_2, and so on. The aim is to compute the value of the function in such a way that no party learns anything about the other parties' inputs beyond what the output reveals. An important special case is when $n = 2$; this is called two-party computation (2PC).

Garbled circuits (GCs) are a prominent approach for the 2PC problem, introduced by Andrew Yao in the 1980s [165, 166]. The GC protocol assumes that the function to be computed is given as a Boolean circuit consisting of only AND and XOR gates. A subset of the input bits form a secret of one party, while the remaining input bits form a secret of the other party. In the context of GC, the parties are called garbler and evaluator. On a high level (cf. Figure 6), the idea is as follows:

(1) The garbler creates an "encrypted," also called *garbled*, version of the Boolean circuit, by encrypting the values on the wires and the truth tables of the gates.
(2) After the garbler transfers the garbled circuit description to the evaluator, the evaluator can evaluate the garbled circuit on the encrypted inputs.
(3) Once in the possession of the encrypted circuit and all encrypted inputs, the evaluator can evaluate the circuit gate by gate to compute the output.
(4) The end of the protocol depends on which party should learn the output: the garbler, the evaluator, or both. Depending on this, the output computed by the evaluator may or may not be encrypted, and there may be a final step in which the evaluator sends the (potentially encrypted) output to the garbler.

An in-depth discussion of GCs can be found in References [46, 66]. The described protocol has a constant number of communication rounds, which is independent of the computed functionality. However, the overall computation and communication associated with the protocol can be quite large. Over the years, many variations and optimizations have been introduced [11, 91, 92, 95, 118, 131, 148, 167] that reduce the number of ciphertexts that must be transferred and/or speed up the garbling and evaluation processes. Most notably, the Free-XOR optimization [92] enables the evaluator to locally compute XOR operations without the requirement for the garbler to create and transfer a garbled truth table. As a consequence, the most relevant metric for optimizing circuits for GC evaluation is the number of AND gates in a circuit, also known as the multiplicative size of the circuit. Together with the "half gates" optimization [167], the communication cost for each AND gate is 2κ and the computation cost is dominated by four AES evaluations on the garbler's and two AES evaluations on the evaluator's side.

Recently, a novel optimization reduced the communication overhead further to $1.5\kappa+5$ bits [148] at the cost of a higher number of AES evaluations per gate (with at most six and three AES evaluations on the garbler's and evaluator's side, respectively). Another recent trend is extending the concept of GCs to enable features like conditional branching [68], vector gates [69], and oblivious array operations [70].

2.5 Additive Secret Sharing

Secret sharing is an approach for distributing a secret value via two or more "shares" that separately do not reveal any information about the secret. The secret value can only be reconstructed if all (or, depending on the specific scheme, a sufficient number of) shares are combined. With additive secret sharing (A-SS), to share a secret number x among two parties, one selects random x_0, x_1 such that $x = x_0 + x_1$ (typically, in a given ring). One party gets x_0, the other gets x_1. Neither of the parties alone can find out anything about the value of x.

It is possible to perform arithmetic operations on secret-shared numbers, obtaining secret-shared outputs. Linear arithmetic operations, such as adding two secret-shared numbers, increasing a secret-shared number by a publicly known constant, or multiplying a secret-shared number by a publicly known constant, can be performed by simple local computations of the parties, i.e., without communication between them. For example, assume that x is secret-shared as above and c is a publicly known constant. Multiplying x by c can be performed by having both parties multiply their share of x by c; the result is a secret-sharing of $c \cdot x$, since $c \cdot x_0 + c \cdot x_1 = c \cdot (x_0 + x_1) = c \cdot x$.

Multiplying two secret-shared numbers is also possible, although significantly more complicated and also requires communication between the parties. Multiplication can be eased with Beaver multiplication triplets (MTs) [9]. MTs consist of values a, b, c such that $c = a \cdot b$. If the parties have access to shares of pre-computed MTs, then they can multiply secret-shared numbers with low computation and communication complexity. It is important to note that a MT can be used only once; otherwise, it would lead to data leakage.

Since operations on secret-shared numbers produce secret-shared numbers, such operations can be composed (i.e., executed one after the other) to yield more complex functions. This way, any function given as an arithmetic circuit (i.e., a circuit composed of addition and multiplication gates) can be evaluated on secret-shared numbers. Thus, A-SS combined with Beaver triplets leads to a solution for the 2PC problem, assuming the function to compute is given as an arithmetic circuit:

- The secret inputs of the parties are secret-shared between them.
- The arithmetic circuit is evaluated, gate by gate, using secret-shared numbers.
- The output is reconstructed from the final shares.

It is important that reconstruction only happens at the end; in all previous steps, parties work only with their own shares, so as not to reveal anything about the secret inputs.

An important special case of A-SS is *Boolean secret sharing*, where individual bits are shared between the parties, and all operations are performed modulo 2. A secret bit x is shared as

(x_0, x_1), where x_0 and x_1 are bits such that $x = x_0 \oplus x_1$, and $\oplus$ denotes XOR. On secret-shared bits, we can again perform operations: Addition becomes XOR and multiplication becomes AND. This approach can be used for computing Boolean circuits over private inputs, which is known as the MPC protocol of Goldreich, Micali, and Wigderson (GMW) [62]. An advantage of Boolean sharing is that AND gates can be evaluated relatively efficiently by means of OT, without requiring Beaver triplets.

In the following, we summarize the flow of securely computing a function using additive or Boolean secret sharing between two parties P_0, P_1 who have respective private inputs x and y. First, both parties generate shares of their input: (x_0, x_1) and (y_0, y_1). Then, P_0 sends x_1 to P_1 and P_1 sends y_0 to P_0. Computing linear gates (XOR or addition gates) can be done without interaction between the parties, similarly to GCs with FreeXOR [92]. For this, the parties both locally apply the respective operation on the shares they hold. For non-linear gates (AND or multiplication gates), interaction between the parties is necessary. The parties use Beaver triplets or OTs for this purpose. Finally, after all gates of the circuit are evaluated, the shares of the output wires can be reconstructed to obtain the overall computation result.

Shares of Beaver triplets can be pre-computed, for example, using random OTs [5, 48, 87] or HE [35, 88, 121, 139]. Recent pre-computation techniques also include *silent* protocols with a communication complexity that is sub-linear in the circuit size [24, 25]. As a result of pre-computation, during the online phase of the protocol, the communication per non-linear gate is $2l$ bits per party, where l is the bitlength of the shares. With a recent function-dependent preprocessing technique, it is possible to reduce this overhead to l bits [125].

Note that in contrast to GCs, this protocol requires one round of interaction per non-linear layer of the circuit. Thus, it might not be well suited for high-latency network settings and circuits should be optimized to have a low multiplicative depth.

3 CHARACTERISTICS OF SECURE NEURAL NETWORK INFERENCE APPROACHES

Before going into *how* SNNI approaches work (cf. Section 4), this section investigates *what* such approaches assume and what they guarantee. Table 1 gives an overview about the most important characteristics of some selected approaches. Details are provided in the following subsections.

3.1 Number and Roles of Participants

Most works on SNNI consider two participants (cf. Figure 1). In the beginning, the client has the input, and the server has the neural network. By the end of the process, the client obtains the output of the NN on the given input. Thus, SNNI is regarded as a two-party computation problem.

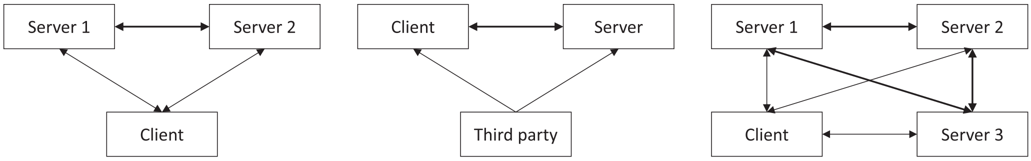
Beyond this standard model, which was adopted by the majority of existing works, some papers used more than two participants or other types of participants. For example, SecureML [116] assumes the participation of two servers (cf. Figure 7(a)). Both servers are untrusted, but it is assumed that they do not collude. In other words, an adversary may compromise one of the servers, but not both. For example, the two servers could be run by two competing cloud providers. The client sends shares of its input to the two servers, and the two servers engage in a protocol to compute the result, the shares of which are then sent to the client.

Chameleon [144] assumes, beyond the client and the server, a third party (cf. Figure 7(b)). This third party does not take part in the actual inference on the client's input. However, in a preceding phase, the third party sends information to the other parties that helps them perform the SNNI process efficiently, such as correlated randomness for OT or pre-computed Beaver triplets. Here again it is crucial to assume that no collusion between the third party and one of the computing parties occurs. For additional assurance, this third party may be implemented using trusted hardware tokens or execution environments.

Table 1. Characteristics of Selected SNNI Approaches

Approach	Parties (Section 3.1)			Attack (Section 3.2)		Secret (Section 3.2)				NN limitations (Section 3.3)	Link to training (Section 3.4)	Interactive (Section 3.5)
	Client	Server	Other	HbC	Malicious	Input	Result	Weights	Architecture			
Chameleon [144]	1	1	1	✓		✓	✓	✓				Yes
COINN [77]	1	1		✓		✓	✓	✓			Pre- & postprocessing	Yes
CrypTFlow [97]	0	3		✓	✓	✓	✓	✓				Yes
CrypTFlow2 [138]	1	1		✓		✓	✓	✓				Yes
CryptoNets [60]	1	1		✓		✓	✓	✓	✓	Polynomials, limited depth	Postprocessing	No
DeepSecure [149]	1	1		✓		✓	✓	✓			Pre- & postprocessing	No
Delphi [113]	1	1		✓		✓	✓	✓			ReLU modification	Yes
Falcon [160]	0	3		✓	✓	✓	✓	✓				Yes
Gazelle [83]	1	1		✓		✓	✓	✓				Yes
MiniONN [104]	1	1		✓		✓	✓	✓				Yes
SecureML [116]	0	2		✓		✓	✓	✓				Yes
SIMC [37]	1	1		✓	✓	✓	✓	✓				Yes
SiRnn [137]	1	1		✓		✓	✓	✓				Yes
XONN [143]	1	1		✓		✓	✓	✓	✓	Binary weights	Pre- & postprocessing	No

A "0" in the "Client" column means that the client does not take part in the protocol after providing the input. HbC: honest but curious.



(a) Client and two servers (b) Client, server, and third party (c) Client and three servers

Fig. 7. Some possible setups of the participants. Thick arrows show the main flow of the protocol, and thin arrows show the flow of additional information.

Falcon [160] takes the idea of SecureML further and uses three instead of two servers (cf. Figure 7(c)). The assumption is that the servers do not collude and at most one of them may be compromised. The additional server helps improve both the efficiency and the security guarantees of the protocol. However, the additional server also requires additional assumptions, making the practical application of the scheme more challenging. Further works in the three-party setting include ABY^3 [115], ASTRA [39], BLAZE [126], SecureNN [159], and SWIFT [93].

Some works like FLASH [31], Trident [40], and Tetrad [94] take it even further by operating in a four-party setting. They claim that due to the efficiency gains on protocol level the total monetary cost for operating four servers is lower than for three-party approaches like ABY^3 [115].

3.2 Security Properties

3.2.1 Adversary Model. Most works on SNNI adopt the *honest-but-curious* (abbreviated as HbC, and also called semi-honest or passive) adversary model. This means that participants follow the protocol but try to find out more information than what they are entitled to find out. In contrast, the *malicious* adversary model means that participants can deviate arbitrarily from the protocol to gain some advantage (e.g., to learn some secret or to deceive other participants). Obviously, protecting against malicious adversaries is more difficult than protecting against honest-but-curious adversaries. The HbC adversary model is generally used in research on secure MPC [72] and is dominant also in SNNI research. The security of SNNI approaches against HbC adversaries often follows from known results about the security of MPC protocols in this adversary model. Less work has been done in the malicious security model:

- The Aramis component of CrypTFlow provides a generic method for transforming an MPC protocol that is secure against HbC adversaries into one that is secure against malicious adversaries [97]. Aramis relies on special hardware with integrity guarantees (in particular, Intel SGX technology). The idea of the transformation is to furnish each protocol message with a signature, so that adherence to the protocol can be verified. This transformation introduces non-negligible overhead: Experimental results show that the time needed for performing an inference is roughly 3× as high with Aramis than without the transformation.
- ABY^3 provides protocols for SNNI in a three-party setting in both the HbC and the malicious adversary model [115]. The protocols for the malicious adversary model have higher complexity, but they were not implemented, so their practical performance was not evaluated.
- Falcon devises and also implements specific protocols for security against malicious adversaries [160]. Falcon uses three servers and assumes an honest majority, which allows it to discover malicious activities and abort the protocol in such a case. The experimental evaluation reveals that the protocols providing security against malicious adversaries are significantly less efficient than the ones that only provide security against HbC adversaries.
- Works like FLASH [31], SWIFT [93], Trident [40], and Tetrad [94] provide malicious security with at most one corruption and additionally achieve notions of fairness or robustness. Here, fairness means that all or none of the parties obtain the output of the computation, whereas robustness, or guaranteed output delivery, ensures that honest parties always receive the correct computation result.

Most approaches either assume the HbC model for both client and server or the stronger malicious model for both client and server. MUSE introduces a hybrid model, assuming an HbC server and a malicious client [102]. The rationale behind this model is that clients are more numerous, less regulated, and not so well protected as service providers, so that the chance of a malicious adversary acting as client or compromising a client is high. MUSE provides protocols that are faster than approaches protecting against both malicious clients and servers, although not as efficient as the approaches that only protect against HbC adversaries.

3.2.2 Secrecy Goals. Independently of the adversary model, SNNI approaches also differ in what information they keep secret. Depending on the application scenario, the secrecy of different types of information is important. Typical secrecy goals are as follows:

- The input to the inference remains the client's secret.
- The result of the inference is only learned by the client.
- The parameters of the neural network (such as weights and biases in fully connected layers) remain the server's secret.
- The architecture of the neural network (including the number, types, and sizes of layers) remains the server's secret.

Practically all proposed SNNI approaches aim for the first three points, i.e., the secrecy of the client's input, of the inference result, and of the NN's parameters. The fourth point is addressed only by some approaches. In particular, interactive approaches (cf. Section 3.5) assume that the architecture of the NN is known to all parties, thus disregarding the fourth of the above points.

3.2.3 Black-box Attacks. If the secrecy goals of Section 3.2.2 are satisfied, then the client does not learn anything about the NN *beyond what the result of the inference reveals*. The result of the

Table 2. NN Layer Types Supported by Selected SNNI Approaches

Approach	FC	Conv	Tanh	Sigmoid	ReLU	Square	Sign	MaxPool	AvgPool	SumPool	ArgMax	BN	$\frac{1}{\sqrt{x}}$
CrypTFlow2 [138]	✓	✓			✓			✓	✓		✓		
CryptoNets [60]	✓	✓				✓				✓			
Falcon [160]	✓	✓			✓			✓				✓	
Gazelle [83]	✓	✓			✓	✓		✓					
SiRnn [137]	✓	✓	✓	✓	✓			✓			✓	✓	✓
XONN [143]	✓	✓					✓	✓				✓	

FC: fully connected; BN: batch normalization.

inference may leak some information about the NN though. Using a sufficient number of inference queries, the client may be able to infer secret information [102, 143]. For example, the client can try to find out the parameters of the model (model extraction attack), determine prototypical samples with given labels (model inversion attack), or find out if a given input appeared in the training set (membership inference attack). This is an intrinsic property of the MLaaS model, independently of the specific techniques used to make the inference process secure. Potential solutions could include limiting the number of queries that a client can make or limiting the information that a client gets from an inference [143]. Most of the work on SNNI does not address this topic.

3.3 Supported Neural Networks

A generic technique that would support secure computation on all types of NNs in an efficient and effective way is not known. Therefore, most work on SNNI focuses on developing specific techniques that work well for a useful set of neural network types. Either the supported types of layers or the supported ranges of weights are limited.

3.3.1 Supported Types of Layers. Each type of NN layer requires specific types of computations, so each SNNI approach supports some types of layers (cf. Table 2). For example, many approaches target convolutional NNs. For this purpose, an approach should ideally support the following types of layers: fully connected, convolutional, activation functions like ReLU or sigmoid, max-pooling, and softmax or argmax. There are approaches that support all these layer types. For example, CrypTFlow2 supports fully connected layers, convolutional layers, ReLU, max-pooling, and argmax [138]. Other approaches only support a subset of these layer types, and potentially some approximations of the layer types not supported directly. For example, some approaches support the square function as activation function instead of ReLU or sigmoid and summing instead of max-pooling [60].

Some approaches also support other layer types. For example, Falcon supports batch normalization [160], while SiRnn supports reciprocal of square root, which is used in some recurrent neural networks (RNNs) [137].

3.3.2 Supported Ranges of Weights. There are also approaches that constrain the types of numbers that can be worked on in the NN. For example, some approaches are limited to *discretized* NNs, where weights are integers while inputs and outputs of neurons can only be from $\{1, -1\}$ [22], or *binary* NNs, where both the weights and activation values can only take the values $\{1, -1\}$ [143]. These limitations can be exploited for efficiently implementing the involved arithmetic operations but may threaten the accuracy of the NN.

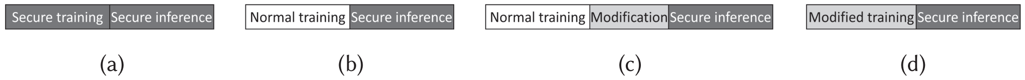


Fig. 8. Possibilities for the integration with training.

3.4 Connection to Training

There is a large variance in how SNNI approaches relate to training. On the one extreme, there are approaches that cover secure training and secure inference in the same framework (cf. Figure 8(a)), such as Falcon [160]. On the other extreme, several approaches are completely independent from the training process, starting from a pre-trained model and not modifying it (cf. Figure 8(b)), such as in the case of CryptFlow [97].

Several authors suggested to take a pre-trained model and transform it to make it more appropriate for SNNI (cf. Figure 8(c)). For example, CryptoNets simplifies the trained NN by combining subsequent linear layers into a single layer and removing monotonic activation functions like softmax in the output layer [60]. FHE-DiNN discretizes weights and exchanges activation functions in the trained NN to make it compatible with the proposed inference process [22]. DeepSecure uses a combination of transforming the input data space, pruning connections in the NN with weights of low absolute value, and fine-tuning the training of the NN to improve the efficiency of the subsequent secure inference process [149].

Some approaches require more significant changes to the training process (cf. Figure 8(d)). Delphi replaces some ReLU layers with square functions and uses special training techniques (gradient and activation clipping, gradual activation exchange) to efficiently train such NNs [113]. In XONN, the neurons in each layer are replicated to compensate the decrease in accuracy stemming from limiting the weights to $\{1, -1\}$ [143]. This is followed by the normal training process. Afterwards, neurons are successively pruned with a greedy procedure, as long as accuracy stays above a given limit, to increase efficiency.

While such optimizations can contribute to an increase in efficiency, they might hamper the applicability in existing machine learning pipelines.

3.5 Interactivity

Most existing SNNI approaches use one of two fundamentally different communication patterns:

- In *interactive* approaches, client and server communicate with each other anew after evaluating every layer of the neural network.
- In *non-interactive* approaches, communication between client and server only happens at the beginning and at the end of the protocol.

The *round complexity*, i.e., the number of communication rounds between client and server, is $O(n)$ for interactive and $O(1)$ for non-interactive approaches, where n is the number of layers.

These two classes of approaches have significantly different properties. On the one hand, interactive approaches are more flexible, since they can use different protocols for different types of layers. By using the most appropriate protocol for each layer, interactive approaches can achieve superior efficiency [83].

On the other hand, interactive approaches have two drawbacks: First, they leak information about the structure of the NN (number and types of layers) from the server to the client, in contrast to non-interactive approaches where the structure of the NN can be kept secret (cf. the last property in Section 3.2.2). Second, interactive approaches require the client's active involvement throughout the process, which may be problematic for example for mobile devices. Overall, whether an interactive or a non-interactive approach is better depends on the specific application context.

3.6 Offline Preprocessing

In a number of approaches, some parts of the protocol are independent of the specific input of the client. This makes it possible to split the protocol into two phases: an offline preprocessing phase that is independent of the client input, and the online phase that depends on the specific input. Approaches that use such a split include SecureML [116], MiniONN [104], Chameleon [144], Delphi [113], and Falcon [160]. Typical activities for the offline phase include the generation of Beaver multiplication triplets or other sets of correlated numbers for masking secrets (cf. Section 2.5).

Moving some parts of the protocol to an offline phase is beneficial in some situations. Such a situation could be if there is a long-standing relation among the parties, with inference requests arising only occasionally. The times in which no active inference is taking place can be used to perform offline preprocessing. The results of this preprocessing can then be used to speed up the execution of upcoming inference requests. In such situations, reducing the duration of the online phase is the primary goal.

In other situations, the split between offline and online phases is less useful. If new clients arrive to the system frequently together with their inference requests, then there is no idle time available for preprocessing. In such situations, the total duration of the computation should be minimized.

4 SOLUTION APPROACHES

After investigating the assumptions and characteristics of different SNNI approaches in Section 3, we now look at how these approaches actually work. In Sections 4.1, 4.2, and 4.3, we describe how homomorphic encryption, garbled circuits, and additive secret sharing can be used to realize SNNI, respectively. In Section 4.4, we describe approaches that combine different cryptographic primitives. In each of Sections 4.1–4.4, we first describe the basic idea of using the given technique for SNNI and then the progress made so far and future perspectives. Section 4.5 presents a high-level comparison of approaches, while Section 4.6 discusses aspects that are relevant for different types of solution approaches.

4.1 Homomorphic Encryption

4.1.1 Basic Idea. Implementing SNNI by means of HE is a natural idea, since HE allows one to perform computation over encrypted data (see Section 2.2). The client encrypts the input using an HE scheme and sends the resulting ciphertext to the server. The server performs inference on the encrypted input and sends the result in encrypted form back to the client, which decrypts the result. Since the NN is a black box for the client, and the server only sees the input data in encrypted form, this approach satisfies all security properties listed in Section 3.2.2.

Using HE for SNNI was already suggested in the earliest work on the topic [8, 120]. However, turning this idea into a practical method has proved quite challenging. In the following, we discuss some important aspects.

4.1.2 Progress so Far.

Handling linear layers. Evaluating linear layers (like fully connected or convolutional layers) of an NN requires multiplying matrices with vectors, which in turn requires multiplications and additions of numbers. Since these operations are supported by FHE (or leveled FHE) schemes, securely evaluating linear layers with HE is simple. It is interesting to note that, since the operations are carried out by the server, only the input vector must be encrypted, and the weight matrix can be used as plaintext. Thus, one operand of the multiplication is encrypted, the other can be in the plain, which may enable a faster homomorphic multiplication.

Handling non-linear functions. Computing non-linear activation and pooling functions such as ReLU, sigmoid, or max is challenging for homomorphic encryption schemes, since these functions cannot be computed by using additions and multiplications. One approach is to use a *polynomial approximation* of such functions, since polynomials can be evaluated by HE schemes that support both addition and multiplication. However, a close approximation typically requires a high-degree polynomial, and homomorphically evaluating high-degree polynomials is often expensive. Being "expensive" can mean different things in practice, depending on the HE scheme: an increase in computation time, a large increase of noise, or an increased message space size. Another approach is to use lookup tables to achieve a low-precision approximation of a more complex function (such as the activation function) [49]. We return to the issues of polynomial approximation in Section 4.6.6.

The problem can be (partially) circumvented by simplifying the activation function. For example, CryptoNets uses the square function as activation function and summing for pooling [60]. This approach was reported to deliver good results in a neural network withfive layers (from which two layers used the square function). However, the authors remarked that the square function can lead to problems in training, especially for deeper neural networks, because its derivative is unbounded. The square function is also used by the Low Latency Privacy Preserving Inference (LoLa) SNNI approach [29]. A different solution is proposed in FHE-DiNN [22]. Here, the sign function is used as non-linear activation function. The authors develop a method to compute the sign of an encrypted number and perform bootstrapping at the same time.

The problem is circumvented entirely in Reference [19] where Ivakhnenko's group method of data handling [81] is used (these inductive algorithms are also known as polynomial neural networks). This approach does not require any activation functions.

Since homomorphically evaluating non-linear layers is difficult, some authors suggested to use HE only for the linear layers of the network and use some other MPC technique for the non-linear layers [83]. We come back to such hybrid usage of privacy-preserving techniques in Section 4.4.

Homomorphic encryption schemes and frameworks. There are a number of different HE approaches and schemes. A survey on (fully) homomorphic encryption schemes can be found in Reference [1], and Reference [111] surveys the engineering aspects of FHE. Also partially homomorphic schemes, which can do either additions or multiplications on encrypted data, have been considered in the setting of NNs [8, 120], but they can only be used in a very limited way. Since the first concrete FHE scheme was found by Gentry [59], multiple newer generations of FHE schemes appeared, offering improved possibilities also for SNNI. Examples of the second generation of FHE schemes are BGV [27], BFV [26, 57], and yashe [20]. These have a slower noise growth and are more efficient compared to the first generation. The third generation improved the bootstrapping [28, 43, 55], while the fourth and latest generation of schemes such as CKKS [42] work especially well with applications that use floating-point arithmetic.

In the encryption scheme used by CryptoNets [60] (and also in some other HE schemes), ciphertexts are high-degree polynomials, in which each coefficient can carry some information. This makes it possible to pack the encryption of multiple plaintext messages into a single ciphertext. Performing an operation on the ciphertext translates to performing an operation on multiple plaintext messages. This way, Single-Instruction-Multiple-Data (SIMD) processing is possible, which has the potential to amortize overhead over a large number of concurrently handled inputs. Although the latency (i.e., the time for performing one secure inference) of CryptoNets is quite high, up to 4,096 inputs can be processed simultaneously, potentially leading to a high throughput. However, SIMD processing is only possible if a sufficient number of inputs have to be processed at the same time.

The LoLa framework [29] is based on the second generation FHE scheme BFV. Here, every plaintext message can be regarded as a vector of a given dimension n. Supported homomorphic operations are the coordinate-wise addition and multiplication of two vectors and the rotation of a vector. LoLa defines several different ways of representing a set of numbers in one or more vectors.

The different representations have different advantages and disadvantages, making them more or less appropriate for different aims within the SNNI process. For example, one of the representations is specifically developed to support convolutions. In addition, matrices can be composed of vectors in a column-major or row-major way. Depending on this and the used representation of numbers in vectors, there are several different ways of using matrix-vector multiplications, which may also have the side-effect of changing between different vector representations. With the appropriate use of the different vector and matrix representations in different steps of the process, LoLa achieves significant improvements over CryptoNets. In addition, while CryptoNets applies batching to a set of inputs, LoLa can batch items belonging to the *same* input to benefit from SIMD processing.

The Shift-accumulation-based LHE-enabled deep neural network (SHE) [106] framework is based on a different scheme: the Torus FHE scheme [43]. Under the hood, ciphertexts are expressed over the torus modulo one, which results in fast binary operations over encrypted bits. This makes it possible to implement the ReLU activation and max pooling by Boolean operations.

4.1.3 Future Perspectives. Despite the impressive progress in HE, SNNI based on HE alone remains challenging. The first problem is the already mentioned intrinsic incompatibility between HE and non-linear layers. Although some workarounds have been proposed, their applicability is limited. Finding a widely applicable solution to this problem will likely require fundamental innovation either in NNs (HE-friendly non-linear layers) or in HE (homomorphic evaluation of typical non-linear layers of NNs). The second problem relates to the overhead and cumbersome usage of state-of-the-art HE schemes. Bootstrapping is costly; thus, it is often better to use leveled FHE instead of proper FHE to avoid bootstrapping. State-of-the-art leveled FHE techniques have several parameters that have to be set on a case-by-case basis, with significant impact on computation and communication complexity. Finding the most appropriate HE technique and the most appropriate parameter configuration for that technique is a challenging task on its own [140].

4.2 Garbled Circuits

4.2.1 Basic Idea. If the bitlength of the involved numbers is fixed, then all operations in an NN can be realized by Boolean circuits. Thus, the whole NN can be encoded as a single, potentially huge, Boolean circuit. Yao's GC protocol can be used to evaluate this Boolean circuit in a secure way (cf. Section 2.4). The client's input to the protocol is the inference input, while the server's input is the set of weights and other parameters. For this to work, the structure of the NN must be known to both parties. Thus, the last secrecy goal from Section 3.2.2 is not met, but Yao's protocol ensures that the parties do not learn each other's inputs. This leads to a general solution of the SNNI problem but can be prohibitively inefficient.

4.2.2 Progress so Far. DeepSecure was the first practical SNNI approach based mainly on GCs [149]. It makes use of the "Free-XOR" optimization [92], a known technique to reduce the overhead of the GC protocol by making the cost of XOR gates negligible. DeepSecure uses an industrial hardware synthesis tool to synthesize Boolean circuits for typical building blocks of NNs with a minimum number of non-XOR gates. Since the synthesis tool optimizes for chip area, the desired optimization goal is achieved by setting the area of XOR gates to 0 and the area of non-XOR gates to 1. In addition, DeepSecure uses preprocessing techniques (cf. Section 3.4), which are independent of the GC protocol.

A different approach is followed by XONN [143]. The main idea is to use binary NNs, i.e., restricting the numbers in the NN (weights, biases, activation values) to $\{1, -1\}$. This restriction considerably simplifies the Boolean circuits that encode the operations in the NN. In particular, multiplications can be replaced by XNOR operations, which can be efficiently evaluated using the already-mentioned Free-XOR optimization. In contrast to DeepSecure, the Boolean circuits used in XONN for typical operations of binary NNs are manually crafted. The first layer of the NN requires special attention, because the inputs of the network are not assumed to be binary; thus, the first layer is the only one that needs to operate with non-binary numbers. XONN implements two distinct approaches for handling the first layer. The first approach is a dedicated Boolean circuit, which can be evaluated as part of the GC protocol together with the other layers. The second approach uses other techniques, namely secret sharing and oblivious transfers; the GC then only starts with the second layer.

The natural way of using GCs for SNNI entails that the client is the garbler and the server is the evaluator. GCs can also be used as part of mixed-protocol approaches to evaluate certain parts of an NN. In this case, it may be useful to reverse the roles. For example, Delphi uses GCs for evaluating non-linear layers, using the server as garbler and the client as evaluator [113].

4.2.3 Future Perspectives. Although GCs have the potential to support the secure evaluation of complete NNs, they do not seem to be competitive with other techniques (such as HE and A-SS) on linear layers. Unless some significant innovation changes this, GCs will likely remain limited to being a sub-protocol used for evaluating (some types of) non-linear layers in mixed-protocol approaches (cf. Section 4.4).

4.3 Additive Secret Sharing

4.3.1 Basic Idea. The techniques described so far (HE and GCs) allow a non-interactive evaluation of the NN on the server side (cf. Section 3.5). In contrast, the approaches based on A-SS that have been proposed so far are interactive: Each layer of the NN is evaluated using a separate interaction among the parties. The parties may be the client and the server, but their role may be symmetric after the initial secret-sharing step, and also more than two parties are possible (cf. Section 3.1).

Approaches based on A-SS typically maintain the following *invariant*. At the beginning of evaluating the ith layer, the parties hold additive shares of all involved numbers, including the input but also the weights or other parameters of the layer. At the end of evaluating the ith layer, the parties hold additive shares of the output of the layer, which will be used as input to the next layer. At the beginning of the inference process, to reach a state in which the invariant holds, secret shares of the inputs are created and distributed among the parties. At the end of the inference process, the shares of the output are sent to the client, which combines them to retrieve the output.

Evaluating *linear layers* requires addition and multiplication of secret-shared numbers. As described in Section 2.5, adding secret-shared numbers is easy, whereas multiplying them can be done if a Beaver multiplication triplet is available. Generating Beaver triplets efficiently is a non-trivial task and is discussed further below.

For evaluating *non-linear layers* efficiently, no general recipe is known. Rather, a large variance of different techniques has been proposed, as discussed below.

4.3.2 Progress so Far.

Generating Beaver triplets. Beaver triplets do not depend on the input data and can thus be generated in an offline preprocessing phase (cf. Section 3.6). If there are only two parties, then the generation of Beaver triplets is non-trivial, since it has to be ensured that each party only learns its own shares of the triplet elements. This requires a cryptographic protocol. There are two well-known protocols for this purpose: One of them uses homomorphic encryption, and the other uses oblivious transfers. For example, MiniONN uses the protocol based on HE [104]. SecureML offers both protocols as alternatives, because in some cases one is more efficient, and in other cases the other [116]. If there is a third party that does not collude with the first two parties, then this makes the generation of Beaver triplets much easier. In this case, the third party can generate the Beaver triplets locally and then send the appropriate shares to the first two parties, without the need for a cryptographic protocol. It should be noted that for this purpose, the third party does not need to participate in the online phase, thus it does not need to get access to the actual inference input or the NN model. This approach is used, for example, by Chameleon [144]. It should also be noted that the three-party setup offers also a different kind of additive secret sharing (called 2-out-of-3 secret sharing), with which multiplications can be performed without precomputed Beaver triplets, as is done for example in Falcon [160].

Evaluating non-linear layers. Several different methods have been suggested to evaluate specific types of non-linear layers in the framework of A-SS. One possibility is to use a polynomial approximation of the given non-linear function and use standard addition and multiplication protocols for secret-shared numbers to evaluate the polynomial. In particular, the square function was investigated in SecureML [116] and in Delphi [113].

Another option is to apply a general cryptographic protocol like GCs. For example, the ReLU activation function can be realized with a simple Boolean circuit, which can be evaluated using Yao's GC protocol, as was suggested in SecureML [116]. MiniONN also uses GCs for ReLU, max-pooling, and a piecewise linear approximation of sigmoid [104]. Note that these methods, in contrast to the approaches of Section 4.2, use GCs only for one specific function and not for the whole NN. The main approach is still A-SS. Hence, the used circuits first reconstruct the input values from their shares, perform the non-linear function, and then return shares of the output.

Instead of GCs, OT can also be used. For example, the protocols of CrypTFlow2 for ReLU, max-pooling, and argmax rely on new, OT-based protocols for basic operations like comparison and logical AND [138]. Compared to the methods using GCs mentioned above that follow a similar scheme, the methods based on OTs require more creative, proprietary ideas.

Using even more proprietary ideas, the techniques of CrypTFlow2 are further improved and extended in SiRnn [137]. In particular, SiRnn uses lookup tables and iterative approximation to compute the exponential function and the inverse function. Using these, the sigmoid activation function can be computed in the framework of A-SS. Also Falcon uses an iterative approximation to implement division; for implementing max-pooling, it uses binary search [160].

4.3.3 Future Perspectives. Similarly to HE, also A-SS is well suited for linear layers but ill-suited for non-linear layers. However, A-SS offers a good framework for composing different protocols for different types of layers. Thus, we see a trend of A-SS being more and more used as an overall framework, in which different protocols can be plugged in to evaluate different types of layers (cf. Section 4.4). The combination of A-SS with OT seems especially promising. Future research should focus on devising more general solution techniques for efficiently evaluating non-linear layers on secret-shared inputs, potentially also involving searching for different secret-sharing mechanisms and/or for different non-linear functions to be used in NNs.

4.4 Mixed-protocol Approaches

4.4.1 Basic Idea. The available techniques that can be used to implement SNNI (HE, GCs, A-SS) all have some disadvantages that limit their appropriateness for certain types of layers. Since these disadvantages relate to different types of layers, it makes sense to combine multiple techniques into a joint approach that uses for each layer of the NN the most appropriate technique. This recipe has led to highly efficient approaches for SNNI. However, such approaches are necessarily interactive, which also leads to some drawbacks (cf. Section 3.5).

Table 3. Comparison of Solution Approaches Surveyed in Sections 4.1–4.4

Solution	Comp. complexity	Comm. complexity	Operations	Batching	Const. rounds
HE (§ 4.1)	●○○○○	●●●●○	ADD, MUL	✓	✓
GC (§ 4.2)	●●●○○	●●●○○	AND, XOR	✗	✓
A-SS (§ 4.3)	●●●●○	●●○○○	AND, XOR / ADD, MUL	✗	✗
Mixed (§ 4.4)	●●●○○	●●●○○	AND, XOR, ADD, MUL	(✓)	✗

Batching for mixed-protocol approaches is only supported for HE-based layers.

4.4.2 Progress So Far. The general scheme for evaluating an NN with L layers is as follows:

(1) Initialization to prepare the input to the evaluation sub-protocol of the first layer.[5]
(2) For each layer $i = 1, \ldots, L - 1$:
  (a) Evaluate layer i.
  (b) Prepare the input to the evaluation of layer $i + 1$ from the outputs of layer i.
(3) Evaluate layer L.
(4) Create the final protocol output.

In many cases, A-SS is used as a framework. This means that the evaluation of each layer starts with the inputs to the evaluation of the layer being secret-shared by the parties and ends with the output of the layer secret-shared between them. At the beginning of the whole protocol, the shares of the inputs to the evaluation of the first layer are created. At the end of the protocol, the server sends its share of the output to the client, so that the client can reconstruct the output.

Different incarnations of this general recipe have been proposed, depending on which technique is used for which type of layer. For example, SecureML and MiniONN use A-SS for linear layers and GCs for non-linear layers [104, 116]. Chameleon uses A-SS for linear layers, the GMW protocol for ReLU activations, and GCs for argmax [144]. Gazelle uses HE for linear layers and GCs for non-linear layers [83]. CrypTFlow2 uses either HE or OTs for linear layers and A-SS with proprietary sub-protocols for non-linear layers [138]. Delphi uses A-SS for linear layers and for square activations and GCs for ReLU activations [113].

It should be noted that all these combinations are within the online phase. Independently of this, the offline phase may use further protocols (cf. Sections 3.6 and 4.3.2).

4.4.3 Future Perspectives. Current research focuses on devising more and more efficient combinations of techniques in mixed-protocol approaches, which is a tedious process, mostly based on intuition. In the future, we expect to see research toward more systematic approaches. On the one hand, standardized interfaces are needed, so that different protocols can be easily plugged together. On the other hand, automation could be exploited to find the best set of protocols for a given configuration.

4.5 Qualitative Comparison

Table 3 gives a high-level qualitative comparison of the approaches surveyed in Sections 4.1–4.4. The table gives an indication of the typical characteristics of these approaches, in terms of computing and communication complexity, supported operations, possibilities for batch processing of multiple inputs, and whether the protocol finishes in a constant number of communication rounds. It should be noted that there is considerable variability within each family of techniques, so that deviations from this table are possible. More details on comparing different approaches is given in Section 6.

[5] The input to the "evaluation sub-protocol" of a layer may include—besides the data fed into the layer as input—also the weights or other parameters of the given layer.

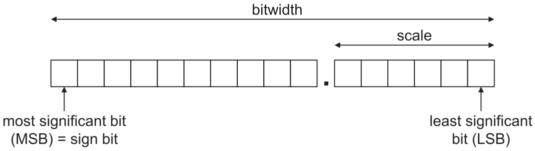


Fig. 9. Encoding a real number on a finite number of bits.

4.6 Other Aspects

Here, we review some further aspects that are important for different types of SNNI approaches.

4.6.1 Use of Oblivious Transfer. In contrast to HE, GCs, and A-SS, OT is typically not used as the main method of the proposed approaches. However, OTs are used as an important building block in many approaches and in several different roles:

- Yao's GC protocol makes use of OT in the secure transfer of the encryption of the evaluator's input (cf. Section 2.4). Thus, all approaches that use GCs implicitly also use OT.
- OT offers a possible way for generating Beaver multiplication triplets (cf. Sections 2.5 and 4.3.2). Thus, some of the approaches using A-SS, such as SecureML [116], also use OT for the generation of Beaver triplets.
- OTs are used in varied ways in proprietary sub-protocols for evaluating non-linear layers on secret-shared numbers. For example, the sub-protocols of CrypTFlow2 for ReLU, max-pooling, and argmax are all based on (different types of) OT [138].

4.6.2 Encoding Numbers. In theory, the inputs and outputs, as well as the weights and other parameters of an NN may be any real numbers. In a computer implementation though, all these numbers are represented using a finite number of bits. Choosing a number representation involves several decisions: type of representation (floating-point, fix-point, integer), signed/unsigned, bitwidth (i.e., total number of bits, see also Figure 9), and scale (also called precision, the number of bits for the fractional part). These decisions may have significant impact, resulting in different tradeoffs between efficiency and the achievable accuracy. In particular, reducing bitwidth may significantly improve efficiency, depending on the used techniques [36]. For example, if GCs are used, then the size of the circuit may decrease if the bitwidth is reduced, thus leading to less computation and less communication. However, a reduced bitwidth may limit the achievable accuracy.

Computers often have native support for operations on numbers of given bitwidth. Exploiting such a hardware-supported bitwidth may make the practical implementation simpler and more efficient. For example, Delphi and Falcon use a fixed bitwidth of 32 [113, 160], while SecureML and SecureNN use a fixed bitwidth of 64 [116, 159]. However, the bitwidth does not have to be uniform throughout the NN. For example, SiRnn introduces protocols for securely increasing or decreasing the bitwidth, making it possible to dynamically adapt the bitwidth [137]. Operations may change the bitwidth. For example, adding two n-bit numbers leads to an $n + 1$-bit number, while multiplying them leads to $2n$ bits. If the bitwidth is to be kept constant, then a truncation (i.e., division by a power of 2) is necessary after operations like addition or multiplication [116, 160]. Truncation can be seen as an additional non-linear function to compute.

For the precision of the fractional part, different solutions have been proposed. Some approaches use fixed-point representation with a uniform scale. For example, Delphi uses 15 bits and Falcon uses 16 bits to represent the fractional part [113, 160]. Other approaches determine the precision dynamically. For example, CrypTFlow2 determines both the bitwidth and the scale by means of autotuning [138]. Some approaches, such as MiniONN, work with integers, represented on a fixed number of bits [104]. In this case, care is needed to ensure that there is no overflow.

For encoding signed numbers, typically two's complement is used. This has important implications. For example, evaluating ReLU entails determining whether a number is positive. In two's complement representation, this boils down to determining the most significant bit of the number (which is a non-trivial task if the number is secret-shared or encrypted).

In the case of HE, several encryption schemes are based on polynomials. Plaintext numbers are encoded as coefficients of polynomials. To make the best use of the message space, some approaches like CryptoNets pack multiple numbers into a single coefficient, or vice versa, use multiple coefficients to represent one number [60]. These transformations are made possible by the Chinese remainder theorem. An efficient encoding method for fixed-point numbers tailored for homomorphic function evaluation is given in Reference [18].

4.6.3 Parallelization. In today's computers, more and more parallel computing units are available, including multicore CPUs and GPUs [109]. NN inference offers good opportunities for exploiting parallel processing units, as the computations within one layer typically do not depend on each other and can thus be performed in parallel. However, the use of cryptographic protocols may make it more challenging to exploit parallelism.

Some authors engineer their approaches specifically to enhance parallelism. For example, Huang et al. define their operations for vectors and matrices instead of individual numbers to benefit from efficient vectorization [75]. As mentioned in Section 4.1.2, some HE schemes support SIMD processing, making it possible to perform the same operation on a batch of different inputs [60] or even batching numbers belonging to the same input [22, 29, 83].

The offline phase (cf. Section 3.6) may also offer several opportunities for parallelization. For example, SecureML vectorizes Beaver triple generation, i.e., generates Beaver triplets for masking the multiplication of vectors instead of separate numbers, leading to increased efficiency [116]. A similar approach is used also by MiniONN [104].

4.6.4 Correlated Randomness. To mask the transfer of secrets, several approaches make use of correlated random numbers, i.e., a set of random numbers owned by different parties such that the numbers fulfill some relation. Beaver triplets are an example of correlated random numbers. As another example, Falcon uses random shares of 0, i.e., random numbers owned by different parties that add up to 0 [160].

Securely generating such correlated random numbers is a non-trivial task. We have already discussed this for Beaver triplets (cf. Section 4.3.2), but the problem is more general. Typical solutions either require some cryptographic protocol, such as HE or OT, or an additional party that generates the correlated random numbers and then distributes them to the parties that need them.

The process of generating correlated random numbers can be streamlined by using pseudo-random generators (PRG) and pseudo-random function families. In Chameleon, where this idea was used probably for the first time in the context of SNNI, a semi-honest third party sends appropriate random seeds to the two parties, who can then generate correlated (pseudo-)random numbers by using a local PRG, without further communication [144]. In a three-party setup, Falcon uses pairwise shared random keys (one for each pair of parties) to generate different types of correlated random numbers [160].

4.6.5 Neural Architecture Search. As discussed in Section 3.4, some approaches to SNNI also make modifications to the architecture of the NN. Indeed, an architecture that performs well when used in the plain may not be optimal for SNNI.

Finding the best NN architecture is a tedious process, involving trying many architectures, training them and assessing the achieved accuracy. This process can be automated by a neural architecture search (NAS) algorithm. This idea can also be leveraged in connection with SNNI.

Delphi uses NAS for a specific purpose [113]: It replaces some ReLU activations in the NN with the square function, aiming at finding a good tradeoff between accuracy and inference efficiency. Which ReLUs to replace is determined by the NAS algorithm.

NASS goes a step further and puts NAS into the focus of their approach [13]. While searching for the best neural architecture for secure inference, NASS takes into account the overhead incurred by different cryptographic primitives. As part of the search process, NASS also automatically tunes the parameters of HE used for linear layers of the NN, which would be tedious to do manually. A similar approach is followed also by HEMET [107].

4.6.6 Polynomial Approximation. Both HE and A-SS are more appropriate for evaluating polynomial functions than non-polynomial functions. In the case of HE, this is because most HE schemes only support the homomorphic evaluation of addition and multiplication. In the case of A-SS, addition is easy, only involving local computations, and multiplication is possible using Beaver triplets, but other operations require different, sophisticated protocols.

For both HE and A-SS, a possible solution is to replace non-polynomial functions with polynomial approximations. In both cases, this leads to the problem that high-degree polynomials are costly, because multiplications are costly. As a result, using the square function has been proposed by multiple authors. However, with low-degree polynomials, a good approximation is hardly possible, especially in a large domain. In addition, the derivative of polynomials of degree at least 2 is non-bounded, which can cause problems in the training of networks with such activation functions [113].

One possible way of resolving this is by using piecewise polynomial approximation, such as in MiniONN [104] and in the approach of Huang et al. [75]. By splitting the domain of possible input numbers into several smaller intervals, better approximation can be achieved in each interval by a different function, even with linear functions.

Delphi proposed further ideas [113]: First, they only replace a subset of ReLUs with square functions. Second, they apply gradient and activation clipping during training to avoid unbounded gradients. Third, they use a smooth transition from ReLU to the square function during training.

4.6.7 Accelerating Convolutions. Convolutions can be represented as matrix multiplications, making it possible to handle convolutional layers the same way as fully connected layers. This is convenient from a theoretical point of view but may lead to an inefficient implementation. In particular, if Beaver triplets are used to implement matrix multiplication in the framework of A-SS, then this leads to a waste of Beaver triplets, because the same matrix elements are masked by multiple Beaver triplets. A more careful implementation of convolutions leads to a significant reduction of the number of required Beaver triplets, as is done in the Porthos component of CrypTFlow [97].

Another optimization is proposed in CrypTFlow2 [138]. Here, convolutions are implemented with HE. Through a smart use of SIMD, ciphertexts of the same offset in the convolution can be grouped and added, resulting in a decrease of the number of necessary rotation operations.

4.6.8 Security Analysis. The security of solutions based on MPC (in terms of input privacy) in most cases relies on the security of their building blocks and their composability. In particular, the security of OT and GC protocols is commonly established via simulation-based security proofs [103]. Here, for the semi-honest case, it is argued that the distribution of a party's view during real protocol executions is computationally indistinguishable from the distribution of artificial transcripts produced by a simulator who has only access to the party's input and output for the computation. If this indistinguishability is proven, then an adversary cannot learn any further information from the protocol execution where incoming protocol messages can be observed. In most SNNI solutions, the composition of the building blocks appears to be intuitively secure; however, most papers do not provide a rigorous proof, e.g., in the universal composability framework [33].

A-SS protocols are information-theoretically secure, meaning they do not rely on any computational hardness assumption and hence will not be outdated by potentially upcoming quantum computers or advances in cryptanalysis. While A-SS protocols in this aspect are superior to the aforementioned other MPC protocols, it is important to note that this does not hold for the OT- or HE-based pre-computation to generate multiplication triplets. Hence, in practice, the information-theoretic security property is currently not an argument to favor A-SS protocols, except that it is easier to exchange the pre-computation component if attacks arise.

Only few solutions design custom MPC protocols for particular components and thus are required to provide additional security proofs (e.g., References [39, 94]).

Many of the fully homomorphic encryption schemes are based on a variant of the *ring-LWE* problem. Work in this area began with Ajtai's seminal paper [3] (cf. the survey [127] for a comprehensive list of relevant references). Regev's work introduced the learning with errors (LWE) problem [141], which relates to solving a "noisy" linear system modulo a known integer. A specialized version of this is often used in practice: the ring-LWE problem [108, 128]. This additional algebraic structure offers significant storage and efficiency improvements. Concretely, ring-LWE relies on the (worst-case) hardness of problems in *ideal* lattices. Ideal lattices correspond to ideals in certain algebraic structures, such as polynomial rings. Due to the usage of this hard problem in post-quantum cryptographic schemes, where the announced winners that will be standardized [4] are based on the same problems and techniques as used in FHE; this has received a high level of scrutiny in the last years.

4.7 Compilers and Hardware Acceleration

For developers without expertise in cryptography, it can be challenging to utilize the solutions discussed so far. Therefore, researchers have developed compilers that either integrate with existing machine learning frameworks or automate the translation from high-level code to secure computation frameworks. Examples of the first category are COINN [77] and Rosetta [41]. With Rosetta, by importing a Python package and defining private inputs, regular TensorFlow code is securely evaluated in SecureNN [159]. ngraph-HE [17] and ngraph-HE2 [16] extend Intel's graph compiler ngraph with an HE backend that automatically takes care of HE-specific optimizations such as packing. MP2ML [15] supports both HE and MPC with automatic conversions. CHET [51] translates a domain-specific language (DSL) into optimized encrypted circuits for different HE schemes with optimal parameters. Also EzPC [38] proposes a DSL, which is automatically compiled for ABY [53] or EMP [161] using a heuristic to split between arithmetic and garbled circuits. Cerebro [169] compiles a DSL to MPC, also considering the deployment environment for planning an optimal execution of ML workloads. There exist compilers that translate high-level query languages [132], hardware description languages [52, 154], or programming languages [30, 71] for evaluation by secure computation frameworks, some of which have also been applied to ML tasks [30, 143].

CrypTen [89] is one of the few frameworks that also support GPU acceleration—which is standard for regular ML tasks but not yet in SNNI. It provides an API that closely resembles that of PyTorch while mapping the operations to hardware-accelerated arithmetic and binary versions of the GMW protocol (cf. Section 2.5). CryptGPU [155] is another GPU-accelerated framework that builds on CrypTen but implements a different MPC protocol with a different secret-sharing scheme.

Table 4. Implementation Aspects of Selected MPC- and HE-based SNNI Solutions; DSL Denotes a Custom Domain Specific Language, OS the Availability of an Open Source Implementation, MLI Whether the Solution is Integrated into an ML Framework, and SC the Utilized Secure Computation Techniques

Solution	Language		Secure Computation Libraries	OS	MLI	SC Technique			
	Front-End	Back-End				OT	GC	A-SS	HE
ABY2.0 [125]	C++		ENCRYPTO Utils [56]			✓	✓	✓	
ABY^3 [115]	C++		libOTe [130]	✓		✓	✓	✓	
Chameleon [144]	C/C++		ABY [53]			✓	✓	✓	
Cheetah [76]	C++		EMP [161], SEAL [112]	✓		✓		✓	✓
COINN [77]	Python	C++	EMP [161]	✓		✓	✓	✓	
CrypTFlow2 [138]	Python	C++	EMP [161], SEAL [112]	✓	✓	✓		✓	✓
CryptoNets [60], LoLa [29]	C#		SEAL [112]	✓					✓
Delphi [113]	Python	C++, Rust	fancy-garbling [58], SEAL [112]	✓		✓	✓	✓	✓
EzPC [38]	DSL	C++	ABY [53], EMP [161]	✓		✓	✓	✓	
Gazelle [83]	C++		JustGarble [11], libOTe [130], OpenFHE [6]	✓		✓	✓	✓	✓
MiniONN [104]	C++		ABY [53], SEAL [112]	✓		✓	✓	✓	✓
MP2ML [15]	Python	C++	ABY [53], SEAL [112]	✓	✓	✓	✓	✓	✓
Muse [102]	C++, Rust		JustGarble [11], MP-SPDZ [85], SEAL [112]	✓		✓	✓	✓	✓
ngraph-HE / HE2 [16, 17]	Python	C++	SEAL [112]	✓	✓				✓
SecureML [116]	C++		EMP [161]			✓	✓	✓	✓
SIMC [37]	C++		EMP [161], SEAL [112]	✓		✓	✓	✓	✓
XONN [143]	Python, DSL	C++	libOTe [130]		✓	✓	✓		

5 IMPLEMENTATION OF SNNI APPROACHES

In this section, we review how proposed SNNI solutions have been implemented and assess their usability. We summarize our findings in Table 4.

Most of the available implementations are research prototypes rather than production-ready software. They were created as a proof-of-concept and to run performance tests. Due to the limited budget of academic institutes for including software engineers in their research teams and lacking incentives to deliver high-quality code in addition to a publication, such prototype implementations usually have not undergone professional quality assurance processes. Several implementations are available as open source code, but they should not be applied on real confidential data. However, they may be useful as an inspiration for companies considering re-implementing a solution for production. There are a few (start-up) companies (e.g., Ciphermode, Duality, Enveil, and Opaque) that offer SNNI services; however, their (back-end) code is mostly not available for our review.

5.1 Used Technology

For the sake of efficient execution, secure computation protocols are commonly implemented as C/C++ code rather than in interpreted languages. This also applies to many SNNI solutions that build upon such secure computation techniques. There are a few exceptions that use Python due to the language's popularity for data analytics and integration with Python-based ML frameworks (which we discuss in the following section). While some solutions are purely implemented in Python (e.g., Reference [170]), it is more common to use Python only for the integration or front-end part and rely on a C/C++ implementation for the cryptographic back-end (e.g., References [15–17, 97, 113, 135, 138, 152]). Several SNNI solutions build on established OT, MPC, or HE libraries such as the libOTe OT library [130], the mixed MPC-protocol framework ABY [52], or the SEAL FHE library [112].

GPU acceleration is standard in ML training and inference; however, there are few successful attempts of GPU-accelerated secure computation [163]. Thus, few solutions support (partial) GPU acceleration for SNNI [113]. AES native instructions (AES-NI) are another type of hardware acceleration that is critical for MPC-based solutions. This is because garbled circuit and OT-based protocols make heavy use of this block cipher [11]. Support for AES-NI is provided by most Intel and AMD CPUs, as well as recent ARM chips as part of their cryptography extensions.

5.2 Ease of Use

We now discuss how easily existing SNNI implementations can be adapted to custom use cases and are usable for people without a background in cryptography. In terms of adaption, we see a wide range of approaches. In some implementations, the evaluation of specific NN architectures is hard-coded [76], while others have a more flexible domain-specific language [38], and some directly integrate with well-known ML frameworks such as TensorFlow or PyTorch (e.g., References [15–17, 97, 138, 143]). The latter category provides a cryptography back-end for a privacy-preserving execution, which allows to run existing code in a secure manner, ideally without modifications. Integration with ML frameworks certainly helps non-experts in using such solutions.

Another issue is the limited time that research groups maintain their code, which makes it hard to build and execute their solutions due to outdated dependencies, e.g., in terms of specific versions of system libraries that are only shipped with outdated operating systems. To counter this issue, the project of Hastings et al. [65] provides pre-built MPC frameworks as Docker containers. Unfortunately, such an initiative is, to our knowledge, not yet available for many SNNI solutions.

6 EVALUATION OF SOLUTION APPROACHES

This section discusses how SNNI approaches are experimentally evaluated in the literature. We describe the typical NNs and datasets used in the evaluation (Section 6.1), the technical setup of experiments (Section 6.2), the used metrics (Section 6.3), and discuss how different approaches are compared (Section 6.4). Finally, we discuss evaluation results for the surveyed SNNI solution approaches (Section 6.5).

6.1 Machine Learning Problems, NNs, and Datasets

Most works on SNNI are implemented and evaluated in the context of image classification problems. Mostly standard CNNs were used to evaluate solutions [60, 160]. CNNs are particularly well studied [2, 14], and many are available online, including pre-trained models that spare users a lengthy training process. Some of the benchmark CNNs used across studies are quite small, such as LeNet and AlexNet (composed of seven and eight layers, respectively). Larger CNNs used in newer studies are VGG16, VGG19, ResNet50, DenseNet121, and SqueezeNet [14].

Most of these CNNs are trained to perform two simple image classification tasks, using two customary datasets: MNIST [54] and CIFAR-10 [96]. Both datasets are composed of small low-resolution images falling into 10 categories (28×28 handwritten digits for MNIST and 32×32 animals and vehicles for CIFAR-10).

Some studies [16, 144, 160] also evaluate their solutions using CNNs that were developed by other work in SNNI, like DeepSecure [149], MiniONN [104], SecureNN [159], GAZELLE [83], or SecureML [116], all using the MNIST or CIFAR-10 classification task.

Other types of NNs are less commonly used in this field. Some works focused on RNNs, a type of NN that is able to evaluate input data based on previous inputs and thus is particularly suited for sequential data. The DeepSecure paper [149] proposed both CNNs and RNNs as novel benchmarks, with RNNs developed to classify audio and sensor data. Reference [104] additionally developed a Long-short-term-memory (LSTM) RNN trained on the Penn Treebank dataset [158] to predict likely next words given previous words. References [153] and [7] performed classification of six datasets of time series (e.g., EEG measurements and motion sensors). Reference [74] addressed RNNs with gated recurrent units, a sub-type of RNNs with greater internal memory but did not provide experimental results to evaluate their design.

Graph neural networks, able to classify data composed of nodes and information on their relations, have also been studied in the context of SNNI. References [142] and [82] addressed classification of nodes from two widespread citation network datasets, CORA (to classify 2,708 publications into seven classes) [32] and CITESEER [34] (to classify 3,312 scientific publications into six classes).

Recently, Transformers, capable of complex natural language processing (such as translating sentences or providing complex answers to questions), have also been considered for SNNI. Transformers typically have significantly higher parameter counts (often by several orders of magnitude) than standard CNNs, have variable-length input and output, and can handle multi-modal inputs. Reference [64] evaluates a novel HE-based protocol for SNNI applied to four well-known Transformers based on the BERT architecture [114] with four standard natural language processing tasks. Reference [162] analyzed the impact of several compression techniques on SNNI runtime for models that have Transformer-like characteristics (e.g., embedding tables, multi-headed attention matrix multiplication).

6.2 Technical Setup

Many authors performed experiments with laptops and on-premise servers, usually using machines with Intel Core i7 or Xeon E5 processors with around 3.5 GHz [44, 76, 78, 143, 144, 149]. In such setups, typically, the computations of the client are run on a laptop and those of the server are run on a local server. In the experiments in Reference [104], for example, both the server and the client run on an Intel Core i5 CPU with four cores, with the server having more RAM (16 GB versus 8 GB) and slightly higher clock frequency (3.3 GHz versus 3.2 GHz) than the client. Several of those experimental setups do not specify network characteristics like bandwidth and network latency.

Other authors used cloud-based virtual machines, often hosted in Amazon's AWS cloud. For example, SecureML was evaluated on two Amazon EC2 c4.8xlarge machines with up to 36 vCPUs and 60 GB of RAM each [116]. Cloud-based experiments are often done in two settings: WAN, using machines in different regions, and LAN, using machines in the same region. In the WAN setting, the bandwidth is around 9 to 70 MB/s with a ping time of around 60 ms, while the LAN setting offers a bandwidth up to 1 GB/s and a ping time of less than 1 ms [113, 159, 160].

6.3 Evaluation Metrics

The biggest concern in SNNI has been the overhead incurred by the used cryptographic techniques. Accordingly, metrics relating to efficiency play a major role in the evaluation of such approaches. The time needed to perform one inference (called *execution time* or *latency*) is often used as an evaluation metric. For approaches that use offline preprocessing (cf. Section 3.6), there have been variations whether only the time of the online phase is used or the total time (offline plus online).

Besides execution time, the amount of *communication*, i.e., the total number of bytes transferred between parties, is often considered. The amount of communication is important, because some cryptographic techniques may lead to several GBs of communication even for relatively small NN architectures and very small inputs (e.g., images of size 28 × 28 pixels for MNIST).

For approaches using batching (e.g., CryptoNets, cf. Section 4.1.2), the overhead (in terms of both time and communication) may be amortized over multiple inputs, and *average execution time* and *average communication* may be used as evaluation metrics. Also the *throughput*, i.e., the number of inferences per time unit, is a meaningful metric for capturing the effects of batching.

Accuracy is typically not directly influenced by the applied security techniques. However, approaches that constrain or modify the NN (cf. Sections 3.3 and 3.4) may have an effect on accuracy. Thus, accuracy is an important evaluation metric for such approaches.

Few papers attempt to evaluate security/privacy empirically. For example, Hou et al. use various methods for demonstrating that the client does not learn any useful information about the NN [73]. Osia et al. investigate to what extent the server may extract information about some sensitive attributes of the input from the information it gets from the client [122].

6.4 Comparison

Showing that a new approach improves the state of the art typically includes comparing it empirically against previous approaches. Such a comparison is complicated by multiple factors.

First, the comparison should take into account all the dimensions discussed in Section 3. For example, if approach A is faster than approach B, but approach A is less secure than B, then it cannot be clearly stated which approach is better. Second, some of the dimensions discussed in Section 3, such as security properties, are notoriously hard to measure or to quantify. Third, even deciding which of two approaches is faster may be difficult, because the answer may depend on several factors (e.g., the size of the NN or the types of layers in the NN). If only a small number of experiments is carried out, then there is a considerable risk that the resulting answer may not generalize to other situations.

Unfortunately, in many papers, evaluation is limited to efficiency and accuracy metrics, and experiments are performed on a small number of NNs and in just one or two technical setups. Comparison with previous work is often based on values published in previous papers, without running the different approaches in the same environment and comparing them directly.

6.5 Selected Results

As outlined in the previous subsections, making direct comparisons between the different SNNI solution approaches surveyed in Section 4 is a difficult endeavour, and it even warrants future work (cf. Section 7.4). Nevertheless, we want to give an impression of the performance that state-of-the-art approaches can achieve. Here, we focus on representative works for each category and approximate the order of magnitude in terms of runtime and communication.

For homomorphic encryption there is slow but steady progress. The work introducing the LoLa framework [29] presents performance figures for MNIST using a six-layer network (using the square function as an approximation for non-linear activation layers). Aiming for 99 % accuracy, LoLa can compute inference in 2 s while previous approaches such as CryptoNets [60] require 30 s on the same hardware. Computing inference for a single CIFAR-10 sample takes over 12 min using a 10-layer NN with 74 % accuracy [29]. Recent work [101] also applies the fourth-generation FHE scheme CKKS [42] to CIFAR-10. It is shown that using a relatively large machine with more than a hundred cores, one can reach 92.4 % accuracy; however, inference takes almost 3 hours.

There are not many works purely based on GCs (cf. Section 4.2). The most recent work in that category, XONN [143], requires roughly 5 s and 1.5 GB communication to classify one sample from the CIFAR-10 dataset with 80 % accuracy using a pruned 13-layer network. For small 3-layer networks that only contain fully connected but no convolution layers (yet are useful in some healthcare use cases), it is possible to classify samples in the order of 100 ms with less than 0.5 MB of communication.

For A-SS, performance is improving rapidly. One of the first protocols proposed was SecureML [116] in 2017. With a 3-layer CNN for MNIST it achieved more than 98 % accuracy, and inference time of about 5 s. In comparison, the 2022 two-party protocol AriaNN [151] was evaluated using a similar 3-layer CNN, also achieving more than 98 % accuracy but in less than 1 s in a similar setting. In terms of three-party A-SS protocols, Banners [78] in 2021 had an inference time of 6 s for a single CIFAR-10 image and required 1.5 GB communication to achieve 78 % accuracy with the same 13-layer NN used in Reference [143].

In the category of mixed-protocol approaches (cf. Section 4.4), Cheetah [76] is one of the most recent two-party approaches. Cheetah was evaluated on larger NNs than most previous works. Inference on the ResNet50 benchmark, a CNN with over 23 million trainable parameters, took about 80 s in LAN settings and 135 s in WAN settings and incurred about 2.3 GB of communication.

7 DISCUSSION AND INSIGHTS

The field of SNNI made huge progress in recent years. Nevertheless, we identified several limitations of the state of the art, which we categorized by theme in previous sections. Here, we discuss future research areas to address these limitations using the same structure as in Sections 3–6. We note that many limitations do not have an order of priority for future research; rather, we recommend that researchers consider each of these limitations, when applicable in their work, to jointly improve both methodologies and resulting solutions, thereby ultimately improving SNNI as a whole.

7.1 Characteristics of SNNI Approaches (cf. Section 3)

Security vs. efficiency tradeoff. A key problem in SNNI is how to achieve high efficiency and a high level of security simultaneously. The first solution approaches were inefficient. Much work since then aimed at making SNNI more efficient, for example, by introducing mixed-protocol approaches or adding a third party. However, these changes may lead to a decrease in security by leaking information about the NN architecture or by necessitating assumptions of non-collusion among parties. In addition, approaches that guarantee security against malicious adversaries only started to appear recently and typically incur higher overhead. An interesting future research direction is to realize a two-party approach that protects against malicious adversaries and guarantees all secrecy goals of Section 3.2.2 with minimal overhead.

Impact of the application context. Much research has been done to find the “best” SNNI approach. However, what the best approach is may depend on the specific context in which the approach is applied. For example, in one application, keeping the architecture of the NN secret may be essential, while it may be irrelevant in other applications. In one application, only the time needed for the online phase matters, while for another application, the differentiation between offline and online phase may not help. Understanding the impact of the practical security requirements of the application on the solution space and the interplay of such properties is an interesting area to gain better understanding.

SNNI in Internet of Things and edge computing. A particular application context in which SNNI could play an important role is the Internet of Things and the related edge computing paradigm [98, 110]. A challenge of this application context is that the clients are resource constrained. This limits the applicability of interactive approaches that put a significant burden on the client. New approaches that can offload the computationally intensive part of the protocol to the sever or provide the client with more computational resources (such as dedicated hardware solutions) are interesting new directions.

Secure-inference-friendly NN architecture and training. Some papers have shown examples for improving efficiency by considering not only the inference phase but also the choice of NN architecture and the training (cf. Sections 3.3 and 3.4). For example, different activation functions may require different protocols for secure inference, leading to faster or slower inference. Choices of the NN architecture, the allowed ranges for parameter values, and the way the network is trained and possibly post-processed may have significant impact on the performance of the inference phase. Considering the whole process from architecture selection until inference as an integrated optimization problem may lead to improved results, for example by selecting NN architectures that support more efficient secure inference. A systematic investigation of such possibilities is needed, especially using the recommendations for experimental frameworks that we put forward in Section 7.4.

Defense against black-box attacks. As pointed out in Section 3.2, most proposed SNNI approaches do not protect against black-box attacks, such as model extraction. Some potential measures to protect against such attacks, like limiting the number of queries a client can make, are orthogonal to the operation of SNNI. The secure inference process itself may be hardened against such specialized attacks [143]. Directions to protect against black-box attacks are interesting new research.

Post-quantum cryptography. Some protocols often used in SNNI, like A-SS or HE based on lattices (such as ring-LWE), have the intrinsic property to protect against adversaries with access to quantum computers. However, typical SNNI approaches also use other cryptographic building blocks that are not secure against quantum attacks. Developing fully quantum-secure SNNI approaches is an area for future research.

7.2 Solution Approaches (cf. Section 4)

New building blocks. Many of the solution frameworks surveyed in Section 4 are based on cryptographic techniques from the era of 2016–2019. As mentioned in Section 2, there are exciting new developments in secure computation building blocks, for example, "silent" OT and MPC protocols that introduce a new communication-computation tradeoff. Only some recent developments make use of such building blocks [76]. We suggest revisiting the solutions proposed beforehand to evaluate whether upgrading their building blocks could improve their performance. Such upgrades might change what protocols are most appropriate for given types of NN layers, so that the selection and assignment of building blocks to layer types might also have to be revisited.

Modular solutions. Most solutions proposed for SNNI so far operate with a single or a small set of secure computation approaches. Few offer exchanging some building blocks (e.g., generating multiplication triplets via either OT or HE [116]) or address two different levels of security [94]. This limits the flexibility and versatility of these solution approaches by making them appropriate for only a specific situation. In the future, more modular solutions could be developed that allow to scale the number of computing parties, support different adversary models and secrecy goals, work for a wide range of model architectures, and so on. This would enable users to stay flexible instead of being required to adopt a different solution when requirements change.

Scalability of solutions. Through the progress of recent years, current state-of-the-art SNNI approaches can cope with NNs with tens of millions of parameters, such as ResNet50. However, recent language models reach sizes with billions of parameters. This is way beyond the possibilities of current SNNI solutions in terms of memory, computation, and communication demand. In this respect, SNNI is lagging behind AI developments by several years. Hence, it needs more effort to find ways to scale SNNI techniques, for example by distributing the overhead of secure computation techniques among multiple nodes to be able to handle the workload with commodity hardware.

Integration with other ML techniques. In existing work, SNNI is often regarded as a standalone process. This limits the possibilities to benefit from improvements in the general field of machine learning. For example, NAS or model compression are techniques that are often used in the ML community to improve NN inference. Applying these techniques in connection with SNNI in the most appropriate way requires further research [36]. In particular, knowledge distillation, an approach to drastically reduce the number of layers and parameters in a NN, is gaining widespread traction in the field of AI research [63] but has not yet been investigated in the context of SNNI.

7.3 Implementation (cf. Section 5)

ML framework integration and unified API. As observed in Section 5, many of the approaches proposed so far are difficult to use, especially for users who are not cryptography experts. The highest level of usability is provided by those approaches that offer a cryptography back-end for popular ML frameworks such as TensorFlow or PyTorch. This allows running existing ML code almost without modifications in a secure manner. In order for more solutions to move into this direction, a *unified cryptography API* would be helpful. If such an API is accepted as a de-facto standard, then future SNNI solutions would have stronger incentives to also ensure conformance to the same API, allowing easy integration with other technologies conforming to this API. This would also contribute to improving the comparability between solutions, as an ML task could then be easily evaluated on different back-ends, making them directly comparable. However, designing such an API is challenging. On the one hand, the burden put on the user to use the API (e.g., annotations to mark private inputs) should be minimal. On the other hand, an over-simplified or not sufficiently flexible API could limit the potential for optimizations in the design of solution approaches.

GPU support. As described in Section 5.1, although regular ML training and inference are often accelerated using GPUs, SNNI approaches rarely make use of these accelerators and run almost exclusively on CPUs. Several SNNI approaches could also benefit from using GPUs. For example, evaluating linear layers using A-SS involves simple arithmetic operations over large matrices, which could be well parallelized using GPUs. However, finding the best way to utilize GPUs in SNNI—also taking into account the evaluation of non-linear layers, which may be less GPU-friendly—as well as designing new SNNI approaches from the start to make use of the GPU hardware are interesting new research directions.

Container builds. As pointed out in Section 5.2, available SNNI implementations are often hard to install, build, and execute. This is one of the reasons why new approaches are often compared to numbers reported in prior publications instead of executing the (open source) implementation in a benchmarking environment (cf. Section 6.4). A promising way to improve the situation is the approach of Reference [65] to package implementations in container formats such as Docker to make them readily available for experimentation. This makes it easier for other researchers and developers to reuse and further develop already proposed solutions. In addition, with the proposed unified API and ML framework integration, this helps in enabling fair comparisons between SNNI solutions (cf. Section 6.4). An automated and unified way to reproduce results is an interesting research direction.

7.4 Evaluation of Solution Approaches (cf. Section 6)

Improved comparisons. As described in Section 6.4, the current practice of comparing different approaches is limited. This should mature in the future, with authors performing more direct and fair empirical comparisons, running multiple competing solutions in the same system configuration. There is also a need for papers whose main goal is to perform unbiased large-scale empirical comparative studies to develop a richer and more reliable understanding of the pros and cons of the different SNNI approaches in various situations.

More varied benchmarks. As described in Section 6.1, evaluation is currently limited to a few benchmarks, typically using CNNs for image classification tasks. To obtain more meaningful results, experiments with a larger variety of ML tasks, NN architectures, and datasets are needed. Domains other than image classification should be explored, as well as NN types other than CNNs. For example, the application of SNNI to transformers is an interesting research direction.

Realistic technical setup. Section 6.2 described the typical technical setup used in the evaluation of SNNI approaches, which might be very different from a real-world setup. Experiments with resource-constrained clients, with servers serving many different clients, and so on, should also be performed.

Metrics beyond efficiency. As described in Section 6.3, evaluation is currently mostly focused on efficiency. However, the practicality of an approach also depends on many other aspects that should also be evaluated and quantified as much as possible. Metrics to be considered include energy consumption, memory demand, security, privacy, and usability. Only by considering an extensive mix of metrics can we obtain a realistic assessment of the practical applicability of SNNI approaches.

8 CONCLUSIONS

We gave an overview of recent developments in the field of secure neural network inference. Although the page limit did not allow us to present each relevant paper in detail, we reviewed the main characteristics of SNNI approaches, the main solution techniques, implementation issues, and evaluation practices. Through the intensive work of the last couple of years, SNNI has made large progress toward becoming practical. The most recent approaches make secure inference fast at least for small to medium-sized NNs with execution times in the order of seconds and communication overhead in the order of megabytes. Nevertheless, several challenges remain, for example, in terms of ease of use and integration into existing machine learning pipelines. Thus, we expect sustained further research interest in this field for the coming years.

REFERENCES

[1] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. 2018. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.* 51, 4 (2018), 79:1–79:35.

[2] Arohan Ajit, Koustav Acharya, and Abhishek Samanta. 2020. A review of convolutional neural networks. In *ic-ETITE*. IEEE, 1–5.

[3] Miklós Ajtai. 1996. Generating hard instances of lattice problems (extended abstract). In *STOC*. ACM, 99–108.

[4] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, et al. 2022. *Status Report on the Third Round of the NIST Post-quantum Cryptography Standardization Process*. Technical Report NISTIR 8413. National Institute of Standards and Technology.

[5] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2017. More efficient oblivious transfer extensions. *J. Cryptol.* 30, 3 (2017), 805–858.

[6] Ahmad Al Badawi, Jack Bates, Flávio Bergamaschi, et al. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915. (2022).

[7] Maya Bakshi and Mark Last. 2020. CryptoRNN - Privacy-preserving recurrent neural networks using homomorphic encryption. In *CSCML'20*. Springer, 245–253.

[8] Mauro Barni, Claudio Orlandi, and Alessandro Piva. 2006. A privacy-preserving protocol for neural-network-based computation. In *MM&Sec*. ACM, 146–151.

[9] Donald Beaver. 1991. Efficient multiparty protocols using circuit randomization. In *CRYPTO*. Springer, 420–432.

[10] Donald Beaver. 1995. Precomputing oblivious transfer. In *CRYPTO*. Springer, 97–109.

[11] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. 2013. Efficient garbling from a fixed-key blockcipher. In *SP*. IEEE Computer Society, 478–492.

[12] Mihir Bellare and Silvio Micali. 1989. Non-interactive oblivious transfer and applications. In *CRYPTO*. Springer, 547–557.

[13] Song Bian, Weiwen Jiang, Qing Lu, Yiyu Shi, and Takashi Sato. 2020. NASS: Optimizing secure inference via neural architecture search. In *ECAI (Frontiers in Artificial Intelligence and Applications)*, Vol. 325. IOS Press, 1746–1753.

[14] Simone Bianco, Rémi Cadène, Luigi Celona, and Paolo Napoletano. 2018. Benchmark analysis of representative deep neural network architectures. *IEEE Access* 6 (2018), 64270–64277.

[15] Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. 2020. MP2ML: A mixed-protocol machine learning framework for private inference. In *ARES*. ACM, 14:1–14:10.

[16] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE2: A High-throughput framework for neural network inference on encrypted data. In *WAHC@CCS*. ACM, 45–56.

[17] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE: A graph compiler for deep learning on homomorphically encrypted data. In *CF*. ACM, 3–13.

[18] Charlotte Bonte, Carl Bootland, Joppe W. Bos, Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. 2017. Faster homomorphic function evaluation using non-integral base encoding. In *CHES*. Springer, 579–600.

[19] Joppe W. Bos, Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. 2017. Privacy-friendly forecasting for the smart grid using homomorphic encryption and the group method of data handling. In *AFRICACRYPT*. 184–201.

[20] Joppe W. Bos, Kristin E. Lauter, Jake Loftus, and Michael Naehrig. 2013. Improved security for a ring-based fully homomorphic encryption scheme. In *IMACC*. Springer, 45–64.

[21] Amine Boulemtafes, Abdelouahid Derhab, and Yacine Challal. 2020. A review of privacy-preserving techniques for deep learning. *Neurocomputing* 384 (2020), 21–45.

[22] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2018. Fast homomorphic evaluation of deep discretized neural networks. In *CRYPTO*. Springer, 483–512.

[23] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. 2019. Efficient two-round OT extension and silent non-interactive secure computation. In *CCS*. ACM, 291–308.

[24] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. 2020. Efficient pseudorandom correlation generators from ring-LPN. In *CRYPTO*. 387–416.

[25] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2021. Sublinear GMW-style compiler for MPC with preprocessing. In *CRYPTO*. Springer, 457–485.

[26] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO*. Springer, 868–886.

[27] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS*. ACM, 309–325.

[28] Zvika Brakerski and Vinod Vaikuntanathan. 2014. Lattice-based FHE as secure as PKE. In *ITCS*. ACM, 1–12.

[29] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. 2019. Low latency privacy preserving inference. In *ICML (Proceedings of Machine Learning Research)*, Vol. 97. PMLR, 812–821.

[30] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. 2018. HyCC: Compilation of hybrid protocols for practical secure computation. In *CCS*. ACM, 847–861.

[31] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. 2020. FLASH: Fast and robust framework for privacy-preserving machine learning. *Proc. Priv. Enhancing Technol.* 2020, 2 (2020), 459–480.

[32] Cecile Cabanes, Antoine Grouazel, Karina von Schuckmann, et al. 2013. The CORA dataset: Validation and diagnostics of in-situ ocean temperature and salinity measurements. *Ocean Sci.* 9, 1 (2013), 1–18.

[33] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*. IEEE Computer Society, 136–145.

[34] Cornelia Caragea, Jian Wu, Alina Maria Ciobanu, Kyle Williams, Juan Pablo Fernández Ramírez, Hung-Hsuan Chen, Zhaohui Wu, and C. Lee Giles. 2014. CiteSeer x : A scholarly big dataset. In *ECIR*. Springer, 311–322.

[35] Dario Catalano, Mario Di Raimondo, Dario Fiore, and Irene Giacomelli. 2020. Mon$\mathbb{Z}_{2^k}$a: Fast maliciously secure two party computation on $\mathbb{Z}_{2^k}$. In *PKC*. 357–386.

[36] Daphnee Chabal, Dolly Sapra, and Zoltán Ádám Mann. 2023. On achieving privacy-preserving state-of-the-art edge intelligence. *PPAI*.

[37] Nishanth Chandran, Divya Gupta, Sai Lakshmi Bhavana Obbattu, and Akash Shah. 2022. SIMC: ML inference secure against malicious clients at semi-honest cost. In *USENIX Security Symposium*. USENIX Association, 1361–1378.

[38] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2019. EzPC: Programmable and efficient secure two-party computation for machine learning. In *EuroS&P*. IEEE, 496–511.

[39] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: High throughput 3pc over rings with application to secure prediction. In *CCSW@CCS*. ACM, 81–92.

[40] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2020. Trident: Efficient 4PC framework for privacy preserving machine learning. In *NDSS*. The Internet Society.

[41] Yuanfeng Chen, Gaofeng Huang, Junjie Shi, Xiang Xie, and Yilin Yan. 2020. Rosetta: A Privacy-Preserving Framework Based on TensorFlow. Retrieved from https://github.com/LatticeX-Foundation/Rosetta.

[42] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*. Springer, 409–437.

[43] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast fully homomorphic encryption over the torus. *J. Cryptol.* 33, 1 (2020), 34–91.

[44] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. 2018. Faster CryptoNets: Leveraging sparsity for real-world encrypted inference. *CoRR* abs/1811.09953 (2018).

[45] Tung Chou and Claudio Orlandi. 2015. The simplest protocol for oblivious transfer. In *LATINCRYPT*. 40–58.

[46] Ashish Choudhury and Arpita Patra. 2022. *Secure Multi-Party Computation Against Passive Adversaries*. Springer.

[47] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. 2021. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In *CRYPTO*. Springer, 502–534.

[48] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority. In *CRYPTO*. Springer, 769–798.

[49] Jack L. H. Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup. 2018. Doing real work with FHE: The case of logistic regression. In *WAHC@CCS*. ACM, 1–12.

[50] Ivan Damgård, Jesper Buus Nielsen, and Claudio Orlandi. 2008. Essentially optimal universally composable oblivious transfer. In *ICISC*. Springer, 318–335.

[51] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: An optimizing compiler for fully-homomorphic neural-network inferencing. In *PLDI*. ACM, 142–156.

[52] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. 2015. Automated synthesis of optimized circuits for secure computation. In *CCS*. ACM, 1504–1517.

[53] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*. The Internet Society.

[54] Li Deng. 2012. The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Sign. Process. Mag.* 29, 6 (2012), 141–142.

[55] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT*. Springer, 617–640.

[56] ENCRYPTO Group. 2021. ENCRYPTO Utils. Retrieved from https://github.com/encryptogroup/ENCRYPTO_utils/.

[57] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. IACR Cryptol. ePrint Arch., Paper 2012/144. (2012).

[58] Galois Inc. 2019. fancy-garbling. https://github.com/GaloisInc/fancy-garbling.

[59] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *STOC*. ACM, 169–178.

[60] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML (JMLR Workshop and Conference Proceedings)*, Vol. 48. JMLR.org, 201–210.

[61] Oded Goldreich. 2004. *The Foundations of Cryptography—Volume 2: Basic Applications*. Cambridge University Press.

[62] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*. ACM, 218–229.

[63] Jianping Gou, Baosheng Yu, Stephen J. Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *Int. J. Comput. Vis.* 129 (2021), 1789–1819.

[64] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. 2022. Iron: Private inference on transformers. *Adv. Neural Inf. Process. Syst.* 35 (2022), 15718–15731.

[65] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General purpose compilers for secure multi-party computation. In *SP*. IEEE, 1220–1237.

[66] Carmit Hazay and Yehuda Lindell. 2010. *Efficient Secure Two-Party Protocols—Techniques and Constructions*. Springer.

[67] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *ICCV*. IEEE Computer Society, 1026–1034.

[68] David Heath and Vladimir Kolesnikov. 2020. Stacked garbling—Garbled circuit proportional to longest execution path. In *CRYPTO*. Springer, 763–792.

[69] David Heath and Vladimir Kolesnikov. 2021. One hot garbling. In *CCS*. ACM, 574–593.

[70] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. 2022. EpiGRAM: Practical garbled RAM. In *EUROCRYPT*. Springer, 3–33.

[71] Tim Heldmann, Thomas Schneider, Oleksandr Tkachenko, Christian Weinert, and Hossein Yalame. 2021. LLVM-based circuit compilation for practical secure computation. In *ACNS*. Springer, 99–121.

[72] Johannes Heurix, Peter Zimmermann, Thomas Neubauer, and Stefan Fenz. 2015. A taxonomy for privacy enhancing technologies. *Comput. Secur.* 53 (2015), 1–17.

[73] Jiahui Hou, Huiqi Liu, Yunxin Liu, Yu Wang, Peng-Jun Wan, and Xiang-Yang Li. 2022. Model protection: Real-time privacy-preserving inference service for model privacy at the edge. *IEEE Trans. Depend. Secur. Comput.* 19, 6 (2022), 4270–4284.

[74] Shou-Ching Hsiao, Zi-Yuan Liu, Raylin Tso, Da-Yu Kao, and Chien-Ming Chen. 2020. PrivGRU: Privacy-preserving GRU inference using additive secret sharing. *J. Intell. Fuzzy Syst.* 38, 5 (2020), 5627–5638.

[75] Kai Huang, Ximeng Liu, Shaojing Fu, Deke Guo, and Ming Xu. 2021. A lightweight privacy-preserving CNN feature extraction framework for mobile sensing. *IEEE Trans. Depend. Secur. Comput.* 18, 3 (2021), 1441–1455.

[76] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and fast secure two-party deep neural network inference. In *USENIX Security Symposium*. USENIX Association, 809–826.

[77] Siam Umar Hussain, Mojan Javaheripi, Mohammad Samragh, and Farinaz Koushanfar. 2021. COINN: Crypto/ML codesign for oblivious inference via neural networks. In *CCS*. ACM, 3266–3281.

[78] Alberto Ibarrondo, Hervé Chabanne, and Melek Önen. 2021. Banners: Binarized neural networks with replicated secret sharing. In *IH&MMSec*. ACM, 63–74.

[79] Russell Impagliazzo and Steven Rudich. 1989. Limits on the provable consequences of one-way permutations. In *STOC*. ACM, 44–61.

[80] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending oblivious transfers efficiently. In *CRYPTO*. Springer, 145–161.

[81] A. G. Ivakhnenko. 1970. Heuristic self-organization in problems of engineering cybernetics. *Automatica* 6, 2 (1970), 207–219.

[82] Yixin Jie, Yixuan Ren, Qingtao Wang, Yankai Xie, Chi Zhang, Lingbo Wei, and Jianqing Liu. 2022. Multi-party secure computation with intel sgx for graph neural networks. In *ICC*. IEEE, 528–533.

[83] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha P. Chandrakasan. 2018. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security Symposium*. USENIX Association, 1651–1669.

[84] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography, Second Edition*. CRC Press.

[85] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*. ACM, 1575–1590.

[86] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2015. Actively secure OT extension with optimal overhead. In *CRYPTO*. Springer, 724–741.

[87] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS*. ACM, 830–842.

[88] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2018. Overdrive: Making SPDZ great again. In *EUROCRYPT*. Springer, 158–189.

[89] Brian Knott, Shobha Venkataraman, Awni Y. Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. CrypTen: Secure multi-party computation meets machine learning. In *NeurIPS*. 4961–4973.

[90] Vladimir Kolesnikov and Ranjit Kumaresan. 2013. Improved OT extension for transferring short secrets. In *CRYPTO*. Springer, 54–70.

[91] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. 2014. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In *CRYPTO*. Springer, 440–457.

[92] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved garbled circuit: Free XOR gates and applications. In *ICALP*. Springer, 486–498.

[93] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and robust privacy-preserving machine learning. In *USENIX Security Symposium*. USENIX Association, 2651–2668.

[94] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2022. Tetrad: Actively secure 4PC for secure training and inference. In *NDSS*. The Internet Society.

[95] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. 2012. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium*. USENIX Association, 285–300.

[96] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Master's thesis. University of Toronto.

[97] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow: Secure TensorFlow inference. In *SP*. IEEE, 336–353.

[98] Clemens Lachner, Zoltán Ádám Mann, and Schahram Dustdar. 2021. Towards understanding the adaptation space of AI-assisted data protection for video analytics at the edge. In *ICDCS Workshops*. IEEE, 7–12.

[99] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[100] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[101] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2022. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access* 10 (2022), 30039–30054.

[102] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. 2021. Muse: Secure inference resilient to malicious clients. In *USENIX Security Symposium*. USENIX Association, 2201–2218.

[103] Yehuda Lindell. 2017. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*. Springer International Publishing, 277–346.

[104] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious neural network predictions via MiniONN transformations. In *CCS*. ACM, 619–631.

[105] Qiang Liu, Pan Li, Wentao Zhao, Wei Cai, Shui Yu, and Victor C. M. Leung. 2018. A survey on security threats and defensive techniques of machine learning: A data driven view. *IEEE Access* 6 (2018), 12103–12117.

[106] Qian Lou and Lei Jiang. 2019. SHE: A fast and accurate deep neural network for encrypted data. In *NeurIPS*. 10035–10043.

[107] Qian Lou and Lei Jiang. 2021. HEMET: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture. In *ICML (Proceedings of Machine Learning Research)*, Vol. 139. PMLR, 7102–7110.

[108] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On ideal lattices and learning with errors over rings. In *EUROCRYPT*. Springer, 1–23.

[109] Zoltán Ádám Mann. 2011. GPGPU: Hardware/Software co-design for the masses. *Comput. Inf.* 30, 6 (2011), 1247–1257.

[110] Zoltán Ádám Mann. 2021. Security- and privacy-aware IOT application placement and user assignment. In *Computer Security–ESORICS 2021 International Workshops*. Springer, 296–316.

[111] Paulo Martins, Leonel Sousa, and Artur Mariano. 2018. A survey on fully homomorphic encryption: An engineering perspective. *ACM Comput. Surv.* 50, 6 (2018), 83:1–83:33.

[112] Microsoft Research. ([n.d.]). Microsoft SEAL. https://github.com/Microsoft/SEAL.

[113] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference service for neural networks. In *USENIX Security Symposium*. USENIX Association, 2505–2522.

[114] Athar Hussein Mohammed and Ali H. Ali. 2021. Survey of BERT Bidirectional Encoder Representation Transformer types. In *Journal of Physics: Conference Series*, Vol. 1963. IOP Publishing, 012173.

[115] Payman Mohassel and Peter Rindal. 2018. ABY^3: A mixed protocol framework for machine learning. In *CCS*. ACM, 35–52.

[116] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A system for scalable privacy-preserving machine learning. In *SP*. IEEE Computer Society, 19–38.

[117] Moni Naor and Benny Pinkas. 2001. Efficient oblivious transfer protocols. In *SODA*. ACM/SIAM, 448–457.

[118] Moni Naor, Benny Pinkas, and Reuban Sumner. 1999. Privacy preserving auctions and mechanism design. In *EC*. ACM, 129–139.

[119] Usman Naseem, Imran Razzak, Shah Khalid Khan, and Mukesh Prasad. 2021. A comprehensive survey on word representation models: From classical to state-of-the-art word representation language models. *ACM Trans. Asian Low Resour. Lang. Inf. Process.* 20, 5 (2021), 74:1–74:35.

[120] Claudio Orlandi, Alessandro Piva, and Mauro Barni. 2007. Oblivious neural network computing via homomorphic encryption. *EURASIP J. Inf. Secur.* 2007 (2007).

[121] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. 2020. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In *CT-RSA*. Springer, 254–283.

[122] Seyed Ali Osia, Ali Shahin Shamsabadi, Sina Sajadmanesh, Ali Taheri, Kleomenis Katevas, Hamid R. Rabiee, Nicholas D. Lane, and Hamed Haddadi. 2020. A hybrid deep learning architecture for privacy-preserving mobile analytics. *IEEE IoT J.* 7, 5 (2020), 4505–4518.

[123] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*. Springer, 223–238.

[124] Nicolas Papernot, Patrick D. McDaniel, Arunesh Sinha, and Michael P. Wellman. 2018. SoK: Security and privacy in machine learning. In *EuroS&P*. IEEE, 399–414.

[125] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved mixed-protocol secure two-party computation. In *USENIX Security Symposium*. USENIX Association, 2165–2182.

[126] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing fast privacy-preserving machine learning. In *NDSS*. The Internet Society.

[127] Chris Peikert. 2016. A decade of lattice cryptography. *Found. Trends Theor. Comput. Sci.* 10, 4 (2016), 283–424.

[128] Chris Peikert, Oded Regev, and Noah Stephens-Davidowitz. 2017. Pseudorandomness of ring-LWE for any ring and modulus. In *STOC*. ACM, 461–473.

[129] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. 2008. A framework for efficient and composable oblivious transfer. In *CRYPTO*. Springer, 554–571.

[130] Lance Roy Peter Rindal. libOTe: An efficient, portable, and easy to use Oblivious Transfer Library. Retrieved from https://github.com/osu-crypto/libOTe.

[131] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. 2009. Secure two-party computation is practical. In *ASIACRYPT*. Springer, 250–267.

[132] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. 2021. Senate: A maliciously-secure MPC platform for collaborative analytics. In *USENIX Security Symposium*. USENIX Association, 2129–2146.

[133] John M. Pollard. 1971. The fast Fourier transform in a finite field. *Math. Comp.* 25, 114 (1971), 365–374.

[134] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria E. Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. 2019. A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv.* 51, 5 (2019), 92:1–92:36.

[135] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkel, Pramod Bhatotia, and Christof Fetzer. 2020. SecureTF: A secure TensorFlow framework. In *Middleware*. ACM, 44–59.

[136] Michael O. Rabin. 1981. How to exchange secrets with oblivious transfer. Technical Report TR-81, Aiken Computation Lab, Harvard University (1981).

[137] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. SiRnn: A math library for secure RNN inference. In *SP*. IEEE, 1003–1020.

[138] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-party secure inference. In *CCS*. ACM, 325–342.

[139] Deevashwer Rathee, Thomas Schneider, and K. K. Shukla. 2019. Improved multiplication triple generation over rings via RLWE-based AHE. In *CANS*. Springer, 347–359.

[140] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent T. Lee, Hsien-Hsin S. Lee, Gu-Yeon Wei, and David Brooks. 2021. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *HPCA*. IEEE, 26–39.

[141] Oded Regev. 2005. On lattices, learning with errors, random linear codes, and cryptography. In *STOC*. ACM, 84–93.

[142] Yixuan Ren, Yixin Jie, Qingtao Wang, Bingbing Zhang, Chi Zhang, and Lingbo Wei. 2021. A hybrid secure computation framework for graph neural networks. In *PST*. IEEE, 1–6.

[143] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. 2019. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security Symposium*. 1501–1518.

[144] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A hybrid secure computation framework for machine learning applications. In *AsiaCCS*. ACM, 707–721.

[145] Mauro Ribeiro, Katarina Grolinger, and Miriam A. M. Capretz. 2015. MLaaS: Machine learning as a service. In *ICMLA*. IEEE, 896–902.

[146] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. 1978. On data banks and privacy homomorphisms. *Found. Secure Comput.* 4, 11 (1978), 169–180.

[147] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

[148] Mike Rosulek and Lawrence Roy. 2021. Three halves make a whole? beating the half-gates lower bound for garbled circuits. In *CRYPTO*. Springer, 94–124.

[149] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. 2018. DeepSecure: Scalable provably-secure deep learning. In *DAC*. ACM, 2:1–2:6.

[150] Lawrence Roy. 2022. oftSpokenOT: Communication-computation tradeoffs in OT extension. In Springer.

[151] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis R. Bach. 2022. AriaNN: Low-interaction privacy-preserving deep learning via function secret sharing. *Proc. Priv. Enhanc. Technol.* 2022, 1 (2022), 291–316.

[152] Alexander Schlögl and Rainer Böhme. 2020. eNNclave: Offline inference with model confidentiality. In *AISec@CCS*. ACM, 93–104.

[153] Y. Son, K. Han, Y. S. Lee, J. Yu, Y. H. Im, and S. Y. Shin. 2021. Privacy-preserving breast cancer recurrence prediction based on homomorphic encryption and secure two party computation. *PLoS ONE* 16, 12 (2021).

[154] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. 2015. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *SP*. IEEE Computer Society, 411–428.

[155] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. 2021. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *SP*. IEEE, 1021–1038.

[156] Tjerk Timan and Zoltan Mann. 2021. Data protection in the era of artificial intelligence: Trends, existing solutions and recommendations for privacy-preserving technologies. In *The Elements of Big Data Value: Foundations of the Research and Innovation Ecosystem*. Springer International Publishing, 153–175.

[157] Florian Tramèr and Dan Boneh. 2019. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR*. OpenReview.net.

[158] David Vadas and James R. Curran. 2011. Parsing noun phrases in the penn treebank. *Comput. Linguist.* 37, 4 (2011), 753–809.

[159] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-party secure computation for neural network training. *Proc. Priv. Enhanc. Technol.* 2019, 3 (2019), 26–49.

[160] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proc. Priv. Enhanc. Technol.* 2021, 1 (2021), 188–208.

[161] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. Retrieved from https://github.com/emp-toolkit.

[162] Yongqin Wang, G. Edward Suh, Wenjie Xiong, Benjamin Lefaudeux, Brian Knott, Murali Annavaram, and Hsien-Hsin S. Lee. 2022. Characterization of MPC-based private inference for transformer-based models. In *ISPASS*. IEEE, 187–197.

[163] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. 2022. Piranha: A GPU platform for secure computation. In *USENIX Security Symposium*. USENIX Association.

[164] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. 2020. Ferret: Fast extension for correlated OT with small communication. In *CCS*. ACM, 1607–1626.

[165] Andrew Chi-Chih Yao. 1982. Protocols for secure computations (extended abstract). In *FOCS*. 160–164.

[166] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets (extended abstract). In *FOCS*. 162–167.

[167] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*. Springer, 220–250.

[168] Qiao Zhang, Chunsheng Xin, and Hongyi Wu. 2021. Privacy-preserving deep learning based on multiparty secure computation: A survey. *IEEE IoT J.* 8, 13 (2021), 10412–10429.

[169] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. 2021. Cerebro: A platform for multi-party cryptographic collaborative learning. In *USENIX Security Symposium*. USENIX Association, 2723–2740.

[170] Wenxing Zhu, Mengqi Wei, Xiangxue Li, and Qiang Li. 2022. SecureBiNN: 3-party secure computation for binarized neural network inference. In *ESORICS*. Springer, 275–294.

Received 15 October 2022; revised 2 September 2023; accepted 26 September 2023