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

Zoltán Ádám Mann[1], Christian Weinert[2], Daphnee Chabal[1], and Joppe W. Bos[3]

[1] University of Amsterdam
[2] Royal Holloway, University of London
[3] NXP Semiconductors

Abstract. Neural networks (NNs) have become one of the most important tools for artificial intelligence (AI). 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 private data remains secret. This challenge belongs to the "privacy and data governance" dimension of trustworthy AI.
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 towards 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 survey paper is to give an overview of the main approaches proposed so far, their different properties, and the techniques used. In addition, remaining challenges towards large-scale deployments are identified.

Keywords: neural network, machine learning, deep learning, privacy preservation, homomorphic encryption, secure multi-party computation

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) [107]. Deep NNs are now routinely used in a variety of applications, including image recognition, natural language understanding, and drug discovery [174]. In these and many other domains, deep NNs have outperformed other ML techniques in terms of accuracy and are often competitive with the performance of humans [72].

Deep learning is fueled by data. In many cases, the involved data may be sensitive. For example, it can be personal data, subject to individuals' privacy concerns and data protection laws. Also non-personal data can be sensitive, for example if it represents business secrets or other intellectual property. If the involved data is sensitive, this puts constraints on how it can be used in DL. To foster privacy and data governance, a key dimension of trustworthy AI, there is a need for approaches that enable DL while ensuring that security and privacy requirements are met [23, 174]. 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 trade-off 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 [108], the input encodes a 28×28 grayscale picture of a hand-written 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 find the best parameter values of a NN. In the inference phase, the already

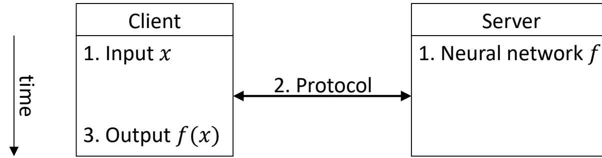


Fig. 1: Typical setup of secure neural network inference. The client obtains $f(x)$ through a protocol that ensures that both x and f remain secret.

trained NN is applied to a new input. While the operations performed in the two phases are similar, the requirements in terms of security and privacy are very different in the two phases. In this paper, 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 "Machine Learning as a Service" (MLaaS) scenario [155]. In this scenario, a company offers inference with a pre-trained NN as a service to its clients. As Fig. 1 shows, the client holds an input x, while the NN 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[4]) problem entails calculating $f(x)$ 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 approaches achieve different trade-offs 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 trade-offs are achieved, the fast progress has made it difficult to get an overview of this field.

The aim of this paper is to provide a guided tour of this fascinating research area. Previous relevant surveys presented a high-level overview of larger areas, covering different types of ML models, different attack possibilities, and different phases of the ML pipeline [23, 113, 134, 174]. 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[5].

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

2 Preliminaries

In the following, we detail relevant background information on NNs as well as secure computation techniques that are frequently used to build secure neural network inference solutions. As comprehending the construction of the relevant secure computation techniques requires advanced cryptographic knowledge, we refer the reader to [65,71,91] for more detailed explanations and background information.

2.1 Neural networks

A neural network (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

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

[5] 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 pre-prints.

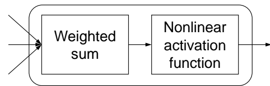


Fig. 2: A neuron.

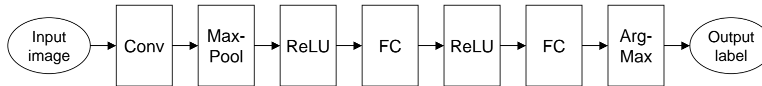


Fig. 3: An example convolutional neural network (based on [154]).

types of data [143]. For instance, words in a sentence can be represented by numbers such that words with similar meanings are represented with similar values [128]. 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 Fig. 2, the neuron gets as input a list of numbers $x_1, \ldots, x_n \in \mathbb{R}$. The neuron first computes $y = w_1 x_1 + \ldots + w_n x_n + b$, and then applies a non-linear function $f : \mathbb{R} \to \mathbb{R}$ 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 $\mathrm{ReLU}(y) = \max(0, y)$.

The NN consists of a sequence of layers[6]. 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 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[7]. 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 $\mathbb{R} \to \mathbb{R}$ 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, etc.
- Pooling layer: Computes output numbers by sliding a window of size k over the input, and applying an $\mathbb{R}^k \to \mathbb{R}$ 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. Fig. 3 for an example). In addition, the sizes of the layers are determined (note

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

[7] 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.

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 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 the slow wide-scale adoption and acceptance of new techniques such as processing sensitive data as is required in the MLaaS scenario. The privacy of sensitive information can be guaranteed if this data is encrypted by the data owner before being uploaded to a cloud service. In that way, only the legitimate data owner can access the data by decrypting it using their private decryption key. But this encryption limits the possibility to outsource *computation* on the externally stored information.

This problem can be solved with homomorphic encryption (HE), a privacy-enhancing technology introduced in the late 1970s by Rivest et al. [156]: the ability to compute meaningful operations on encrypted data. It took until 2009 until a concrete instantiation was found by Gentry [63]. This *fully homomorphic encryption* (FHE) allows an untrusted party to carry out arbitrary computation on *encrypted data* without learning anything about the content of this data.

The key to allowing arbitrary computations is that an FHE scheme allows both homomorphic addition and multiplication operations on the encrypted data: sufficient operations to implement any arithmetic circuit. Previous HE schemes were *partially homomorphic*, i.e., they only provided one of the two operations and therefore were not fully homomorphic. For example, the RSA cryptosystem [157] allows to compute an unrestricted number of modular multiplications on the encrypted data, while the Paillier cryptosystem [133] allows for as many modular additions as needed. Typically, the FHE schemes work with a polynomial ring $\mathbf{R}_q = \mathbb{Z}_q[X]/(f(X))$, where $f(X) \in \mathbb{Z}$ is a monic degree-d irreducible polynomial. In practice, $f(X)$ is chosen to be a power-of-two cyclotomic polynomial: $X^d + 1$, where $d = 2^k$ for some positive integer k. This allows for efficient arithmetic in $\mathbf{R}_q$ using the fast Fourier transform [142].

Ciphertexts of current FHE schemes inherently contain a certain amount of noise, which grows during homomorphic operations. This noise "pollutes" the ciphertext and makes correct decryption impossible, even with the legitimate decryption key, if the noise grows too large. To enable an *unrestricted* number of operations, the idea is to perform a re-encryption procedure homomorphically. This is called bootstrapping: it resets the noise in the ciphertext. Unfortunately, bootstrapping has a performance cost in practice. The research field of FHE is now in its fourth generation of schemes and approaches where the overall performance is getting more practical (especially for use cases such as NN inference) and the schemes support more features.

When the algorithm applied to the encrypted data is known in advance, one can use a *somewhat homomorphic encryption* (SHE) scheme, which only allows to perform a limited number of computational steps on the encrypted data. This enables one to fine-tune the parameters for this use case and avoid the usage of bootstrapping.

2.3 Oblivious transfer

Oblivious transfer (OT) is a cryptographic two-party protocol that allows the receiving party to obliviously select one of the sending party's inputs [147]. 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 a choice bit b. After invoking the OT protocol, the receiver learns x_b (and no information about x_{1-b}), and the sender learns nothing. As will be discussed in the following sections, this functionality is an instrumental building block for interactive secure computation protocols.

Several protocols were proposed to securely realize the described OT functionality [14,49,52,126,139]. Unfortunately, it was shown in [85] that OT inherently requires some form of cryptographic hardness assumptions and therefore has to utilize computationally expensive asymmetric cryptography. For example, the protocol of [49] is based on the Diffie-Hellman key-exchange protocol, which in turn requires modular exponentiations. To benefit from significantly faster symmetric primitives (e.g., via hardware-accelerated AES evaluations), *OT extension* protocols were constructed [5, 87, 93, 97, 160] that can generate a large number of OTs from a small constant number of base OTs, and for this extension step do not require asymmetric cryptography.

The amortized communication overhead of passively secure 1-out-of-2 OT extension protocols on l bit inputs [5, 87] is $\kappa + 2l$ bits per OT, where κ is the symmetric security parameter. There also exist variants of the OT functionality that have a reduced communication overhead and are sufficient for many applications. For example, in random OT, the inputs x_1, x_2 are randomly chosen by the protocol, which reduces the amortized communication overhead to κ bits per OT [5, 87]. It is also possible to pre-compute OT [12] to ensure fast protocol execution when the inputs are known.

Recently, a new line of work proposed *silent* OT extension [25,50,170] based on the stronger learning parity with noise (LPN) assumption, which introduces a computation-communication performance trade-off as it now is possible to generate a larger number of OT instances from small correlated seeds without further communication. For example, the communication of the protocol of [25] for random OT extension is amortized to 0.1 bit per random OT for 10^7 instances.

2.4 Garbled circuits

Garbled circuits (GCs) are a prominent approach for secure multi-party computation (MPC) introduced by Andrew Yao in the 1980s [171,172]. GCs allow two parties, a garbler and an evaluator, to interactively evaluate any efficiently computable function that can be expressed as a Boolean circuit consisting of only AND and XOR gates.

The protocol flow is as follows: The garbler creates the GC from a Boolean circuit that consists of the aforementioned gate types. For this, the garbler, for each wire w in the circuit, first assigns two wire labels $\tilde{w}^0, \tilde{w}^1$ that represent the logical values 0 and 1 and are random bitstrings in the length of the symmetric security parameter κ. Further, the garbler replaces the truth table for each gate g in the circuit with a garbled truth table $\tilde{g}$. The garbled truth table contains ciphertexts of the labels of the gate's output wire. It is constructed such that for each possible combination of two wire inputs, the output wire label that represents the logical value to which the gate would evaluate is encrypted using the corresponding wire labels as keys. This ensures that for the gate's two input wires w_i, w_j and one output wire w_o it holds that $\tilde{g}(\tilde{w}_i, \tilde{w}_j) = \tilde{w}_o^{g(w_i, w_j)}$. After transferring the GC, the evaluator can compute the output wire labels by decrypting the circuit gate by gate. Note that it is important for the garbler to randomly permute the entries of the garbled truth tables before the transfer as the evaluator can otherwise infer the plain logical values from the positions of the truth table entries it can decrypt. Finally, the garbler has the necessary information to map the output wire labels back to their corresponding logical values.

However, the evaluator must learn the wire labels that correspond to both parties' inputs for decrypting the first layer of gates. For the inputs of the garbler, the garbler simply selects the wire labels that correspond to its input bits and transfers them to the evaluator. The procedure for obtaining the wire labels that correspond to the evaluator's inputs is more complicated. For this, the parties run OT instances (one per input bit of the evaluator) as described in § 2.3, where the garbler is the sender in the OT protocol and the evaluator the receiver. The inputs for the OT sender are the wire labels corresponding to the logical values 0 and 1; the input for the OT receiver is the bit of the evaluator's input for which the wire label should be obtained.

The described protocol has a constant number of communication rounds, which is independent of the computed functionality. Over the years, many variations and optimizations for the garbling routine have been introduced [13,98,99,103,127,141,158,173] 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 [99] introduced the idea to let wire labels for 0 and 1 differ by a global constant, which 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 [173], the communication cost for each AND gate is 2κ and the computation cost is dominated by 4 AES evaluations on the garbler's and 2 AES evaluations on the evaluator's side.

Recently, a novel optimization reduced the communication overhead further to $1.5\kappa + 5$ bits [158] at the cost of a higher number of AES evaluations per gate (with at most 6 and 3 AES evaluations on the garbler's and evaluator's side, respectively). Another recent trend is extending the concept of GCs to enable garbled RAM computation by introducing features like conditional branching [74], vector gates [75], and oblivious array operations [76].

2.5 Additive secret sharing

Secret sharing is an approach for distributing a secret value via two or more "shares" that on their own do not reveal any information. 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 an input 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 the value of x. This approach can be used for interactively computing Boolean or arithmetic circuits over private inputs, which is known as the MPC protocol of Goldreich, Micali, and Wigderson (GMW) [66].

In the following, we describe the basics for securely computing a function using the GMW protocol between two parties P_0, P_1 who have respective private inputs x and y. First, both parties use A-SS to generate shares of their input x_0, x_1, 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 gates for $\mathbb{Z}_2$ or addition gates for larger rings) can be done without interaction between the parties, similar to GCs with FreeXOR [99]. For this, the parties both locally apply the respective operation on the shares they hold.

For non-linear gates (AND gates for $\mathbb{Z}_2$ or multiplication gates for larger rings), interaction between the parties is necessary. The parties exchange the shares of the multiplication inputs masked with *Beaver multiplication triplets* (MTs) [11]. MTs consist of values a, b, c such that $c = a \cdot b$. Shares of such triplets can be pre-computed, for example, using random OTs [5, 51, 94] or HE [41, 95, 131, 150]. Recent pre-computation techniques also include *silent* protocols with a communication complexity that is sub-linear in the circuit size [26, 27]. 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 [135].) Finally, after all layers of the circuit are evaluated, the shares of the output wires can be reconstructed to obtain the overall computation result.

Note that in comparison 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* secure neural network inference (SNNI) approaches work (cf. § 4), this section investigates *what* such approaches assume and what they guarantee. Tab. 1 gives an overview about the most important characteristics of some selected approaches. Details are provided in the following subsections.

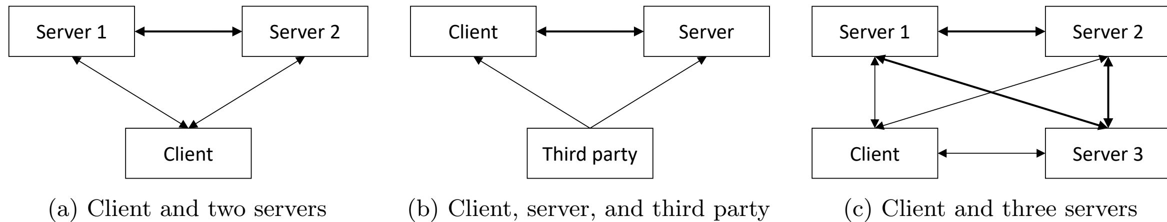
3.1 Number and roles of participants

Most works on SNNI consider two participants (cf. Fig. 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 [125] assumes the participation of two servers (cf. Fig. 4a). 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

Table 1: Characteristics of selected SNNI approaches. 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.

Approach	Parties (§ 3.1)			Attack (§ 3.2)		Secret (§ 3.2)				NN limitations (§ 3.3)	Link to training (§ 3.4)	Interactive (§ 3.5)
	Client	Server	Other	HbC	Malicious	Input	Result	Weights	Architecture			
Chameleon [154]	1	1	1	✓		✓	✓	✓				Yes
CrypTFlow [105]	0	3		✓	✓	✓	✓	✓				Yes
CrypTFlow2 [149]	1	1		✓		✓	✓	✓				Yes
CryptoNets [64]	1	1		✓		✓	✓	✓	✓	Polynomials, limited depth	Postprocessing	No
DeepSecure [159]	1	1		✓		✓	✓	✓			Pre- & postprocessing	No
Delphi [122]	1	1		✓		✓	✓	✓			ReLU modification	Yes
Falcon [167]	0	3		✓	✓	✓	✓	✓				Yes
Gazelle [90]	1	1		✓		✓	✓	✓				Yes
MiniONN [112]	1	1		✓		✓	✓	✓				Yes
SecureML [125]	0	2		✓		✓	✓	✓				Yes
SiRnn [148]	1	1		✓		✓	✓	✓				Yes
XONN [153]	1	1		✓		✓	✓	✓	✓	Binary weights	Pre- & postprocessing	No



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

Fig. 4: Some possible setups of the participants. Thick arrows show the main flow of the protocol, thin arrows show the flow of additional information.

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 [154] assumes, beyond the client and the server, a third party (cf. Fig. 4b). 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 MT pre-computation. 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.

Falcon [167] takes the idea of SecureML further and uses three instead of two servers (cf. Fig. 4c). 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. On the other hand, 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 [124], ASTRA [43], BLAZE [136], SecureNN [166], and SWIFT [100].

Some works like FLASH [37], Trident [44], and Tetrad [101] take it even further by operating in a 4-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 [124].

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 multi-party computation (MPC) [78], 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 [105]. 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 $3\times$ as high with Aramis than without the transformation.
- ABY^3 provides specific protocols for SNNI in a 3-party setting, either in the HbC or in the malicious adversary model [124]. The protocols for the malicious adversary model have a higher complexity, but they were not implemented, so their practical performance was not evaluated.
- XONN is claimed to be easily upgradeable to security against malicious adversaries [153], but no practical experience with such an upgrade is provided.
- Falcon devises and also implements specific protocols for security against malicious adversaries [167]. 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 [37], SWIFT [100], Trident [44] and Tetrad [101] 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 (GOD), 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 [110]. 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:

- The input to the inference remains the client's secret.
- The result of the inference is only learned by the client.
- The weights and biases of the neural network 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, second, and third point, i.e., the secrecy of the client's input, the inference result, and the server's weights and biases. The fourth point is addressed only by some approaches. In particular, interactive approaches (cf. § 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 § 3.2.2 are satisfied, the client does not learn anything about the NN *beyond what the result of the inference reveals.* The result of the inference

Table 2: Neural network layer types supported by selected SNNI approaches. FC: fully-connected; BN: batch normalization.

Approach	FC	Conv	Tanh	Sigmoid	ReLU	Square	Sign	MaxPool	AvgPool	SumPool	ArgMax	BN	$\frac{1}{\sqrt{x}}$
CrypTFlow2 [149]	✓	✓			✓			✓	✓		✓		
CryptoNets [64]	✓	✓				✓				✓			
Falcon [167]	✓	✓			✓			✓				✓	
Gazelle [90]	✓	✓			✓	✓		✓					
SiRnn [148]	✓	✓	✓	✓	✓			✓			✓	✓	✓
XONN [153]	✓	✓					✓	✓				✓	

may leak some information about the NN though. Using a sufficient number of inference queries, the client may be able to infer secret information [110, 153]. 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 [153]. 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. Tab. 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 [149]. 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 [64].

Some approaches also support other layer types. For example, Falcon supports batch normalization [167], while SiRnn supports reciprocal of square root, which is used in some RNNs [148].

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 $\{1, -1\}$ [24], or *binary* NNs, where both the weights and activation values can only take the values $\{1, -1\}$ [153]. These limitations can be exploited for efficiently implementing the involved arithmetic operations, but may threaten the accuracy of the NN.

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. Fig. 5a), such as Falcon [167]. On the other extreme, several approaches are completely independent from the training process, starting from a pre-trained model and not modifying it (cf. Fig. 5b), such as in the case of CryptFlow [105].

Several authors suggested to take a pre-trained model and transform it to make it more appropriate for SNNI (cf. Fig. 5c). 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 [64]. FHE-DiNN discretizes weights and exchanges activation functions in the trained NN to make it compatible with the proposed inference process [24]. 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 [159].

Some approaches require more significant changes to the training process (cf. Fig. 5d). 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 [122]. In XONN, the neurons in each layer are replicated to compensate the decrease in accuracy stemming from limiting the weights to $\{1, -1\}$ [153]. 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 [90].

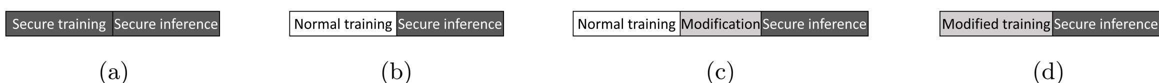
On the other hand, interactive approaches have two drawbacks: First, they leak information about the structure of the NN (number and types of layers) from the server to the client, in contrast to non-interactive approaches where the structure of the NN can be kept secret (cf. the last property in § 3.2.2). Second, they 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 [125], MiniONN [112], Chameleon [154], Delphi [122], and Falcon [167]. Typical activities for the offline phase include the generation of Beaver multiplication triplets or other sets of correlated numbers for masking secrets (cf. § 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



(a) (b) (c) (d)

Fig. 5: Possibilities for the integration with training.

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 § 3, we now look at the way how these approaches actually work. In § 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 § 4.4, we describe approaches that combine different cryptographic primitives, while § 4.5 discusses aspects that are relevant for different types of solution approaches. Finally, in § 4.6, we present some other approaches that are not based on the above cryptographic primitives.

4.1 Homomorphic encryption

Implementing privacy-preserving inference by means of homomorphic encryption (HE) is a natural idea since fully homomorphic encryption allows one to compute any functionality over encrypted data. 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 encrypted data, this approach satisfies all security properties listed in § 3.2.2.

Using HE for SNNI was already suggested in the earliest work on the topic [9, 130]. However, turning this idea into a practical method has proved quite challenging. In the following, we discuss the most important aspects.

4.1.1 Handling non-linear functions Computing the linear part in each node of the network can be done with most HE schemes as $\mathrm{Enc}(y_{ij}) = w_{i0} + \sum_j \mathrm{Enc}(x_{ij}) \cdot w_{ij}$, where the in- and output are encrypted but the weights and bias are in plain. Computing non-linear activation functions such as ReLU, sigmoid, or max becomes the main problem for (fully) homomorphic encryption schemes. 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, although this is often still 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.

This problem can be (partially) circumvented by simplifying the activation function. For example, CryptoNets uses the square function as activation function and summing for pooling [64]. This approach was reported to deliver good results in a neural network with 5 layers (from which 2 layers use 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. This same square function is also used by LoLa [33]. We return to the issues of polynomial approximation in § 4.5.6. A different approach is proposed in FHE-DiNN [24]. 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. This problem is circumvented entirely in [21] where Ivakhnenko's group method of data handling [88] 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 [90]. We come back to such hybrid usage of privacy-preserving techniques in § 4.4.

4.1.2 Homomorphic encryption schemes There are quite a number of different HE approaches and schemes. A survey on (fully) homomorphic encryption schemes can be found in [1], and [120] surveys the engineering aspects of FHE. As outlined in § 2.2, there are the partially homomorphic schemes which can do either additions or multiplications on encrypted data. These approaches have

Table 3: Performance comparison between selected HE schemes as used in HE-based solutions: CryptoNets (CN), Faster CryptoNets (FCN), Low Latency Privacy Preserving Inference (LoLa) and Shift-accumulation-based LHE-enabled deep neural network (SHE). The number of homomorphic multiplications between plaintext and ciphertext (P × C), ciphertext and ciphertext (C × C), homomorphic comparisons, inference (inf in seconds), and accuracy (acc, a percentage) are shown for MNIST and CIFAR-10.

approach	MNIST [57]					CIFAR-10 [104]				
	#muls P × C	#muls C × C	#cmp	inf (sec)	acc (%)	#muls P × C	#muls C × C	#cmp	inf (sec)	acc (%)
CN [64]	296k	945	0	205	99.0	-	-	-	-	-
FCN [48]	24k	945	0	98.7	39.1	350M	64k	0	39k	76.7
LoLa [33]	10.5k	1.6k	0	2.2	99.0	61k	15k	0	730	74.1
SHE [115]	945	0	3k	9.3	99.5	13k	0	16k	2258	92.5

been considered in the setting of NNs [9, 130]. The first concrete instantiation that could do both and is *fully homomorphic* was found by Gentry [63]. Examples of the second generation of FHE schemes are BGV [29], BFV [28, 60], and yashe [22]. These have a slower noise growth and are more efficient compared to the first generation. The third generation improved the bootstrapping [30, 47, 58] while finally the fourth and latest generation of schemes such as CKKS [46] work especially well with applications that use floating point arithmetic such as NNs.

In the encryption scheme used by CryptoNets [64] (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 cleartexts into a single ciphertext. Performing an operation on the ciphertext translates to performing an operation on multiple cleartexts. 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. In the specific NN implemented with CryptoNets, up to 4096 inputs can be packed into a single ciphertext. Although the latency of CryptoNets is quite high, up to 4096 inputs can be processed simultaneously, potentially leading to a high throughput. Similarly, while ciphertext size is quite high for a single input, the same ciphertext size can accommodate up to 4096 cleartext inputs, leading to a much lower ciphertext size per input. However, SIMD processing is only possible if a sufficient number of inputs have to be processed at the same time.

The Low Latency Privacy Preserving Inference (LoLa) framework [33] is based on the second generation BFV FHE scheme. Here, every cleartext 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 (i.e., shifting its coordinates to the right, moving the ith coordinate to the position $(i+k) \bmod n$ for a given k). This opens up new possibilities, as illustrated by the following toy example from [33]. Assume we have a neuron that calculates its output y from its 4 inputs $x_1, \ldots, x_4$ and 4 weights $w_1, \ldots, w_4$. CryptoNets would encrypt each input as a separate message and then use 4 multiplications and 3 additions to calculate $y = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$. In contrast, LoLa packs the inputs into the 4-dimensional vector $x = (x_1, x_2, x_3, x_4)$ and the weights into the 4-dimensional vector $w = (w_1, w_2, w_3, w_4)$. The coordinate-wise multiplication of these two vectors results in $p = (w_1x_1, w_2x_2, w_3x_3, w_4x_4)$. Rotating p by 1 results in $p' = (w_4x_4, w_1x_1, w_2x_2, w_3x_3)$. Adding p and p' results in $s = (w_1x_1 + w_4x_4, w_2x_2 + w_1x_1, w_3x_3 + w_2x_2, w_4x_4 + w_3x_3)$. Rotating s by 2 results in $s' = (w_3x_3 + w_2x_2, w_4x_4 + w_3x_3, w_1x_1 + w_4x_4, w_2x_2 + w_1x_1)$. Finally, adding s and s' results in the vector (y, y, y, y) which contains the desired result in all of its coordinates. Thus, the result was achieved using 1 multiplication, 2 additions, and 2 rotations, which is potentially much more efficient than the normal method with 4 multiplications and 3 additions.

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) [115] framework is based on a different scheme: the Torus FHE scheme (TFHE) [47]. Under the hood, ciphertexts are expressed over the torus modulo one, which results in fast binary operations over encrypted binary bits. This, in turn, means that this is ideal to implement the ReLU activation and the max pooling by Boolean operations. In order to give an indication about the practicality of these approaches, the popular databases MNIST [57] and CIFAR-10 [104] are often used to verify and benchmark the privacy-preserving inference. Tab. 3 shows inference latency in seconds and accuracy figures for several popular approaches. This shows that FHE approaches get faster but cannot be considered practical yet: e.g., SHE manages to achieve over 90 % accuracy on CIFAR-10 but each single inference on encrypted input takes almost 40 minutes.

Recent work [109] also applies the fourth generation FHE scheme CKKS [46] to CIFAR-10. It is shown that using a relatively large machine with 112-cores, one can reach 92.4 % accuracy with almost three hours per inference.

4.2 Garbled circuits

If the bitlength of the involved numbers is fixed, 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 garbled circuit (GC) protocol can be used to evaluate this Boolean circuit in a secure way (cf. § 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. The structure of the NN is known to both parties. Thus, the last secrecy goal from § 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.

DeepSecure was the first practical SNNI approach based mainly on GCs [159]. It makes use of the "Free-XOR" optimization [99], 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. § 3.4), which are independent of the GC protocol.

A different approach is followed by XONN [153]. The main idea is to use binary NNs, i.e., restrict 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 [122].

4.3 Additive secret sharing

The techniques described so far (HE and garbled circuits) allow a non-interactive evaluation of the NN on the server side (cf. § 3.5). In contrast, the approaches based on additive secret sharing (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 first layer, and also more than two parties are possible (cf. § 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 § 2.5, adding two 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 in § 4.3.1.

For evaluating *non-linear layers* efficiently, no general recipe is known. Rather, a large variance of different techniques has been proposed, as discussed in § 4.3.2.

4.3.1 Generating Beaver triplets Beaver triplets do not depend on the input data, and can thus be generated in an offline preprocessing phase (cf. § 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, the other uses oblivious transfers. For example, MiniONN uses the protocol based on HE [112]. SecureML offers both protocols as alternatives because in some cases one is more efficient, in other cases the other [125].

If there is a third party that does not collude with the first two parties, 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 [154]. It should also be noted that with three-party 2-out-of-3 secret sharing in the online phase, multiplications can be performed without precomputed Beaver triplets, as is done for example in Falcon [167].

4.3.2 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 [125] and in Delphi [122].

Another option is to apply a general cryptographic protocol like garbled circuits (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 [125]. MiniONN also uses GCs for ReLU, max-pooling, and a piecewise linear approximation of sigmoid [112]. Note that these methods, in contrast to the approaches of § 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 GCs, oblivious transfers (OTs) 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, and MUX [149]. 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 [148]. 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 [167].

4.4 Mixed-protocol approaches

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. On the other hand, such approaches are necessarily interactive, which also leads to some drawbacks (cf. § 3.5).

The general scheme for evaluating an NN with L layers entails the following steps:

1. Initialization to prepare the input to the evaluation sub-protocol of the first layer[8].
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 [112, 125]. Chameleon uses A-SS for linear layers, the Goldreich, Micali, and Wigderson (GMW) protocol for ReLU activations, and GCs for argmax [154]. Gazelle uses HE for linear layers and GCs for non-linear layers [90]. CrypTFlow2 uses either HE or OTs for linear layers, and A-SS with proprietary sub-protocols for non-linear layers [149]. Delphi uses A-SS for linear layers and for square activations, and GCs for ReLU activations [122].

It should be noted that all these combinations are within the online phase. Independently of this, the offline phase may use further protocols (cf. § 3.6 and § 4.3.1).

4.5 Other aspects

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

4.5.1 Use of oblivious transfer In contrast to HE, GCs, and A-SS, oblivious transfer (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. § 2.4). Thus, all approaches that use GCs implicitly also use OT.
- OT offers a possible way for generating Beaver multiplication triplets (cf. § 2.5 and § 4.3.1). Thus, some of the approaches using A-SS, such as SecureML [125], 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 [149].

4.5.2 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 Fig. 6), scale (also called precision, the number of bits for the fractional part). These decisions may have significant impact, resulting in different trade-offs between efficiency and the achievable accuracy. In particular, reducing bitwidth may significantly improve efficiency, depending on the used techniques. E.g., if GCs are used, the size of the circuit may decrease if the bitwidth is reduced, thus leading to less computation and less communication. On the other hand, a reduced bitwidth may limit the achievable accuracy.

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

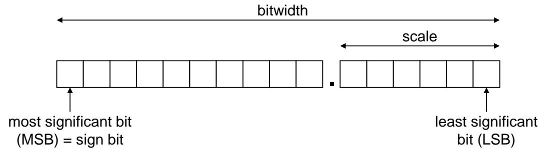


Fig. 6: Encoding a real number on a finite number of bits.

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 [122, 167], while SecureML and SecureNN use a fixed bitwidth of 64 [125, 166]. On the other hand, 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 [148]. 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, a truncation (i.e., division by a power of 2) is necessary after operations like addition or multiplication [125, 167].

Regarding the precision of the fractional part, different solutions have been proposed. Some approaches use a fixed-point representation with a uniform scale. For example, Delphi uses 15 bits to represent the fractional part, whereas Falcon uses 16 bits for the same purpose [122, 167]. Other approaches determine the precision dynamically. For example, CrypTFlow2 determines both the bitwidth and the scale by means of autotuning [149]. Some approaches, such as MiniONN, work with integers, represented on a fixed number of bits [112]. 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 [64]. These transformations are made possible by the Chinese remainder theorem. It is possible to work with fractional (non-integer) numbers using FHE methods by representing these numbers in a fixed-point representation. This requires mapping such numbers to the polynomial ring used in practice: an efficient encoding method for fixed-point numbers tailored for homomorphic function evaluation is given in [20].

4.5.3 Parallelization

In today's computers, more and more parallel computing units are available, including multicore CPUs and GPUs [118]. 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 [82]. As mentioned in § 4.1.2, some HE schemes support SIMD processing, making it possible to perform the same operation on a batch of different inputs [64] or even batching numbers belonging to the same input [24, 33, 90].

The offline phase (cf. § 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 [125]. A similar approach is used also by MiniONN [112].

4.5.4 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 [167].

Securely generating such correlated random numbers is a non-trivial task. We have already discussed this for Beaver triplets (cf. § 4.3.1), 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 (PRF). 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 [154]. In a 3-party setup, Falcon uses pairwise shared random keys (one for each pair of parties) to generate different types of correlated random numbers [167].

4.5.5 Neural Architecture Search (NAS) As discussed in § 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 different candidate architectures, training them and assessing the achieved accuracy. This process can be automated by a neural architecture search (NAS) algorithm.

Delphi uses NAS for a specific purpose [122]: It replaces some ReLU activations in the NN with the square function, aiming at finding a good trade-off 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 [15]. 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 [116].

4.5.6 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 appropriate 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 [122].

One possible way of resolving this is by using piecewise polynomial approximation, such as in MiniONN [112] and in the approach of Huang et al. [82]. 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 [122]: 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.

4.5.7 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, 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 [105].

Another optimization is proposed in CrypTFlow2 [149]. 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.

4.5.8 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 [111]. 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, 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 selected building blocks appears to be intuitively secure, however, the papers mostly do not provide a dedicated proof, e.g., in the universal composability (UC) framework [39].

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 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 in case attacks arise.

Only few solutions design custom MPC protocols for particular components and thus are required to provide additional security proofs (e.g., [43, 101]).

Many of the fully homomorphic encryption schemes are based on a variant of the ring-LWE-based HE scheme. Work in this area began with Ajtai's seminal paper [3] (cf. the survey [137] for a comprehensive list of relevant references). Regev's work introduced the "learning with errors" (LWE) problem [151], which relates to solving a "noisy" linear system modulo a known integer. A specialized version if this is often used in practice: the ring-LWE problem [117, 138]. 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 which 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.

4.6 Other approaches

The majority of the proposed approaches fit into the categories discussed so far. However, there are some approaches that use completely different underlying techniques, such as model splitting (cf. § 4.6.1) or trusted execution environments (cf. § 4.6.2). Moreover, there are approaches that provide compilers from high-level machine learning code to cryptographic protocols (cf. § 4.6.3).

4.6.1 Model splitting The idea of model splitting is to split the NN into two or more parts that are allocated to different parties, thus limiting the knowledge of each party. Such approaches have the promise of achieving a useful level of secrecy without computationally costly cryptographic operations. However, it is not easy to obtain security guarantees, or even good practical levels of security, by model splitting alone [73, 132, 145].

To preserve the privacy of the input data, [132] proposes that the client extracts features from the input data by computing the initial layers of the NN. The client additionally obfuscates the extracted features and sends them to the server. The server performs denoising and dimensionality reconstruction, before being able to evaluate the remaining layers of the NN. The authors empirically demonstrated that the information about the input that the server can reconstruct diminishes, if a sufficient number of layers is allocated to the client. On the other hand, the used obfuscation techniques may lead to an accuracy/privacy trade-off.

[73] demonstrated that adding noise does not prevent model inversion attacks, and that features from the input data can be reconstructed from intermediary results after at least 6 layers. The authors

proposed two privacy-enhancing solutions. The first, setting to zero the output of some neurons every layer, led to significant accuracy drop, while the second, assigning more layers to the client, including at least one fully connected layer, would give the client much information about the neural network's architecture as well as a large amount of computation to perform.

In another variant of model splitting, the server offloads some computations during inference to a third party. An example, [144], performs computations for the first half of the layers on a local server and uses HE to evaluate the remainder of the layers in the cloud, before sending encrypted results back to the server. This method however does nothing to keep the input and output secret from the server. [114] proposes a method to preserve both input and output data privacy during model splitting. Their solution consists of mixing together several inputs, during both training and inference, so that the input vector of the neural network does not reveal personal identities. Intermediary outputs are returned to the client who holds a demixing algorithm to obtain individual inference results. It is estimated that mixing 8 images is enough to prevent attacks. This method also suffers from some accuracy drop.

[6] proposed splitting an NN between up to 120 devices, ensuring that no device had knowledge of a full layer of the NN. This solution was designed for CNNs using images as inputs, and predicated on the fact that in the input layer filters of small dimensions are applied to extract specific features from the input images. The resulting vectors, called feature maps, are then distributed between nodes. This solution did not lead to an accuracy drop, and higher privacy levels were even associated with lower latency, as splitting the model between more devices increased possibilities of parallel computing. However, this solution has no security defenses in cases of collusion between devices.

4.6.2 Trusted execution environment Hardware-based trusted execution environments (TEEs) provide a practical solution to enable secure computation in an untrusted environment. For this, TEEs use hardware-level isolation and encryption techniques to guarantee confidentiality of data and integrity of execution. Via mechanisms for remote attestation, the authenticity of the TEE can be proven to parties providing confidential inputs for the computation.

A prominent example of widely deployed TEEs is Intel's Software Guard Extensions (SGX). SGX provides developers the opportunity to create secure *enclaves*, i.e., programs that can be executed in isolation from all other software running on the same device, including the operating system. TEEs are potentially a viable alternative to HE and MPC protocols described in § 2, operating at almost native speed and with barely any communication overhead.

SGX has been criticized for a) providing no built-in protection against software side-channel attacks, leaving this non-trivial task to enclave developers and b) for the significant trust in Intel's hardware and Intel's key management required from users of SGX. Enclave developers must carefully harden their code by ensuring that no secret-dependent branching or memory accesses occur and that secret-dependent operations run in constant time. Confidential information could be extracted from enclaves due to this problematic [32, 35, 67, 68, 123], although there are approaches to automate the hardening process [61]. Researchers were also able to present speculative execution-based attacks on SGX that do not require a vulnerability of the enclave code [34].

A major concern regarding the performance of SGX for memory-intensive machine learning tasks was the initially limited amount of available enclave page cache (EPC) memory (only up to 128 MB). The EPC is an area of protected processor reserved memory (PRM) that essentially acts as the enclave's main memory, whereas higher memory requirements are met by running costly paging mechanisms where parts of the EPC are encrypted and securely outsourced to untrusted main memory. However, this restriction was recently lifted with the introduction of the "Ice Lake" processor family that supports up to 1 TB of EPC memory on the Intel Xeon SP [86].

A number of works have demonstrated the suitability of SGX for secure neural network inference. Slalom [164] builds an SGX library for inference that securely outsources compute-intensive tasks to untrusted devices (e.g., a co-located GPU). secureTF [146] allows to run unmodified TensorFlow applications in SGX enclaves while considering distributed setups and aforementioned memory restrictions of SGX for intense workloads. eNNclave [161] addresses the issue of limited EPC memory by splitting TensorFlow models in public and private layers, where only the private layers are executed inside an SGX enclave; this setup is suitable, for example, in transfer learning (cf. § 4.6.1). MLCapsule [69] encapsulates neural network evaluation in an SGX enclave and provides defenses against reverse engineering of model specifications, model stealing, and membership inference. Ohrimenko et

al. [129] address software side-channel attacks, which are possible as SGX reveals data-dependent access patterns. The authors propose data-oblivious primitives (e.g., for comparisons and array accesses) and ML algorithms, including neural network inference. VoicGuard [31] demonstrates the feasibility of privacy-preserving real-time speech recognition based on SGX.

Another example of widely deployed TEEs is ARM TrustZone, predominantly shipped in mobile devices, for example, in most Android-based smartphones. The concept of TrustZone is to simultaneously run a normal and a secure world. In the secure world, a trusted operating system and trusted applications provided by verified vendors are executed. With TrustZone, SNNI can be enabled on the mobile device instead of a remote server, which can be useful in offline usage scenarios, or when software vendors want to advertise increased input security to their customers, as confidential client data can remain on the mobile device. A TrustZone-based security framework is utilized to demonstrate secure offline speech recognition in real time by "offline model guard" (OMG) [10]. OMG provides a "TensorFlow Lite for Microcontroller" environment as a user-space enclave and can be generalized for other SNNI tasks.

4.6.3 Compilers For developers without a background in cryptography, it can be challenging or even impossible to apply some of the solutions discussed in § 4. For example, it can be unclear how to choose HE parameters, or which sub-protocols to choose for different layers in hybrid solutions. As a remedy, some researchers developed compilers that either directly integrate with existing machine learning frameworks or at least automate the translation from high-level machine learning code to secure computation frameworks.

Examples from the first group are Rosetta [45] and secureTF [146]. Developers create regular NN inference code in standard ML frameworks like TensorFlow, and the operations are automatically mapped to a secure back-end that performs the evaluation in a privacy-preserving manner. In Rosetta [45], simply by importing a Python package and defining private inputs, regular TensorFlow code will be securely evaluated using the 3-party framework SecureNN [166] as a cryptographic back-end. ngraph-HE [19] and ngraph-HE2 [18] extend Intel's graph compiler ngraph, which interfaces between ML frameworks like TensorFlow or PyTorch and hardware architectures, with an HE backend that allows for privacy-preserving evaluation of NNs and automatically takes care of HE-specific optimizations such as packing. Building on top, MP2ML [17] supports both HE and MPC as cryptography backends for ngraph with conversions between the two secure computation paradigms. CHET [53] translates a domain-specific language (DSL) for describing tensor circuits into optimized encrypted circuits that can be securely evaluated using different HE schemes. This compilation includes an automatic and optimal parameter selection for the chosen HE scheme. Also EzPC [42] proposes a DSL, which is automatically compiled and securely executed using the two-party frameworks ABY [56] or EMP [168]. The framework uses a heuristic to automatically find the best split between execution as an arithmetic or garbled Boolean circuit. CrypTen [96] provides an API that closely resembles that of PyTorch while mapping the operations to arithmetic and binary versions of the GMW protocol (cf. § 2.5). secureTF [146] allows to run unmodified TensorFlow applications in Intel SGX enclaves.

There are also compilers that translate high-level programming languages to circuit formats that can then be evaluated by secure computation frameworks based on HE, GCs, and A-SS [36, 77, 79, 102]. HyCC [36, 55] translates ANSI C code implementing NN evaluation into a circuit format that can be securely evaluated by the two-party framework ABY [56]. Other works applied hardware synthesis tools for secure computation circuit compilation [54, 163]. Such tools are also used to compile optimized building blocks, e.g., for XONN [153].

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 Tab. 4.

Most of the available implementations are research prototypes rather than production-ready software. They were specifically created to serve 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.

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 [135]	C++		ENCRYPTO Utils [59]			✓	✓	✓	
ABY^3 [124]	C++		libOTe [140]	✓		✓	✓	✓	
Chameleon [154]	C/C++		ABY [56]			✓	✓	✓	
Cheetah [83]	C++		EMP [168], SEAL [121]	✓		✓		✓	✓
CrypTFlow2 [149]	Python	C++	EMP [168], SEAL [121]	✓	✓	✓		✓	✓
CryptoNets [64], LoLa [33]	C#		SEAL [121]	✓					✓
Delphi [122]	Python	C++, Rust	fancy-garbling [62], SEAL [121]	✓		✓	✓	✓	✓
EzPC [42]	DSL	C++	ABY [56], EMP [168]	✓		✓	✓	✓	
Gazelle [90]	C++		JustGarble [13], libOTe [140], OpenFHE [7]	✓		✓	✓	✓	✓
MiniONN [112]	C++		ABY [56], SEAL [121]	✓		✓	✓	✓	✓
MP2ML [17]	Python	C++	ABY [56], SEAL [121]	✓	✓	✓	✓	✓	✓
Muse [110]	C++, Rust		JustGarble [13], MP-SPDZ [92], SEAL [121]	✓		✓	✓	✓	✓
ngraph-HE / HE2 [18,19]	Python	C++	SEAL [121]	✓	✓				✓
SecureML [125]	C++		EMP [168]			✓	✓	✓	✓
XONN [153]	Python, DSL	C++	libOTe [140]		✓	✓	✓		

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., [175]), it is more common to use Python only for the integration or front-end part and rely on aC/C++ implementation for the cryptographic back-end (e.g., [17–19, 105, 122, 146, 149, 161]). Several SNNI solutions build on established OT, MPC, or HE libraries such as the libOTe OT library [140], the mixed MPC-protocol framework ABY [54], or the SEAL FHE library [121].

GPU acceleration is standard in ML training and inference; however, there are few successful attempts of GPU-accelerated secure computation [169]. Thus, few solutions support (partial) GPU acceleration for SNNI [122]. 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 [13]. Support for AES-NI is provided by most Intel and AMD CPUs, as well as recent ARM chips as part of their cryptography extensions (CE).

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 [83], while others have a more flexible domain-specific language [42], and some directly integrate with well-known ML frameworks such as TensorFlow or PyTorch (e.g., [17–19, 105, 149, 153]). 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. Most TEE-based solutions reviewed in § 4.6.2 satisfy this criterion, as they simply package an ML framework inside an enclave (e.g., [10, 146, 161]).

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. [70] 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 (§ 6.1), the technical setup of experiments (§ 6.2), the used metrics (§ 6.3), and discuss how different approaches are compared (§ 6.4).

6.1 Machine learning problems, NNs, and datasets

Most works on SNNI focus on image classification problems. Mostly standard open-source convolutional neural networks (CNNs) were used to evaluate solutions [64, 167]. CNNs are particularly well studied [2, 16], 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, AlexNet (composed of 7 and 8 layers, respectively). Larger CNNs used regularly are VGG16, VGG19, ResNet50, DenseNet121, and SqueezeNet [16].

Most of these CNNs are trained to perform two simple image classification tasks, using two customary datasets: MNIST [57] and CIFAR-10 [104]. Both datasets are composed of small low-resolution images falling into ten categories (28x28 handwritten digits for MNIST and 32x32 animals and vehicles for CIFAR-10).

Some studies [18, 154, 167] also evaluate their solutions using CNNs that were developed by other work in SNNI, like DeepSecure [159], MiniONN [112], SecureNN [166], GAZELLE [90], or SecureML [125], all using the MNIST or CIFAR-10 classification task.

Other types of NNs are less commonly used in this field. Some works focused on recurrent neural networks (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 [159] proposed both CNNs and RNNs as novel benchmarks, with RNNs developed to classify audio and sensor data. [112] additionally developed a long-short-term-memory (LSTM)-RNN trained on the Penn Treebank dataset [165] to predict likely next words given previous words. [162] and [8] performed classification of 6 datasets of time series (e.g., EEG measurements and motion sensors). [81] 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 spatial location compared to other nodes, have also been studied in the context of SNNI. [152] and [89] addressed classification of nodes from two widespread citation network datasets, CORA (to classify 2708 publications into 7 classes) [38] and CITESEER [40] (to classify 3312 scientific publication into 6 classes).

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 [48, 83, 84, 153, 154, 159]. 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 of [112], for example, both the server and the client run on an Intel Core i5 CPU with 4 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. Most of those experimental setups do not mention whether using a WAN or LAN setting and what is the resulting network latency, although WAN is unlikely to be used in the lab environment (except when simulated by means of traffic control) [174].

Other authors used cloud-based virtual machines, often hosted in Amazon's AWS cloud. For example, SecureML was run on two Amazon EC2 c4.8xlarge machines with up to 36 vCPUs and 60 GB of RAM each [125]. 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 [122, 166, 167].

6.3 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. § 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 considered. Communication is important because some cryptographic techniques incur significant overhead and 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. § 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. § 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 [80]. 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 [132].

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 § 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 § 3, such as security properties, are notoriously hard to measure or even 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, 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.

7 Future research directions

The field of SNNI made huge progress in recent years. Nevertheless, we identified several challenges but also promising new directions that will likely lead to even more intensive research in this field. In the following, we describe these future research areas using the same structure that we used for reviewing the state of the art in § 3–§ 6.

7.1 Characteristics of SNNI approaches (cf. § 3)

Security vs. efficiency trade-off. A key problem in SNNI is how to achieve high efficiency and a high level of security at the same time. The first solution approaches were very 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. In

addition, approaches that guarantee security against malicious adversaries only started to appear recently, and typically incur higher overhead. The quest for a two-party approach that protects against malicious adversaries and guarantees all secrecy goals of § 3.2.2, while only incurring minimal overhead, is still ongoing.

Impact of the application context. Much research has been done to find the "best" approach. However, what the best approach is may depend on the specifics of the context in which the approach is applied. For example, in one application, keeping the architecture of the NN secret may be important, while it may be unimportant in another application. 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, and so on. More research is needed to understand how properties of the application context influence the appropriateness of different approaches.

SNNI in IoT and edge computing. A particular application context in which SNNI could play an important role is the Internet of Things (IoT) and the related edge computing paradigm [106, 119]. A challenge of this application context is that the resources available especially on the client side are severely constrained. This limits the applicability of interactive approaches that put a significant burden on the client. Devising efficient approaches for this context remains a challenging and important task for future research.

Secure-inference-friendly NN and training. Some papers have already shown examples how efficiency can be improved by considering not only the inference phase, but also the choice of NN architecture and the training (cf. § 3.3 and § 3.4). Choices of the NN architecture, the allowed numbers, the way the network is trained and 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. A systematic investigation of such possibilities is needed.

Defence against black-box attacks. As pointed out in § 3.2, most SNNI approaches do not protect against black-box attacks, such as model extraction attacks. Some potential countermeasures to protect against such attacks, such as limiting the number of queries a client can make, are orthogonal to the operation of SNNI. However, also the secure inference process itself may be hardened against such attacks, as done to some extent in the case of XONN [153]. We expect to see more research on this in the future.

7.2 Solution approaches (cf. § 4)

New building blocks. Many of the solution frameworks surveyed in § 4 are based on techniques from the era of 2016-2019. However, as hinted at in § 2, there are exciting new developments in terms of secure computation building blocks, for example, "silent" OT and MPC protocols that introduce a new communication-computation trade-off. Since only very recent developments make use of such building blocks [83], we suggest to revisit the solutions proposed beforehand and re-evaluate whether retrospectively "upgrading" their building blocks can have a significant positive impact on their overall performance. Doing so might even challenge the fundamental architecture of solutions, especially if they are hybrid solutions where now the selection and assignment of building blocks to layer types must be revisited.

Modular solutions. Most solutions proposed for SNNI so far operate with a single or a small set of secure computation approaches. A few offer exchanging parts of the building blocks (e.g., generating multiplication triplets via either OT or HE [125]) or address two different levels of security (e.g., [101]). In the future, much more modular solutions could be developed that allow to arbitrarily scale the number of computing parties, support different adversaries models and secrecy goals, work for a wide range of model architectures, and allow variations of other characteristics of SNNI solutions. This would enable users to stay flexible instead of being required to adopt a completely different solution when requirements change.

Scalability of solutions. While recent language models reach sizes in the order of three billion parameters, SNNI solutions today struggle even with much smaller NN architectures such as ResNet50 in terms of memory, computation, and communication requirements. Hence, it needs more efforts to find ways to securely distribute the inherent overhead of secure computation techniques among multiple nodes to be able to handle the workload with commodity hardware.

7.3 Implementation (cf. § 5)

ML framework integration and unified API. As observed in § 5, only a subset of solutions is implemented in a way that allows users who are not cryptography experts to deploy and adapt them to their use case. Those that are most usable provide a cryptography back-end for popular ML frameworks such as TensorFlow or PyTorch, which allows to run existing code almost without modifications in a secure manner. In order for more solutions to move in that direction, a *unified cryptography API* would be helpful. If such an API exists and is accepted as a de-facto standard, future solutions for SNNI would have stronger incentives to provide an integration. This would also contribute to the challenge of limited comparability between solutions, as a certain ML task could then be easily evaluated on different back-ends. However, designing such an API is challenging: On the one hand, the API should not introduce too much additional burden for the user (e.g., in the form of annotations to mark private inputs). On the other hand, an over-simplified API could limit the potential for optimizations in the design of efficient solutions.

GPU support. As described in § 5.1, although ML training and inference are often GPU-driven, SNNI approaches rarely make use of these accelerators and run almost exclusively on CPUs. However, especially for A-SS-based solutions, there is significant potential to operate on GPUs. This is because – after the generation of multiplication triplets – A-SS protocols execute simple arithmetic operations over large matrices consisting of shares represented by 32 or 64 bit integers.

Container builds. As pointed out in § 5.2, available implementations of SNNI solutions are often hard to 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 the same benchmarking environment (cf. § 6.4). We suggest to follow the approach of [70] and package implementations in container formats such as Docker to make them readily available for experimentation. Together with the proposed unified API and ML framework integration, this is one important step towards fair performance comparisons between SNNI solutions (cf. § 6.4).

7.4 Evaluation of solution approaches (cf. § 6)

Improved comparisons. As described in § 6.4, the current practice of comparing different approaches is quite limited. This should mature in the future, with authors performing more direct and fair empirical comparisons, running multiple competing solutions in the same setup. There is also a need for papers whose main goal is to perform large-scale empirical comparative studies.

More varied benchmarks. As described in § 6.1, evaluation is currently mostly limited to a few benchmarks. To obtain more meaningful results, experiments with a larger variety of machine learning problems, NN architectures, and datasets are needed. Domains other than image classification should be explored, as well as NN types other than CNNs.

Realistic technical setup. In § 6.2, we described the typical technical setup used in the evaluation, which might be very different from a real-world setup. Experiments with resource-constrained clients, with servers serving many different clients etc. should also be performed.

Metrics beyond efficiency. As described in § 6.3, evaluation is currently mostly focused on efficiency. However, the practicality of an approach depends also on many other aspects that should also be evaluated and quantified as much as possible. Metrics to be considered include energy consumption, security, privacy, and usability.

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 towards 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.

Acknowledgements

The work of Z.Á. Mann and D. Chabal was partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement no. 871525 (FogProtect).

References

1. Acar, A., Aksu, H., Uluagac, A. S., and Conti, M. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv. 51*, 4 (2018), 79:1–79:35.
2. Ajit, A., Acharya, K., and Samanta, A. A review of convolutional neural networks. In *2020 international conference on emerging trends in information technology and engineering (ic-ETITE)* (2020), IEEE, pp. 1–5.
3. Ajtai, M. Generating hard instances of lattice problems (extended abstract). In *STOC* (1996), ACM, pp. 99–108.
4. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R., et al. Status report on the third round of the nist post-quantum cryptography standardization process. *NIST, Tech. Rep. NISTIR 8413* (2022).
5. Asharov, G., Lindell, Y., Schneider, T., and Zohner, M. More efficient oblivious transfer extensions. *J. Cryptol. 30*, 3 (2017), 805–858.
6. Baccour, E., Erbad, A., Mohamed, A., Hamdi, M., and Guizani, M. Distprivacy: Privacy-aware distributed deep neural networks in iot surveillance systems. In *GLOBECOM* (2020), IEEE, pp. 1–6.
7. Badawi, A. A., Bates, J., Bergamaschi, F., et al. Openfhe: Open-source fully homomorphic encryption library. https://github.com/openfheorg/.
8. Bakshi, M., and Last, M. Cryptornn - privacy-preserving recurrent neural networks using homomorphic encryption. In *CSCML* (2020), Springer, pp. 245–253.
9. Barni, M., Orlandi, C., and Piva, A. A privacy-preserving protocol for neural-network-based computation. In *MM&Sec* (2006), ACM, pp. 146–151.
10. Bayerl, S. P., Frassetto, T., Jauernig, P., Riedhammer, K., Sadeghi, A., Schneider, T., Stapf, E., and Weinert, C. Offline model guard: Secure and private ML on mobile devices. In *DATE* (2020), IEEE, pp. 460–465.
11. Beaver, D. Efficient multiparty protocols using circuit randomization. In *CRYPTO* (1991), Springer, pp. 420–432.
12. Beaver, D. Precomputing oblivious transfer. In *CRYPTO* (1995), Springer, pp. 97–109.
13. Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy* (2013), IEEE Computer Society, pp. 478–492.
14. Bellare, M., and Micali, S. Non-interactive oblivious transfer and spplications. In *CRYPTO* (1989), Springer, pp. 547–557.
15. Bian, S., Jiang, W., Lu, Q., Shi, Y., and Sato, T. NASS: optimizing secure inference via neural architecture search. In *ECAI* (2020), vol. 325 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, pp. 1746–1753.
16. Bianco, S., Cadène, R., Celona, L., and Napoletano, P. Benchmark analysis of representative deep neural network architectures. *IEEE Access 6* (2018), 64270–64277.
17. Boemer, F., Cammarota, R., Demmler, D., Schneider, T., and Yalame, H. MP2ML: a mixed-protocol machine learning framework for private inference. In *ARES* (2020), ACM, pp. 14:1–14:10.
18. Boemer, F., Costache, A., Cammarota, R., and Wierzynski, C. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *WAHC@CCS* (2019), ACM, pp. 45–56.
19. Boemer, F., Lao, Y., Cammarota, R., and Wierzynski, C. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In *CF* (2019), ACM, pp. 3–13.
20. Bonte, C., Bootland, C., Bos, J. W., Castryck, W., Iliashenko, I., and Vercauteren, F. Faster homomorphic function evaluation using non-integral base encoding. In *CHES* (2017), Springer, pp. 579–600.
21. Bos, J. W., Castryck, W., Iliashenko, I., and Vercauteren, F. Privacy-friendly forecasting for the smart grid using homomorphic encryption and the group method of data handling. In *AFRICACRYPT* (2017), pp. 184–201.
22. Bos, J. W., Lauter, K. E., Loftus, J., and Naehrig, M. Improved security for a ring-based fully homomorphic encryption scheme. In *IMACC* (2013), Springer, pp. 45–64.
23. Boulemtafes, A., Derhab, A., and Challal, Y. A review of privacy-preserving techniques for deep learning. *Neurocomputing 384* (2020), 21–45.
24. Bourse, F., Minelli, M., Minihold, M., and Paillier, P. Fast homomorphic evaluation of deep discretized neural networks. In *CRYPTO* (2018), Springer, pp. 483–512.

25. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Rindal, P., and Scholl, P. Efficient two-round OT extension and silent non-interactive secure computation. In *CCS* (2019), ACM, pp. 291–308.
26. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., and Scholl, P. Efficient pseudorandom correlation generators from ring-lpn. In *CRYPTO* (2020), pp. 387–416.
27. Boyle, E., Gilboa, N., Ishai, Y., and Nof, A. Sublinear gmw-style compiler for MPC with preprocessing. In *CRYPTO* (2021), Springer, pp. 457–485.
28. Brakerski, Z. Fully homomorphic encryption without modulus switching from classical gapsvp. In *CRYPTO* (2012), Springer, pp. 868–886.
29. Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (leveled) fully homomorphic encryption without bootstrapping. In *ITCS* (2012), ACM, pp. 309–325.
30. Brakerski, Z., and Vaikuntanathan, V. Lattice-based FHE as secure as PKE. In *ITCS* (2014), ACM, pp. 1–12.
31. Brasser, F., Frassetto, T., Riedhammer, K., Sadeghi, A., Schneider, T., and Weinert, C. Voiceguard: Secure and private speech processing. In *INTERSPEECH* (2018), ISCA, pp. 1303–1307.
32. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., and Sadeghi, A. Software grand exposure: SGX cache attacks are practical. In *WOOT* (2017), USENIX Association.
33. Brutzkus, A., Gilad-Bachrach, R., and Elisha, O. Low latency privacy preserving inference. In *ICML* (2019), vol. 97 of *Proceedings of Machine Learning Research*, PMLR, pp. 812–821.
34. Bulck, J. V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium* (2018), USENIX Association, pp. 991–1008.
35. Bulck, J. V., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security Symposium* (2017), pp. 1041–1056.
36. Büscher, N., Demmler, D., Katzenbeisser, S., Kretzmer, D., and Schneider, T. Hycc: Compilation of hybrid protocols for practical secure computation. In *CCS* (2018), ACM, pp. 847–861.
37. Byali, M., Chaudhari, H., Patra, A., and Suresh, A. FLASH: fast and robust framework for privacy-preserving machine learning. *Proc. Priv. Enhancing Technol. 2020*, 2 (2020), 459–480.
38. Cabanes, C., Grouazel, A., von Schuckmann, K., et al. The cora dataset: validation and diagnostics of in-situ ocean temperature and salinity measurements. *Ocean Science 9*, 1 (2013), 1–18.
39. Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS* (2001), IEEE Computer Society, pp. 136–145.
40. Caragea, C., Wu, J., Ciobanu, A. M., Williams, K., Ramírez, J. P. F., Chen, H., Wu, Z., and Giles, C. L. Citeseer x : A scholarly big dataset. In *ECIR* (2014), Springer, pp. 311–322.
41. Catalano, D., Raimondo, M. D., Fiore, D., and Giacomelli, I. Mon$\mathbb{Z}_{2^k}$a: Fast maliciously secure two party computation on $\mathbb{Z}_{2^k}$. In *Public-Key Cryptography – PKC 2020* (2020), pp. 357–386.
42. Chandran, N., Gupta, D., Rastogi, A., Sharma, R., and Tripathi, S. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *EuroS&P* (2019), IEEE, pp. 496–511.
43. Chaudhari, H., Choudhury, A., Patra, A., and Suresh, A. ASTRA: high throughput 3pc over rings with application to secure prediction. In *CCSW@CCS* (2019), ACM, pp. 81–92.
44. Chaudhari, H., Rachuri, R., and Suresh, A. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS* (2020), The Internet Society.
45. Chen, Y., Huang, G., Shi, J., Xie, X., and Yan, Y. Rosetta: A Privacy-Preserving Framework Based on TensorFlow. `https://github.com/LatticeX-Foundation/Rosetta`, 2020.
46. Cheon, J. H., Kim, A., Kim, M., and Song, Y. S. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT* (2017), Springer, pp. 409–437.
47. Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol. 33*, 1 (2020), 34–91.
48. Chou, E., Beal, J., Levy, D., Yeung, S., Haque, A., and Fei-Fei, L. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *CoRR abs/1811.09953* (2018).
49. Chou, T., and Orlandi, C. The simplest protocol for oblivious transfer. In *LATINCRYPT* (2015), Springer, pp. 40–58.
50. Couteau, G., Rindal, P., and Raghuraman, S. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In *CRYPTO* (2021), Springer, pp. 502–534.
51. Cramer, R., Damgård, I., Escudero, D., Scholl, P., and Xing, C. Spd$\mathbb{F}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO* (2018), Springer, pp. 769–798.
52. Damgård, I., Nielsen, J. B., and Orlandi, C. Essentially optimal universally composable oblivious transfer. In *ICISC* (2008), Springer, pp. 318–335.
53. Dathathri, R., Saarikivi, O., Chen, H., Laine, K., Lauter, K. E., Maleki, S., Musuvathi, M., and Mytkowicz, T. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *PLDI* (2019), ACM, pp. 142–156.

54. Demmler, D., Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., and Zeitouni, S. Automated synthesis of optimized circuits for secure computation. In *CCS* (2015), ACM, pp. 1504–1517.
55. Demmler, D., Katzenbeisser, S., Schneider, T., Schuster, T., and Weinert, C. Improved circuit compilation for hybrid MPC via compiler intermediate representation. In *SECRYPT* (2021), SCITEPRESS, pp. 444–451.
56. Demmler, D., Schneider, T., and Zohner, M. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS* (2015), The Internet Society.
57. Deng, L. The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Process. Mag. 29*, 6 (2012), 141–142.
58. Ducas, L., and Micciancio, D. FHEW: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT* (2015), Springer, pp. 617–640.
59. ENCRYPTO Group. Encrypto utils.
60. Fan, J., and Vercauteren, F. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* (2012), 144.
61. Felsen, S., Kiss, Á., Schneider, T., and Weinert, C. Secure and private function evaluation with intel SGX. In *CCSW@CCS* (2019), ACM, pp. 165–181.
62. Galois Inc. fancy-garbling. https://github.com/GaloisInc/fancy-garbling.
63. Gentry, C. Fully homomorphic encryption using ideal lattices. In *STOC* (2009), ACM, pp. 169–178.
64. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K. E., Naehrig, M., and Wernsing, J. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML* (2016), vol. 48 of *JMLR Workshop and Conference Proceedings*, JMLR.org, pp. 201–210.
65. Goldreich, O. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.
66. Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC* (1987), ACM, pp. 218–229.
67. Götzfried, J., Eckert, M., Schinzel, S., and Müller, T. Cache attacks on intel SGX. In *EUROSEC* (2017), ACM, pp. 2:1–2:6.
68. Hähnel, M., Cui, W., and Peinado, M. High-resolution side channels for untrusted operating systems. In *USENIX Annual Technical Conference* (2017), USENIX Association, pp. 299–312.
69. Hanzlik, L., Zhang, Y., Grosse, K., Salem, A., Augustin, M., Backes, M., and Fritz, M. Mlcapsule: Guarded offline deployment of machine learning as a service. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (2021), pp. 3300–3309.
70. Hastings, M., Hemenway, B., Noble, D., and Zdancewic, S. Sok: General purpose compilers for secure multi-party computation. In *IEEE Symposium on Security and Privacy* (2019), IEEE, pp. 1220–1237.
71. Hazay, C., and Lindell, Y. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010.
72. He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV* (2015), IEEE Computer Society, pp. 1026–1034.
73. He, Z., Zhang, T., and Lee, R. B. Attacking and protecting data privacy in edge-cloud collaborative inference systems. *IEEE Internet Things J. 8*, 12 (2021), 9706–9716.
74. Heath, D., and Kolesnikov, V. Stacked garbling - garbled circuit proportional to longest execution path. In *CRYPTO* (2020), Springer, pp. 763–792.
75. Heath, D., and Kolesnikov, V. One hot garbling. In *CCS* (2021), ACM, pp. 574–593.
76. Heath, D., Kolesnikov, V., and Ostrovsky, R. Epigram: Practical garbled RAM. In *EUROCRYPT* (2022), Springer, pp. 3–33.
77. Heldmann, T., Schneider, T., Tkachenko, O., Weinert, C., and Yalame, H. Llvm-based circuit compilation for practical secure computation. In *Applied Cryptography and Network Security* (2021), pp. 99–121.
78. Heurix, J., Zimmermann, P., Neubauer, T., and Fenz, S. A taxonomy for privacy enhancing technologies. *Comput. Secur. 53* (2015), 1–17.
79. Holzer, A., Franz, M., Katzenbeisser, S., and Veith, H. Secure two-party computations in ANSI C. In *CCS* (2012), ACM, pp. 772–783.
80. Hou, J., Liu, H., Liu, Y., Wang, Y., Wan, P.-J., and Li, X.-Y. Model protection: Real-time privacy-preserving inference service for model privacy at the edge. *IEEE Trans. on Dependable and Secure Computing* (2021).
81. Hsiao, S., Liu, Z., Tso, R., Kao, D., and Chen, C. Privgru: Privacy-preserving GRU inference using additive secret sharing. *J. Intell. Fuzzy Syst. 38*, 5 (2020), 5627–5638.
82. Huang, K., Liu, X., Fu, S., Guo, D., and Xu, M. A lightweight privacy-preserving CNN feature extraction framework for mobile sensing. *IEEE Trans. Dependable Secur. Comput. 18*, 3 (2021), 1441–1455.
83. Huang, Z., Lu, W., Hong, C., and Ding, J. Cheetah: Lean and fast secure two-party deep neural network inference. In *USENIX Security Symposium* (2022), USENIX Association.

84. Ibarrondo, A., Chabanne, H., and Önen, M. Banners: Binarized neural networks with replicated secret sharing. In *IH&MMSec* (2021), ACM, pp. 63–74.
85. Impagliazzo, R., and Rudich, S. Limits on the provable consequences of one-way permutations. In *STOC* (1989), ACM, pp. 44–61.
86. Intel Corporation. Intel xeon scalable platform built for most sensitive workloads, 2020.
87. Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. In *CRYPTO* (2003), Springer, pp. 145–161.
88. Ivakhnenko, A. Heuristic self-organization in problems of engineering cybernetics. *Automatica 6*, 2 (1970), 207–219.
89. Jie, Y., Ren, Y., Wang, Q., Xie, Y., Zhang, C., Wei, L., and Liu, J. Multi-party secure computation with intel SGX for graph neural networks. In *ICC* (2022), IEEE, pp. 528–533.
90. Juvekar, C., Vaikuntanathan, V., and Chandrakasan, A. P. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security Symposium* (2018), USENIX Association, pp. 1651–1669.
91. Katz, J., and Lindell, Y. *Introduction to Modern Cryptography, Second Edition.* CRC Press, 2014.
92. Keller, M. MP-SPDZ: A versatile framework for multi-party computation. In *CCS* (2020), ACM, pp. 1575–1590.
93. Keller, M., Orsini, E., and Scholl, P. Actively secure OT extension with optimal overhead. In *CRYPTO* (2015), Springer, pp. 724–741.
94. Keller, M., Orsini, E., and Scholl, P. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *CCS* (2016), ACM, pp. 830–842.
95. Keller, M., Pastro, V., and Rotaru, D. Overdrive: Making SPDZ great again. In *EUROCRYPT* (2018), Springer, pp. 158–189.
96. Knott, B., Venkataraman, S., Hannun, A. Y., Sengupta, S., Ibrahim, M., and van der Maaten, L. Crypten: Secure multi-party computation meets machine learning. In *NeurIPS* (2021), pp. 4961–4973.
97. Kolesnikov, V., and Kumaresan, R. Improved OT extension for transferring short secrets. In *CRYPTO* (2013), Springer, pp. 54–70.
98. Kolesnikov, V., Mohassel, P., and Rosulek, M. Flexor: Flexible garbling for XOR gates that beats free-xor. In *CRYPTO* (2014), Springer, pp. 440–457.
99. Kolesnikov, V., and Schneider, T. Improved garbled circuit: Free XOR gates and applications. In *ICALP* (2008), Springer, pp. 486–498.
100. Koti, N., Pancholi, M., Patra, A., and Suresh, A. SWIFT: super-fast and robust privacy-preserving machine learning. In *USENIX Security Symposium* (2021), USENIX Association, pp. 2651–2668.
101. Koti, N., Patra, A., Rachuri, R., and Suresh, A. Tetrad: Actively secure 4pc for secure training and inference. In *NDSS* (2022), The Internet Society.
102. Kreuter, B., Shelat, A., Mood, B., and Butler, K. R. B. PCF: A portable circuit format for scalable two-party secure computation. In *USENIX Security Symposium* (2013), USENIX Association, pp. 321–336.
103. Kreuter, B., Shelat, A., and Shen, C. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium* (2012), USENIX Association, pp. 285–300.
104. Krizhevsky, A. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009.
105. Kumar, N., Rathee, M., Chandran, N., Gupta, D., Rastogi, A., and Sharma, R. Cryptflow: Secure tensorflow inference. In *IEEE Symposium on Security and Privacy* (2020), IEEE, pp. 336–353.
106. Lachner, C., Mann, Z. Á., and Dustdar, S. Towards understanding the adaptation space of ai-assisted data protection for video analytics at the edge. In *ICDCS Workshops* (2021), IEEE, pp. 7–12.
107. LeCun, Y., Bengio, Y., and Hinton, G. E. Deep learning. *Nature 521*, 7553 (2015), 436–444.
108. LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE 86*, 11 (1998), 2278–2324.
109. Lee, J., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y., and No, J. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access 10* (2022), 30039–30054.
110. Lehmkuhl, R., Mishra, P., Srinivasan, A., and Popa, R. A. Muse: Secure inference resilient to malicious clients. In *USENIX Security Symposium* (2021), USENIX Association, pp. 2201–2218.
111. Lindell, Y. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography.* Springer International Publishing, 2017, pp. 277–346.
112. Liu, J., Juuti, M., Lu, Y., and Asokan, N. Oblivious neural network predictions via minionn transformations. In *CCS* (2017), ACM, pp. 619–631.
113. Liu, Q., Li, P., Zhao, W., Cai, W., Yu, S., and Leung, V. C. M. A survey on security threats and defensive techniques of machine learning: A data driven view. *IEEE Access 6* (2018), 12103–12117.
114. Liu, Z., Wu, Z., Gan, C., Zhu, L., and Han, S. Datamix: Efficient privacy-preserving edge-cloud inference. In *ECCV (11)* (2020), Springer, pp. 578–595.

115. Lou, Q., and Jiang, L. SHE: A fast and accurate deep neural network for encrypted data. In *NeurIPS* (2019), pp. 10035–10043.
116. Lou, Q., and Jiang, L. HEMET: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture. In *ICML* (2021), vol. 139 of *Proceedings of Machine Learning Research*, PMLR, pp. 7102–7110.
117. Lyubashevsky, V., Peikert, C., and Regev, O. On ideal lattices and learning with errors over rings. In *EUROCRYPT* (2010), Springer, pp. 1–23.
118. Mann, Z. Á. GPGPU: hardware/software co-design for the masses. *Comput. Informatics 30*, 6 (2011), 1247–1257.
119. Mann, Z. Á. Security- and privacy-aware iot application placement and user assignment. In *Computer Security – ESORICS 2021 International Workshops* (2021), Springer, pp. 296–316.
120. Martins, P., Sousa, L., and Mariano, A. A survey on fully homomorphic encryption: An engineering perspective. *ACM Comput. Surv. 50*, 6 (2018), 83:1–83:33.
121. Microsoft Research. Microsoft SEAL. `https://github.com/Microsoft/SEAL`.
122. Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., and Popa, R. A. Delphi: A cryptographic inference service for neural networks. In *USENIX Security Symposium* (2020), USENIX Association, pp. 2505–2522.
123. Moghimi, A., Irazoqui, G., and Eisenbarth, T. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES* (2017), Springer, pp. 69–90.
124. Mohassel, P., and Rindal, P. Aby^3: A mixed protocol framework for machine learning. In *CCS* (2018), ACM, pp. 35–52.
125. Mohassel, P., and Zhang, Y. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE Symposium on Security and Privacy* (2017), IEEE Computer Society, pp. 19–38.
126. Naor, M., and Pinkas, B. Efficient oblivious transfer protocols. In *SODA* (2001), ACM/SIAM, pp. 448–457.
127. Naor, M., Pinkas, B., and Sumner, R. Privacy preserving auctions and mechanism design. In *EC* (1999), ACM, pp. 129–139.
128. Naseem, U., Razzak, I., Khan, S. K., and Prasad, M. 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.
129. Ohrimenko, O., Schuster, F., Fournet, C., Mehta, A., Nowozin, S., Vaswani, K., and Costa, M. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium* (2016), pp. 619–636.
130. Orlandi, C., Piva, A., and Barni, M. Oblivious neural network computing via homomorphic encryption. *EURASIP J. Inf. Secur. 2007* (2007).
131. Orsini, E., Smart, N. P., and Vercauteren, F. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In *CT-RSA* (2020), Springer, pp. 254–283.
132. Osia, S. A., Shamsabadi, A. S., Sajadmanesh, S., Taheri, A., Katevas, K., Rabiee, H. R., Lane, N. D., and Haddadi, H. A hybrid deep learning architecture for privacy-preserving mobile analytics. *IEEE Internet Things J. 7*, 5 (2020), 4505–4518.
133. Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT* (1999), Springer, pp. 223–238.
134. Papernot, N., McDaniel, P. D., Sinha, A., and Wellman, M. P. Sok: Security and privacy in machine learning. In *EuroS&P* (2018), IEEE, pp. 399–414.
135. Patra, A., Schneider, T., Suresh, A., and Yalame, H. ABY2.0: improved mixed-protocol secure two-party computation. In *USENIX Security Symposium* (2021), USENIX Association, pp. 2165–2182.
136. Patra, A., and Suresh, A. BLAZE: blazing fast privacy-preserving machine learning. In *NDSS* (2020), The Internet Society.
137. Peikert, C. A decade of lattice cryptography. *Found. Trends Theor. Comput. Sci. 10*, 4 (2016), 283–424.
138. Peikert, C., Regev, O., and Stephens-Davidowitz, N. Pseudorandomness of ring-lwe for any ring and modulus. In *STOC* (2017), ACM, pp. 461–473.
139. Peikert, C., Vaikuntanathan, V., and Waters, B. A framework for efficient and composable oblivious transfer. In *CRYPTO* (2008), Springer, pp. 554–571.
140. Peter Rindal, L. R. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. `https://github.com/osu-crypto/libOTe`.
141. Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. Secure two-party computation is practical. In *ASIACRYPT* (2009), Springer, pp. 250–267.
142. Pollard, J. M. The fast Fourier transform in a finite field. *Mathematics of computation 25*, 114 (1971), 365–374.
143. Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M. E. P., Shyu, M., Chen, S., and Iyengar, S. S. A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv. 51*, 5 (2019), 92:1–92:36.

144. Qiang, W., Liu, R., and Jin, H. Defending CNN against privacy leakage in edge computing via binary neural networks. *Future Gener. Comput. Syst. 125* (2021), 460–470.
145. Qiu, H., Zheng, Q., Zhang, T., Qiu, M., Memmi, G., and Lu, J. Toward secure and efficient deep learning inference in dependable iot systems. *IEEE Internet Things J. 8*, 5 (2021), 3180–3188.
146. Quoc, D. L., Gregor, F., Arnautov, S., Kunkel, R., Bhatotia, P., and Fetzer, C. securetf: A secure tensorflow framework. In *Middleware* (2020), ACM, pp. 44–59.
147. Rabin, M. O. How to exchange secrets with oblivious transfer. *TR-81 edition, Aiken Computation Lab, Harvard University* (1981).
148. Rathee, D., Rathee, M., Goli, R. K. K., Gupta, D., Sharma, R., Chandran, N., and Rastogi, A. Sirnn: A math library for secure RNN inference. In *IEEE Symposium on Security and Privacy* (2021), IEEE, pp. 1003–1020.
149. Rathee, D., Rathee, M., Kumar, N., Chandran, N., Gupta, D., Rastogi, A., and Sharma, R. Cryptflow2: Practical 2-party secure inference. In *CCS* (2020), ACM, pp. 325–342.
150. Rathee, D., Schneider, T., and Shukla, K. K. Improved multiplication triple generation over rings via rlwe-based AHE. In *CANS* (2019), Springer, pp. 347–359.
151. Regev, O. On lattices, learning with errors, random linear codes, and cryptography. In *STOC* (2005), ACM, pp. 84–93.
152. Ren, Y., Jie, Y., Wang, Q., Zhang, B., Zhang, C., and Wei, L. A hybrid secure computation framework for graph neural networks. In *PST* (2021), IEEE, pp. 1–6.
153. Riazi, M. S., Samragh, M., Chen, H., Laine, K., Lauter, K. E., and Koushanfar, F. XONN: xnor-based oblivious deep neural network inference. In *USENIX Security Symposium* (2019), USENIX Association, pp. 1501–1518.
154. Riazi, M. S., Weinert, C., Tkachenko, O., Songhori, E. M., Schneider, T., and Koushanfar, F. Chameleon: A hybrid secure computation framework for machine learning applications. In *AsiaCCS* (2018), ACM, pp. 707–721.
155. Ribeiro, M., Grolinger, K., and Capretz, M. A. M. Mlaas: Machine learning as a service. In *ICMLA* (2015), IEEE, pp. 896–902.
156. Rivest, R. L., Adleman, L., and Dertouzos, M. L. On data banks and privacy homomorphisms. *Foundations of secure computation 4*, 11 (1978), 169–180.
157. Rivest, R. L., Shamir, A., and Adleman, L. M. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (1978), 120–126.
158. Rosulek, M., and Roy, L. Three halves make a whole? beating the half-gates lower bound for garbled circuits. In *CRYPTO* (2021), Springer, pp. 94–124.
159. Rouhani, B. D., Riazi, M. S., and Koushanfar, F. Deepsecure: scalable provably-secure deep learning. In *DAC* (2018), ACM, pp. 2:1–2:6.
160. Roy, L. oftspokenot: Communication-computation tradeoffs in OT extension. In *CRYPTO* (2022), Springer.
161. Schlögl, A., and Böhme, R. ennclave: Offline inference with model confidentiality. In *AISec@CCS* (2020), ACM, pp. 93–104.
162. Son, Y., Han, K., Lee, Y., Yu, J., Im, Y., and Shin, S. Privacy-preserving breast cancer recurrence prediction based on homomorphic encryption and secure two party computation. *Plos one 16*, 12 (2021), e0260681–e0260681.
163. Songhori, E. M., Hussain, S. U., Sadeghi, A., Schneider, T., and Koushanfar, F. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *IEEE Symp. Security and Privacy* (2015), pp. 411–428.
164. Tramèr, F., and Boneh, D. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR* (2019), OpenReview.net.
165. Vadas, D., and Curran, J. R. Parsing noun phrases in the penn treebank. *Comput. Linguistics 37*, 4 (2011), 753–809.
166. Wagh, S., Gupta, D., and Chandran, N. Securenn: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol. 2019*, 3 (2019), 26–49.
167. Wagh, S., Tople, S., Benhamouda, F., Kushilevitz, E., Mittal, P., and Rabin, T. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proc. Priv. Enhancing Technol. 2021*, 1 (2021), 188–208.
168. Wang, X., Malozemoff, A. J., and Katz, J. Emp-toolkit: Efficient multiparty computation toolkit.
169. Watson, J., Wagh, S., and Popa, R. A. Piranha: A GPU platform for secure computation. In *USENIX Security Symposium* (2022), USENIX Association.
170. Yang, K., Weng, C., Lan, X., Zhang, J., and Wang, X. Ferret: Fast extension for correlated OT with small communication. In *CCS* (2020), ACM, pp. 1607–1626.
171. Yao, A. C. Protocols for secure computations (extended abstract). In *FOCS* (1982), IEEE Computer Society, pp. 160–164.
172. Yao, A. C. How to generate and exchange secrets (extended abstract). In *FOCS* (1986), IEEE Computer Society, pp. 162–167.

173. Zahur, S., Rosulek, M., and Evans, D. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT* (2015), Springer, pp. 220–250.
174. Zhang, Q., Xin, C., and Wu, H. Privacy-preserving deep learning based on multiparty secure computation: A survey. *IEEE Internet Things J. 8*, 13 (2021), 10412–10429.
175. Zhu, W., Wei, M., Li, X., and Li, Q. Securebinn: 3-party secure computation for binarized neural network inference. In *ESORICS* (2022), Springer, pp. 275–294.